|  |  |
| --- | --- |
|  | Francesco Cavallini  Matricola: 920835  f.cavallini8@campus.unimib.it  Corso di studi di Informatica Magistrale |
| Università degli studi Milano Bicocca  Facoltà di informatica magistrale  Milano, Padiglione U24  Anno accademico 23/24 |  |

|  |  |
| --- | --- |
|  | |
| DCT, DCT2 e compressione di immagini  Metodi del Calcolo Scientifico  Relazione Progetto 2  **Consegna:**  **“““**  Lo scopo di questo progetto è di utilizzare l’implementazione della DCT2 in un ambiente open source e di studiare gli effetti di un algoritmo di compressione tipo jpeg (senza utilizzare una matrice di quantizzazione) sulle immagini in toni di grigio.  **”””**  Riferimenti : | |
| Repository git-hub: | <https://github.com/VR3ED/Scientific_Calculus_2> |

Sommario

[0. Introduzione 3](#_Toc1)

[0.1. Obbiettivi 3](#_Toc2)

[0.2. Scelte implementative 3](#_Toc3)

[1. Parte-1: Sviluppo DCT & DCT2 4](#_Toc4)

[1.1. Requisiti tecnici: 4](#_Toc5)

[1.2. Le basi per lo sviluppo: cosa ci aspettiamo: 4](#_Toc6)

[1.3. Le basi per lo sviluppo: come è strutturato il codice: 5](#_Toc7)

[1.4. Utilss.py: Implementazione DCT: 7](#_Toc8)

[1.5. Utilss.py: Implementazione DCT2: 9](#_Toc9)

[1.6. Utilss.py: Metodi di libreria per DCT e DCT2: 11](#_Toc10)

[1.7. Analisi dei risultati e conclusioni parte-1: 12](#_Toc11)

[2. Parte-2: Software con interfaccia grafica 14](#_Toc12)

[2.1. Requisiti tecnici: 14](#_Toc13)

[2.2. Le basi per lo sviluppo: cosa ci aspettiamo: 15](#_Toc14)

[2.3. Le basi per lo sviluppo: come è strutturato il codice: 16](#_Toc15)

[2.4. ProcessImages.py: generate\_blocks: 20](#_Toc16)

[2.5. ProcessImages.py: apply\_dct2: 21](#_Toc17)

[2.6. ProcessImages.py: apply\_idct2: 22](#_Toc18)

[2.7. ProcessImages.py: save\_compressed\_image: 24](#_Toc19)

[2.8. Analisi dei risultati prodotti dall’app: 25](#_Toc20)

[2.9. Problemi riscontrati con l’applicazione: 29](#_Toc21)

[2.10. Conclusioni parte-2: 29](#_Toc22)

# 0. Introduzione

## 0.1. Obbiettivi

Come definito nella pagina iniziale, l’obbiettivo del progetto è quello di creare un implementazione custom della DCT2. In particolare il lavoro è suddiviso in 2 parti in cui abbiamo:

* **Parte 1**: Implementare la DCT2, come spiegata a lezione, in un ambiente open source a vostra scelta e confrontare i tempi di esecuzione con la DCT2 ottenuta usando la libreria dell’ambiente utilizzato (che si presuppone essere nella versione FFT)
* **Parte 2**: Implementare compressione di immagini (nel formato .bmp), sfruttando la DCT2: una metodologia comunemente impiegata per emulare la compressione JPEG. L'obiettivo principale consiste nello sviluppo di un'applicazione che consenta all'utente di selezionare un'immagine e regolare il livello di compressione, offrendo la possibilità di visualizzare sia l'immagine originale che quella compressa.

## 0.2. Scelte implementative

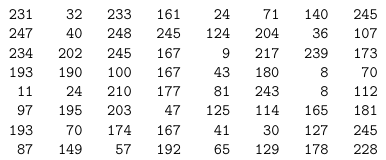
Si sceglie di utilizzare il linguaggio python per lo sviluppo di entrambe le parti del progetto, utilizzando visual studio code come ambiente di sviluppo. Le motivazioni dietro a questa scelta sono:

* Vasta disponibilità di librerie open source per i paragoni necessari
* Vasta gamma di librerie per lo sviluppo di grafici
* Sintassi semplice ed intuitiva, renderà le fasi di analisi del codice più scorrevoli, e, allo stesso tempo, diversamente da C# (il linguaggio utilizzato per lo sviluppo del progetto 1) questo linguaggio è più adatto alla stesura di codice in ambito di calcolo scientifico.

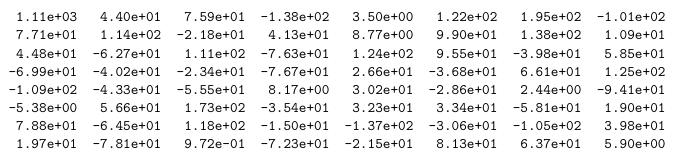
# 1. Parte-1: Sviluppo DCT & DCT2

## 1.1. Requisiti tecnici:

I requisiti necessari per lo sviluppo della prima parte sono quelli di prestare molta attenzione a come viene scalata la DCT2 (o la DCT). Come caso test è necessario verificare che il seguente blocchetto 8×8:



venga trasformato in questo modo dalla DCT2:



È inoltre necessario controllare per la DCT monodimensionale che la prima riga del blocchetto 8×8 precedentemente mostrato venga trasformata in:



## 1.2. Le basi per lo sviluppo: cosa ci aspettiamo:

Per questo progetto, analizzeremo le prestazioni della nostra implementazione della DCT2 rispetto a quella fornita da una libreria di riferimento, concentrandoci sui tempi di esecuzione e sulle complessità teoriche dei due approcci. In particolare, studieremo come i tempi di calcolo variano al crescere delle dimensioni della matrice fornita in input.

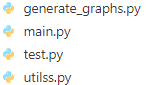
Ci si aspetta che la nostra implementazione manuale della DCT2 dovrebbe presentare una complessità temporale proporzionale a , mentre la versione basata su una libreria ottimizzata, tipicamente costruita utilizzando FFT, dovrebbe raggiungere una complessità . Questa differenza nasce dalle diverse strategie algoritmiche impiegate:

* la nostra implementazione si basa su una definizione diretta della DCT2,
* le librerie ottimizzate sfruttano algoritmi avanzati per ridurre il numero di operazioni necessarie.

Nel corso dell'analisi, confronteremo i tempi di esecuzione ottenuti per matrici di dimensioni crescenti, valutando in che misura i risultati sperimentali confermano le complessità teoriche attese.

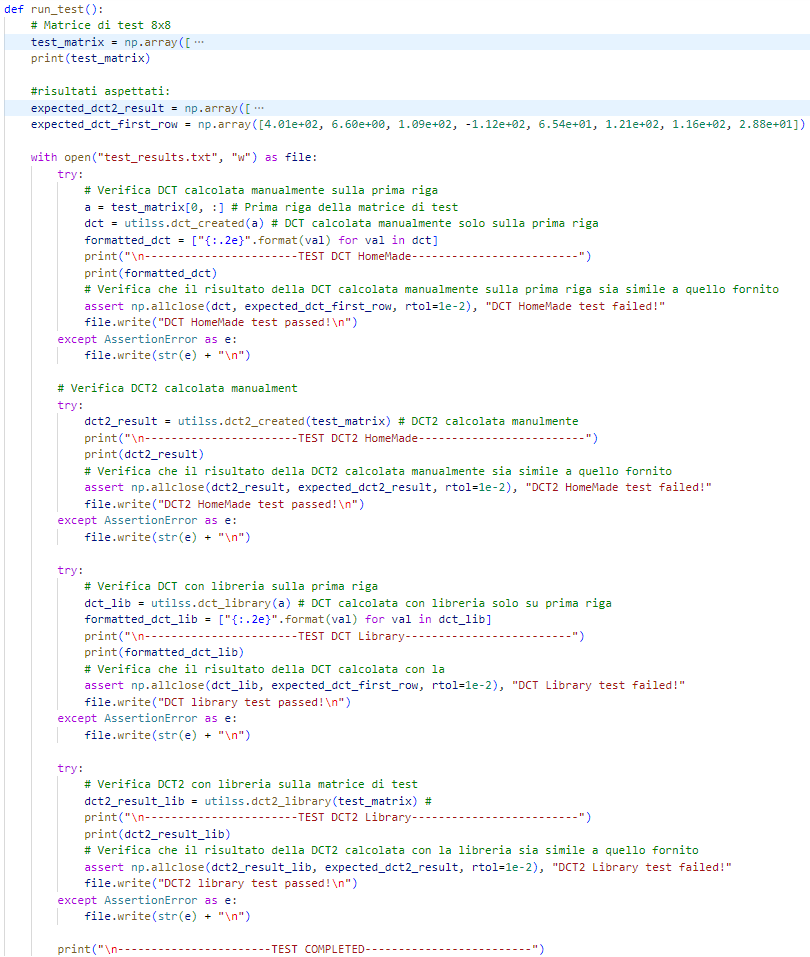
## 1.3. Le basi per lo sviluppo: come è strutturato il codice:

