Università degli Studi di Milano-Bicocca
Corso di Laurea Magistrale in Informatica
2023-2024

Metodi del Calcolo Scientifico
Progetto 2

Compressione JPEG tramite Discrete Cosine Transform

Varisco Alberto 866109
Quaggio Stefano 866504

Indice

1 Prima Parte **3**
- 1.1 Discrete Cosine Transform (DCT) . . . 3
- 1.2 Obiettivo . . . 3
- 1.3 Codice sorgente . . . 4
  - 1.3.1 Informazioni preliminari . . . 4
  - 1.3.2 Funzione DCT2 home made . . . 4
  - 1.3.3 Funzione di misurazione dei tempi . . . 6
  - 1.3.4 Esecuzione del test . . . 6
- 1.4 Risultati . . . 6

2 Seconda Parte **8**
- 2.1 Compressione JPEG . . . 8
- 2.2 Codice sorgente . . . 8
  - 2.2.1 Informazioni preliminari . . . 8
  - 2.2.2 Upload dell'immagine . . . 8
  - 2.2.3 Esecuzione della compressione . . . 10
- 2.3 Risultati . . . 13
  - 2.3.1 F = dimensione dell'immagine . . . 13
  - 2.3.2 F = 8 . . . 15

3 Conclusioni **16**

1 Prima Parte

1.1 Discrete Cosine Transform (DCT)

La Discrete Cosine Transform (DCT) è una trasformazione matematica utilizzata per decomporre un segnale in una serie di funzioni coseno armoniche. La sua grande utilità, viene applicata in informatica per ridurre la quantità di informazioni salvate, in particolare la troviamo nell'ambito della compressione delle immagini. Grazie alla DCT, infatti, è possibile memorizzare solamente i coefficienti chiave di informazione relativi all'immagine in ingresso, riducendo quindi la quantità di dati utilizzati. La DCT è collegata alle trasformate di Fourier, sulle quali essa si appoggia: in particolare ci troviamo nel caso discreto, dove il valore della funzione è conosciuta solamente in alcuni punti e, nei rimanenti, avviene un'approssimazione a "scalini". Matematicamente, quindi, la DCT ci permette di costruire i coefficienti a_k dato un vettore R^N in ingresso, utilizzando la seguente formula:

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

Le sue componenti riguardano:

- k è l'indice del coefficiente DCT che stiamo calcolando
- N è il numero totale di elementi nel segnale v_i
- $\cos\left(\pi k \frac{2i+1}{2N}\right)$ sono i pesi di ciascun v_i, in base alla loro posizione e alla frequenza k.
- $w^k \cdot w^k$ fattore di normalizzazione

Da notare in particolare come il termine relativo al coseno permetta di decomporre il segnale, trasformandolo da un dominio spaziale ad un dominio delle frequenze, permettendone una manipolazione e analisi più efficiente.

1.2 Obiettivo

L'obbiettivo della prima parte del progetto 2 è quella di realizzare una versione personale della DCT, che sfrutti quando detto della sezione precedente ed applicandolo in versione bidimensionale, quindi a matrici di dati. Sapendo che lo spazio di matrici è uno spazio vettoriale, troviamo anche qui il concetto di elementi linearmente indipendenti, generatori e basi. La DCT monodimensionale, in quanto anche trasformata ortogonale, viene sfruttata iterando su righe e colonne della matrice creando così la DCT2 bidimensionale (proprio quello che ci serviva!)

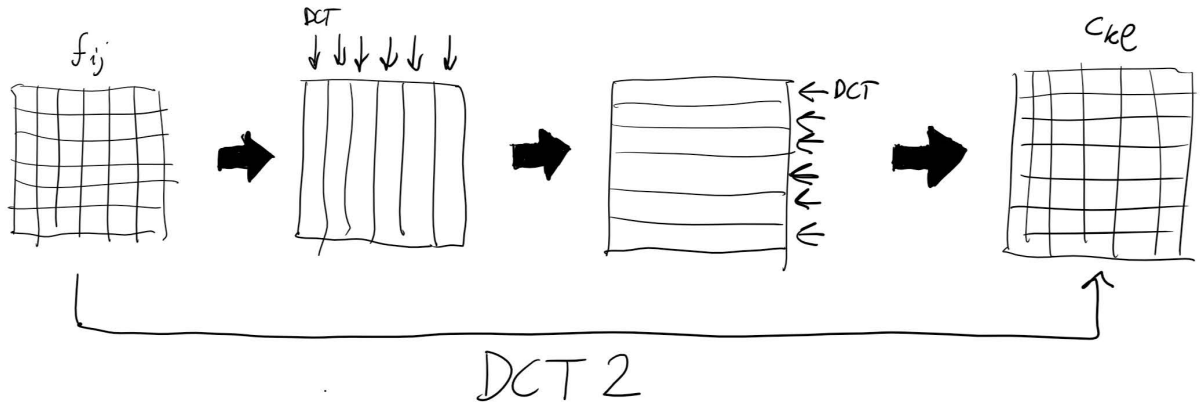

Figura 1: Applico la DCT prima sulle colonne e poi sulle righe, ottenendo la DCT2

