

Università degli Studi di Milano Bicocca

Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di Laurea Magistrale in Informatica - LM 18

Metodi del Calcolo Scientifico Progetto 2

Compressione JPEG

Compressione di immagini tramite DCT

Componenti del gruppo:

Bancora Davide - M.905588

Donato Benedetta - M.905338

Dubini Emanuele - M.904078

Anno Accademico 2022-2023

Indice

1 DCT – Discrete Cosine Transform	
Prima parte	1
1.1 Introduzione	1
1.2 Discrete Cosine Transform (DCT)	2
1.2.1 Descrizione del codice sorgente	3
1.2.2 Risultati ottenuti	9
2 Compressione JPEG	
Seconda parte	13
2.1 Introduzione	13
2.2 Il codice	14
2.2.1 Descrizione del codice implementato	14
2.2.2 Analisi approfondita del codice	15
2.3 Risultati ottenuti	25
2.3.1 Casi limite	25
2.3.2 Ulteriori risultati ottenuti	27
2.4 Conclusioni	32
2.4.1 Considerazioni finali	32

Elenco delle figure

1.1	Tempi di esecuzione della DCT2 al variare della dimensione N	10
2.1	Interfaccia grafica	16
2.2	Caso 1	26
2.3	Caso 2	27
2.4	Caso 3	28
2.5	Caso 4	29
2.6	Caso 5	30
2.7	Caso 6	31

1

DCT – Discrete Cosine Transform Prima parte

1.1 Introduzione

Il progetto presentato all'interno di questa relazione ha lo scopo di eseguire un'analisi comparativa delle prestazioni ottenibili da un'implementazione manuale ed una basata sulla libreria Fast Fourier Transform (FFT) per la Discrete Cosine Transform di secondo tipo (DCT2).

A tal fine, sarà utile valutare come i tempi di esecuzione delle due implementazioni variano al crescere delle dimensioni della matrice fornita in input, includendo la nostra implementazione della DCT2 e quella di una libreria di riferimento.

In questo capitolo esamineremo anche come le loro prestazioni si confrontano con le complessità teoriche attese.

Come da aspettativa, è possibile prevedere a priori che l'implementazione manuale della DCT2 abbia un tempo di esecuzione approssimativamente proporzionale a N^3 mentre l'implementazione basata sulla libreria FFT avrà un tempo di esecuzione proporzionale a $N^2 * \log(N)$: tale differenza è dovuta dalle diverse strategie algoritmiche utilizzate nei due approcci.

L'analisi dei tempi di esecuzione dei due algoritmi contribuirà a valutarne l'efficienza

in relazione alle dimensioni della matrice data in input.

1.2 Discrete Cosine Transform (DCT)

La Discrete Cosine Transform (DCT) è un concetto chiave nell'elaborazione delle immagini, in quanto consente la rappresentazione delle stesse in termini di frequenze cosinusoidali.

Di fatto, questo tipo di rappresentazione è da considerarsi fondamentale per la compressione delle immagini, dato che consente di concentrare le informazioni rilevanti in un numero ridotto di coefficienti riducendo, di conseguenza, la quantità di dati necessari alla memorizzazione di ciascun file.

Possiamo quindi affermare che la DCT è una trasformata matematica in grado di convertire un vettore di numeri reali (f) in un vettore di frequenze (c) calcolato come la moltiplicazione tra la matrice di trasformazione e il vettore di coefficienti reali.

La matrice di trasformazione è ottenuta tramite la formula:

$$D_{i,j} = a_l^N \cos\left(l\pi \frac{2j + 1}{2N}\right)$$

dove i coefficienti a_l^N hanno valore $\frac{1}{\sqrt{N}}$ se l è uguale a 0, oppure $\sqrt{\left(\frac{2}{N}\right)}$ se l è diversa da 0 come di seguito mostrato:

$$a_l^N = \begin{cases} \frac{1}{\sqrt{N}} & \text{se } l = 0 \\ \sqrt{\frac{2}{N}} & \text{altrimenti} \end{cases} \quad (1.1)$$

La trasformata inversa della DCT prende il nome di IDCT consentendo di ottenere numeri reali a partire da un vettore di frequenze.

Inoltre, anche nel caso bidimensionale, possono essere utilizzate la DCT e la IDCT calcolando inizialmente la DCT sulle colonne e successivamente applicando nuovamente la stessa formula sulle righe di un'ipotetica matrice: questa operazione prende il nome di DCT2.

L'utilizzo della DCT2, applicata al mondo delle immagini, si estende anche alla ricostruzione del file originale. Inoltre, tramite la trasformata inversa della DCT2, chiamata IDCT2, è possibile ricostruire l'immagine originale partendo dai coefficienti generati dalla DCT2.

Questo è un processo fondamentale per la compressione delle immagini poiché consente di ripristinare un'approssimazione dell'immagine originale dopo aver eseguito la trasformata.

1.2.1 Descrizione del codice sorgente

In questa sezione verranno elencate e descritte le funzioni sviluppate per raggiungere gli obiettivi del progetto.

Il codice sorgente completo è disponibile al link GitHub: https://github.com/dbancora/ProgettoCalcoloScientifico_DCT2.

Il linguaggio di programmazione da noi scelto per lo sviluppo è Python, decisione è motivata da un insieme di vantaggi offerti dall'ambiente stesso in ambito di elaborazione delle immagini e sviluppo di applicazioni desktop. Inoltre, la semplicità ed intuitività della sintassi richiesta da Python rappresentano aspetti chiave:

- Esso è un linguaggio di programmazione noto per la sua semplicità di lettura e scrittura del codice sorgente, caratteristica di fondamentale importanza nell'affrontare compiti complessi come nel caso di elaborazione di immagini.
- La natura intuitiva del linguaggio agevola lo sviluppo del codice e la sua comprensione, rendendo l'intero processo di sviluppo più efficiente.
- Anche la presenza di una ampia selezione di librerie specializzate per l'elaborazione delle immagini e l'analisi dei dati, rappresenta un vantaggio significativo. Python offre una vasta gamma di librerie di terze parti che coprono ogni aspetto dell'elaborazione delle immagini, consentendo agli sviluppatori di sfruttare strumenti potenti e altamente ottimizzati senza la necessità di svilupparli manualmente.

Seguendo con l'implementazione del progetto, è presente un confronto tra la funzione DCT, da noi sviluppata, con la funzione fornita dal package FFT, contenuto all'interno della libreria Scipy.

Quest'ultimo costituisce una versione ottimizzata dell'algoritmo DCT2 presentando alta efficienza nelle operazioni matematiche della trasformata.

Come detto precedentemente, l'obiettivo del progetto sarà quello di confrontare le due funzioni tra loro in termini di: tempo necessario per eseguire la trasformata DCT2 e in termini di complessità computazionale.

Per una maggiore sicurezza inoltre, il codice prevede di effettuare dei test preliminari con l'obiettivo di verificare l'effettiva correttezza della funzione DCT da noi

implementata: nel caso in cui i risultati restituiti dalla trasformata corrispondano ai valori contenuti all'interno della traccia del progetto, sarà possibile proseguire effettuando il confronto tra le due funzioni.

L'implementazione delle funzioni DCT e DCT2 da noi create e di quelle fornite dalla libreria `Scipy.fft` è disponibile nel file `utils.py`.

Al suo interno troviamo:

- La funzione `dct_created` che riceve in input un vettore monodimensionale dove, a sua volta, viene passato come parametro alla funzione `create_transformation_matrix` con il compito di creare la matrice di trasformazione necessaria per il calcolo della DCT.

Una volta calcolata questa viene restituita dalla funzione e utilizzata per il calcolo della trasformata ottenuta moltiplicando la matrice appena creata con il vettore passato come parametro.

```

1 # Funzione per DCT creata
2 def dct_created(input_vector):
3     # Creazione della matrice di trasformazione
4     transformation_matrix=create_transformation_matrix(input_vector)
5
6     # Calcolo del risultato (c) come la moltiplicazione tra la
7     # matrice di trasformazione con il vettore dato in input (f)
8
9     dct_result = np.dot(transformation_matrix, input_vector)
10    return dct_result

```

Listing 1.1: Funzione `dct_created`

- La funzione `create_transformation_matrix` che prende in input un vettore passatogli come parametro di cui ne verrà calcolata la rispettiva lunghezza, salvandola nella variabile `n`.