La parte 1 è stata sviluppata suddividendo il codice nei seguenti file:



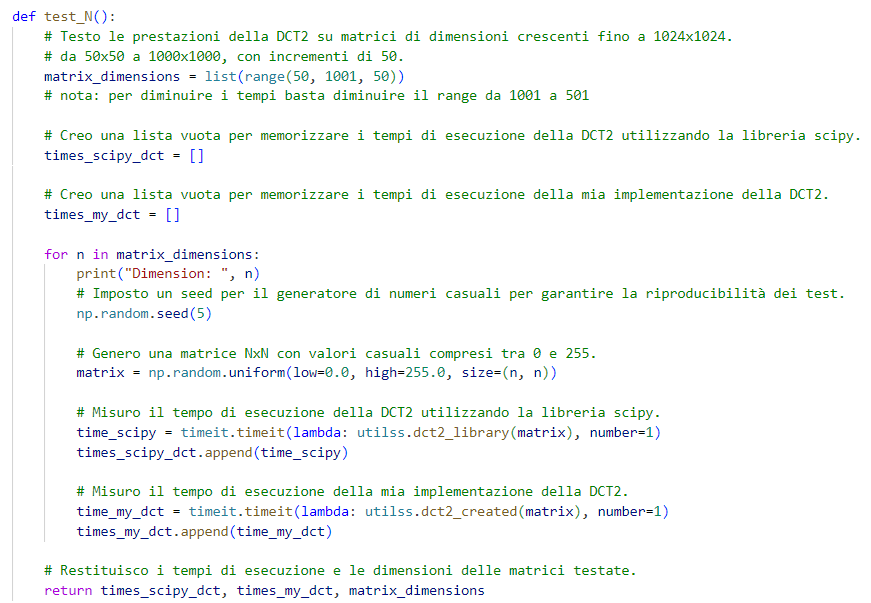
Dove abbiamo che:

test.py: Implementa i requisiti di controllo sull’accuratezza precedentemente accennati:



I commenti nel codice mostrato dovrebbero fornire tutte le informazioni necessarie per individuare tutti i test che vengono svolti per rispettare i requisiti tecnici imposti dalla consegna. Inoltre si può visualizzare che ogni test è inserito all’interno di un try-catch se il test si concluderà con successo allora verrà scritto in un file txt che il test è stato completato correttamente, altrimenti, verrà scritto il messaggio di errore.

Sempre nel file tests.py viene poi implementato un altro metodo per misurare il tempo necessario per eseguire computazioni di DCT e DCT2 su matrici di grandezze progressivamente sempre più grandi (da ad ):



Anche qui abbiamo che i commenti forniscono tutte le informazioni necessarie per comprendere il codice: tutto quello che succede è l’esecuzione ricorsiva di metodi di DCT2 (creato a mano e di libreria) cambiando ad ogni iterazione la grandezza di . Ad ogni iterazione viene anche misurato il tempo di esecuzione di modo da poterlo seguentemente mostrare in formato di grafico.

main.py: Avvia i test definiti nel file test.py

generate\_graphs.py: Contiene una sola funzione che colleziona i time-stamps di esecuzione forniti dai metodi in test.py e plotta un grafico per confrontare le prestazioni temporali della Trasformata Discreta del Coseno (DCT) e della Trasformata Discreta del Coseno bidimensionale (DCT2) tra due implementazioni: una fornita dalla libreria SciPy e una sviluppata manualmente. Si noti che di questo file non verrà mostrato il codice in quanto non rilevante ai fini della relazione, ma verranno mostrati i grafici da essa prodotti.

utilss.py: Cuore del progetto, contiene infatti i due metodi creati manualmente per l’implementazione di DCT e DCT2. Per ciascuna di queste verrà dedicato un capitolo a parte che ne spiega l’implementazione. Questo file contiene inoltre altri 2 metodi wrapper che servono a chiamare la versione da libreria (Scipy) delle implementazioni di DCT e DCT2 con FFT.

## 1.4. Utilss.py: Implementazione DCT:

L’idea di base per la realizzazione di DCT è creare una trasformata matematica utilizzata per rappresentare un segnale discreto (dal dominio del tempo) di lunghezza uni come una somma di funzioni coseno (conversione al dominio della frequenza) con pesi opportuni. Si può usare poi questo principio per scartare le parti con minor quantitativo informativo (troncamento di frequenze) per creare compressioni di segnali ed immagini. Infatti DCT è un’operazione reversibile tramite l’operazione IDCT, tuttavia, la ricostruzione non è perfetta caso in cui ci sia un troncamento delle frequenze (come vedremo nel progetto 2).

Più nello specifico abbiamo che la DCT funziona secondo questo principio:

1. **Calcolo vettore :** Il calcolo di α è semplicemente una fase di normalizzazione necessaria per costruire la matrice di trasformazione della DCT seguendo la definizione teorica. Questo vettore contiene infatti i fattori di normalizzazione che dipendono dalla posizione (indice ) nella matrice di trasformazione:
   * Per : abbiamo che
   * Per : abbiamo che
2. **Calcolo DCT tramite matrice di trasformazione (T)**: Abbiamo che il vettore DCT () utilizza funzioni coseno con frequenze diverse per rappresentare il segnale, ossia abbiamo per ogni elemento ( del vettore risultante che:

Dove rappresenta il campione del segnale originale (nel dominio spazio/tempo)

Per semplicità implementativa possiamo però dividere questa trasformazione in:

1. **Calcolare la matrice di trasformazione T**: Ossia una matrice unicamente composta da vettori riga () che rappresentano funzioni coseno a diverse frequenze; ogni posizione singola di questa matrice è calcolata in questo modo:

Nota che la parte della formula applica la trasformazione cosinuoidale, utilizzando il vettore assicuriamo che la trasformazione sia ortogonale; ossia:

* + - I vettori riga saranno mutuamente ortogonali: Il prodotto scalare tra due vettori riga e della matrice deve essere zero per . Questo garantisce che le frequenze diverse (associate a e ) siano indipendenti l'una dall'altra.
    - I vettori riga saranno normalizzati: Il prodotto scalare di ciascun vettore con sé stesso deve essere

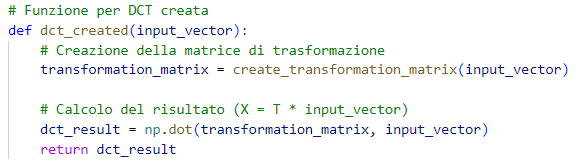
1. **Applicare la trasformazione al segnale originale**: ossia moltiplicare il segnale originale per il vettore di trasformazione:

Dove contiene i coefficienti della DCT, che rappresentano le ampiezze delle diverse frequenze.

Se andiamo a visualizzare quindi l’implementazione del metodo di creazione manuale della DCT all’interno del file utilss.py possiamo osservare che verranno applicati gli stessi principi teorici appena descritti:

Si decide di esportare il calcolo di e di in un altro metodo a parte

Applica la matrice di trasformazione al segnale originale:



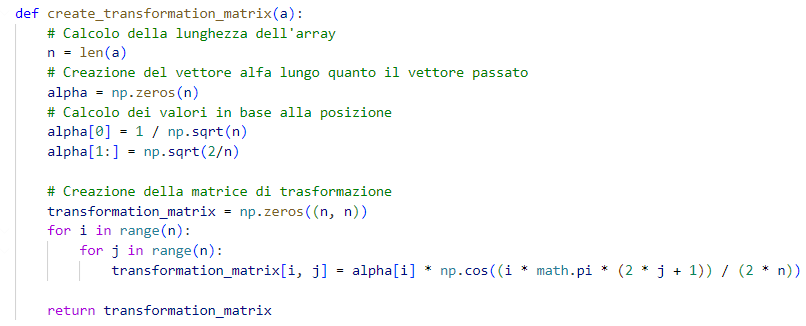
Come possibile osservare il calcolo di e di viene spostato in un metodo diverso (questo semplicemente per migliorare la leggibilità del codice), questo metodo è il seguente:

Calcolo matrice T cella per cella:

Calcolo di con:

•

• (per )



Svolgendo i calcoli nell’ordine di esecuzione abbiamo quindi:

1. Calcolo di
2. Calcolo di
3. Applicazione di al segnale originale

Questa serie di operazioni corrisponde quindi perfettamente con la teoria precedentemente descritta.