1.3 Codice sorgente

1.3.1 Informazioni preliminari

Il codice sorgente per questo progetto è disponibile al seguente link: Link GitHub
Caratteristiche del codice:

- Linguaggio di programmazione utilizzato: **Python**
- Per un supporto ad una scrittura più chiara e documentata sono stati utilizzati i **Notebook (.ipynb)**
- Libreria open-source utilizzata per fare benchmark e compararla con la nostra DCT: **Scipy**
- Sub-package della libreria Scipy, specifico per la presenza della Fast Fourier Transform: **fft**

Al posto della classica DCT (che si trova sempre all'interno della libreria *Scipy*), viene utilizzata una sua più specifica variante, ovvero la versione Fast Fourier Trasform, la quale riduce i tempi di calcolo della DCT fino a $N^2 \log(N)$.

1.3.2 Funzione DCT2 home made

Nella seguente sezione entriamo più nel dettaglio del codice, descrivendo la funzione principale relativa al calcolo della DCT2 seguendo la formula e le nozioni fornite durante le lezioni.

```
def dct2_home_made(input_matrix):
    N = input_matrix.shape[0]
    j = np.arange(N)
    i = j[:, None]

    cos = np.cos(np.pi * i  * (2 * j + 1) / (2 * N))

    input_matrix_transformed = np.zeros_like(input_matrix)

    for m in range(N):
        input_matrix_transformed[m] = np.sum(cos[m] * input_matrix, axis=1)

    for n in range(N):
        input_matrix_transformed[:, n] = np.sum(input_matrix_transformed * cos[n], axis=0)

    Wsr = np.sqrt(2/N) * np.ones(N)
```

```
    Wsr[0] = np.sqrt(1/N)

    Wsr_array = Wsr[:, None] * Wsr

    output_matrix = input_matrix_transformed * Wsr_array

    return output_matrix
```

Listing 1: Funzione completa DCT2 home made

Nel dettaglio, gli step sono stati:

1. Come prima cosa, si presuppone l'utilizzo di matrici quadrate, per questo viene calcolato N basandosi solamente su una singola dimensione dell'array in input. Viene creato un vettore riga j e colonna i, che contengono i valori da 0 a $N-1$: ci verranno utili successivamente.

```
    N = input_matrix.shape[0]
    j = np.arange(N)
    i = j[:, None]
```

2. Pre calcolo della matrice dei coseni, tramite la formula $\cos\left(\pi i \frac{2j+1}{2N}\right)$, dove i e j variano da 0 a $N-1$. Sfruttiamo la potenza dell'operazione broadcasting, calcolando istantaneamente tutti i valori al variare di i e j. Il broadcasting ci ha permesso di ottimizzare di addirittura più di 20 volte il tempo di calcolo del coseno per ogni elemento, rispetto ai 2 classici cicli *for* innestati. Alla fine di questa procedura, otterremo la matrice di trasformazione dei coseni di grandezza N x N. Questa matrice sarà molto più facile da gestire quando si andrà ad applicare ad ogni elemento del nostro array di valori da modificare.

```
    cos = np.cos(np.pi * i  * (2 * j + 1) / (2 * N))
```

3. Primo ciclo *for*: per ogni riga m da 0 a $N-1$, si calcola il prodotto elemento per elemento della riga m della matrice dei coseni con la matrice in input dei valori, quindi si sommano i risultati lungo l'asse delle colonne.
   Secondo ciclo *for*: per ogni riga n da 0 a $N-1$, si calcola il prodotto elemento per elemento della colonna m della matrice dei coseni con la matrice calcolata al ciclo for precedente, quindi si sommano i risultati lungo le righe. Così facendo otteniamo una nuova matrice risultante dalla trasformazione di righe e colonne applicando la formula della DCT monodimensionale.

```
    input_matrix_transformed = np.zeros_like(input_matrix)

    for m in range(N):
        input_matrix_transformed[m] = np.sum(cos[m] * input_matrix, axis=1)

    for n in range(N):
        input_matrix_transformed[:, n] = np.sum(input_matrix_transformed * cos[n],
    axis=0)
```