Ottenuta la dimensione del vettore, gli unici valori mancanti per effettuare il calcolo della matrice di trasformazione sono rappresentati dai coefficienti `alpha`, che vengono inseriti all'interno di un vettore creato e inizializzato a 0: il primo valore è pari a $\frac{1}{\sqrt{n}}$, mentre per calcolare i successivi, si utilizza la formula $\sqrt{\left(\frac{2}{n}\right)}$.

In seguito al calcolo dei coefficienti, si dispone ora di tutti i valori per creare la matrice di trasformazione seguendo i passaggi elencati:

- Viene creata una matrice vuota di dimensioni $n \times n$;
- Ogni elemento della matrice viene calcolato seguendo la formula descritta precedentemente;
- L'elemento appena calcolato viene aggiunto nella posizione i, j secondo due cicli **for** innestati tra loro;
- L'ordine di riempimento prevede l'inserimento dei valori prima per le righe e successivamente per le colonne.

Una volta terminati i cicli, viene restituita la matrice. Di seguito viene riportato il codice di quanto appena illustrato:

```

1 def create_transformation_matrix(a):
2     # Calcolo della lunghezza dell'array
3     n = len(a)
4
5     # Creazione del vettore alfa lungo quanto il vettore passato
6     alpha = np.zeros(n)
7
8     # Calcolo dei valori in base alla posizione
9     alpha[0] = 1 / np.sqrt(n)
10    alpha[1:] = np.sqrt(2/n)
11
12    # Creazione della matrice di trasformazione
13    transformation_matrix = np.zeros((n, n))
14    for i in range(n):
15        for j in range(n):
16            transformation_matrix[i, j] = alpha[i] *
17                np.cos((i * math.pi * (2 * j + 1)) / (2 * n))
18
19    return transformation_matrix

```

Listing 1.2: Funzione `create_transformation_matrix`

Successivamente è stata sviluppata l'implementazione della DCT2 che richiama due volte la funzione `dct_created()` sopra descritta, prima per le colonne e successivamente per le righe di una matrice che viene passata come parametro.

Successivamente è riportato il codice dell'implementazione della funzione `dct2_created()`:

```

1 # Funzione per DCT2 creata
2 def dct2_created(input_matrix):
3     n, m = input_matrix.shape
4
5     # Creazione della matrice
6     dct2_result = np.copy(input_matrix.astype('float64'))
7
8     # DCT per ogni colonna
9     for j in range(m):
10         dct2_result[:, j] = dct_created(dct2_result[:, j])
11
12     # DCT per ogni riga
13     for i in range(n):
14         dct2_result[i, :] = dct_created(dct2_result[i, :])
15
16     return dct2_result

```

Listing 1.3: Funzione `dct2_created`

Una volta terminate le operazioni, viene restituito il risultato della DCT2.

Per implementare una funzione che permetesse di calcolare la DCT e la DCT2 tramite una libreria esterna, si è ricorso all’ utilizzo della la libreria **open-source Scipy** sopra descritta: importando il pacchetto FFT, è stato possibile utilizzare la funzione DCT della libreria per poi successivamente confrontare i tempi di esecuzione tra la funzione da noi implementata con la funzione resa disponibile da Scipy.

Di seguito viene riportato il codice delle funzioni `dct_library()` e `dct2_library()`:

```

1 # Funzione DCT implementata da libreria esterna
2 def dct_library(f):
3     return dct(f.T, norm='ortho')
4
5 # Funzione DCT2 implementata da libreria esterna
6 def dct2_library(f):
7     return dct(dct(f.T, norm='ortho').T, norm='ortho')

```

Listing 1.4: Funzione `dct_library`

La prima funzione calcola la DCT di un array monodimensionale, essa riceve in ingresso i seguenti parametri:

- L’array monodimensionale di cui deve essere calcolata la DCT.

- Il parametro `T`, posto in successione all'array, permette di calcolare il suo traspoto così da replicare il calcolo della DCT da noi implementato che prevede l'applicazione della trasformata prima sulle colonne e successivamente sulle righe.
- Il parametro `ortho` utile per indicare alla funzione `DCT` di considerare anche i valori alpha durante il calcolo della matrice di trasformazione.

Per quanto riguarda la seconda funzione, serve a calcolare la `DCT2` su una matrice bidimensionale passatale come argomento, ed è stata implementata chiamando due volte la funzione `DCT` della libreria `Scipy`.

Una volta sviluppato il codice necessario per la definizione delle trasformate, è possibile eseguire i test richiesti: il file `test.py` è progettato per eseguire una serie di test e misurazioni di performance delle trasformate discrete del coseno (`DCT` e `DCT2`) applicati su matrici di dati.

Analizzando il codice da noi implementato, è prevista inizialmente l'importazione delle librerie necessarie come `numpy`, utile per eseguire operazioni matematiche e `timeit` che ha il compito di misurare i tempi di esecuzione delle funzioni.

La funzione `run_test()`, di seguito riportata, esegue i test stampando i risultati delle `DCT` e `DCT2` da noi sviluppate insieme ai risultati ottenuti utilizzando le `DCT` e `DCT2` fornite dalla libreria `Scipy` nel pacchetto `fft`.

```
1 def run_test():
2     # Test DCT:
3     a = np.array([231, 32, 233, 161, 24, 71, 140, 245])
4
5     dct = utils.dct_created(a)
6
7     formatted_dct = ["{:.2e}".format(val) for val in dct]
8     print("\n-----TEST DCT HomeMade-----")
9     print(formatted_dct)
10
11    input_matrix = np.array([[231, 32, 233, 161, 24, 71, 140, 245],
12                            [247, 40, 248, 245, 124, 204, 36, 107],
13                            [234, 202, 245, 167, 9, 217, 239, 173],
14                            [193, 190, 100, 167, 43, 180, 8, 70],
15                            [11, 24, 210, 177, 81, 243, 8, 112],
16                            [97, 195, 203, 47, 125, 114, 165, 181],
```

```

17                     [193, 70, 174, 167, 41, 30, 127, 245],
18                     [87, 149, 57, 192, 65, 129, 178, 228]])

19
20     dct2_result = utils.dct2_created(input_matrix)
21     print("\n-----\n")
22     print("-----TEST DCT2 HomeMade-----")
23     print(dct2_result)

24
25     dct = utils.dct_library(a)
26     formatted_dct = ["{:.2e}".format(val) for val in dct]
27     print("\n-----\n")
28     print("-----TEST DCT Library-----")
29     print(formatted_dct)

30
31     dct2_result = utils.dct2_library(input_matrix)
32     print("\n-----\n")
33     print("-----TEST DCT2 Library-----")
34     print(dct2_result)

```

Listing 1.5: Funzione run_test

Questo codice è utile per verificare se le funzioni da noi implementate restituiscono dei risultati corretti, che devono essere confrontati con i dati presenti nella consegna del progetto. Eseguendo il codice, si nota che i risultati coincidono, suggerendo che l'implementazione della trasformata è corretta.

La funzione `test_N()` esegue una serie di test di performance su matrici di dimensioni $N \times N$, misurando anche il tempo impiegato per la risoluzione di DCT e DCT2.

```

1 def test_N():
2     # Dimensioni delle matrici NxN (da 50 a 900 con passo 50)
3     matrix_dimensions = list(range(50, 951, 50))
4
5     times_scipy_dct = []
6     times_my_dct = []
7
8     for n in matrix_dimensions:
9         print("Dimension: ", n)
10        # Creazione di una matrice random
11        np.random.seed(5)
12        matrix = np.random.uniform(low=0.0, high=255.0, size=(n,n))
13
14        # Calcolo del tempo di esecuzione con scipy DCT2

```

```

15     time_scipy = timeit.timeit
16         (lambda: utils.dct2_library(matrix), number=1)
17     times_scipy_dct.append(time_scipy)
18
19     # Calcolo del tempo di esecuzione con la tua DCT2
20     time_my_dct = timeit.timeit
21         (lambda: utils.dct2_created(matrix), number=1)
22     times_my_dct.append(time_my_dct)
23
24     return times_scipy_dct, times_my_dct, matrix_dimensions

```

Listing 1.6: Funzione test_N

1.2.2 Risultati ottenuti

In seguito alla descrizione del codice implementato, vengono ora presentati i confronti tra le varie implementazioni di DCT2 con l'obiettivo di valutarne i tempi di esecuzione in relazione alle dimensioni delle matrici coinvolte.