## 1.5. Utilss.py: Implementazione DCT2:

La DCT2 bi-dimensionale (2D) è un'estensione della DCT al dominio delle matrici. Il principio fondamentale della DCT2 è che concentra la maggior parte dell'informazione utile nei coefficienti a bassa frequenza (tipicamente in prossimità dell'angolo in alto a sinistra della matrice trasformata), rendendo possibile una significativa riduzione della dimensionalità del dato e una compressione efficace qualora si decidesse di effettuare un troncamento delle frequenze.

Si ha che ogni cella () della matrice risultante dalla trasformazione DCT2 viene calcolata applicando separatamente DCT lungo le righe e le colone alla matrice originale, ossia:

Applicazione DCT sulle righe

Applicazione DCT sulle colonne

Dove:

* è il valore dell'elemento della matrice di input (prima della trasformazione DCT2)
* e sono le dimensioni della matrice di input.
* Insieme a sono i 2 fattori di normalizzazione per le rispettive DCT.

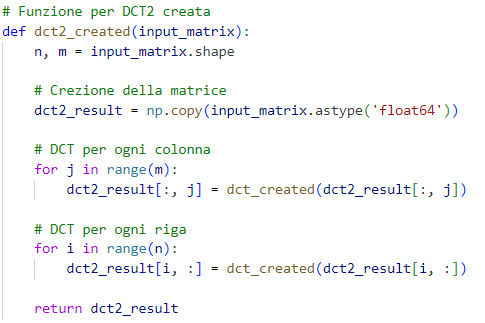
Essendo questa trasformazione direttamente derivata da DCT è facile capire che anche in questo caso la trasformazione è reversibile (mantenendo sempre il problema di ricostruzione perfetta a causa di un troncamento delle frequenze che potrebbe avvenire durante la trasformazione DCT2).

È fondamentale evidenziare che la tecnica appena descritta di applicazione DCT2 e troncamento frequenze, seppur molto simile, non corrisponde perfettamente a quella impiegata dal formato JPEG per comprimere le immagini. Questo perché alcune operazioni critiche potrebbero generare distorsioni e artefatti nella ricostruzione finale. Il formato JPEG, invece, utilizza strategie specifiche per superare tali problematiche, garantendo una compressione efficiente senza compromettere eccessivamente la qualità dell'immagine.

**NOTA**: DCT2 nel formato JPEG

Avendo quindi già spiegato il funzionamento della funzione DCT nel capitolo precedente diventa molto facile spiegare il codice generato per l’implementazione di DCT2.

Se andiamo quindi a visualizzare l’implementazione del metodo di creazione manuale della DCT2 (sempre all’interno del file utilss.py) possiamo osservare che verranno applicati gli stessi principi teorici appena descritti:



La DCT2 implementata nel codice antecedente segue una strategia "a separazione di variabili". Ossia viene sfruttata la linearità della trasformazione consentendo di ridurre il problema bidimensionale in **due trasformazioni monodimensionali** consecutive:

1. Prima si calcola la DCT lungo una dimensione (colonne).
2. Poi si calcola la DCT lungo l'altra dimensione (righe).

Questo approccio è computazionalmente efficiente perché riduce la complessità computazionale rispetto al calcolo diretto della DCT2, specialmente per matrici di grandi dimensioni. Abbiamo infatti che la definizione diretta della DCT2 per una matrice è:

Questo implica di avere due somme nidificate () su tutte le celle della matrice. Dove ogni somma richiede moltiplicazioni e somme per calcolare un singolo elemento ​. Per calcolare tutti i coefficienti della DCT2, la complessità totale diventa:

Invece, utilizzando il principio della scomposizione di due operazioni lineari abbiamo che:

* Per la trasformazione lungo le colonne: per una colonna sola richiede operazioni, quindi per tutte le colonne:
* Per la trasformazione lungo le righe: analogamente per una riga sola richiede operazioni, quindi per tutte le colonne:

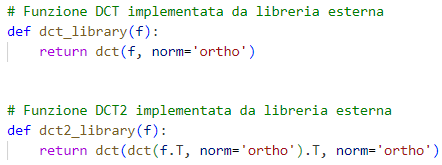
Avendo quindi che la complessità totale utilizzando questo metodo è:  
  
Avendo quindi che questa strategia è computazionalmente più efficiente (e rispecchia anche la complessità di implementazione che ci aspettavamo all’inizio)

**NOTA**: matrici quadrate

Nota che per entrambi i risultati finali se operiamo con una matrice quadrata allora i “” diventano “”

## 1.6. Utilss.py: Metodi di libreria per DCT e DCT2:

Come già anticipato si usano i metodi della libreria SciPy per avere un paragone con altre implementazioni di DCT e DCT2, di seguito vengono mostrati i metodi wrapper che richiamano le funzioni di libreria:



La funzione DCT2 prende in input una matrice, su questa matrice viene richiamata due volte la funzione DCT. Si noti che si da in pasto al metodo DCT2 la matrice trasposta 2 volte appunto per creare un analogia con il codice scritto manualmente: qui, la trasposizione fa sì che la prima trasformazione venga applicata lungo le righe (tramite la prima trasposta ), e la seconda lungo le colonne (invertendo con un altro ). Se non si usasse la trasposizione T nella funzione dct2\_library, la trasformazione sarebbe applicata solo lungo una dimensione della matrice (ad esempio, solo lungo le righe o solo lungo le colonne), anziché su entrambe le dimensioni.

**NOTA**: Coefficienti di normalizzazione

Si vuole far notare però che questa libreria implementa il fattore di scaling α in maniera diversa da come lo abbiamo appena visto dalle descrizioni teoriche che abbiamo dato nelle sezione [1.4](#_1.4._Utils) (ossia come abbiamo visto l’implementazione teorica della DCT).

Abbiamo infatti visto durante le lezioni teoriche che la formula per la DCT sarebbe:  
  
dove: (con vettore base della DCT)

* se allora (che possiamo trasformare quindi in moltiplicazione per )
* se allora (che possiamo trasformare quindi in moltiplicazione per )

A questi si sceglie di aggiungere la radice quadrata per fare normalizzazione, senza normalizzazione il prodotto scalare di un vettore base con sé stesso sarebbe proporzionale a N. La radice quadrata viene utilizzata per compensare questa dipendenza.

Ma se andiamo ad esplorare la [documentazione di SciPyci](https://docs.scipy.org/doc/scipy/reference/generated/scipy.fftpack.dct.html) (sotto dct-type2 e norm=ortho) possiamo vedere che questa implementa il calcolo dei coefficienti di normalizzazione in maniera leggermente diversa, ma molto simile, a quella che abbiamo deciso di applicare:

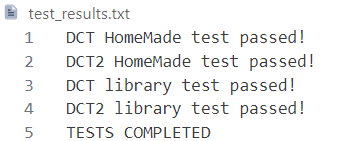
* se allora (trasformabile quindi in moltiplicazione per )
* se allora (trasformabile quindi in moltiplicazione per )

## 1.7. Analisi dei risultati e conclusioni parte-1:

Una volta definiti tutti i metodi utilizzati per l’implementazione di DCT2 possiamo dunque eseguirli.