4. L'ultimo passaggio della DCT, è quello di andare a normalizzare tutti i nostri valori ottenuti in precedenza. Sfruttando una piccola furbizia, invece che calcolare il prodotto interno $W_{sr} : W_{sr}$, creiamo una matrice N x N di valori semplicemente controllando gli indici s ed r, in particolare:

   - La prima riga e colonna di valori che compone la matrice è $\frac{1}{\sqrt{N}}$

- I restanti valori sono $\sqrt{\frac{2}{N}}$

Infine moltiplichiamo la nostra matrice dei valori modificati con la matrice degli *alpha*, ottenendo la matrice trasformata tramite DCT2.

```
    Wsr = np.sqrt(2/N) * np.ones(N)
    Wsr[0] = np.sqrt(1/N)

    Wsr_array = Wsr[:, None] * Wsr

    output_matrix = input_matrix_transformed * Wsr_array
```

1.3.3 Funzione di misurazione dei tempi

Il seguente blocco di codice è relativo alla funzione utilizzata per calcolare le tempistiche della funzione passata come parametro

```
def measure_time(func, x):
    start_time = time.perf_counter()
    func(x)
    end_time = time.perf_counter() - start_time
    return end_time
```

Semplicemente viene avviato un timer, si esegue la funzione e, una volta terminata, si calcola il tempo trascorso facendo la differenza tra il timestamp di avvio e quello corrente

1.3.4 Esecuzione del test

Per effettuare la fase di test, sono state utilizzate 7 matrici, le quali raddoppiavano dimensione di volta in volta. Per prima cosa si è quindi definito un vettore Numpy, contenente la dimensione N di ogni matrice:

```
sizes = np.array([64, 128, 256, 512, 1024, 2048, 4096], dtype=np.int64)
```

Si itera, successivamente, sul vettore appena creato, generando all'interno di una variabile x, una matrice **size x size** i cui valori hanno minimo 0 e massimo 255 (come il valore di un singolo pixel all'interno delle immagini). Questa matrice viene utilizzata come input per la nostra dct2 home made e la dct2 (in versione FFT) della libreria *Scipy*. Inizialmente si era utilizzata la versione classica della DCT, applicandola 2 volte, ma questa non aveva performance buone da poter essere considerata paragonabile alla versione Fast. Una volta terminata l'esecuzione, poi, si salvano i valori all'interno di un vettore, per poi utilizzarli in seguito per creare il grafico

```
times_fft2 = []
times_dct2_home_made = []

for size in sizes:
    x = np.random.uniform(low=0.0, high=255.0, size=(size, size))

    time_dct2_home_made = measure_time(dct2_home_made, x)
    times_dct2_home_made.append(time_dct2_home_made)

    time_fft2 = measure_time(fft2_scipy, x)
    times_fft2.append(time_fft2)
```

1.4 Risultati

In questa sezione analizziamo i risultati ottenuti per entrambi gli algoritmi. In particolare analizziamo anche le premesse, ovvero che la Fast Fourier Transform segua circa un andamento di $O(N^2 \cdot \log N)$ mentre la DCT2 home made $O(N^3)$. Dopo aver avviato i vari test, i

valori ottenuti sono stati mostrati graficamente attraverso un grafico che possiede le seguenti caratteristiche:

- Asse x che individua la dimensione N della matrice in esame
- Asse y che individua il tempo di esecuzione, in secondi, della funzione
- Entrambi gli assi vengono adattati con un fattore di scala logaritmica
- Funzioni $y = N^3$ e $y = N^2 \log N$ utilizzate come riferimento