Oltre al tempo necessario per l'elaborazione degli input, viene anche verificato se i tempi di esecuzione seguono una relazione proporzionale rispetto alle dimensioni delle matrici.

I valori attestati sono proporzionali a N^3 per la DCT2 implementata da noi e $N^2 * \log(N)$ per l'implementazione resa disponibile dal pacchetto FFT di Scipy.

Al fine di eseguire correttamente il confronto, vengono raccolti i tempi di esecuzione delle due diverse implementazioni di DCT2 applicate ad un set di matrici quadrate $N \times N$, la cui dimensione varia da 50 a 950, incrementando di 50 unità ad ogni iterazione: come è possibile osservare dal codice sopra riportato, rappresentante la funzione `test_N`, i valori inseriti all'interno delle matrici sono casuali e compresi tra 0 e 255.

Il grafico in Figura 1.1, sotto riportato, rappresenta il tempo necessario per eseguire le due implementazioni di DCT2 all'aumentare della dimensione N delle matrici quadrate costruite: sull'asse delle ascisse, viene indicata la dimensione della matrice considerata (N), mentre sull'asse delle ordinate viene annotato il tempo, in scala logaritmica, che la funzione ha impiegato per eseguire la trasformata.

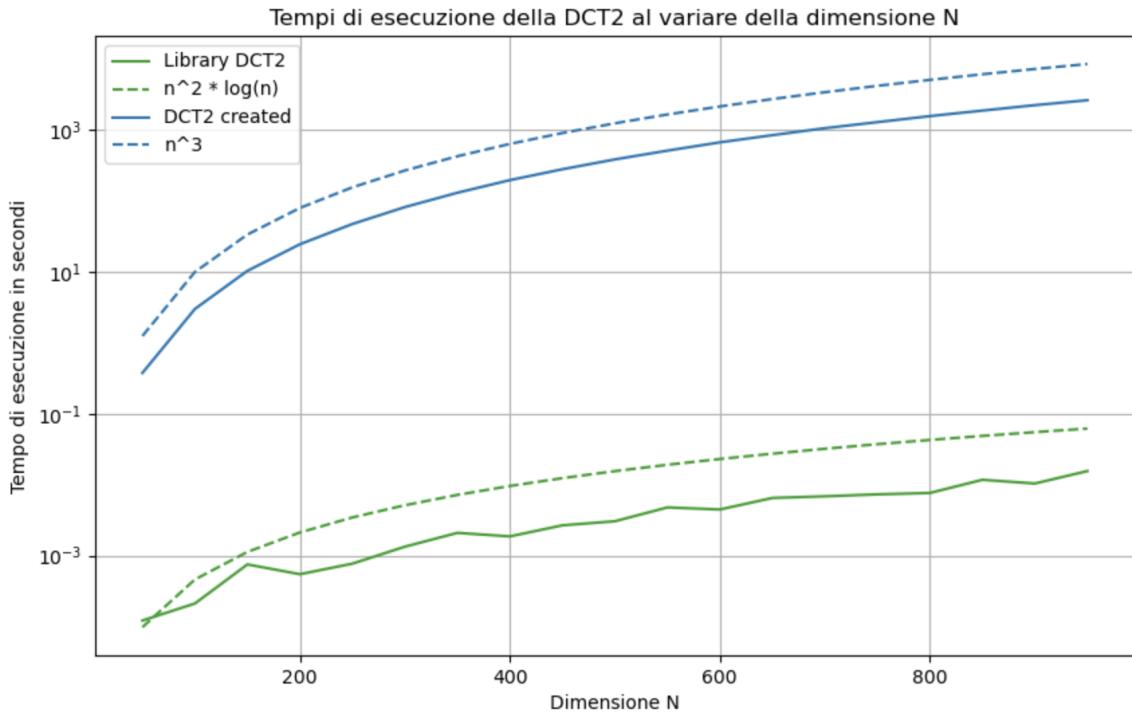


Figura 1.1: Tempi di esecuzione della DCT2 al variare della dimensione N.

Per completezza, è necessario specificare che la linea intera di colore blu rappresenta l'andamento della DCT2 implementata da noi, mentre la linea intera di colore verde rappresenta l'andamento della funzione DCT2 resa disponibile dalla libreria Scipy. Le rispettive linee tratteggiate, anteposte alle linee intere, rappresentano l'andamento N^3 per la funzione da noi implementata e $N^2 * \log(n)$ per la funzione resa disponibile dalla libreria.

Dai risultati ottenuti emerge che i tempi di esecuzione delle implementazioni seguono le previste relazioni di proporzionalità: l'implementazione della funzione DCT2 da noi sviluppata, segue un andamento proporzionale di N^3 , mentre quella basata su Scipy.fft segue un andamento proposizionale a $N^2 * \log(n)$, dimostrando chiaramente che l'utilizzo dell'FFT contribuisce significativamente all'accelerazione dei tempi di esecuzione nella DCT2.

È importante notare che l'andamento del tempo di esecuzione della DCT2, resa disponibile dalla libreria Scipy.fft, non segue un andamento regolare al crescere della dimensione N della matrice. Questo fenomeno può essere attribuito a diversi fattori, tra cui ottimizzazioni interne implementate dalla libreria, l'effetto delle

gerarchie di cache e il coinvolgimento del parallelismo nelle operazioni di calcolo: tali dinamiche possono causare variazioni non monotone nei tempi di esecuzione al variare di N , risultando in comportamenti che possono discostarsi da una semplice relazione lineare.

Pertanto, anche se si potrebbe supporre che il tempo di esecuzione cresca costantemente all'aumentare di N , la presenza di questi fattori può portare a fluttuazioni o a una diminuzione apparentemente anomala in alcuni casi.

2

Compressione JPEG Seconda parte

2.1 Introduzione

La seconda parte del progetto riguarda la compressione di immagini, in formato .bmp utilizzando la Trasformata discreta del coseno di secondo tipo (DCT2), processo utile per simulare la compressione in JPEG.

L’obiettivo principale è implementare un’applicazione che permetta agli utenti di: selezionare un’immagine in scala di grigi, specificare i parametri di compressione (variabile F e parametro di quantizzazione d), applicare la DCT2 e la sua versione quantizzata ai blocchi dell’immagine ed infine visualizzare sia l’immagine originale che quella compressa in modo da evidenziarne le differenze.

2.2 Il codice

2.2.1 Descrizione del codice implementato

Come previsto dalla traccia del progetto, l'utente può selezionare un'immagine in scala di grigi (nel formato .bmp) utilizzando il pulsante "Sfoglia" e successivamente sarà in grado di scegliere i parametri di compressione:

- La variabile F : questo valore rappresenta la dimensione dei blocchi in cui l'immagine verrà suddivisa.
- Il parametro di quantizzazione d : questo parametro influisce sulla quantità di informazioni che verranno mantenute nella compressione.
Valori più bassi di d porteranno ad una maggiore compressione ma anche a una perdita di dettagli.

Una volta inseriti entrambi i parametri, l'utente può avviare il processo di compressione.

L'applicazione provvederà a dividere l'immagine in blocchi di dimensione $F \times F$, applicherà la DCT2 a ciascun blocco e quindi quantizzerà i coefficienti DCT2 basandosi sul valore del parametro d . I blocchi compressi verranno poi ricomposti al fine di riottenere l'immagine compressa finale.

Una volta terminata l'esecuzione della DCT2, l'applicazione mostrerà sia l'immagine originale che l'immagine compressa in due finestre separate, permettendo all'utente di valutare visivamente l'effetto ottenuto dalla compressione.

Anche per la seconda parte del progetto, è stato scelto di utilizzare Python, linguaggio di programmazione che risulta essere ben supportato e dispone di librerie efficienti per l'elaborazione e la visualizzazione delle immagini.

In particolare, nel programma da noi sviluppato, abbiamo sfruttato la libreria PIL (Python Imaging Library) libreria di grande rilevanza nel campo dell'elaborazione delle immagini che consente una serie di soluzioni pragmatiche e potenti per la gestione delle immagini.