**1.7.1. Esecuzione test per i requisiti tecnici forniti:**

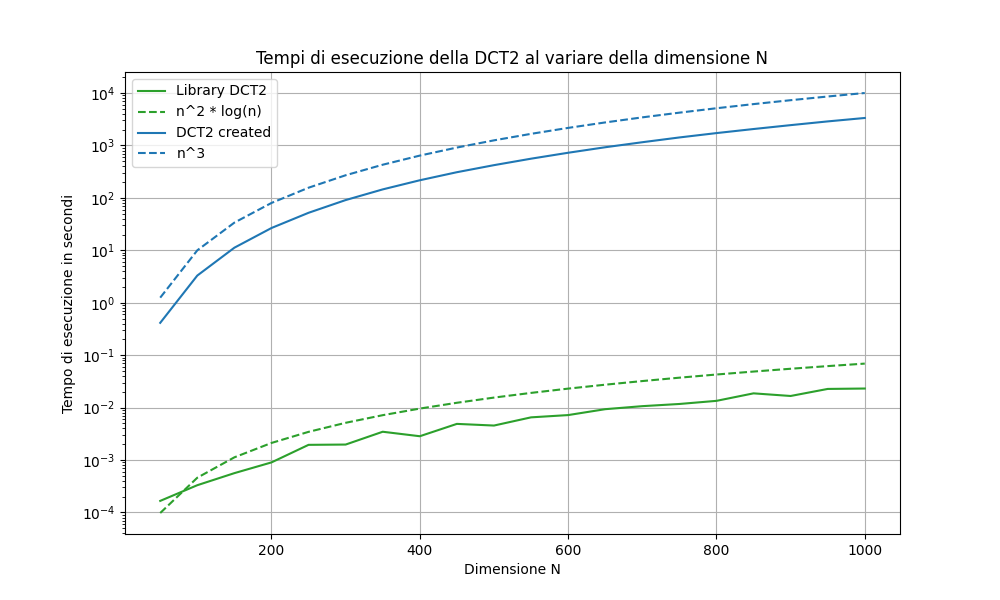
Abbiamo precedentemente mostrato nella sezione [1.3](#_1.3._Le_ba) la funzione run\_test() (all’interno del file test.py). Questa funzione è incaricata di controllare il rispetto dei requisiti tecnici forniti dalla consegna. Abbiamo infatti che per ogni test eseguito viene scritta una riga di codice all’interno del file “test\_results.txt”, di seguito ne vengono mostrati i risultati:



Come possibile visualizzare tutti i test eseguiti vengono eseguiti correttamente (nonostante la piccola differenza di fattore di normalizzazione precedentemente notata). Di conseguenza possiamo affermare con certezza che i risultati prodotti rispettino i requisiti di accuratezza richiesti da consegna.

**1.7.2. Esecuzione test per verificare i tempi di esecuzione:**

Sempre nel capitolo [1.3](#_1.3._Le_ba) abbiamo precedentemente mostrato il funzionamento della funzione test\_N che permette di misurare il tempo necessario per eseguire computazioni di DCT2 (sia la versione scritta manualmente che quella di libreria) su matrici di grandezze progressivamente più grandi (da ad ). Avevamo anche accennato poi alla presenza di un metodo presente nel file generate\_graphs.py che permette di prendere i risultati dei tempi di computazione calcolati con il metodo precedente e mostrarli in un grafico. In seguito all’esecuzione di entrambi i metodi questi sono i risultati riportati:



Analizzando i risultati riportati, si osserva che il tempo di esecuzione delle implementazioni della DCT2 segue, segue un andamento molto simile a quello di ; in maniera allineata a ciò che avevamo previsto nei capitoli precedenti. In particolare, l’utilizzo del metodo di libreria consente un notevole miglioramento delle prestazioni per le matrici nella DCT2. Infatti, come anticipato, questo metodo segue un andamento di .

Possiamo quindi concludere che per matrici di piccole dimensioni (), i tempi di esecuzione delle due implementazioni sono simili: essendo le y in scala logaritmica abbiamo che per entrambi i metodi è necessario al massimo di un secondo (circa) per calcolare entrambi i metodi. Ovviamente però diventa necessario utilizzare il metodo di libreria per calcolare matrici di dimensioni in quanto la differenza tra i due metodi inizia a diventare molto più accennata.

Siamo dunque soddisfatti dei risultati ottenuti sia sui test precedenti che su quelli misurazione delle tempistiche di esecuzione, in quanto, in entrambi i casi siamo riusciti ad arrivare alle nostre aspettative iniziali.

# 2. Parte-2: Software con interfaccia grafica

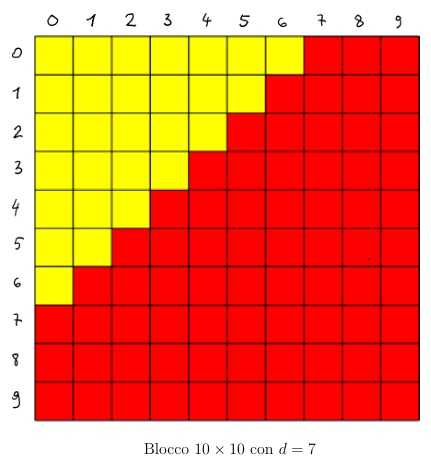
## 2.1. Requisiti tecnici:

**2.1.1: Come deve funzionare l’applicazione:**

Creare un software con interfaccia grafica che permetta di suddividere l’immagine in blocchi quadrati di pixel di dimensioni partendo in alto a sinistra, scartando gli avanzi; Ossia, per ogni blocco eseguire le seguenti operazioni:

* applicare la DCT2 (della libreria): ;
* eliminare le frequenze con : assumendo che le frequenze partano da 0: se le elimino tutte, se elimino solo la più alta, cioè quella con:

In sostanza bisogna eliminare i coefficienti in frequenza a destra della diagonale individuata dall’intero d, come esemplificato qui sotto:



Qui abbiamo e . I coefficienti da eliminare sono indicati in rosso.

* applicare la DCT2 inversa al blocchetto così modificato:
* arrotondare all’intero più vicino, mettere a zero i valori negativi a e si troncano i valori maggiori di 255 (limitandoli a 255) in modo da avere dei valori ammissibili (1 byte);

Una volta fatto questo per ogni blocco si rimette insieme l’immagine mettendo assieme i blocchi ottenuti, mettendoli nell’ordine giusto e paragonando l’immagine ottenuta con l’originale di partenza.

**2.1.2: Requisiti sui parametri in input:**

Una parte fondamentale per lo sviluppo di questa applicazione è la gestione dell’input dell’utente. Più nello specifico è fondamentale garantire che:

* L’utente possa prendere in input solo immagini in formato .bmp
* Il parametro sia un numero intero tale che “Base/Altezza\_immagine caricata”
* Il parametro sia un numero intero tale che

## 2.2. Le basi per lo sviluppo: cosa ci aspettiamo:

**2.2.1: Aspettative a livello qualitativo dell’immagine:**

I parametri e sono fondamentali per fare delle previsioni sui risultati di compressione della nostra applicazione, abbiamo infatti che:

* Per : all’aumentare il valore di F più è possibile che l’immagine venga distorta, questo perché (indicando F la dimensione dei blocchetti dell’immagine che verranno compressi), potenzialmente, se si sceglie una regione molto grande e si eliminano molti coefficienti in relazione alla dimensione si rischia di rovinare la visibilità dell’immagine e di creare grandi artifact di diverso tipo:
  + Artifact di forma quadrata all’interno dell’immagine compressa risultante (che rappresentano i bordi della finestra ), oppure anche creare altri disturbi in cui compare una serie di “discontinuità” nell’immagine ai bordi delle finestre .
  + Fenomeno di Gibbs: si manifesta quando una funzione discontinua viene approssimata mediante una serie di Fourier troncata, producendo oscillazioni vicino ai punti di discontinuità. Questo fenomeno è visibile come artefatti lungo i bordi “netti” delle immagini.
* Per : si può facilmente intuire che questo valore è direttamente proporzionale alla qualità dell’immagine risultante. Al diminuire del valore di più ci si può aspettare che la qualità dell’immagine peggiori, in quanto:
  + utilizzando un alto valore di ci si aspetta di mantenere la maggior parte dei coefficienti di tutti i blocchetti , il che vuol dire che la compressione rimuove solo una piccola parte di informazione dell’immagine. (risultato di alta qualità)
  + Utilizzando un basso valore di ci si aspetta, invece, di eliminare la maggior parte dei coefficienti presenti nel blocchetto portando ad avere un’immagine molto compressa, ma anche con una qualità visiva peggiore (in quanto una gran quantità del valore informativo dell’immagine è stato rimosso)

La chiave per una buona compressione sarà dunque trovare un giusto compromesso per il valore di soglia , in modo da non ridurre eccessivamente la qualità dell’immagine, riuscendo contemporaneamente a utilizzare meno memoria.

