Metodi del Calcolo Scientifico

Progetto 2

Biancini Mattia 865966
Gargiulo Elio 869184

Indice

1 Introduzione

Per questo progetto abbiamo utilizzato Python come linguaggio di programmazione, mentre ci siamo affidati a SciPy come libreria da cui importare le funzioni di `DCT2` e `IDCT2` nella seconda parte del progetto.
Il progetto si divide in due sotto-progetti relativi alle 2 parti che compongono la consegna e una directory per le immagini:

- `imgs` » Directory che contiene le immagine fornite in allegato alla consegna
- `part1` » Sotto-progetto relativo alla prima parte della consegna
  - Implementazione DCT2
  - Analisi della DCT2 fornita da SciPy e confronto con la nostra versione
- `part2` » Sotto-progetto relativo alla seconda parte della consegna
  - Import di un'immagine in toni di grigi
  - Ricostruzione dell'immagine compressa dopo l'elaborazione

Ogni progetto è composto di due file:

- `functions.py` che raccoglie tutte le funzioni da noi usate come se fosse una libreria del nostro progetto
- `main.py` esecutivo del relativo sotto-progetto

2 Implementazione della DCT2 - Parte 1

La `Discrete Cosine Transform 2` (o `DCT2`) è una trasformata che converte un insieme di dati in input in una serie di coefficienti che rappresentano l'ampiezza delle diverse frequenze presenti nei dati originali (si passa alla base dei coseni). La `DCT2` è utilizzata in applicazioni di compressione dei dati, come nel caso dei formati di compressione audio come MP3 e, nel nostro specifico caso, per immagini come JPEG. La `DCT2`, data una matrice generica $A \in \mathbb{R}^{N \times M}$ e un insieme di matrici W_{kl} che sono una base ortogonale rispetto al prodotto interno, restituisce i valori dei coefficienti α_{kl} nell'equazione:

$$A = \sum_{k=0}^{N-1} \sum_{l=0}^{M-1} \alpha_{kl} W_{kl}$$

Per calcolare i coefficienti α_{kl} la `DCT2` utilizza la seguente equazione (ricavata dalla precedente):

$$\alpha_{kl} = \frac{1}{W_{sr} : W_{sr}} \sum_{i=0}^{N-1} b_i \cdot cos\left(s\pi \frac{2i+1}{2N}\right)$$

$$b_i = \sum_{j=0}^{M-1} a_{i,j} \cdot cos\left(r\pi \frac{2j+1}{2M}\right)$$

Analogamente alla `DCT2`, la `DCT` esegue il calcolo dei coefficienti a_k come segue

$$a_k = \frac{\mathbf{v} \cdot \mathbf{w}^k}{\mathbf{w}^k \cdot \mathbf{w}^k} = \frac{\sum_{i=1}^{N} cos\left(\pi k \frac{2i+1}{2N}\right) v_i}{\mathbf{w}^k \cdot \mathbf{w}^k}$$

ottenuto dalla seguente relazione tra un vettore $\mathbf{v} \in \mathbb{R}^N$ e un insieme di vettori $\mathbf{w}^k$ che formano una base per lo spazio vettoriale $\mathbb{R}^N$

$$\mathbf{v} = \sum_{k=0}^{N-1} a_k \mathbf{w}^k$$

È importante notare come il calcolo di $\mathbf{w}^k \cdot \mathbf{w}^k$ sia uguale a:

- $\sqrt{N}$, se $k = 0$
- $\sqrt{\frac{N}{2}}$, altrimenti

Vi è una normalizzazione equivalente a quella utilizzata dalla `DCT2` della libreria, utilizzando la radice quadrata.

2.1 Struttura ed Implementazione del Codice

2.1.1 Discrete Cosine Transform 2

Poichè la `DCT2` è possibile vederla come una doppia esecuzione della funzione `Discrete Cosine Transform` (o `DCT`), eseguita una volta sulle righe e una volta sulle colonne, abbiamo optato per implementare nel nostro codice la funzione `DCT` per poi eseguirla come riportato precedentemente.

```
1  def dct2_custom(matrix):
2      # Creo una matrice di appoggio
3      dct_matrix_row = np.zeros_like(matrix, dtype=float)
4      # Applico la DCT alle righe della matrice
5      for i in range(len(matrix)):
6          dct_matrix_row[i, :] = dct_custom(matrix[i, :])
7      # Applico la DCT alle colonne della matrice dct righe
8      dct_matrix_col = dct_matrix_row.T
9      for i in range(len(dct_matrix_col)):
10         dct_matrix_col[i, :] = dct_custom(dct_matrix_col[i, :])
11     # Rifaccio la traposta e ritorno la matrice
12     return dct_matrix_col.T
```

All'interno del codice si può vedere alle righe 5 e 6, abbiamo la chiamata alla funzione `DCT` sulle righe della matrice e successivamente alle righe da 8 a 10 la chiamata alla `DCT` sulle colonne della matrice ottenuta al passo precedente. Successivamente vediamo come viene implementata la funzione `DCT`.

2.1.2 Discrete Cosine Transform

```
1  def dct_custom(vector):
2      # Inizializzo le variabili
3      n = len(vector)
4      pi = np.pi
5      dct = np.zeros(n)
6      wk = np.zeros(n)
7      cos_base = []
8      # K = 0 a n-1
9      for k in range(n):
10         # Vettori della base dei coseni
11         for i in range(n):
12             # Calcolo le componenti wki come cos(pi * k * (2 * i + 1) / (2 * n))
13             wk[i] = np.cos(pi * k * (2 * i + 1) / (2 * n))
14         # Normalizzazione ortho per confronto con la dct della libreria
15         if k == 0:
16             # Calcolo i coefficenti ak = (v * wk) / (wk * wk)
17             ak = np.dot(vector, wk) / np.sqrt(n)   # wk * wk
18         else:
19             ak = np.dot(vector, wk) / np.sqrt(n/2) # wk * wk
20         # Salvo la base se debug
21         if show_basis:
22             cos_base.append(wk.copy())
23         # Aggiungo alla lista gli ak
24         dct[k] = ak
25     # Plotto i coseni per freq k
26     if show_basis:
```

```
27            plot_cosine_base(cos_base)
28        return dct
```

Nella fase di inizializzazione instanziamo le variabili per poi eseguire la vera `DCT` successivamente.
Abbiamo un ciclo for esterno in cui iteriamo per `k` al cui interno vengono eseguiti:

- Un ciclo for per aggiornare tutti gli elementi del vettore $\mathbf{w}^k$
- Calcolo di a_k
- Aggiornamento del vettore della `DCT` contenente tutti gli a_k calcolati ad ogni passo

2.1.3 Discrete Cosine Transform 2 di SciPy

Nella nostra libreria scelta, la funzione di `DCT` si compone come segue:

```
y = dct(x, type=2, n=None, axis=-1, norm=None, overwrite_x=False)
```

I parametri rappresentano:

- `x` » La matrice o il vettore di input (la matrice A definita precedentemente)
- `type = {1, 2, 3, 4}` » Il tipo di `DCT` da implementare. Esistono 8 tipi di `DCT`, ma SciPy implementa solo le prime 4.
- `n` » È la lunghezza del vettore in output. Se non specificato `n` è la lunghezza del vettore in input, altrimenti se `n < len(x)` l'array viene troncato oppure se `n > len(x)` l'array viene riempito di 0 per le posizioni mancanti.
- `axis` » È la direzione lungo la quale la `DCT` viene calcolata, di default il suo valore è -1 che si riferisce all'ultimo asse. In una matrice (vettore bidimensionale) abbiamo questa situazione:
  - `axis = 0` - Calcolo lungo le colonne
  - `axis = 1` - Calcolo lungo le righe
  - `axis = -1` - Calcolo lungo le righe
- `norm = {None, 'ortho'}` » Di default viene selezionato `None` come modalità di normalizzazione. SciPy nelle sue note spiega come cambiano i valori per entrambe le norme per ogni tipo di `DCT` implementata. Per congruenza col progetto riportiamo solo la differenza tra le norme per la `DCT` di tipo 1 e 2.
  - **DCT** - `norm = None, type = 1`
    Per questa coppia di valori in input il calcolo della funzione viene fatto secondo questa equazione:

$$y_k = x_0 + (-1)^k \cdot x_{N-1} + 2 \sum_{n=1}^{N-2} x_n \cdot cos\left(\frac{\pi k n}{N-1}\right)$$

– **DCT** - `norm = 'ortho', type = 1`
In questa situazione la formula rimane la stessa a patto di fattori moltiplicativi ai termini x_i:

$$y_k = \sqrt{2}x_0 + (-1)^k \cdot \sqrt{2}x_{N-1} + 2\sum_{n=1}^{N-2}\left[f x_n \cdot cos\left(\frac{\pi k n}{N-1}\right)\right]$$

con f definito come segue

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

– **DCT 2** - `norm = None, type = 2`
Per il calcolo della `DCT 2` non viene usato direttamente l'approccio di eseguire 2 volte la DCT prima su righe e poi su colonne, ma viene applicata la formula

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

– **DCT 2** - `norm = 'ortho', type = 2`
Anche in questo caso, per la normalizzazione ortogonale viene applicato un fattore moltiplicativo f:

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

con f definito come segue

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

- `overwrite_x = bool` » Se impostato a True il vettore passato in input può essere distrutto

2.2 Risultati Ottenuti e Considerazioni

```
# Funzione per analizzare i tempi di esecuzione e generare il grafico
def dct2_analyze_graph():
    matrices, N = generate_square_matrices(40, 5)
    # Calcolo i tempi di esecuzione per le due dct2
    dct2_custom_times = measure_dct2_times(matrices, dct2_custom, True)
    dct2_library_times = measure_dct2_times(matrices, dct2_library, False)
    # Scrivo su file i tempi
```

```
    if debug:
        write_times_to_file(dct2_custom_times, dct2_library_times)
    # Curve teoriche per il confronto
    N_cubed = N ** 3
    N_squared_logN = N ** 2 * np.log(N)
    # Normalizzare le curve teoriche per confrontarle con i tempi reali
    N_cubed_normalized = N_cubed / N_cubed[-1] * dct2_custom_times[-1]
    N_squared_logN_normalized = N_squared_logN / N_squared_logN[-1] *
    dct2_library_times[-1]
    # Creazione del grafico
    plt.figure(figsize=(10, 6))
    plt.semilogy(N, dct2_custom_times, 'o-', label='DCT2 Custom')
    plt.semilogy(N, dct2_library_times, 's-', label='DCT2 Libreria')
    # Curve teoriche (tratteggiate)
    plt.semilogy(N, N_cubed_normalized, 'k--', label='$N^3$')
    plt.semilogy(N, N_squared_logN_normalized, 'r--', label='$N^2 \log N$')
    # Labels
    plt.xlabel('Dimensione N')
    plt.ylabel('Tempo di esecuzione (s)')
    plt.title('Confronto Tempi di Esecuzione DCT2')
    plt.legend()
    plt.grid(True)
    plt.xticks(N)
    # Mostra il grafico
    plt.show()
```

Per calcolare i tempi di esecuzione abbiamo generato in modo casuale matrici da 40x40 a 640x640, raddoppiando ogni volta la dimensione partendo da 40x40 (paragrafo 2.3).
Per calcolare il tempo di esecuzione delle funzioni abbiamo creato una funzione di supporto `measure_dct2_times(matrices, function, custom=bool)` che calcola i tempi per ogni matrice generata e poi ne riportano i tempi (paragrafo 2.4). I tempi ottenuti per la DCT2 Custom e Library sono inoltre riportati nella seguente tabella:

Tempo Impiegato (secondi)		
Matrice	DCT2 Custom	DCT2 Library
40 x 40	0.06302900001173839	0.00021550001110881567
80 x 80	0.4911983000347391	6.669998401775956e-05
160 x 160	3.8765637999749742	0.00020820001373067498
320 x 320	31.38378020003438	0.0009302000398747623
640 x 640	251.01030020002509	0.0040401999722234905

2.2.1 Grafico in Scala Semilogaritmica

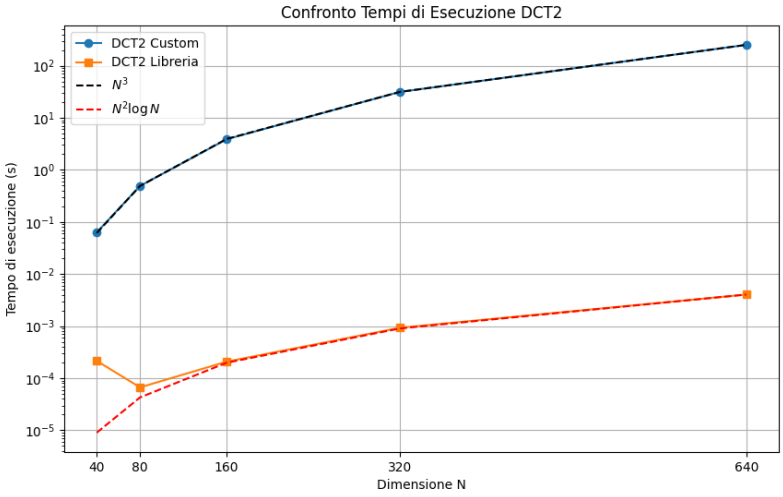

Figura 1: L'andamento dei tempi delle DCT2 rispetto alla dimensione N delle matrici

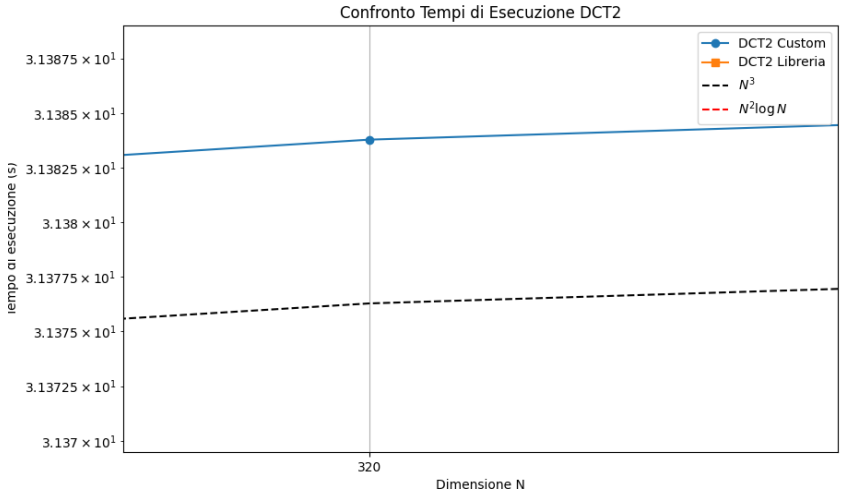

Figura 2: L'andamento della DCT2 Custom ingrandito rispetto a N al cubo

Dal grafico possiamo notare i tempi ottenuti dall'esecuzione di entrambi gli algoritmi con un grafo discreto per punti, mentre in linea tratteggiata si possono vedere le curve per le funzioni:

$$s = N^3$$

per la curva superiore e

$$s = N^2 log(N)$$

per quella inferiore.