La libreria consente il caricamento e il salvataggio di immagini in formati come jpeg, png, bmp e molti altri.

Il vantaggio offerto dalla libreria PIL è la facilità con cui è possibile eseguire operazioni basilari sulle immagini e, con il passare degli anni, la libreria ha subito un'evoluzione sostanziale dando origine a Pillow. Quest'ultima ha assorbito le caratteristiche di PIL e ne ha ampliato le funzionalità anche se, necessitando di un

insieme limitato di operazioni richieste ai fini del progetto, abbiamo optato per l'utilizzo della prima soluzione.

2.2.2 Analisi approfondita del codice

Dopo avere dato una panoramica generale del programma da noi implementato, passiamo ora ad una analisi più dettagliata descrivendo il funzionamento di ogni singola funzione implementata.

Per quanto riguarda la prima parte del codice, ossia la creazione della finestra, è stata realizzata sfruttando la libreria `tkinter` di Python.

Essa fa affidamento ad un insieme di *widget* grafici predefiniti, come finestre, bottoni, caselle di testo, etichette e molto altro, che possono essere disposti e personalizzati per creare l'aspetto desiderato dell'applicazione.

Con l'obiettivo di migliorare la leggibilità del codice, è stata creata la funzione `flow()` all'interno del file `utils.py`, che gestisce il flusso di esecuzione del programma a partire dal controllo dei parametri `F` e `d`, fino ad arrivare alla creazione della finestra per la visualizzazione dell'immagine compressa.

Nel momento in cui viene eseguito il programma, a partire dal file `main.py`, la funzione `create_first_interface()` permette di creare la prima interfaccia grafica che consente all'utente di selezionare l'immagine, in formato `.bmp`, da comprimere ed inserire i parametri `F` e `d` utili per il processo di compressione.

Inoltre, all'interno dell'interfaccia, è stato ritenuto utile anche inserire una `label` che indicasse la dimensione (in pixel) dell'immagine selezionata in quanto, la selezione dei parametri sopra descritti è strettamente legata alle misure dell'immagine stessa.

Il risultato ottenuto dall'esecuzione della funzione `create_first_interface()` è la seguente interfaccia grafica disponibile all'utente:

Compressione JPEG Seconda parte

16

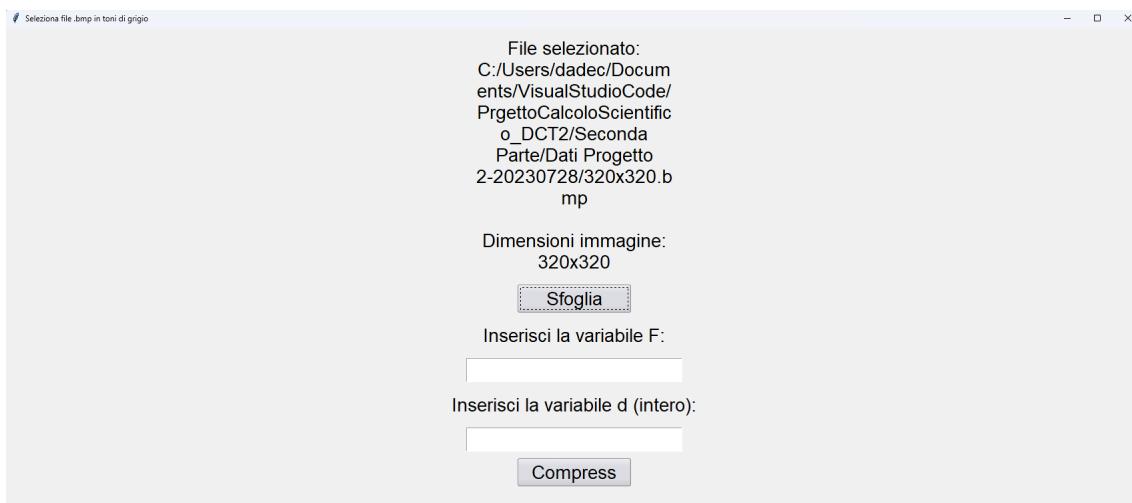


Figura 2.1: Interfaccia grafica

La seconda funzione, denominata `check_variables()`, viene richiamata nel momento in cui l’utente decide di comprimere l’immagine precedentemente selezionata. Essa permette di verificare se i dati inseriti siano corretti ed in particolare:

- La variabile `F`, che rappresenta la dimensione dei macro-blocchi in cui si effettuerà la DCT2, deve essere un numero intero e minore dell’altezza (o larghezza) dell’immagine selezionata;
- La variabile `d`, che rappresenta la soglia di taglio delle frequenze, deve essere un intero compreso tra 0 e $2 * F - 2$.

Viene inoltre verificato che tutti i dati necessari per la compressione siano stati inseriti e che non siano presenti valori nulli. In caso di problemi viene creata una finestra indicante l’errore riscontrato così da agevolare l’utente nel risolvere il problema.

Di seguito è fornito il codice dell’implementazione della funzione appena descritta:

```
1 def check_variables():
2     global entry_variable_f, entry_variable_d, F
3     compressed_image_path = "compressed_image.bmp"
4
5     try:
6         # Verifica se esiste già un file "compressed_image.bmp"
```

```
7     if os.path.exists(compressed_image_path):
8         os.remove(compressed_image_path) #Rimuovi file esistente
9         print("Rimossa l'immagine")
10    except Exception as e:
11        print("Errore durante la rimozione dell'immagine compressa",
12              str(e))
13
14    # Recupera i valori di F e d dalla label
15    F = entry_variable_f.get()
16    d = entry_variable_d.get()
17
18    # Verifica che le label e l'immagine non siano vuote.
19    # In caso contrario, viene mostrato un messaggio di errore
20    if not file_path:
21        messagebox.showerror("Errore", "Seleziona un'immagine.")
22        return
23
24    if not F:
25        messagebox.showerror("Errore", "Inserisci il valore di F.")
26        return
27
28    if not d:
29        messagebox.showerror("Errore", "Inserisci il valore di d.")
30        return
31
32    if not is_integer(F):
33        messagebox.showerror("Errore", "F deve essere un numero
34                           intero")
35        return
36
37    if not is_integer(d):
38        messagebox.showerror("Errore", "d deve essere un numero
39                           intero")
40        return
41
42    F = int(F)
43    d = int(d)
44
45    if F < 0:
46        messagebox.showerror("Errore", "F deve essere > o = a 0")
47        return
48
49    if d < 0:
50        messagebox.showerror("Errore", "d deve essere > o = a 0")
```

```

51     return

52

53     # Verifichiamo che il valore di F non sia superiore
54     # all'altezza/larghezza dell'immagine
55     try:
56         with Image.open(file_path) as img:
57             img_width, img_height = img.size
58
59         if F > img_width or F > img_height:
60             messagebox.showerror("Errore", "Il valore di F non puo'
61                     superare le dimensioni dell'immagine.")
62         return
63
64     # Stampa debug delle dimensioni dell'immagine selezionata
65     print("Image Width:", img_width)
66     print("Image Height:", img_height)
67
68 except Exception as e:
69     messagebox.showerror("Errore", f"Errore durante il recupero
70                     delle dimensioni dell'immagine: {str(e)}")
71     return
72
73     # verifichiamo che il valore di d sia compreso tra 0 e 2F-2
74     if d > 2*F - 2 or d < 0:
75         messagebox.showerror("Errore", "Il valore di d deve essere
76                     compreso tra 0 e 2F-2.")
77     return
78
79     # If all checks pass, proceed with compression
80     print("All test are ok.")
81     return F, d, file_path

```

Listing 2.1: Funzione `check_variables`

Al termine dell'esecuzione, la funzione ritorna le variabili `F` e `d`, oltre che a `file_path` utile per salvare in modo permanente la posizione dell'immagine selezionata precedentemente dall'utente.

La funzione `divide_image_into_blocks()` ha il compito di dividere l'immagine precedentemente selezionata in blocchi quadrati $F \times F$, scartando i pixel in eccesso, a seguito della conversione dell'immagine stessa in scala di grigi.