**2.2.2: Aspettative a livello di tempi di computazione:**

Come abbiamo precedentemente mostrato la procedure di applicazione della DCT2 (e di conseguenza anche quella di IDCT2) è un operazione . Se consideriamo come la dimensione di un’immagine (che solitamente è dell’ordine del migliaia di pixel), avremo che il nostro metodo di compressione “scala male”, ossia potrebbe arrivare ad avere tempi di computazione molto alti. Anche qualora utilizzassimo la funzione di libreria che fa’ utilizzo di FFT (cosa che faremo), il costo rimane elevato: .

Possiamo dunque aspettarci che i tempi di computazione di immagini molto grandi fornite al nostro programma possano diventare altrettanto molto alti.

**NOTA TEORICA**: La compressione Jpeg

Abbiamo appena elencato tutta una serie di problemi che ci possiamo aspettare di avere quando implementiamo il nostro algoritmo di compressione delle immagini. Abbiamo però che, nonostante il metodo di compressione Jpeg sia molto simile a quello descritto nella sezione di [requisiti tecnici](#_2.1._Requi), si ha che l’algoritmo di compressione Jpeg riesce comunque a migliorare la compressione delle immagini applicando delle migliorie ad alcuni step:

* Per risolvere il problema di avere computazioni che richiedono alto tempo di computazione usa finestre di dimensione fissa , la loro piccola dimensione le rende più facili da calcolare, questo aiuta anche a contrastare il fenomeno di Gibbs.
* Per mitigare la creazione di discontinuità lungo i bordi delle finestre Jpeg usa finestre con sovrapposizione. Ossia alcuni pixel contenuti nella finestra precedente sono contenuti anche nella finestra successiva. In questo modo, dopo che si è fatta l’elaborazione di entrambi i blocchi si possono sovrapporre.

## 2.3. Le basi per lo sviluppo: come è strutturato il codice:

Il codice di quest’applicazione è strutturato in 2 file:

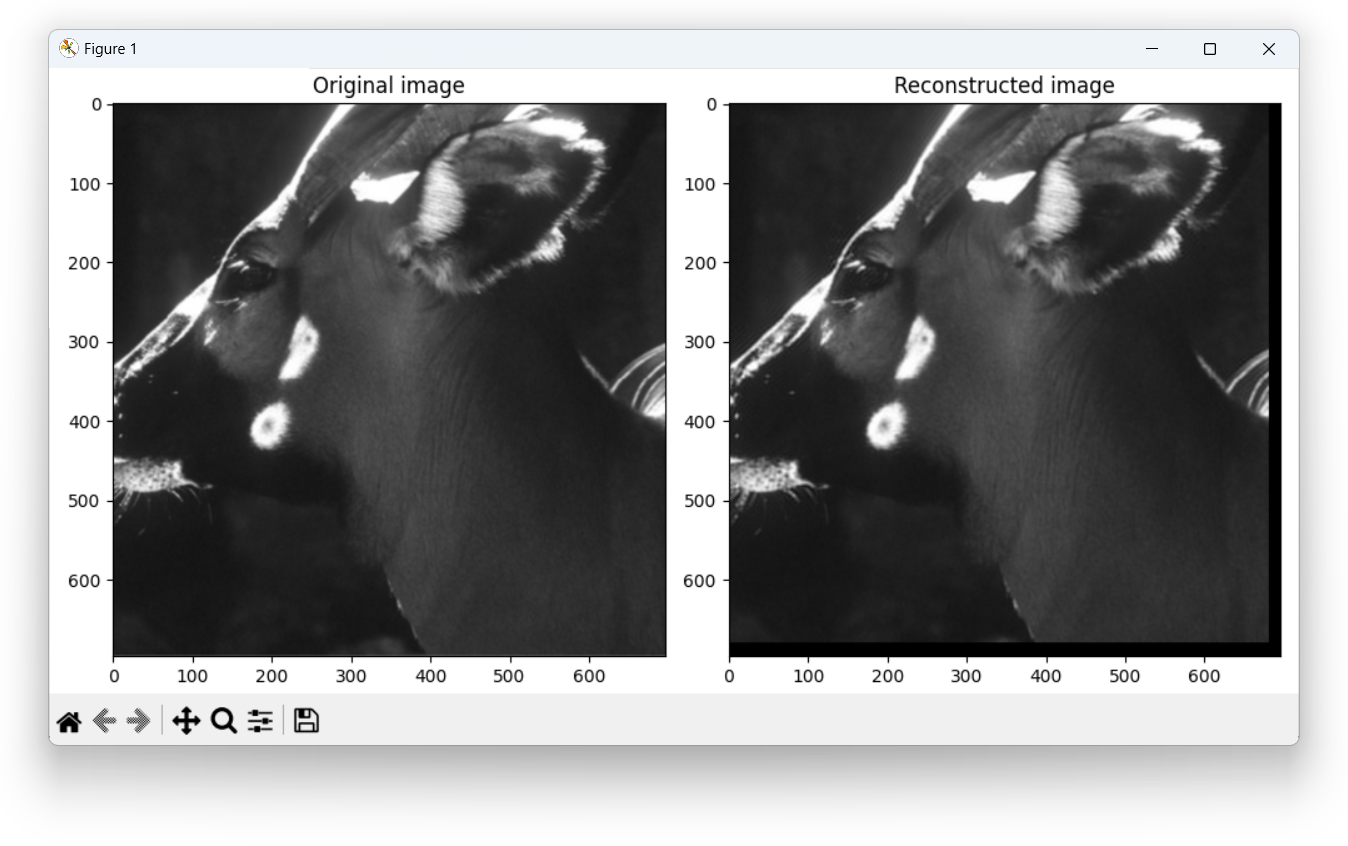
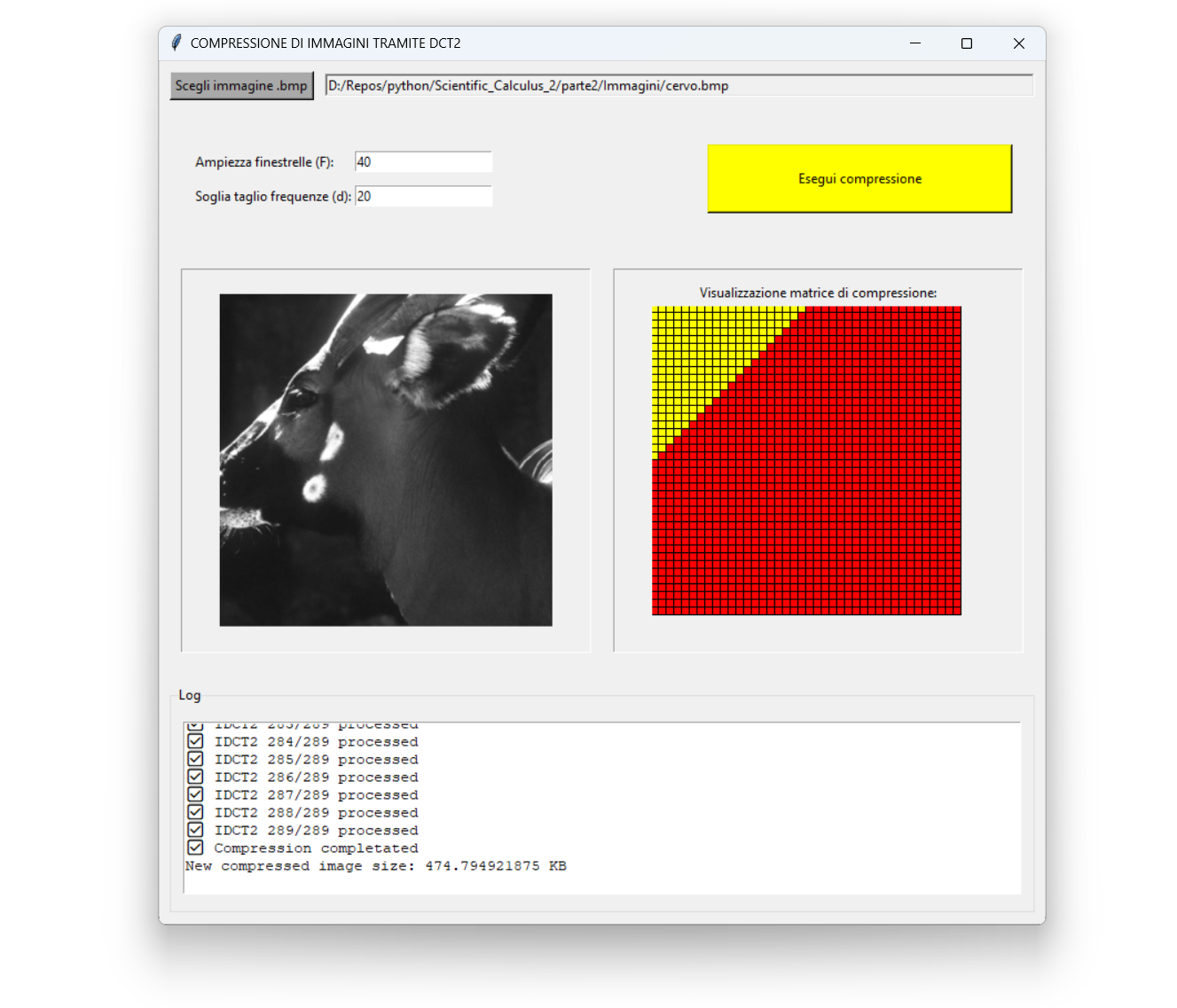
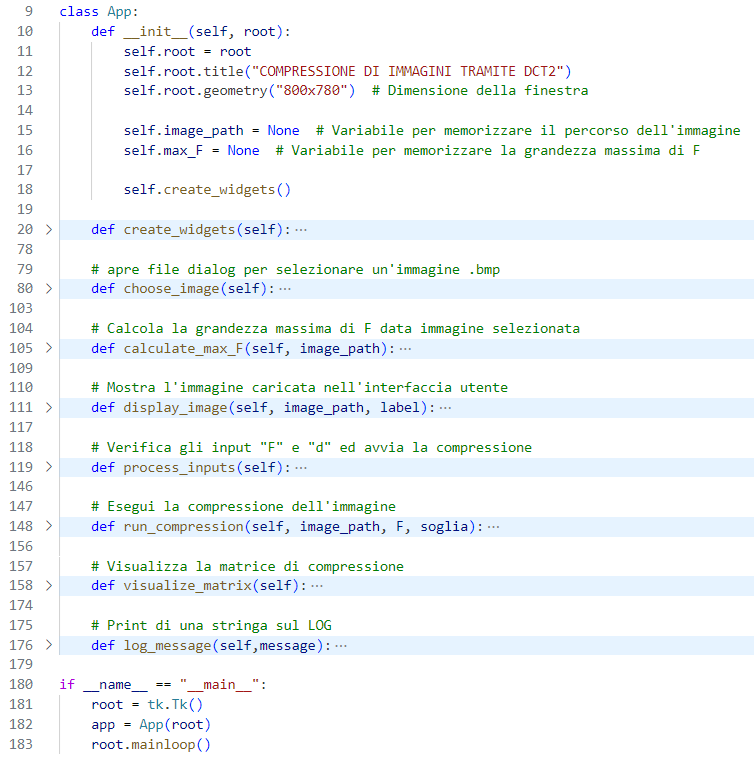