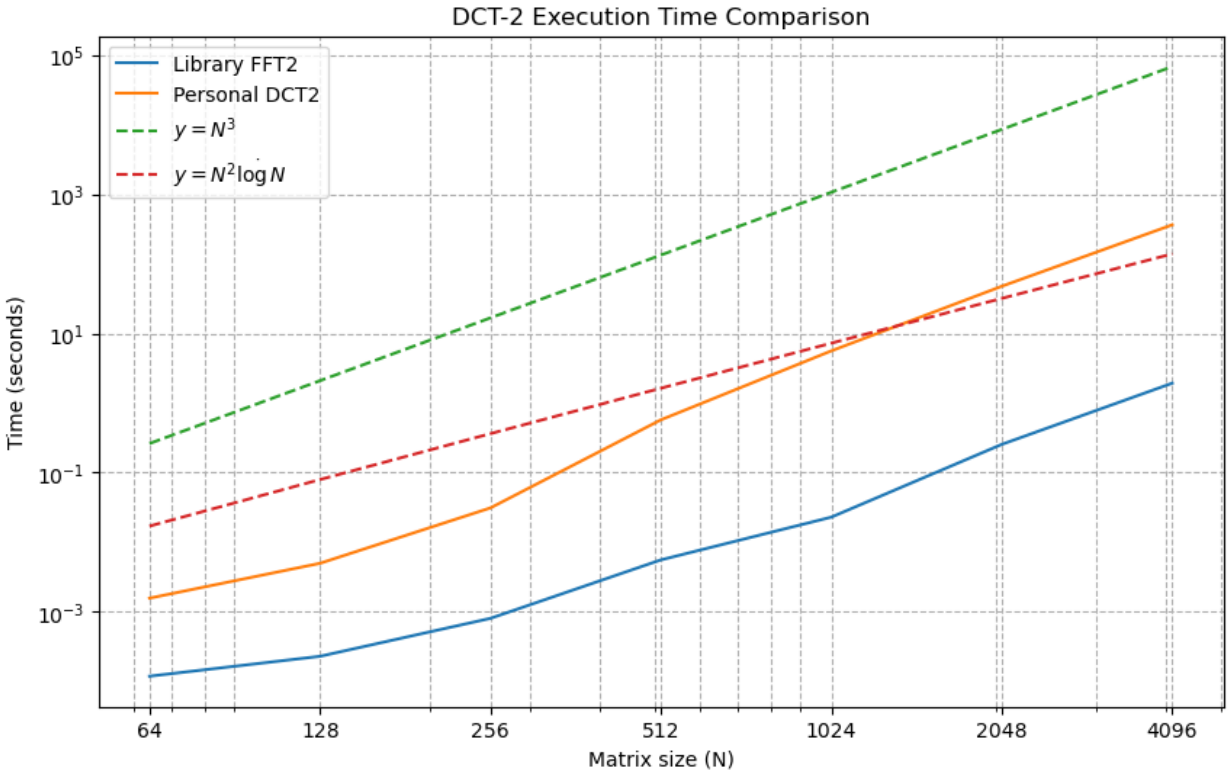

Figura 2: Grafico dei risultati

I risultati mostrano come effettivamente la Fast Fourier Transform della libreria tenda ad assomigliare (e in un certo senso anche overperformare) a $N^2 \log N$, probabilmente dettato dal fatto di possedere ottimizzazioni interne alla libreria per poter sfruttare al meglio operazioni native tra matrici e calcoli complessi. D'altro canto, la nostra versione home made non sfigura affatto e rimane al di sotto del N^3 teorico mantenendo un andamento simile a $N^2 \log N$ quando rimaniamo al di sotto di circa $N = 1500$, grazie soprattutto al precalcolo dei coseni e all'utilizzo di funzioni ottimizzate dalla libreria Numpy, che ci ha permesso di evitare inutili e costosi cicli for innestati. Oltre a questo, anche l'utilizzo della tecnica di broadcasting di Python, riusciamo a processare più dimensioni per un array, considerando solamente un oggetto vettoriale, mentre le operazioni vengono gestite interamente dal linguaggio di programmazione.

2 Seconda Parte

2.1 Compressione JPEG

Utilizzando quanto visto per la prima parte del progetto, è possibile creare algoritmi utili in diversi campi, soprattutto quando si va a trattare la compressione di immagini. Il concetto di DCT viene infatti sfruttato all'interno dell'algoritmo JPEG, essendo l'immagine una matrice di pixel (valori interi). In breve, gli step che vengono seguiti sono i seguenti:

- Ottenere dall'immagine in input una matrice di valori interi
- Applicare la trasformazione DCT2
- Troncare le alte frequenze, ovvero le oscillazioni maggiori nelle trasformate di Fourier
- Applicare la IDCT2 per riottenere l'immagine

Per sopperire poi al fenomeno di Gibbs (o comunque limitarlo), avere un algoritmo veloce e qualità dell'immagine buona, utilizza alcune accortezze. In particolare per velocizzare le operazioni e per ridurre il fenomeno di Gibbs localmente, suddivide l'immagine in blocchetti 8 x 8, mentre per avere una maggiore qualità utilizza la matrice di quantizzazione, che ci permette di quantificare il livello di compressione a scapito della qualità dell'immagine. Nel caso del nostro progetto, non si utilizza la matrice di quantizzazione ma un singolo parametro che indica la qualità della nostra immagine, intesa come soglia di taglio delle frequenze.