Di seguito riportiamo il codice implementato:

```
1 def divide_image_into_blocks(image_path, F):
2     try:
3         with Image.open(image_path) as img:
4             img_width, img_height = img.size
5
6             # Converti l'immagine in scala di grigi
7             img_gray = img.convert('L')
8
9             # Calcoliamo il numero di blocchi in orizzontale e
10            # verticale
11            num_blocks_horizontal = img_width // F
12            num_blocks_vertical = img_height // F
13
14            blocks = []
15
16            # Iteriamo su tutti i blocchi
17            for j in range(num_blocks_vertical):
18                for i in range(num_blocks_horizontal):
19                    # Calcoliamo le coordinate iniziali e finali del
20                    # blocco corrente
21                    x0 = i * F
22                    y0 = j * F
23                    x1 = x0 + F
24                    y1 = y0 + F
25
26                    # Estraiamo il blocco corrente dall'immagine
27                    block = img_gray.crop((x0, y0, x1, y1))
28                    blocks.append(block)
29            return blocks
30
31    except Exception as e:
32        print("Errore durante l'elaborazione dell'immagine", str(e))
33    return blocks
```

Listing 2.2: Funzione `divide_image_into_blocks`

Inizialmente, la funzione apre l'immagine specificata tramite il percorso `image_path` utilizzando la funzione `Image.open(image_path)` della libreria PIL (Python Imaging Library). Il tutto viene eseguito all'interno di un blocco `with`, che assicura la chiusura del file immagine terminate le operazioni successive.

Una volta completata l'apertura dell'immagine, vengono ottenute le dimensioni rappresentate dalla sua larghezza e altezza per poi converte l'immagine originale in scala di grigi: questo passaggio permette di semplificare ulteriormente le operazioni

successive, poiché i blocchi saranno composti solo da valori di intensità luminosa anziché da valori RGB complessi.

Successivamente, viene calcolato il numero di blocchi orizzontali e verticali che possono essere estratti dall'immagine sfruttando il parametro di input, specificato tramite la variabile F.

La funzione procede eseguendo un ciclo `for` innestato che permette di analizzare tutti i possibili blocchi dell'immagine, ottenibili in base al valore del parametro F specificato. Questa operazione viene svolta utilizzando due cicli nidificati, uno per analizzare lo spostamento verticale e uno per quello orizzontale. È utile specificare che in ciascuna iterazione vengono calcolate le coordinate del quadrato corrente rappresentante un blocco dell'immagine.

La dimensione dei blocchi calcolati è rappresentata dalle coordinate dell'angolo superiore sinistro (x_0, y_0) e dell'angolo inferiore destro (x_1, y_1).

Inoltre, per poter estrarre il blocco corrente si sfrutta il metodo `crop()` della libreria PIL che estrae il quadrato definito dalle coordinate calcolate precedentemente e rappresentante una porzione dell'immagine in scala di grigi da elaborare.

Ogni blocco estratto viene poi aggiunto ad una lista denominata `blocks`, restituita al termine dell'esecuzione della funzione, considerata essenziale per la successiva funzione `apply_dct2()` che permette di applicare la trasformata DCT2, di seguito riportata:

```

1 def apply_dct2(blocks, d):
2     # Lista per salvare i blocchi DCT2 quantizzati
3     blocks_dct_quantized = []
4
5     # Iteriamo su tutti i blocchi e applichiamo la DCT2
6     for block in blocks:
7         # Convertiamo il blocco in un array NumPy
8         block_array = np.array(block)
9
10        # Applichiamo la DCT2 al blocco usando scipy
11        block_dct = dct(dct(block_array.T,
12                            norm='ortho').T, norm='ortho')
13
14        # Quantizzazione: Eliminiamo le frequenze con indice di
15        # riga + indice di colonna >= d
16        block_dct_quantized = block_dct *
17            (np.abs(np.add.outer(range(F), range(F))) < d)
18
19        # Aggiungiamo il blocco DCT2 quantizzato alla lista

```

```

20     blocks_dct_quantized.append(block_dct_quantized)
21
22     return blocks_dct_quantized

```

Listing 2.3: Funzione apply_dct2

La funzione inizia dichiarando una lista vuota chiamata `blocks_dct_quantized`, che verrà utilizzata per memorizzare i blocchi di immagini sottoposti alla DCT2 e successivamente quantizzati.

In seguito, viene avviato un ciclo for che itera attraverso ciascun blocco presente nella lista `blocks` precedentemente calcolata. In ciascuna iterazione, il blocco viene convertito in un array NumPy utilizzando la funzione `np.array(block)`, il che permette di manipolare e calcolare operazioni matematiche sui dati dell'immagine in modo efficiente.

All'interno del ciclo, viene applicata la DCT2 al blocco. Questo processo coinvolge due operazioni di DCT consecutive: una DCT lungo le colonne (trasponendo prima il blocco) e una DCT lungo le righe. È importante sottolineare che l'argomento `norm = ortho` indica che vengano utilizzati fattori di normalizzazione specifici per l'ortogonalità, tipici della DCT.

Dopo aver calcolato la DCT2 per il blocco, viene eseguita l'operazione di quantizzazione. Questo passaggio coinvolge la moltiplicazione dell'array della DCT2 per una maschera binaria calcolata in base al parametro `d`.

Per creare la maschera binaria viene utilizzata la funzione

`np.abs(np.add.outer(range(F), range(F))) < d`, che genera una matrice in cui gli elementi sono True se la somma degli indici di riga e colonna è inferiore a `d`, altrimenti sono False.

Tramite questo processo di quantizzazione vengono effettivamente "eliminate" le componenti di frequenza più alte, contribuendo così alla compressione dei dati.

Infine, il blocco risultante dalla DCT2 e dalla quantizzazione viene aggiunto alla lista `blocks_dct_quantized` che viene restituito dalla funzione stessa.

La successiva funzione `apply_idct2()`, presente all'interno del blocco `flow()`, ha il compito di ricostruire l'immagine compressa originale a partire dai blocchi DCT2 quantizzati.

```

1 def apply_idct2(blocks_dct_quantized):
2     # Lista per salvare i blocchi IDCT2 quantizzati e arrotondati
3     blocks_idct_rounded = []
4

```

```

5  # Iteriamo su tutti i blocchi quantizzati e applichiamo la IDCT2
6  for block_dct_quantized in blocks_dct_quantized:
7      # Applichiamo la IDCT2 al blocco usando scipy
8      block_idct = idct(idct(block_dct_quantized.T,
9          norm='ortho').T, norm='ortho')
10
11     # Arrotondiamo i valori della matrice IDCT2 al valore
12     # intero più vicino
13     block_idct_rounded = np.round(block_idct)
14
15     # Impostiamo a 0 i valori negativi
16     block_idct_rounded[block_idct_rounded < 0] = 0
17
18     # Impostiamo a 255 i valori maggiori di 255
19     block_idct_rounded[block_idct_rounded > 255] = 255
20
21     # Convertiamo la matrice risultante in un array di interi
22     # non segnati a 1 byte
23     block_idct_rounded = block_idct_rounded.astype(np.uint8)
24
25     # Aggiungiamo il blocco IDCT2 arrotondato e quantizzato
26     # alla lista
27     blocks_idct_rounded.append(block_idct_rounded)
28
29 return blocks_idct_rounded

```

Listing 2.4: Funzione apply_dct2

La funzione prende come input la lista di blocchi DCT2 quantizzati chiamata `blocks_dct_quantized`.

Eseguendo un ciclo for, la funzione procede con la decodifica per ciascun blocco DCT2 quantizzato e ad ogni blocco viene applicata la IDCT2 (Inverse Discrete Cosine Transform di tipo 2), utilizzando la libreria Scipy, permettendo di convertire ciascun blocco DCT2 quantizzato in un blocco di valori di pixel.

I valori ottenuti dalla IDCT2 sono arrotondati al valore intero più vicino con la funzione `np.round()`: questo passaggio è fondamentale poiché le intensità dei pixel devono essere rappresentate come numeri interi.

Successivamente, i valori dei pixel nella matrice risultante sono regolati per garantire che si trovino nel range corretto, ed in particolare i valori negativi sono impostati a 0 e quelli superiori a 255 sono limitati a 255, rispettando così l'intervallo di intensità dei pixel.