È chiaramente visibile come la DCT2 custom segua perfettamente l'andamento della funzione N^3, mentre per la DCT2 della libreria notiamo come prima di $N = 160$ i tempi non sono proprio sovrapponibili a $N^2 log(N)$ e successivamente in linea con la funzione, come aspettato.

2.2.2 Calcolo dei Tempi

```python
def measure_dct2_times(matrices, function, custom=True):
    """
    Calcola i tempi di esecuzione della funzione DCT2 passata come parametro e restituisce i tempi di esecuzione
    per ogni matrice della lista di matrici
    :param matrices: Lista di Matrici quadrate
    :param function: Funzione di DCT2 (custom o lib)
    :param custom: True sse la funzione e' custom, False altrimenti (utile per il debug)
    :return times: Lista dei tempi di esecuzione per ogni matrice della funzione data
    """
    times = []
    for matrix in matrices:
        # Timeit per un'ottima precisione
        time_taken = timeit.timeit(lambda: function(matrix), number=1)
        times.append(time_taken)
    return times
```

2.3 Utilità: Generazione Matrici casuali

```python
# Genera matrici NxN casuali di dimensione start_size fino a num_doublings raddoppiando in volta
def generate_square_matrices(start_size, num_doublings):
    sizes = [start_size * 2 ** i for i in range(num_doublings)]
    matrices = [np.random.rand(size, size) for size in sizes]
    return matrices, np.array(sizes)
```

Per la generazione delle matrici in modo casuale ci siamo affidati alla generazione random fornita dalla libreria NumPy.
Partendo da una dimensione di partenza `start_size` vengono generate delle matrici quadrate, con dimensioni raddoppiate di volta in volta fino ad un massimo di `num_doublings` volte.
La funzione restituisce una lista di matrici (`matrices`) e una lista contenente le dimensioni delle matrici associate.

2.4 Utilità: Base dei Coseni

```python
# Per debugging stampa delle basi dei coseni per vedere se sono corrette
def plot_cosine_base(base):
    n = len(base[0])  # Lunghezza della base
    num_base = len(base)
    ncols = 3  # Numero di colonne nella griglia
    nrows = -(-num_base // ncols)  # Calcolo del numero di righe
    fig, axes = plt.subplots(nrows, ncols, figsize=(12, 8))
    for i, wk in enumerate(base):
        row = i // ncols
        col = i % ncols
        axes[row, col].bar(range(n), wk, label=f'k={i}')
        axes[row, col].set_xlabel('Indice i fino a n')
        axes[row, col].set_ylabel('Valore')
        axes[row, col].set_title(f'Basi dei Coseni per k={i} oscillazioni')
        axes[row, col].legend()
    plt.tight_layout()
    plt.show()
```

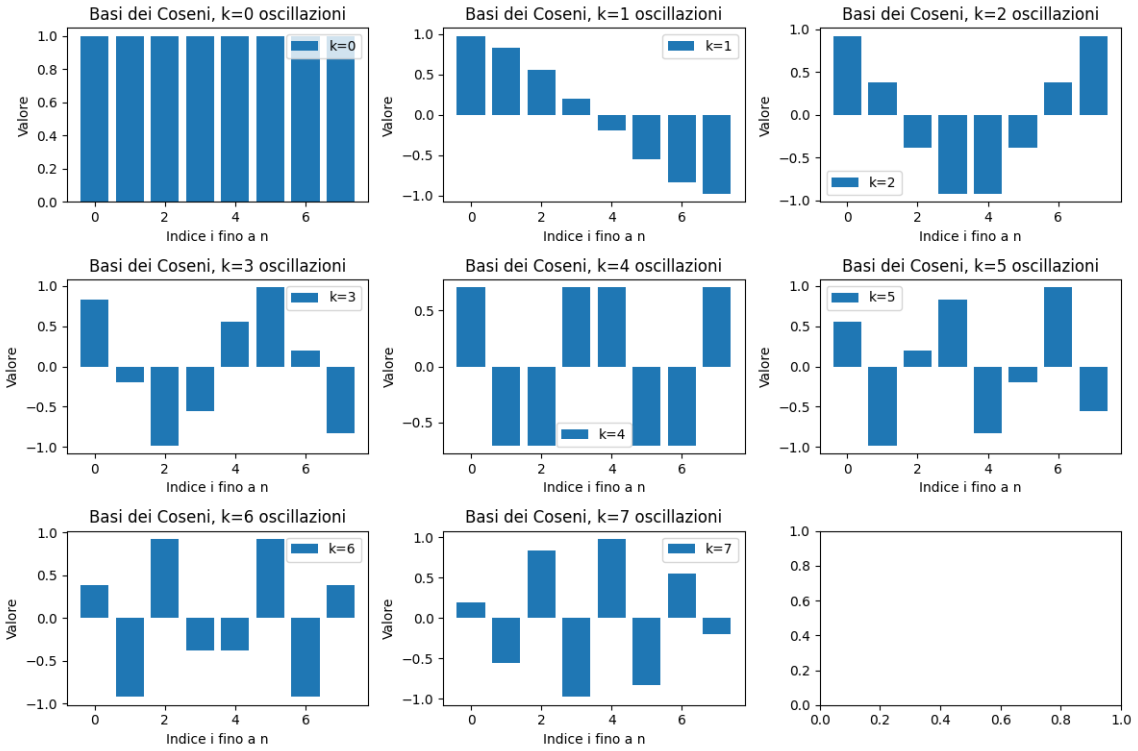

Figura 3: Viene mostrata la base dei coseni per la DCT

Ai fini di debugging è stato implementato un grafico che mostrasse la base dei coseni, in particolare ogni vettore w_k al variare della frequenza k, preso il vettore di test:

$$[231, 32, 233, 161, 24, 71, 140, 245]$$

3 Compressione Immagini - Parte 2

3.1 Introduzione all'Interfaccia Grafica

Per rispettare i requisiti del progetto, e quindi mostrare la compressione dell'immagine avvenuta, è stata implementata una vera e propria interfaccia grafica, utilizzando la libreria `tkinter` del linguaggio Python nel file `main.py`. Essa si presenta come una semplice interfaccia in finestra che permette di:

- Scegliere un'immagine in formato *bitmap*.
- Scegliere l'ampiezza delle finestrelle (blocchi) **F**.
- Scegliere la soglia di taglio delle frequenze **d**.
- Elaborare l'immagine una volta inserito i parametri corretti.
- Salvare, se si desidera, l'immagine elaborata.