Dove abbiamo che:

MainUI.py: Classe che contiene le funzioni di GUI (frontend) per gestire la grafica del programma ed il loro collegamento con il backend. In particolare l’interfaccia è stata sviluppata utilizzando la libreria tkinter, che offre una gamma di widjet predefiniti come finestre, bottoni, caselle di testo, etichette e altro, permettendo di creare e personalizzare l’aspetto dell’applicazione. Più nello specifico abbiamo che in questo file è presente una classe “App” al quale al suo interno è descritto il metodo CreateWidjets dedicato alla creazione della GUI dell’applicazione.

Abbiamo poi che sempre all’interno di questo file è presente il metodo main. All’avvio dell’applicazione questo metodo crea un istanza della classe App, il quale costruttore avvia il metodo CreateWidjets permettendo così di visualizzare la grafica.

A pagina seguente è presentata una breve overview del codice appena descritto e la corrispondente grafica generata tramite applicazione:



Nota che nel file MainUI.py abbiamo anche i controlli necessari per verificare le corrette dimensioni di ed . Abbiamo infatti che, al momento di caricamento di un immagine col il metodo choose\_image (oltre all’avvio del metodo display\_image necessario per mostrare nella UI l’immagine appena selezionata da file explorer) viene avviato anche il metodo calculate\_max\_F; questo metodo serve per salvare la dimensione minima tra la base e l’altezza dell’immagine appena caricata (che corrisponde al valore massimo che potrà assumere il valore ).

Abbiamo poi che quando viene premuto il bottone “Esegui compressione” viene avviato il metodo process\_inputs descritto di seguito:

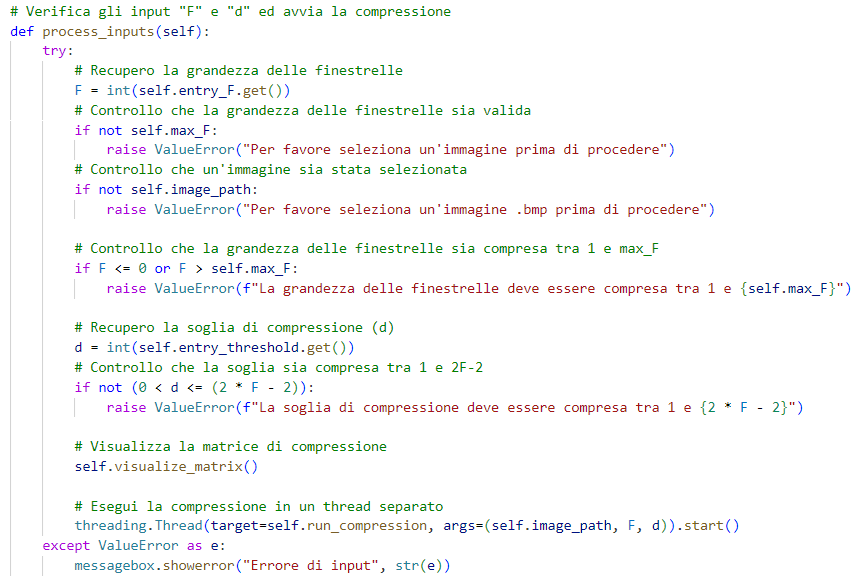
5

4

2

3

1



Abbiamo dunque che nel codice appena mostrato vengono fatti rispettare i vincoli imposti sui parametri (segnalati al capitolo [2.1.2](#_2.1._Requi)):

1. Si verifica che sia stata caricata un’immagine a priori.
2. Si verifica che il parametro sia un numero intero tale che:
3. Si verifica che il parametro sia un numero intero tale che:

Solo dopo aver fatto rispettare i vincoli sui parametri appena descritti allora:

1. Viene fatta visualizzare la matrice di compressione dinamicamente (le dimensioni e la diagonale selezionata cambiano sempre in base ai parametri di e inseriti) grazie al metodo visualize\_matrix.
2. Avvia il metodo run\_compression in un thread separato (in questo modo non viene bloccata la grafica dell’applicazione), questo servirà finalmente ad avviare la compressione dell'immagine.

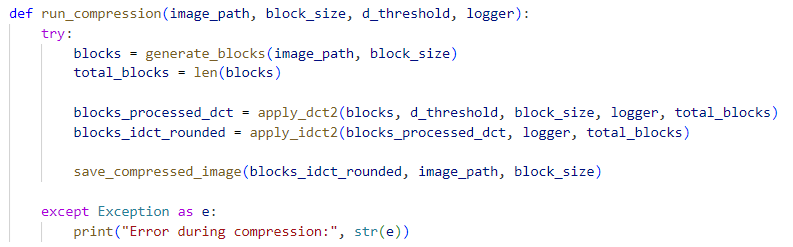
ProcessImages.py: All’interno di questo file sono presenti tutti i metodi che permettono di realizzare la compressione dell’immagine. Abbiamo infatti appena visualizzato che il primo metodo avviato per iniziare il calcolo della compressione è run\_compression, la definizione di questo metodo è infatti presente in questo file e, come vedremo di seguito, questo permette di applicare la compressione esattamente come descritta al capitolo [2.1.1](#_2.1._Requi):

2

3

4

1



Abbiamo infatti che:

1. Tramite il metodo generate\_blocks si suddivide l’immagine in blocchi quadrati () di dimensione
2. Tramite il metodo apply\_dct2 si applica la dct2 ad ogni blocchetto e si eliminano le frequenze al di sotto della diagonale
3. Tramite il metodo apply\_idct2 si applica la trasformazione inversa dei blocchetti (ottenendo )
4. Infine il metodo save\_compressed\_image rimette insieme i blocchetti nella loro posizione originale (in modo da ricostruire l’immagine), salva l’immagine in memoria e mostra il risultato di compressione affiancato all’immagine originale

Nelle sezioni seguenti approfondiremo il codice dietro ciascuno dei metodi appena citati per dimostrare che la compressione avvenga correttamente.