2.2 Codice sorgente

2.2.1 Informazioni preliminari

Caratteristiche del codice:

- Librerie di supporto utilizzate: **Pillow** (fork di Python Image Library), **TKinter** per la visualizzazione di finestre interattive.
- La libreria e la funzione utilizzata per eseguire la Fast Fourier Transform sono le stesse della prima parte.

Nelle sezioni seguenti vengono spiegate nel dettaglio le funzioni implementate

2.2.2 Upload dell'immagine

L'intera parte grafica del nostro programma si basa, come già anticipato, sull'utilizzo della libreria TKinter che ci permette di utilizzare vari tool e funzioni per creare componenti come finestre, text input, pulsanti e labels. Grazie a questa visualizzazione grafica, risulta più facile per l'utente poter utilizzare il programma senza alcun problema.

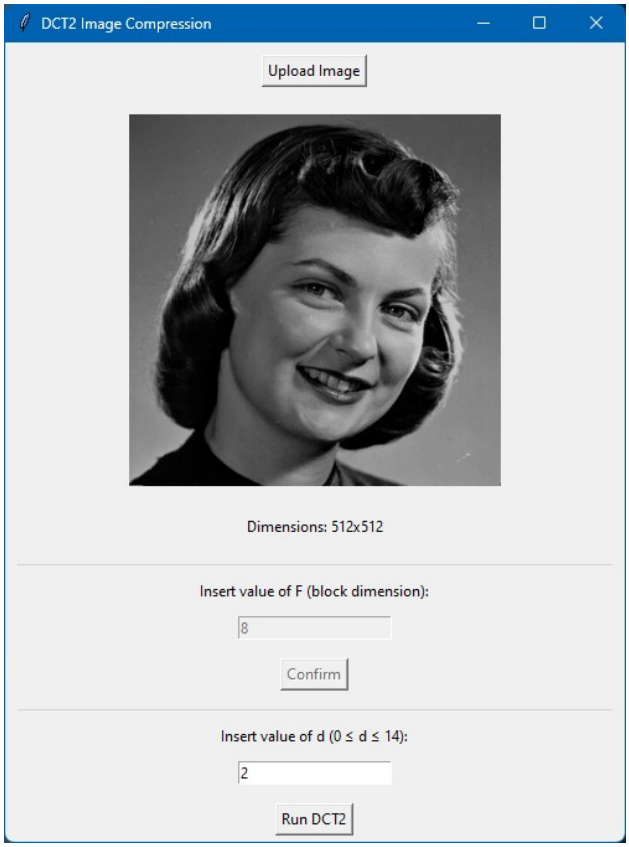

Figura 3: Widget grafico di inserimento

Come mostrato in figura, gli elementi principali della finestra risultano essere:

- Pulsante "Upload Image" per poter sfogliare le cartelle e caricare l'immagine BMP che vogliamo. Attenzione! Il programma limita l'inserimento di file ai soli .bmp, altre estensioni non saranno accettate
- Visualizzazione di una thumbnail dell'immagine scelta
- Una piccola specifica tecnica sulla dimensione dell'immagine, per agevolare la scelta, in seguito, del valore di F
- Text input per l'inserimento di **F**, ovvero l'ampiezza delle finestrelle (macro-blocchi) in cui si eseguirà la DCT2
- Text input per l'inserimento di **d**, valore compreso tra 0 e 2F-2 che sarà la soglia di taglio delle frequenze. Il valore indicato nella sua label viene mostrato dinamicamente in base al valore di F inserito in precedenza.
- Pulsante per l'avvio di tutte le operazioni relative alla DCT2.

Di seguito è riportato il codice relativo all'upload dell'immagine all'interno del nostro programma, mentre le 2 funzioni per creare il widget non vengono riportato all'interno di questa relazione perchè non inerenti all'obbiettivo del progetto (sono comunque nel codice GitHub):

```
def upload_image():
    global img, img_array, original_img
    file_path = filedialog.askopenfilename(filetypes=[("BMP files", "*.bmp")])
    if file_path:
        img = Image.open(file_path)
```

```
        original_img = img
        img.convert('L')
        img_array = np.array(img)

        if img_array.ndim > 2:
            img_array = img_array[:, :, 0]

        img.thumbnail((300, 300))
        img_tk = ImageTk.PhotoImage(img)
        img_label.config(image=img_tk)
        img_label.image = img_tk

    label_dimensions.config(text=f"Dimensions: {img_array.shape[1]}x{img_array.shape[0]}")
    label_dimensions.pack(pady=10)
    div1.pack(padx=10, pady=10, fill="both", expand=True)
    label_F.pack()
    entry_F.pack(pady=10)
    lock_button_F.pack(pady=5)
```