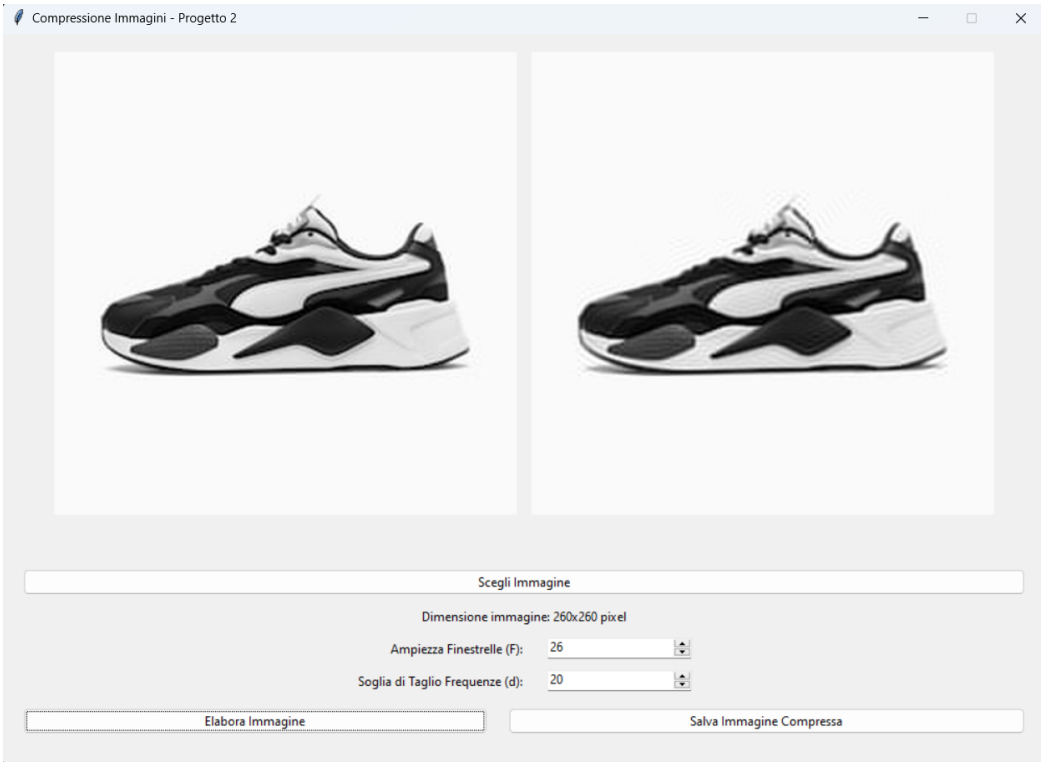

Figura 4: Vengono inoltre riportate le relative immagini e le dimensioni originali.

3.1.1 Gestione dei Parametri

L'interfaccia grafica, oltre a permettere la visualizzazione all'utente delle immagini, comprende una parte di gestione dei parametri (F e d) in modo che siano corretti, e una chiamata all'elaborazione dell'immagine. La funzione principale nel codice che si occupa di tale gestione è la seguente:

```
# Processa l'immagine con i parametri forniti
def process_image(F_entry, d_entry, img_label, proc_img_label):
    global processed_image
    try:
        F = int(F_entry.get())
        d = int(d_entry.get())
        # Valida i dati di input
        is_valid = validate_data(d, F)
        if not is_valid:
            messagebox.showerror("ERRORE", "d non e' compreso tra 0 e 2*F - 2")
            return
        # Crea un'immagine nera se d e' 0, altrimenti comprimi l'immagine
        if d == 0:
            width, height = selected_image.size
            # Utilizzo le dimensioni considerando il taglio dei blocchi
            processed_image = Image.new('L', (F * (width // F), F * (height // F)),
    (0))
        else:
            processed_image = compress_image(selected_image, F, d)
        # Ridimensiona l'immagine elaborata per la visualizzazione
        resize_processed_image = processed_image.resize(get_resized_dimensions(
    processed_image))
        processed_image_tk = ImageTk.PhotoImage(resize_processed_image)
        proc_img_label.config(image=processed_image_tk)
        proc_img_label.image = processed_image_tk
        # Abilita il pulsante per salvare l'immagine
        save_button.config(state=tk.NORMAL)
    except Exception as e:
        messagebox.showerror("ERRORE", f"Impossibile Elaborare l'immagine: {str(e)}")
```

In particolare la funzione svolge le seguenti operazioni:

- Ottiene i parametri **F** e **d**, convertiti in intero.
- Valida la correttezza dei parametri utilizzando la funzione `validate_data(d,F)`
- Controlla che **d** sia uguale a 0, ritornando in caso affermativo un'immagine nera, altrimenti viene fatta partire la compressione dell'immagine selezionata.
- Viene ridimensionata l'immagine elaborata in modo che sia visualizzata correttamente nella finestra
- Abilita il salvataggio dell'immagine elaborata.

3.2 La Compressione

Nel file `functions.py` sono state implementate tutte le funzioni necessarie al fine di permettere la compressione dell'immagine. Essa avviene seguendo un'approccio simil-JPEG, senza l'utilizzo di un vero e proprio quality factor q e la matrice di quantizzazione, avvicinandosi quindi di più ad uno stile di compressione globale. Vi è però comunque una divisione in blocchi F (non necessariamente 8x8 come in JPEG) e un metodo di taglio di frequenze (d) differente da quello globale per la quantizzazione. Analizzando nello specifico, le procedure per la compressione sono le seguenti:

- Utilizzo del parametro **F** per la suddivisione in blocchi **FxF** dell'immagine, partendo in alto a sinistra e scartando gli avanzi (questo si noterà attraverso dei "tagli" nell'immagine elaborata), utilizzando la funzione `get_blocks`.
- Ottenuti i blocchi, essi vengono processati in un ciclo di lunghezza `len(blocchi)`:
  - Viene applicata sul blocco f la **DCT2**, ottenendo c.
  - Vengono eliminate le frequenze c_{kl} con:

    $$k + l \geq d$$

    utilizzando la funzione `compress_block()`.
  - Viene applicata la **IDCT2** a c e arrotondando il risultato ff ad intero, con numeri da 0 a 255 (rgb) da un byte, utilizzando la funzione `idct2_block()`.
- Viene ricostruita l'immagine elaborata dai blocchi ff utilizzando la funzione `reconstruct_image()`.

In seguito il codice dell'implementazione complessiva della compressione:

```
# Effettuo tutte le operazioni per comprimere l'img
def compress_image(image, F, d):
    # Ottengo l'immagine in formato matriciale
    image_matrix = np.array(image)
    # Salvo le dimensioni
    image_height, image_width = image_matrix.shape
    # Valido se ho blocchi corretti
    if not validate_dim(F, image_height, image_width):
        raise Exception("Errore: L'immagine sara' tutta nera e quindi vuota per la dimensione dei blocchi scelta. Scegliere un'altro F")
    # Lista per i blocchi finali
    ff_blocks = []
    # Ottengo i blocchi FxF
    blocks = get_blocks(image_matrix, image_height, image_width, F)
    # Itero sui blocchi
    for f in blocks:
        # Trasformo in array
        f = np.array(f)
        # Applico la dct2
        c = dct2(f)
        # Elimino le frequenze
        c_compressed = compress_block(c, d, F)
```

```
        # Applico la dct2 e casting al formato rgb
        ff = idct_block(c_compressed)
        # Aggiungo il blocco compresso alla lista
        ff_blocks.append(ff)
    # Ricostruisco l'immagine con i blocchi elaborati
    compressed_image = reconstruct_image(ff_blocks, image_width, image_height, F)
    # Ritorno l'immagine compressa
    return compressed_image
```

3.2.1 Suddivisione dell'Immagine in Blocchi

La funzione `get_blocks()` ritorna una lista di blocchi *blocks* di dimensione *FxF*. La suddivisione in blocchi avviene dentro due cicli, di lunghezza pari al numero di blocchi effettivi verticali e orizzontali (arrotondati ad intero), dove vengono estratti i blocchi con `block = matrix[start_row:end_row, start_col:end_col]`

```
# Funzione per suddividere la matrice in blocchi F x F, scartando gli avanzi
def get_blocks(matrix, image_height, image_width, F):
    blocks = []
    # Calcola il numero di blocchi verticali e orizzontali
    num_blocks_v = image_height // F
    num_blocks_h = image_width // F
    # Intero sul numero di blocchi
    for i in range(num_blocks_v):
        for j in range(num_blocks_h):
            # Calcola gli indici di inizio e fine per le righe e le colonne del
            # blocco
            start_row = i * F
            end_row = (i + 1) * F
            start_col = j * F
            end_col = (j + 1) * F
            # Estrae il blocco F x F
            block = matrix[start_row:end_row, start_col:end_col]
            blocks.append(block)
    # Ritorno i blocchi
    return blocks
```