La matrice risultante viene convertita in un array di interi non segnati a 1 byte uti-

lizzando astype(np.uint8), passaggio essenziale per ottenere una rappresentazione corretta dei valori di pixel dell'immagine.

Il blocco ricostruito, arrotondato e quantizzato, viene quindi aggiunto alla lista `blocks_idct_rounded`, che contiene tutti i blocchi dell'immagine ricostruita e, una volta completata l'elaborazione di tutti i blocchi, la funzione restituisce la lista stessa, che rappresenta l'immagine compressa originale ricostruita attraverso il processo di decodifica, una volta terminati i passaggi sopra descritti, è ora possibile ricostruire l'immagine compressa tramite la funzione `save_compressed_image()` che, utilizzando i blocchi IDCT2 arrotondati e quantizzati, è in grado di ricreare l'immagine compressa e salvarla in formato .bmp.

Di seguito viene riportato il relativo codice:

```
1 def save_compressed_image(blocks_idct_rounded):
2
3     compressed_image_path = "compressed_image.bmp"
4     compressed_image = None
5
6     # Verifica se esiste già un file "compressed_image.bmp"
7     if os.path.exists(compressed_image_path):
8         os.remove(compressed_image_path) # Rimuovi file esistente
9         print("Rimossa l'immagine")
10
11    try:
12        # Ricomponi l'immagine compressa usando i blocchi compressi
13        img_width, img_height = Image.open(file_path).size
14        compressed_image = Image.new('L', (img_width, img_height))
15
16        F = entry_variable_f.get()
17        num_blocks_horizontal = img_width // int(F)
18
19        for j in range(img_height // int(F)):
20            for i in range(num_blocks_horizontal):
21                x0 = i * int(F)
22                y0 = j * int(F)
23                block = blocks_idct_rounded.pop(0)
24                compressed_image.paste(Image.fromarray(block),
25                                      (x0, y0))
26
27        # Salva l'immagine compressa nel formato .bmp
28        compressed_image.save(compressed_image_path)
29        compressed_image.close()
30
```

```

31     print("Immagine compressa salvata con successo.")
32 except Exception as e:
33     print("Errore durante il salvataggio dell'immagine
34         compressa:", str(e))

```

Listing 2.5: Funzione `save_compressed_image`

La funzione inizia definendo il percorso in cui verrà salvata l’immagine compressa con il nome `compressed_image.bmp`.

Inizialmente, la variabile `compressed_image` è impostata su `None` e, successivamente, viene controllato se esiste già un file con il nome

`compressed_image.bmp` nel percorso specificato: in caso affermativo, viene rimosso al contrario, si procede senza effettuare alcuna azione.

All’interno del blocco `try`, inizia il processo di ricostruzione dell’immagine compressa: inizialmente vengono recuperate le dimensioni dell’immagine originale utilizzando la funzione `Image.open(file_path).size`, in cui la variabile `file_path` rappresenta il percorso dell’immagine originale.

Successivamente viene creata una nuova immagine vuota di dimensioni (`img_width`, `img_height`) utilizzando il formato ”L” (scala di grigi) che sarà impiegata per ricostruire l’immagine compressa.

Il valore di `F`, che rappresenta la dimensione dei blocchi, viene recuperato dalla variabile `entry_variable_f`, utile per calcolare il numero di blocchi orizzontali in base alle dimensioni dell’immagine originale.

Due cicli nidificati vengono avviati per ricostruire l’immagine compressa iterando attraverso le righe e le colonne in base al numero di blocchi orizzontali e verticali e, per ciascun blocco, vengono calcolate le coordinate iniziali (x_0, y_0) del blocco sulla nuova immagine.

Viene quindi estratto il primo blocco dalla lista `blocks_idct_rounded` utilizzando il metodo `.pop(0)` convertito in un oggetto `Image` sfruttando `Image.fromarray(block)` e quindi incollato sulla nuova immagine compressa utilizzando il metodo `.paste()`.

Una volta che tutti i blocchi sono stati aggiunti all’immagine compressa, questa viene salvata nel formato `.bmp` utilizzando il metodo `.save(compressed_image_path)`. Infine, l’oggetto che rappresenta l’immagine compressa viene chiuso con il metodo `.close()`.

In conclusione, terminata l’esecuzione della funzione, l’immagine compressa viene salvata per essere successivamente visualizzata a schermo tramite l’utilizzo di una nuova finestra in cui sono mostrate contemporaneamente l’immagine originale e quella compressa.

2.3 Risultati ottenuti

In questa sezione verranno mostrati alcuni test effettuati utilizzando immagini in formato .bmp fornite all'interno della consegna del progetto, con lo scopo di verificare se l'implementazione da noi effettuata rispetti le regole per la compressione tramite DCT2.

2.3.1 Casi limite

Con casi limite si intendono esecuzioni del programma in cui vengono compresse immagini utilizzando l'algoritmo DCT2 specificando valori dei parametri parametri F e d al limite della loro correttezza e accettabilità. In particolare considereremo le seguenti esecuzioni:

- compressione di un'immagine specificando il valore di F pari alla misura del lato più piccolo dell'immagine importata, ad esempio: a fronte di un'immagine rettangolare 100x80 pixel verrà assegnato ad F il valore 80.
- compressione di un'immagine specificando il valore di d pari al massimo valore possibile dato da $2 * F - 2$ oppure il valore minimo zero.

Caso 1

Per testare il programma verrà utilizzata l'immagine **80x80.bmp** disponibile al link https://github.com/dbancora/ProgettoCalcoloScientifico_DCT2/tree/main/Seconda%20Parte/Dati%20Progetto%202-20230728. Essendo un'immagine quadrata di dimensione 80x80 pixel assegniamo alla variabile F il valore massimo possibile pari a 80, mentre assegniamo a d il valore 158, anch'esso il valore massimo consentito.

Dal risultato di questa compressione non ci aspettiamo di ottenere notevoli differenze tra l'immagine inizialmente fornita e quella finale dato che la maggior parte delle frequenze, ottenute tramite DCT2, non vengono eliminate. Di seguito riportiamo il risultato ottenuto dalla compressione:

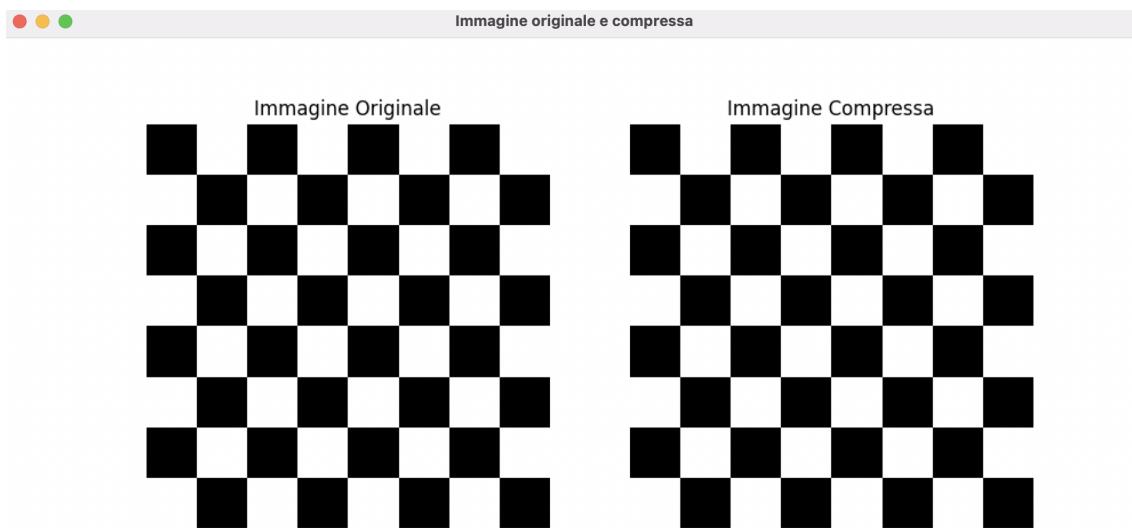


Figura 2.2: Caso 1

Osservando i risultati, come supposto, non sono presenti notevoli cambiamenti tra le due immagini.