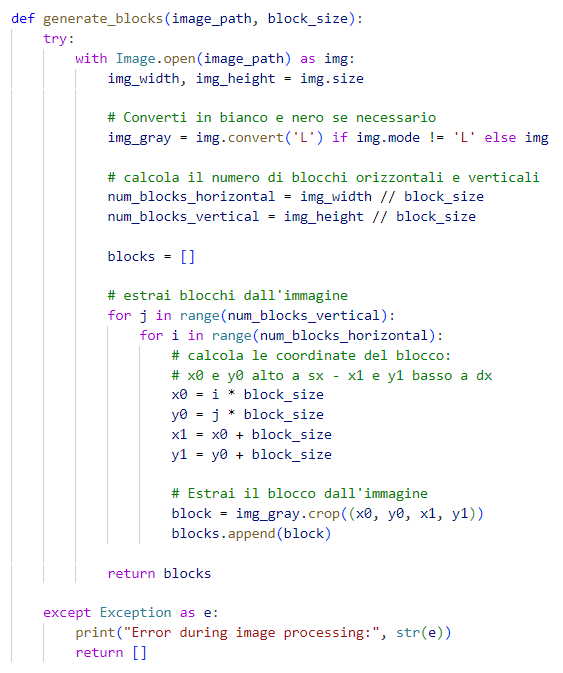
## 2.4. ProcessImages.py: generate\_blocks:

Funzione dedicata alla suddivisione dell’immagine in blocchi quadrati () di dimensione , di seguito se ne riporta il codice:

Calcolo numero di blocchi per ogni riga e per ogni colonna dell’immagine originale

Calcolo delle coordinate di ogni blocco di dimensione da estrarre dall’immagine.

Susseguentemente si preleva (crop) dall’immagine quell’esatto blocchetto . Infine si aggiunge il blocchetto ad un array (che conterrà alla fine del ciclo tutti i blocchi), da ritornare in output



Una volta ritornato la lista di “blocks” riprende l’esecuzione del metodo run\_compression che, come prossima istruzione applicherà la trasformazione DCT2 tramite il metodo apply\_dct

**NOTA**: Crop dell’immagine

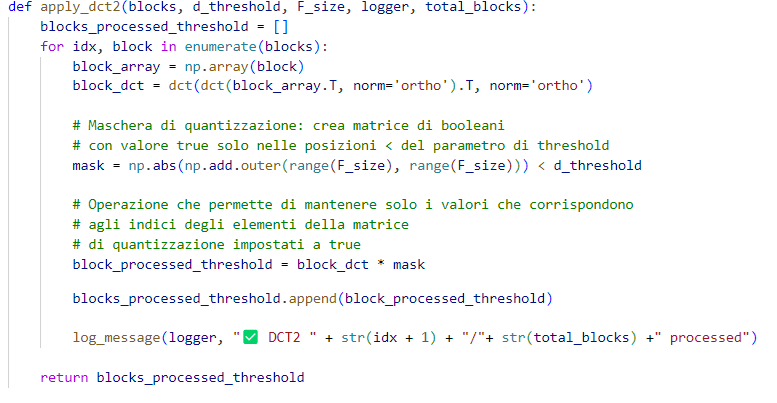
Siccome la divisione che effettuiamo per andare calcolare il numero di blocchi per riga e per colonna dall’immagine è una divisione senza resto ci si può aspettare che se non si seleziona una dimensione di che sia un sottomultiplo del numero di pixel della base/altezza dell’immagine allora questa verrà tagliata alla base/altezza.

## 2.5. ProcessImages.py: apply\_dct2:

In seguito all’aver ottenuto tutti i blocchetti che compongono l’immagine si applica la DCT2 a ciascun blocchetto , inoltre, per ogni blocchetto si elimineranno le frequenze al di sotto della diagonale come riportato di seguito:

2

1

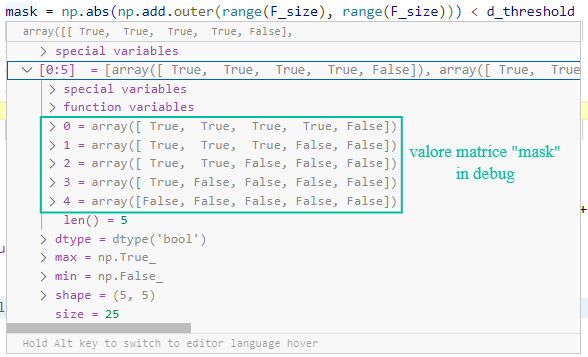
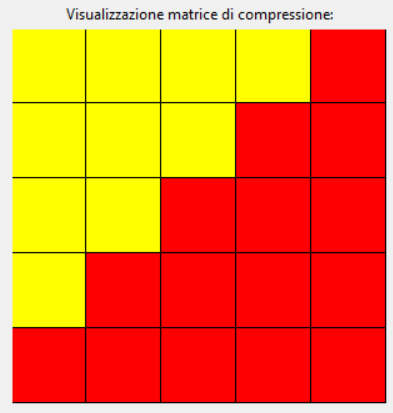


Abbiamo infatti che, nel codice riportato, per ogni blocco ricavato dallo step precedente:

1. si calcola la DCT2 usando la libreria Scipy (esattamente come fatto nel [progetto-1](#_1.6._Utils)). In questo modo otteniamo:
2. Una volta calcolata (nel codice rappresentata dalla variabile “block\_dct”), si troncano le frequenze al di sopra della diagonale , ossia si eliminano le frequenze con , per fare questo si usa la riga di codice:

mask = np.abs(np.add.outer(range(F\_size), range(F\_size))) < d\_threshold

Questa permette di creare una matrice di booleani la quale posizione corrisponde perfettamente con le posizioni con , di seguito un esempio con e :



In python il valore False equivale a 0 ed il valore True equivale ad 1, il che vuol dire che moltiplicare questa matrice “Mask” per il blocchetto DCT2 () otteniamo il troncamento delle frequenze desiderato.

## 2.6. ProcessImages.py: apply\_idct2:

**NOTA**: paginazione della relazione

Prima di passare alla visualizzazione del metodo che implementa IDCT2 se ne da una breve descrizione teorica, in quanto, a differenza del metodo di DCT2, non abbiamo ancora fornito una spiegazione teorica di questo metodo all’interno di questa relazione.

Come abbiamo già accennato IDCT2 è l’operazione inversa di DCT2, questa permette infatti di prendere un segnale 2 dimensionale nel dominio delle frequenze e riconvertirlo al dominio del tempo/spazio. Di conseguenza abbiamo che IDCT (1D) è l’operazione inversa di DCT. Si ha infatti che IDCT viene definita dalla seguente formula:

dove:

* è il coefficiente DCT corrispondente alla frequenza ,
* ​ è lo stesso identico fattore di normalizzazione già usato nella DCT,
* è la funzione che combina le componenti di frequenza per ricostruire il segnale.
* è il valore originale ricostruito nel dominio del tempo/spazio.

Applicare quindi l'IDCT a un segnale trasformato con la DCT restituisce il segnale originale:

Abbiamo dunque che (analogamente a DCT2) IDCT2 bidimensionale è semplicemente l'applicazione sequenziale della IDCT prima sulle colonne e poi sulle righe della matrice dei coefficienti calcolati con DCT2. La formula generale per l'IDCT2 è:

dove:

* è il coefficiente DCT2 corrispondente alla frequenza
* insieme ad sono i 2 fattori di normalizzazione per le rispettive DCT.
* e sono le dimensioni della matrice di input.
* è il valore dell'elemento della matrice di input originale (nel dominio tempo/spazio).