3.2.2 DCT2 e Taglio delle Frequenze

L'applicazione della **DCT2** avviene tramite l'utilizzo della funzione offerta dalla libreria *scipy*, descritta nella parte 1 del progetto in quanto analoga.

```
# DCT2 della libreria
def dct2(matrix):
    return dctn(matrix, norm='ortho')
```

e richiamata nel ciclo di `compress_image()` con `c = dct2(f)`. Ad ogni blocco *c* viene poi applicata la funzione per il taglio delle frequenze:

```
# Elimino le componenti di un blocco ckl con k+l >= d
# ovvero Quantizzazione
def compress_block(block, d, F):
    # Inizializzo il blocco "tagliato"
    compressed_block = np.zeros_like(block)
    # Itero su k e l a partire da 0 fino a lungh del blocco F
```

```
    for k in range(F):
        for l in range(F):
            # Salvo il risultato di k + l
            keep = k + l
            # Se keep e' minore di d, allora mantengo la frequenza
            if keep < d:
                compressed_block[k][l] = block[k][l]
    # Ritorno i blocchi "tagliati"
    return compressed_block
```

3.2.3 IDCT2 e Ricomposizione dell'Immagine

Ottenuti i blocchi con le frequenze eliminate viene applicata la DCT2 inversa, ovvero la **IDCT2**, tornando alla rappresentazione origiale. Successivamente ogni blocco viene arrotondato correttamente all'intero più vicino nel range (0, 255) di un byte, rappresentato dallo spazio RGB. Viene dunque eseguita la funzione `idct_block()`

```
# Applico idct2 per tornare all'img, arrotondando le componenti
# con valori corretti per lo spazio rgb a 1 byte
def idct_block(compressed_block):
    # Applico la IDCT2 della libreria
    ff = idct2(compressed_block)
    # Arrotondo il blocco ad intero
    ff_rounded = np.round(ff)
    # Mi assicuro che il blocco sia nel range (0,255)
    ff_valid = np.clip(ff_rounded, 0, 255)
    # Converto ad un byte
    ff_byte = ff_valid.astype(np.uint8)
    # Ritorno il blocco
    return ff_byte
```

I blocchi restituiti da `idct_block()` saranno i blocchi finali, che una volta ricomposti, formeranno l'immagine compressa. La funzione `reconstruct_image()` si occupa quindi di ricostruire l'immagine finale, collocando nell'ordine corretto i blocchi ottenuti dalle precedenti operazioni. La logica è molto simile alla funzione `get_blocks()`, ma ottenendo i valori dalla lista `blocks` nel seguente modo `blocks[i * num_blocks_h + j]`, dove:

- `i * num_blocks_h` calcola l'offset per la riga corrente.
- `+ j` aggiunge l'offset della colonna corrente all'offset di riga.

Inoltre sono stati eliminati dalla dimensione finale i blocchi neri esclusi durante la divisione in blocchi, al fine di ridurre ulteriormente le dimensioni su disco.

```
# Funzione per ricostruire l'immagine dai blocchi F x F
def reconstruct_image(blocks, image_width, image_height, F):
    # Calcola il numero di blocchi verticali e orizzontali
    num_blocks_v = image_height // F
    num_blocks_h = image_width // F
    # Calcolo le nuove dimensioni dell'immagine
    new_image_height = F * num_blocks_v
    new_image_width = F * num_blocks_h
    # Crea una nuova matrice per l'immagine
```

```
    new_image_matrix = np.zeros((new_image_height, new_image_width), dtype=np.uint8)
    # Itero sui blocchi
    for i in range(num_blocks_v):
        for j in range(num_blocks_h):
            # Calcola gli indici di inizio e fine per le righe e le colonne del
    blocco
            start_row = i * F       # Inizio riga blocco i
            end_row = (i + 1) * F   # Fine riga blocco i
            start_col = j * F       # Inizio colonna blocco j
            end_col = (j + 1) * F   # Fine colonna blocco j
            # Assegna il blocco F x F alla nuova matrice dell'immagine
            # Abbiamo i blocchi salvati in ordine una lista quindi per averli
    possiamo fare:
            # blocco 1 = riga 0 * blocchi in orizzontale + colonna 1 = 1 etc
            new_image_matrix[start_row:end_row, start_col:end_col] = blocks[i *
    num_blocks_h + j]
    return Image.fromarray(new_image_matrix)
```

Una volta quindi ricostruita l'immagine, essa verrà restituita alla procedura `process_image()`, per poi essere ridimensionata e scalata per una presentazione ottimale nell'interfaccia grafica. Il salvataggio dell'immagine invece farà ovviamente riferimento alla dimensione non scalata, quindi corretta.

3.3 Risultati Ottenuti

Per la fase di testing e verifica della correttezza del codice, sono stati effettuati degli esperimenti utilizzando le immagini fornite insieme alla traccia del progetto. Analizzando l'immagine *shoe.bmp* in casi normali, si può notare l'effettiva compressione dell'immagine.

3.3.1 Qualità Alta

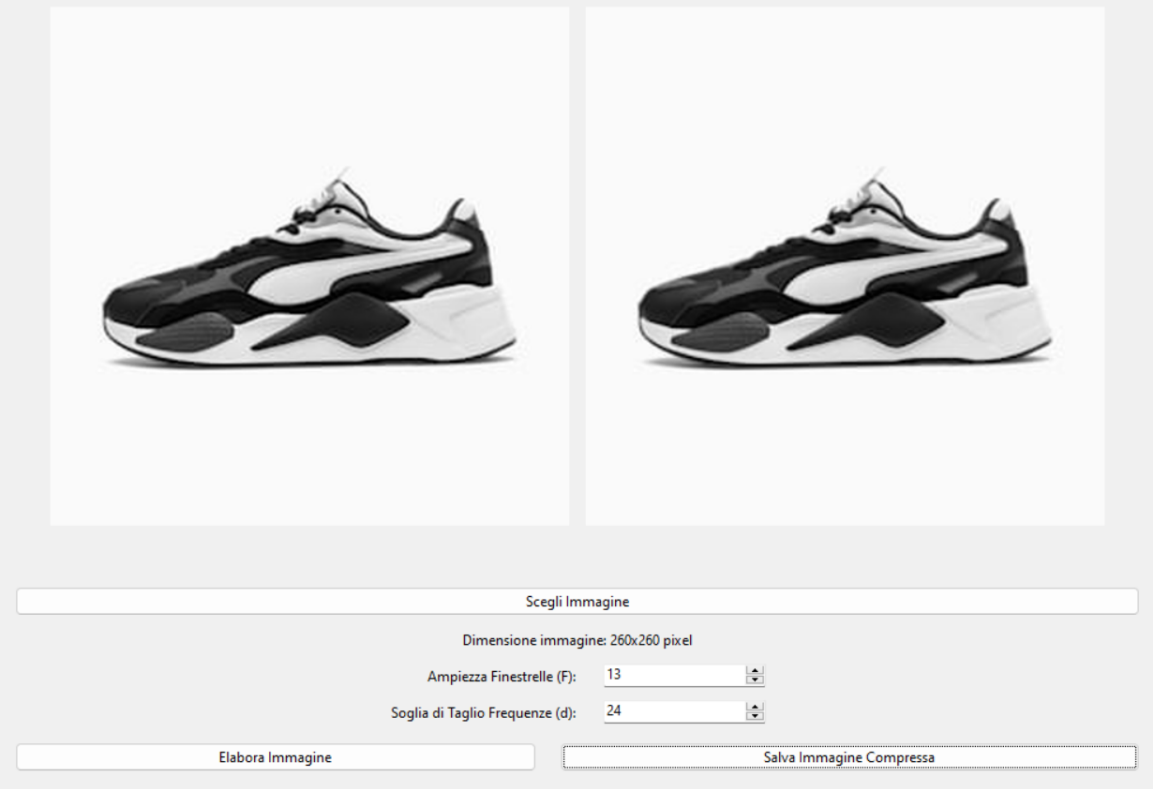

Figura 5: F = 13 e d = 24, taglio solo della frequenza più alta ottenendo la qualità più alta.