Caso 2

Analizziamo ora il secondo caso limite: sempre utilizzando la stessa immagine considerata nel caso precedente, assegneremo ora alla variabile F un valore pari a 80, ossia il massimo possibile. Mentre, per quanto riguarda il valore d assegneremo il valore minimo possibile, corrispondente a zero.

Tale valore permetterà alla funzione DCT2 di assegnare a tutte le frequenze dell'immagine originale il valore zero, aspettandoci di ottenere un'immagine compressa completamente nera, come mostrato di seguito:

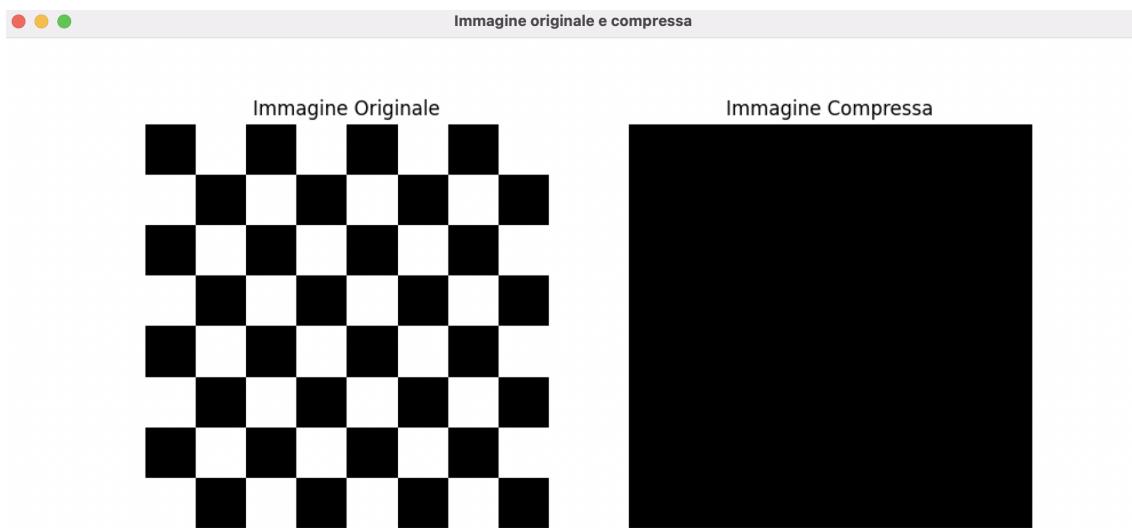


Figura 2.3: Caso 2

Osservando i risultati ottenuti possiamo affermare che, riducendo il parametro d e settandolo a zero, confermiamo la correttezza di quanto ipotizzato. Inoltre notiamo come, riducendo il valore di F nell'intervallo dei suoi valori accettabili e settando la variabile d sempre al valore zero, l'immagine ottenuta dalla compressione risulterà sempre totalmente nera.

2.3.2 Ulteriori risultati ottenuti

In questa sezione proseguiremo nella descrizione di ulteriori prove effettuate, utilizzando il programma da noi sviluppato, con lo scopo finale di denotarne ulteriori considerazioni.

Caso 3

Per eseguire questa prova utilizzeremo l'immagine chiamata `bridge.bmp`, di dimensioni 2749x4049. Al parametro F è stato associato un valore pari a 1567, che non è divisore delle dimensioni dell'immagine importata, mentre per quanto riguarda d associamo il valore di 3132 che, dato il valore di F precedentemente definito, corrisponde al massimo valore possibile. A seguito della compressione, ottenendo il seguente risultato:

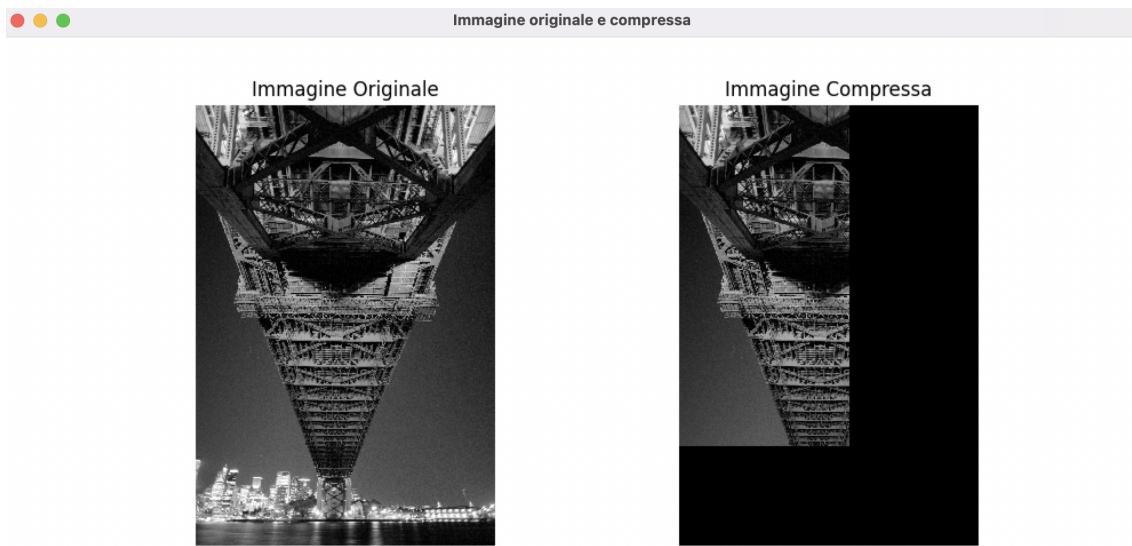


Figura 2.4: Caso 3

Osservando il risultato ottenuto, notiamo una rilevante presenza di alcune zone totalmente nere e due singoli blocchi di pixel compressi. Questo fenomeno è dovuto al fatto che, il valore di F assegnato per la compressione, non è un divisore perfetto della dimensione (sia orizzontale, che verticale) dell'immagine originale e di conseguenza il programma non è in grado di realizzare dei blocchi di dimensione $F \times F$ che permettano di coprire interamente i pixel dell'immagine.

Per quanto riguarda il valore d assegnato, questo risulta essere il massimo valore possibile, quindi l'immagine ricostruita, pur non essendo completa, presenta il minor numero possibile di frequenze tagliate rimanendo così invariata agli occhi dell'osservatore.

Caso 4

Con il quarto test, utilizzando sempre la stessa immagine, assegneremo alla variabile F il valore 1374, circa la metà rispetto alla dimensione orizzontale dell'immagine. Al valore d assegnammo il valore 70 che risulta essere relativamente basso considerando il massimo dato assumibile pari a 2746. In seguito alla compressione ci aspettiamo di ottenere un'immagine visibilmente degradata oltre alla presenza di una porzione di pixel neri dovuti al valore assunto da F .