Da sottolineare più nello specifico, all'interno di questa funzione avviene anche il preprocessing dell'immagine per facilitare le operazioni successive. In particolare si effettua una conversione della nostra immagine BMP in scala di grigi, evitando eventuali problemi riguardo alla molteplicità di livelli che si avrebbe nel gestire un'immagine a colori: avendo un'immagine RGB, infatti, si dovrebbe separare ogni livello di colore (rosso, verde, blu) e trattarlo come matrice a se, per poi ricomporre il tutto. In questo progetto si tratterà quindi solamente la compressione di immagini in bianco-nero. Una volta caricata l'immagine e convertita, viene creata la matrice relativa al valore intero di ogni pixel. In seguito si sblocca il campo per inserire il valore di F, dando la possibilità all'utente di scegliere la dimensione dei blocchetti, con i quali si andrà a "tagliare" l'array.

2.2.3 Esecuzione della compressione

In questa sezione viene analizzata nel dettaglio la funzione principale relativa alla compressione dell'immagine, simulando quanto avviene per il formato JPEG. Siccome è una parte corposa, riportiamo prima il codice per intero e poi lo analizziamo pezzo per pezzo.

```
def apply_dct_operations():

    try:
        F = int(entry_F.get())
        d = int(entry_d.get())
        if d < 0 or d > (2 * F - 2):
            raise ValueError
    except ValueError:
        messagebox.showerror("Warning", f"d must be an integer between 0 and {2*F - 2}")
        return

    height, width,  = img_array.shape

    # Calculate the maximum height and width that are divisible by F
    height = (height // F) * F
    width = (width // F) * F

    # Crop the image to the calculated dimensions
    image_to_compress = img_array[:height, :width]

    block_height = height // F
    block_width = width // F

    img_dct_array = np.zeros((height, width))

    for i in range(block_height):
        for j in range(block_width):
            block_start_i = i * F
            block_end_i = block_start_i + F
            block_start_j = j * F
            block_end_j = block_start_j + F
            block = image_to_compress[block_start_i:block_end_i, block_start_j:block_end_j]
```

```
33
34              c = dctn(block, norm='ortho')
35
36              for k in range(F):
37                  for l in range(F):
38                      if k + l >= d:
39                          c[k, l] = 0
40
41              img_dct_array[block_start_i:block_end_i, block_start_j:block_end_j] = c
42
43              block_idct = idctn(c, norm='ortho')
44
45              # Round and clip block_idct
46              block_idct = np.round(block_idct).astype(int)
47              block_idct = np.clip(block_idct, 0, 255)
48
49              # Update img_dct_array
50              img_dct_array[block_start_i:block_end_i, block_start_j:block_end_j] =
    block_idct
51
52      modified_img = Image.fromarray(img_dct_array.astype(np.uint8))
53
54      create_widgets("Modified Image", root, modified_img, 1000, 700)
55      create_widgets("Original Image", root, original_img, 1000, 700)
```

Il primo blocco riguarda l'ultima parte dei controlli relativi alle text input, in questo caso si verifica che il valore di **d** sia $0 \leq d \leq 2F - 2$.

```
1  try:
2      F = int(entry_F.get())
3      d = int(entry_d.get())
4      if d < 0 or d > (2 * F - 2):
5          raise ValueError
6  except ValueError:
7      messagebox.showerror("Warning", f"d must be an integer between 0 and {2*F - 2}")
8      return
```

A questo punto siamo sicuri che tutte le variabili siano corrette e quindi possiamo procedere con il primo step relativo alla compressione JPEG: suddividere l'immagine in blocchi F x F. Come prima cosa si va a ridimensionare la matrice, in modo tale che possa essere divisibile correttamente da F; si scartano quindi gli avanzi, ovvero un piccolo bordo lungo le ultime righe e ultime colonne. Si calcolano poi quanti blocchi possono essere ottenuti verticalmente e orizzontalmente, salvandone il risultato in 2 variabili.

```
1  height, width,  = img_array.shape
2
3  # Calculate the maximum height and width that are divisible by F
4  height = (height // F) * F
5  width = (width // F) * F
6
7  # Crop the image to the calculated dimensions
8  image_to_compress = img_array[:height, :width]
9
10 block_height = height // F
11 block_width = width // F
```

Procedendo, utilizziamo le variabili appena trovate per costruire 2 cicli for innestati, che scansionano orizzontalmente la matrice come se fossero delle finestre. Quello che si fa è estrarre i nostri blocchetti, salvando la sottomatrice F x F nella variabile *block*. Sulla finestra, si applica la funzione dctn con parametro *norm='ortho'*, il quale specifica che la DCT deve essere normalizzata ortogonalmente. Elimino, poi, le frequenze c_{kl} con $k + l \geq d$, diagonalmente partendo dalle più alte, come mostrato in figura.