Con qualità massima si può notare come la differenza tra l'immagine originale e quella compressa sia quasi nulla, in quanto viene solo scartata la frequenza più alta. Pertanto risulta efficace a livello qualitativo, con una riduzione delle dimensioni da *198 KB* a circa *68 KB*, che rimarrà invariata per le altre qualità. Va notato che per il salvataggio dell'immagine è stato scelto, in modo arbitrario, il formato di partenza ovvero il **bitmap**, dato che non è ulteriormente compresso come possono esserlo gli altri formati *(jpeg, png etc)*.

3.3.2 Qualità Media

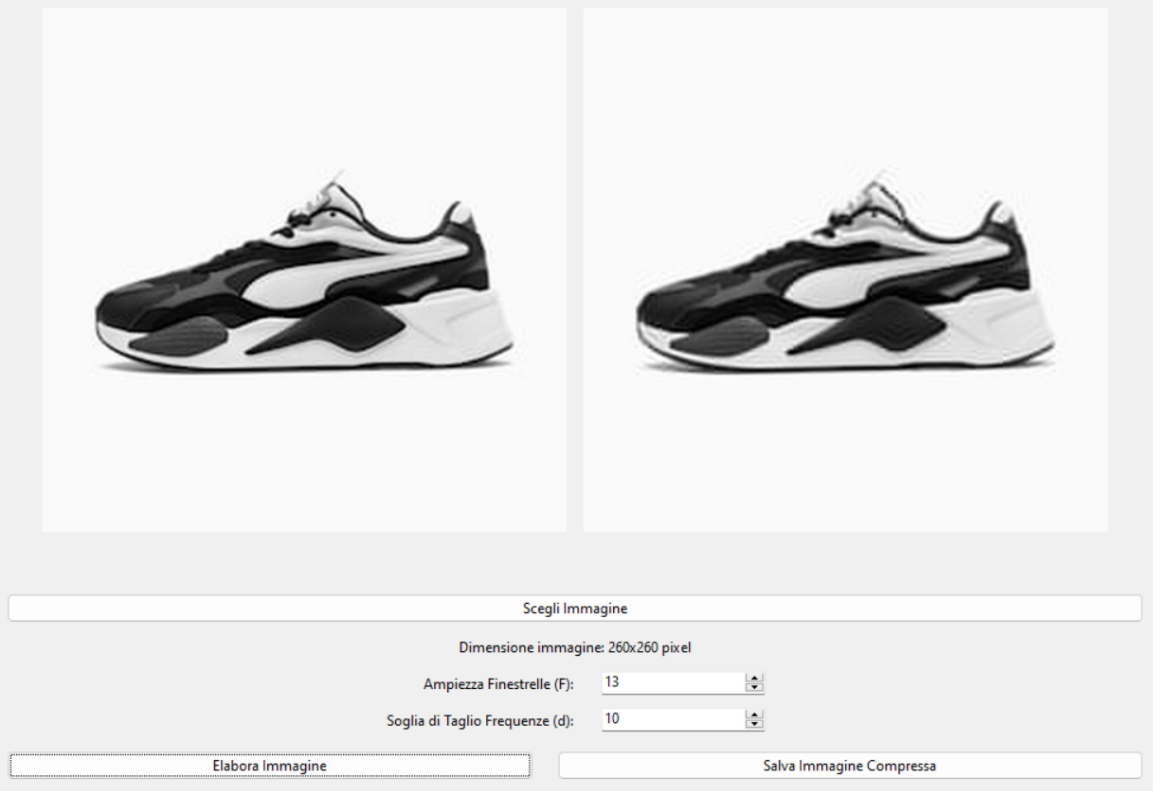

Figura 6: F = 13 e d = 12, qualità media.

Con qualità media si può notare già una differenza non trascurabile con l'immagine originale, soprattutto se ingrandita, evidenziando la leggera presenza del fenomeno di Gibbs (verrà approfondito nei Casi Particolari 8). Questo è dovuto al taglio di più frequenze per ogni blocco, ma ancora garantendo una certa qualità.

3.3.3 Qualità Bassa

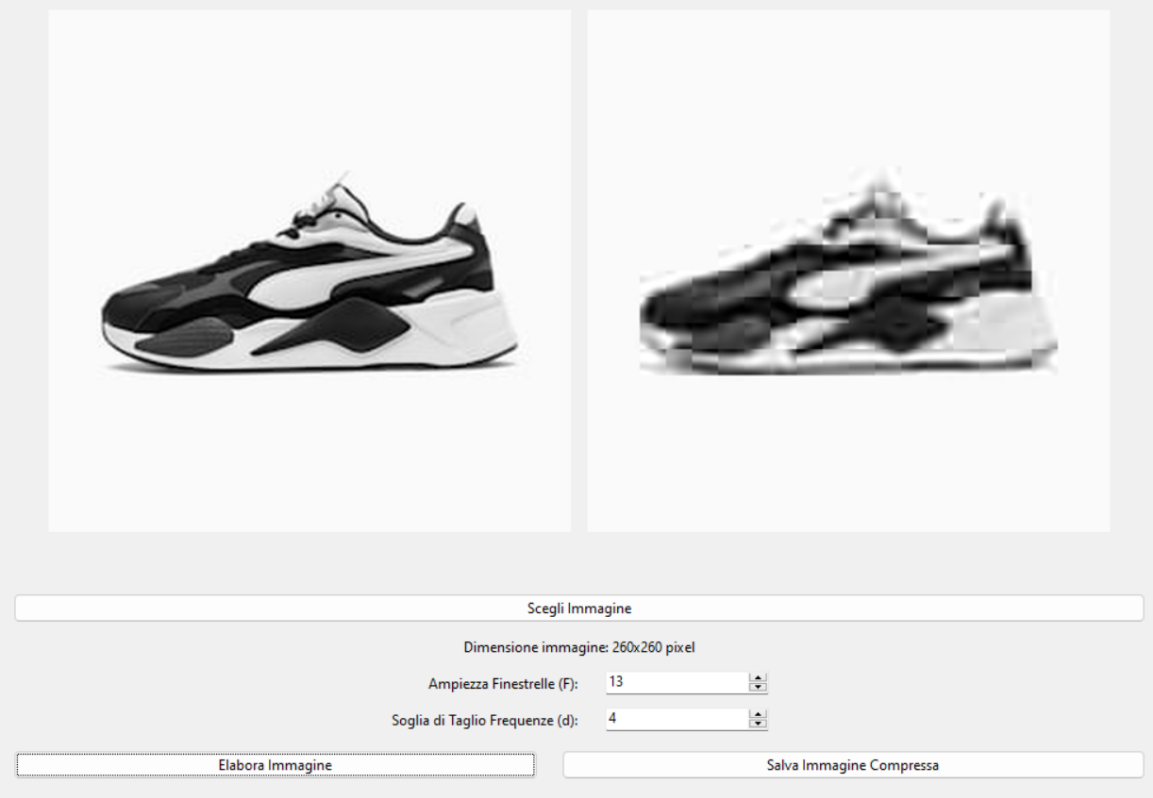

Figura 7: F = 13 e d = 4, la maggioranza delle frequenze tagliate, qualità molto bassa.