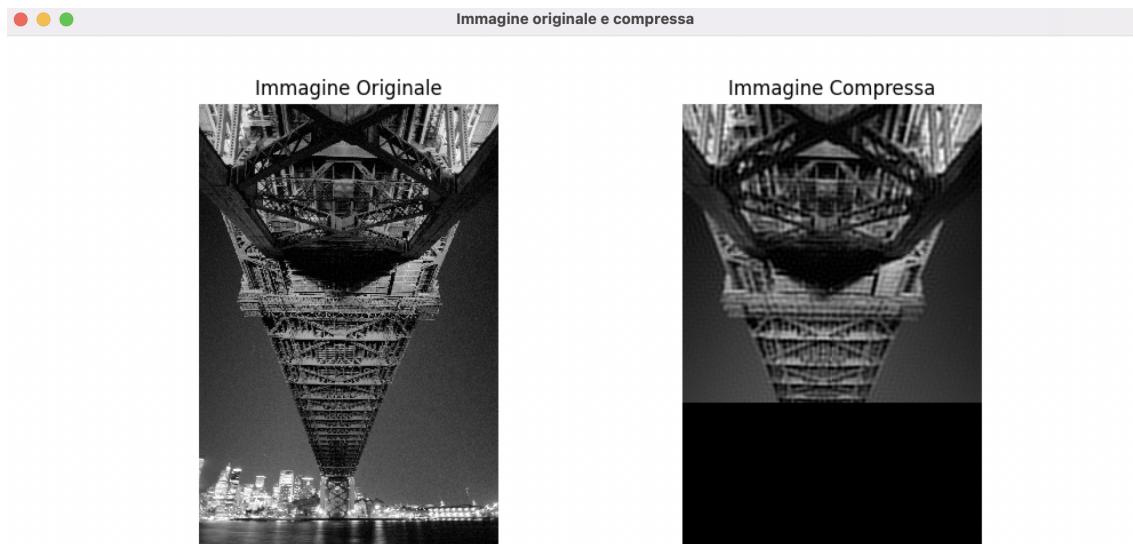


Figura 2.5: Caso 4

A differenza di quanto supposto, l'immagine ottenuta risulta visibilmente degradata, ma non così eccessivamente rispetto a quanto suggerito dal valore assunto da d . Essendo la parte inferiore dell'immagine quella con maggiore valore di contrasto (dovuto ai pixel chiari generati dall'illuminazione degli edifici con il colore scuro dell'acqua) ma non essendo essa elaborata, la restante porzione dell'immagine presa in considerazione risulta avere un minor numero di discontinuità portando, di conseguenza, ad un deterioramento meno marcato rispetto all'immagine originale. Ad occhio nudo è possibile infatti osservare alcune zone maggiormente deteriorate dalla compressione, soprattutto nella parte superiore dell'immagine in cui è presente una zona con un contrasto maggiore.

Caso 5

Analizziamo ora un'immagine ad alto contrasto come, ad esempio, un'immagine raffigurante una scacchiera di dimensioni 640x640 pixel. Prima di eseguire il programma, impostiamo il valore di F pari ad 80, mentre il valore di d pari a 14. A seguito della compressione, otteniamo il seguente risultato:

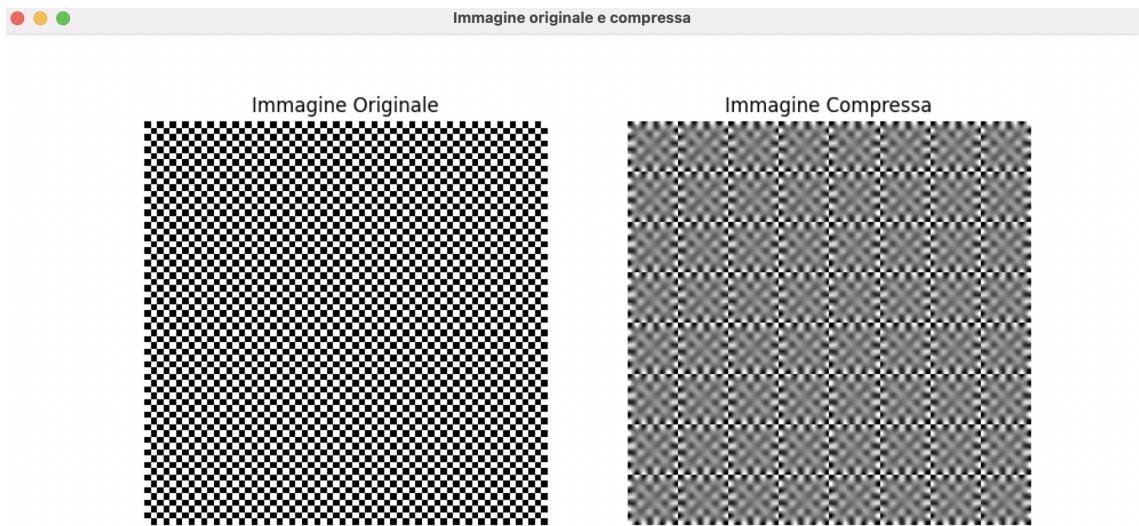


Figura 2.6: Caso 5

L’immagine compressa ottenuta risulta essere altamente degradata, non riuscendo più a distinguere facilmente le porzioni bianche da quelle nere presenti nell’originale. Questo fenomeno si verifica nel caso in cui le immagini sottoposte al processo di compressione presentino un elevato contrasto. Quando i blocchi derivati dalla suddivisione dell’immagine originale vengono sottoposti alla DCT2, questa li trasforma in frequenze, generando una funzione che presenta numerosi punti di discontinuità nelle zone in cui si passa rapidamente dal bianco al nero: la DCT2, in questi casi, produce valori intermedi ed è per questo motivo si nota una forte presenza di grigi nell’immagine ricostruita. È importante sottolineare che i grigi nell’immagine originale erano assenti.

Caso 6

L’ultima prova da noi effettuata è stata eseguita elaborando sempre l’immagine precedente trattata. In questo il valore di F è pari alla dimensione dell’immagine, ovvero 640. Il valore di d , invece, è stato diminuito assegnandogli il valore 5. Di seguito, viene mostrato il risultato ottenuto:

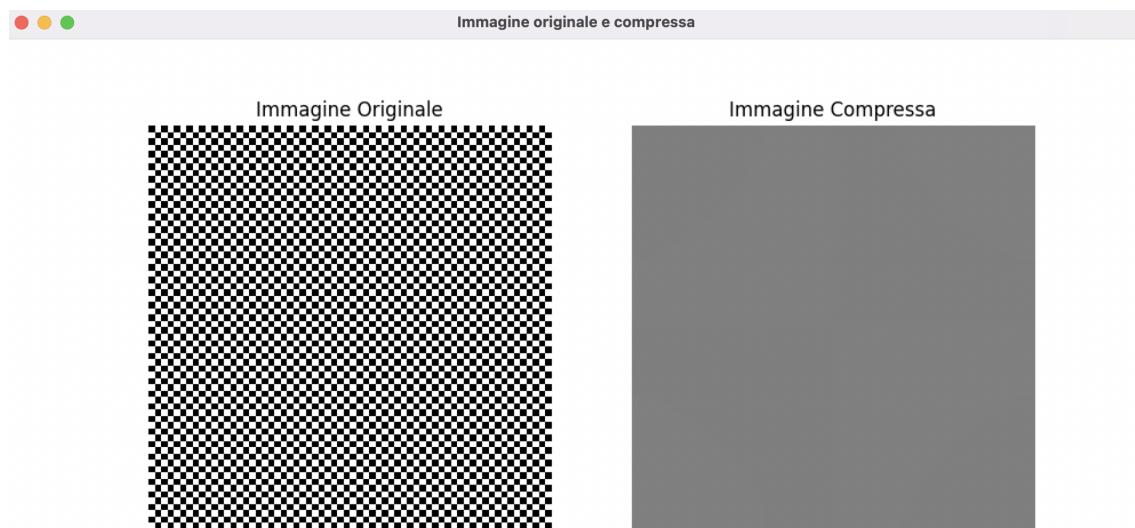


Figura 2.7: Caso 6

L’immagine ottenuta è completamente grigia: rispetto a quanto osservato nella figura 2.3 in cui tutti i pixel risultavano essere di colore nero, in questo caso le frequenze associate ai pixel dell’immagine, avendo un valore del parametro d poco superiore al valore zero, non vengono completamente tagliate rappresentando il colore grigio.

2.4 Conclusioni

In questo progetto abbiamo implementato un algoritmo di compressione basato sulla DCT2 in ambiente Python, studiato gli effetti di questo algoritmo applicato ad immagini in toni di grigio e, in seguito, sono stati condotti una serie di test per valutarne il comportamento.

2.4.1 Considerazioni finali

Abbiamo notato che la scelta dei parametri F e d è critica per il risultato della compressione, e una configurazione errata può portare a una notevole perdita di qualità dell'immagine. Di contro possiamo affermare che, pur diminuendo il valore del parametro d notevolmente, questo non ha portato ad una degradazione eccessiva dell'immagine compressa: di fatto è necessario un basso numero associato al parametro d per ottenere risultati pessimi, e quindi la funzione sembra essere abbastanza robusta in termini di tolleranza dei parametri di compressione. Tuttavia, è importante sottolineare che la scelta dei parametri rimane fondamentale per ottenere risultati di alta qualità nella compressione delle immagini.

In conclusione, il nostro progetto ha raggiunto gli obiettivi stabiliti e ha fornito una panoramica della compressione delle immagini basata sulla DCT2. Tuttavia, ulteriori ricerche e sviluppi sarebbero necessari per rendere questo approccio più pratico ed efficace per applicazioni reali.