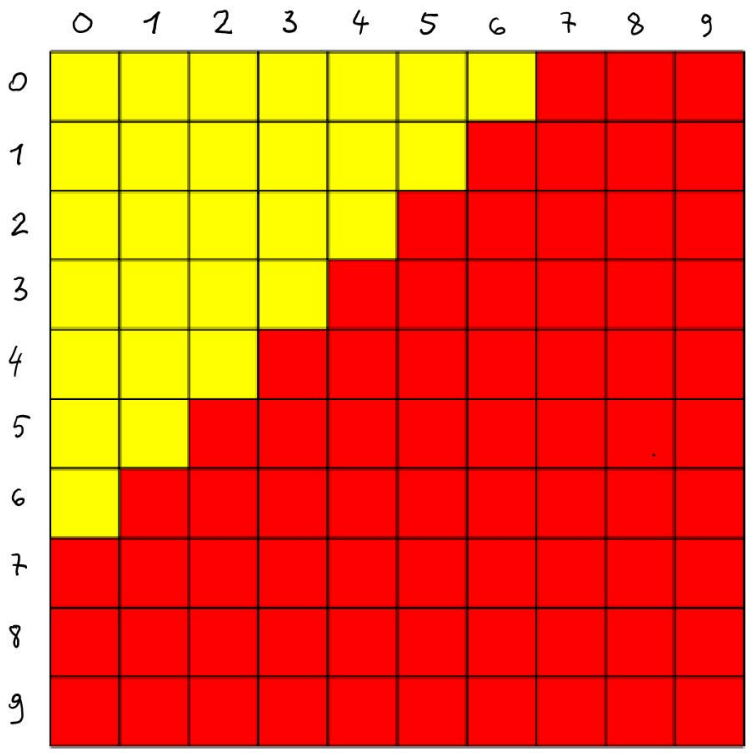

Figura 4: Blocco 10×10 con d = 7

```
for i in range(block_height):
    for j in range(block_width):
        block_start_i = i * F
        block_end_i = block_start_i + F
        block_start_j = j * F
        block_end_j = block_start_j + F
        block = image_to_compress[block_start_i:block_end_i, block_start_j:block_end_j]

        c = dctn(block, norm='ortho')

        for k in range(F):
            for l in range(F):
                if k + l >= d:
                    c[k, l] = 0

        img_dct_array[block_start_i:block_end_i, block_start_j:block_end_j] = c
```

L'ultima fase riguarda la ricostruzione dell'immagine partendo dalla matrice ottenuta dall'operazione di DCT2. Applichiamo subito la funzione IDCT2 della libreria *Scipy* (anche lei con parametro di normalizzazione *ortho*), che ci permette di eseguire l'inversa della Trasformata Discreta del Coseno. Ogni valore della matrice ottenuta, viene arrotondato all'interno più vicino, siccome ovviamente i pixel devono essere rappresentati da numeri interi. Siccome l'intensità dei pixel varia all'interno del range 0-255, effettuiamo un operazione di "clip", ovvero i valori ≤ 0 vengono impostati a 0, mentre quelli $\geq$ di 255, a 255. Infine viene salvato il blocco all'interno della matrice che verrà poi utilizzata, alla fine del ciclo, come visualizzazione grafica della nostra immagine compressa. Le ultime righe di codice, riguardano una funzione che permette di mostrare a schermo l'immagine originale e l'immagine compressa, così da poterle paragonare e trarne risultati e conclusioni.

```
block_idct = idctn(c, norm='ortho')

# Round and clip block_idct
block_idct = np.round(block_idct).astype(int)
```

```
block_idct = np.clip(block_idct, 0, 255)

# Update img_dct_array
img_dct_array[block_start_i:block_end_i, block_start_j:block_end_j] = block_idct

modified_img = Image.fromarray(img_dct_array.astype(np.uint8))

create_widgets("Modified Image", root, modified_img, 1000, 700)
create_widgets("Original Image", root, original_img, 1000, 700)
```

2.3 Risultati

Durante la costruzione del programma, sono stati eseguiti molteplici test utilizzando diverse immagini. In questa sezione riportiamo diversi test, effettuati su un'immagine BMP di dimensioni 512 x 512 pixel. Vengono evidenziati gli effetti che i parametri F e d causano, sia a livello computazionale che sulla qualità dell'immagine.