Con qualità bassa l'immagine elaborata risulta decisamente differente dall'originale, notando la presenza dei blocchi 13x13 e quindi confermando l'applicazione vera e propria dell'algoritmo di compressione. Le frequenze tagliate infatti risultano più della metà, portando ad un risultato scadente.

3.4 Casi Particolari

In questa sezione sono stati analizzati dei casi particolari, riscontrati durante il testing del progetto, riportando alcuni risultati che possono essere interessanti per confermare visivamente alcuni concetti teorici introdotti nel corso. In particolare si è voluta focalizzare l'attenzione su:

- Fenomeno di Gibbs
- Scarti Notevoli
- Soglia del Taglio Delle Frequenze a 0

3.4.1 Fenomeno Di Gibbs

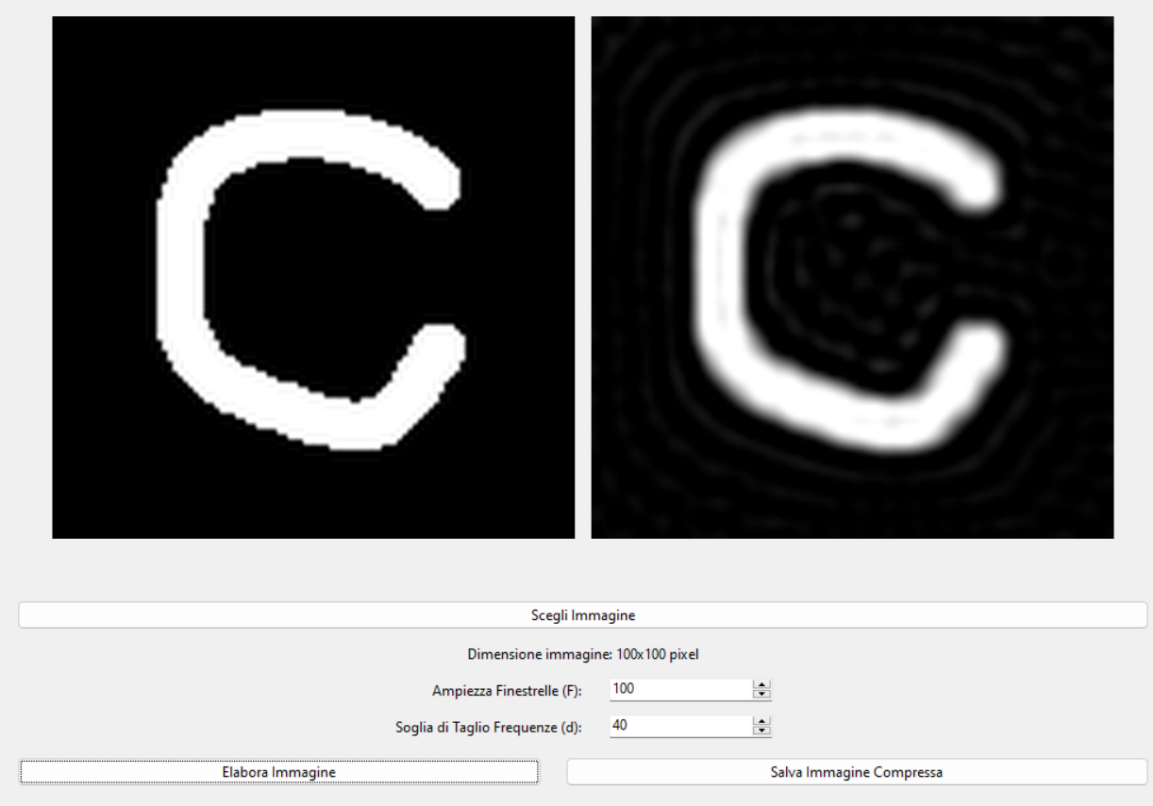

Figura 8: Il fenomeno di Gibbs, manifestato in modo evidente con le "onde" uscenti dalla C

Il fenomeno di Gibbs è un comportamento osservato sulle serie di Fourier e, nel contesto del progetto, sulla Discrete Cosine Transform quando si approssima una funzione discontinua con una somma di funzioni continue (nel caso di immagini quando abbiamo dei "salti" di contrasti, come lo sfondo bianco e il contorno della scarpa). Il fenomeno si manifesta come oscillazioni che non scompaiono mai completamente anche se si aumentano i termini della serie, mentre trattando le immagini, come un'aura contrastante con il resto del blocco. La scelta della dimensione dei blocchi permette di visualizzare il fenomeno di Gibbs in modo evidente se la dimensione F risulta molto elevata, oppure, per immagine mostrata, uguale alla dimensione dell'immagine. L'immagine scelta inoltre presenta un salto estremo (dal bianco al nero), mostrando infatti in modo chiaro e ancora più accentuato il fenomeno di Gibbs.

3.4.2 Scarti Notevoli

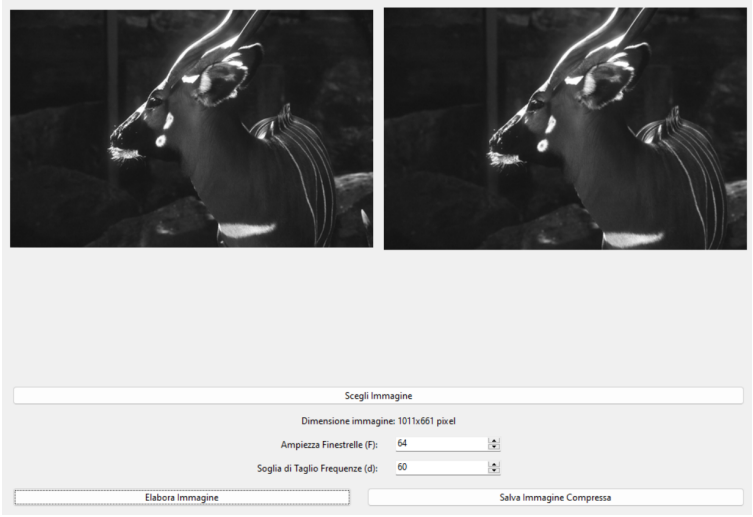

Figura 9: Con F = 64 l'immagine viene leggermente tagliata verso destra e in basso

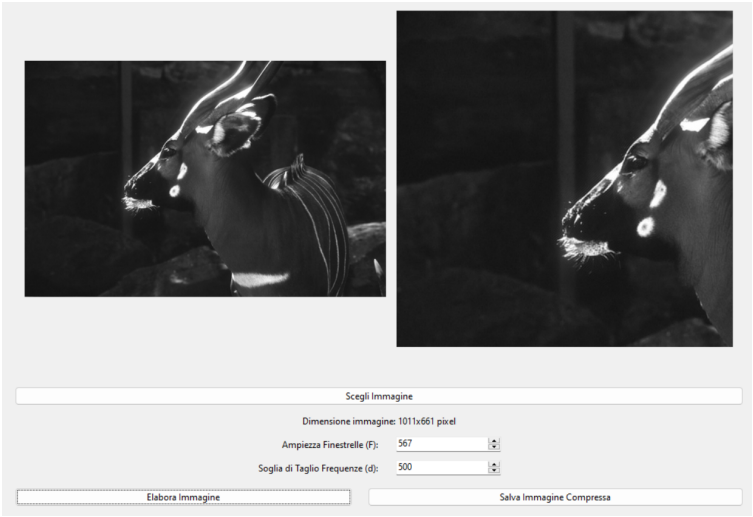

Figura 10: Con F = 567 l'immagine visualizzabile correttamente è meno della metà

Sono stati analizzati dei casi in cui l'immagine utilizzata non fosse divisibile per il parametro F, sostanzialmente andando ad indurre degli scarti dovuti alla divisione in blocchi. Nell'esempio riportato sopra possiamo notare come al variare di F l'immagine venga tagliata (e ridimensionata per una corretta visualizzazione in finestra) in relazione a quanti blocchi "rimangono" dopo la divisione intera per F. I blocchi neri in eccesso sono stati eliminati dalla dimensione finale dell'immagine elaborata permettendo una riduzione maggiore di spazio occupato ed eliminando il contorno nero.

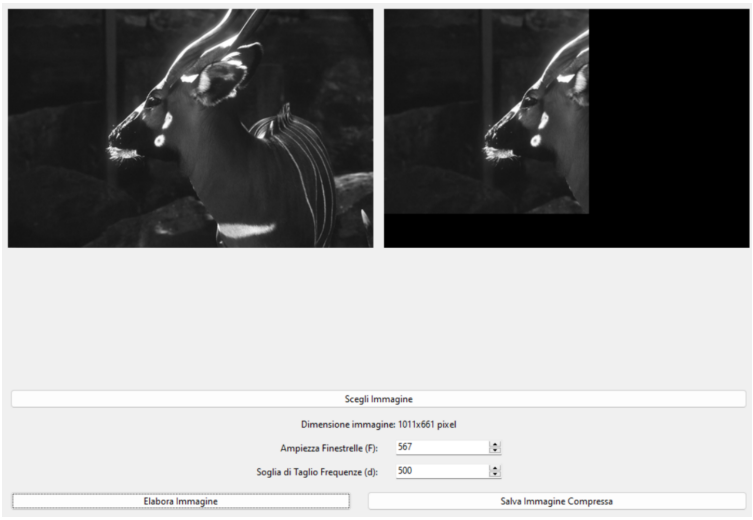

Figura 11: Il risultato ottenuto se lasciati nell'implementazione i blocchi neri

3.4.3 Soglia del Taglio Delle Frequenze a 0

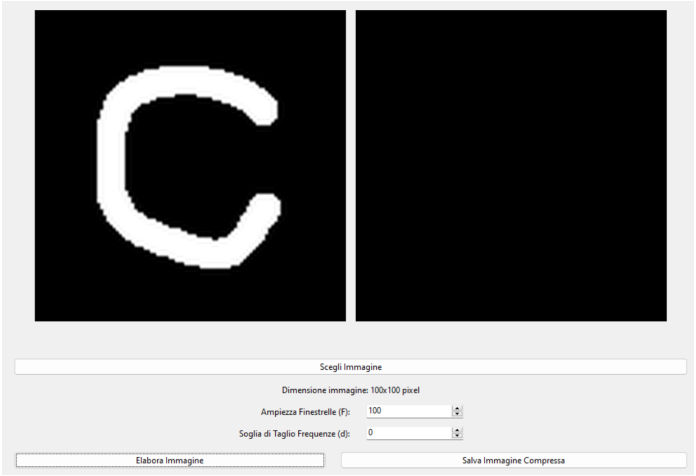

Figura 12: Utilizzando d = 0 l'immagine riportata è effettivamente tutta nera

Come conferma della corretta implementazione è stato testato il caso in cui la soglia del taglio di frequenze d sia uguale a zero. Ci si aspetta un'immagine completamente nera, in quanto tutte le frequenze sono state eliminate, o meglio un'immagine "vuota". Per preferenza di implementazione invece di riportare con un errore la presenza di un'immagine "vuota" è stato riportato un blocco totalmente nero, saltando l'elaborazione vera e propria dell'immagine.

3.4.4 Approfondimento sul Salvataggio e Dimensione

La dimensione dell'immagine ottenuta dall'elaborazione del programma dipenderà solamente dal modo in cui avviene il salvataggio in profondità 8 bit (scala di grigi) di un file bitmap. Esso si può calcolare come:

$$14 + 40 + 1024 + (width * 8 + 31)//32 * 4 * height$$

dove:

- 14 è l'Header del file bitmap
- 40 è l'Header dell'immagine (DIB Header o Bitmap Info Header)
- 1024 è la Color Palette (256 perchè vi sono 8 bit x 4 per il formato RGBA)
- 8 è il numero di bit per rappresentare una riga di pixel
- 31 è il padding
- 32 il numero di bit (4 byte) per ottenere il numero di blocchi da 4 byte necessari per rappresentare una riga, arrotondando verso l'alto con l'operatore $//$.
- 4 permette di convertire il numero di blocchi da 4 byte nella dimensione effettiva in byte di una riga di pixel.
- width e height sono le dimensioni dell'immagine.

Questo risultato è stato riscontrato andando ad analizzare le dimensioni dell'immagine originale e quella compressa, notando la presenza di nessuna relazione tra la scelta di d maggiori o minori sulla dimensione effettiva dell'immagine. Inoltre, questo metodo di salvataggio permette di risparmiare spazio nel caso in cui ci siano degli scarti, come trattato nell'immagine 9, in quanto la dimensione ottenuta sarà:

$$(F * (width//F), F * (height//F))$$

dove // è l'operatore di divisione intera.

3.4.4.1 Esempio Pratico: Considerando i risultati ottenuti da *shoe.bmp* mostrati in precedenza, ovvero una riduzione da *198 KB* a *68 KB* con dimensione 260x260 dell'immagine, si può calcolare la dimensione dell'immagine compressa come:

$$14 + 40 + 1024 + (260 * 8 + 31)//32 * 4 * 260$$

che equivale a *68678 Bytes*, cioè poco più di *68 KB*.

3.5 Considerazioni Finali

In conclusione, il progetto sviluppato ha permesso di confermare ulteriormente i concetti teorici presentati dal corso, producendo dei risultati che erano effettivamente attesi. La variazione dei parametri F e d è stata fondamentale al fine di analizzare le diverse casistiche e generalmente il risultato migliore è ottenuto con blocchi di piccole dimensioni e soglia delle frequenze massima. L'utilizzo di grandi dimensioni per quanto riguarda i blocchi (F) porta l'algoritmo (e quindi la DCT2) ad agire su quasi tutta l'immagine (come nel caso analizzato per il fenomeno di Gibbs 8)), mostrando in modo più evidente gli artefatti e la variazione di qualità portate dalla compressione, mentre una soglia stringente (d molto basso) porta ad una qualità pessima, perdendo dettagli e appiattendo la scala di grigi dell'immagine. Inoltre l'utilizzo improprio del parametro F può portare ad un grande troncamento dell'immagine, dovuto al fatto che le parti escluse dai blocchi vengono scartate. Questo fa apprezzare la soluzione di copiare i pixel ai bordi proposta dal JPEG, oltre che ad una scelta molto piccola dei blocchi F. L'applicazione si è comunque dimostrata abbastanza robusta in termini di tempistiche, producendo risultati in tempi brevi anche per immagini di grandissime dimensioni grazie alla divisione in blocchi (altrimenti inaccettabile), considerando che i tempi di calcolo della DCT2 e IDCT2 della libreria si avvicinano $N^2 log(N)$ come analizzato nella parte 1.