2.3.1 F = dimensione dell'immagine

Il primo caso di test viene eseguito utilizzando il valore massimo possibile per la costante F, in questo caso 512. Avremo quindi che la matrice rappresentante la nostra immagine viene divisa in un singolo blocco, ovvero viene presa interamente. Verifichiamo ora cosa può succedere al variare del valore d:

1. Prendiamo il valore massimo di d, in questo caso 1022:

(a) Immagine originale

(b) Immagine compressa

   Come previsto non ci sono grandi cambiamenti relativi alla qualità dell'immagine, solo un leggerissimo "velo" derivante dall'arrotondamento dei valori della IDCT2.

2. Il valore d=0, è ovviamente relativo ad un'immagine nera, ovvero vengono tagliate tutte le frequenze dell'immagine lasciando una matrice piena di 0.

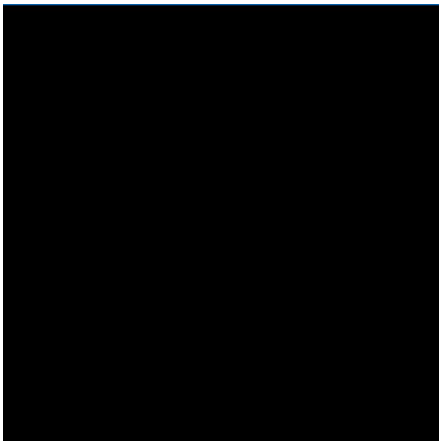

Figura 6: d = 0

3. Utilizziamo ora diversi valori di d:

(a) d = 15

(b) d = 30

(c) d = 70

(d) d = 140

A colpo d'occhio notiamo come l'immagine sia qualitativamente molto pessima man mano che diminuiamo il nostro valore d, permettendo all'algoritmo di tagliare sempre più frequenze della nostra matrice. Tramite questa strategia, abbiamo un secondo problema. Un valore alto del parametro d, come può esserlo 140, oltre ad avere una compressione ancora non sufficiente, presenta il fenomeno di Gibbs. Notiamo, infatti, come nelle aree di maggior contrasto, come il confine capelli-sfondo e all'interno del viso, vi siano le caratteristiche "onde" di questo fenomeno. Questo è dato dal fatto che la DCT2, fa molta fatica nei punti di discontinuità, ovvero quando si passa da un valore elevato ad uno minore (o viceversa), andando a produrre questo effetto indesiderato.

2.3.2 F = 8

Per ovviare a questo problema, consideriamo blocchi con una dimensione più ridotta, proviamo quindi ad utilizzare la stessa tecnica del JPEG. Suddividiamo quindi ora la nostra immagine in blocchetti 8 x 8, notando una cosa strabiliante: non solo abbiamo rimosso il problema del fenomeno di Gibbs, ma eliminando moltissime frequenze abbiamo una qualità dell'immagine molto buona. Rispetto all'effetto "sbiadito" rilevato nella precedente sezione, abbiamo un effetto "censura" dato dal fatto che le sezioni da approssimare sono molto più piccole, e quindi i valori e i salti di frequenze non sono così marcati. Tutto questo mantenendo anche un rapporto qualità/velocità molto alto. Infatti non abbiamo bisogno di impostare il parametro d ad un valore troppo elevato, rischiando di appesantire la IDCT2, ma basta anche solo fermarsi ad 8 per ottenere un'immagine qualitativamente molto buona.

(a) d = 1

(b) d = 2

(c) d = 5

(d) d = 8

3 Conclusioni

Questo progetto ha fornito una comprensione approfondita della DCT2 e delle sue applicazioni nella compressione delle immagini. Abbiamo sperimentato vari algoritmi, analizzandoli e confrontandoli sia con le funzioni già implementate in libreria, sia studiandone l'effetto che le soglie di frequenza sulla qualità dell'immagine compressa causavano. Abbiamo constatato che la scelta dei parametri F e d è determinante per la qualità della compressione e la velocità dell'algoritmo. Valori troppo elevati di F portano al dover accrescere il valore di d e al rischio del presentarsi del fenomeno di Gibbs. Di contro, valori troppo piccoli rischiano di andare ad intaccare la velocità dell'algoritmo e la quantità di valori effettivamente risparmiati.
In conclusione, il progetto ci ha permesso di capire in maniera più pratica e dettagliata quanto succedeva nell'ambito della compressione JPEG, utilizzando la Discrete Cosine Transform, analizzandone tempi e caratteristiche fondamentali.