Abbiamo dunque che, analogamente all’uso di IDCT su DCT, anche applicare IDCT2 sui coefficienti calcolati con DCT2 permette di ritornare al segnale originale:

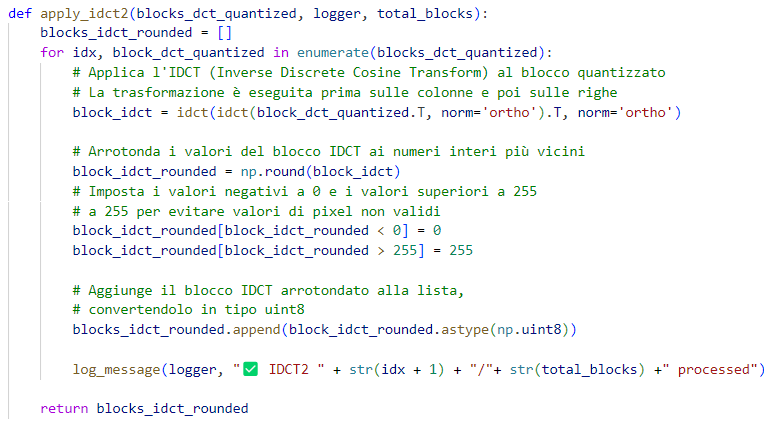
Questo principio verrà utilizzato nel codice che mostreremo a pagina seguente.

Una volta descritti i principi teorici dietro l’applicazione di IDCT2 possiamo continuare con la descrizione del funzionamento del codice dell’applicazione, arrivando ora a descrive il seguente metodo il quale scopo è quello di applicare IDCT2 a tutti i blocchetti precedentemente processati:

1

3

2



Si ha dunque che nel codice appena mostrato succede che per ogni blocchetto :

1. Si applica la IDCT2 inversa ai blocchetti così in modo da ottenere:
2. Si arrotondano tutti i valori di all’intero più vicino e mettendo a zero i valori negativi. Poi si troncano i valori maggiori di 255, limitandoli appunto a 255.
3. Tutti i blocchetti appena processati vengono collezionati in una lista e ritornati in output.

In questo modo, applicando la IDCT2 ed in seguito lo scaling dei valori da 0 a 255, in ogni blocchetto avremo dei valori ammissibili (1 byte) per ricostruire l’immagine. Inoltre, in questo modo vengono rispettati i vincoli tecnici relativi all’applicazione della DCT2 specificati al capitolo [2.1.1](#_2.1._Requi).

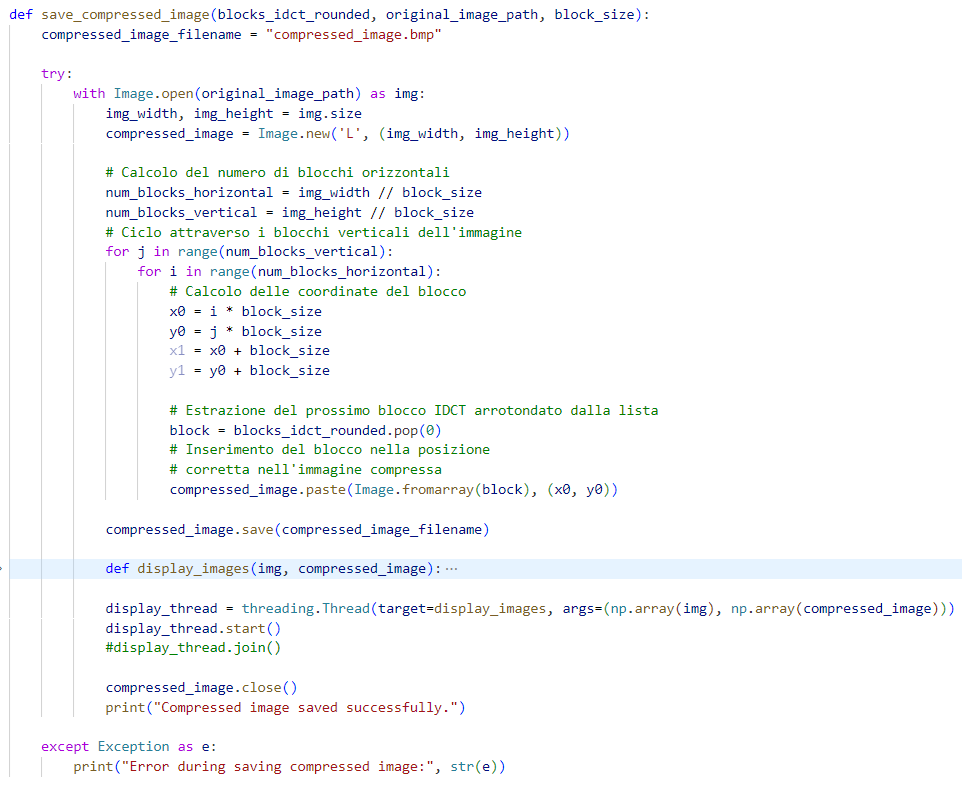
## 2.7. ProcessImages.py: save\_compressed\_image:

Di seguito viene illustrata la funzione dedicata alla ricomposizione dell’immagine partendo da blocchi quadrati () di dimensione . In seguito alla ricomposizione viene mostrata l’immagine originale affiancata a quella compressa e salvata l’immagine compressa in memoria. Più nello specifico si ha:

Salvataggio della nuova immagine nella memoria di sistema. Poi si utilizza la funzione display\_images che permette di visualizzare in un plot l’immagine originale e l’immagine processata affiancate

Calcolo numero di blocchi per ogni riga e per ogni colonna dell’immagine originale

Calcolo delle coordinate di ogni blocco da inserire nella nuova immagine. Ognuno di questi blocchi viene estratto dalla lista che ci siamo precedentemente salvati ed “incollato” nella posizione corretta dell’immagine compressa risultante.



**NOTA IMPLEMENTATIVA**: non blocare la grafica

Per non bloccare la grafica dell’applicazione si mostra avvia la funzione diplay\_images in un thread separato dal main\_thread. Questo permette all’applicazione di poter continuare la sua esecuzione in maniera smooth, senza che la grafica venga interrotta dalla creazione della nuova finestra che si crea per mostrare il paragone delle 2 immagini.

In questo modo siamo riusciti a descrivere completamente il funzionamento dell’applicazione spiegandone come avviene la compressione. Possiamo dunque ora passare all’analisi dei risultati ottenuti dalle elaborazioni di questa app.

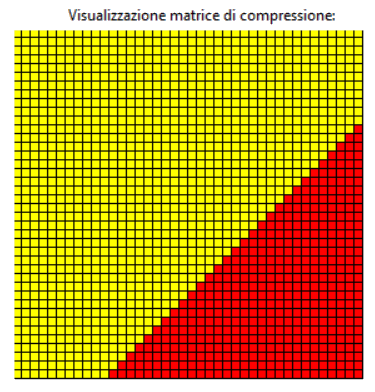
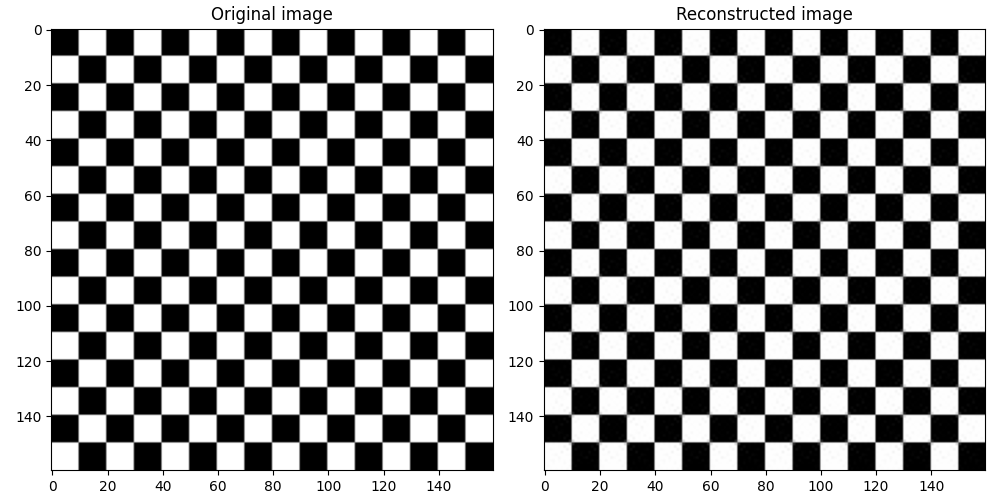
## 2.8. Analisi dei risultati prodotti dall’app:

###### **2.8.1. Immagine 1:**



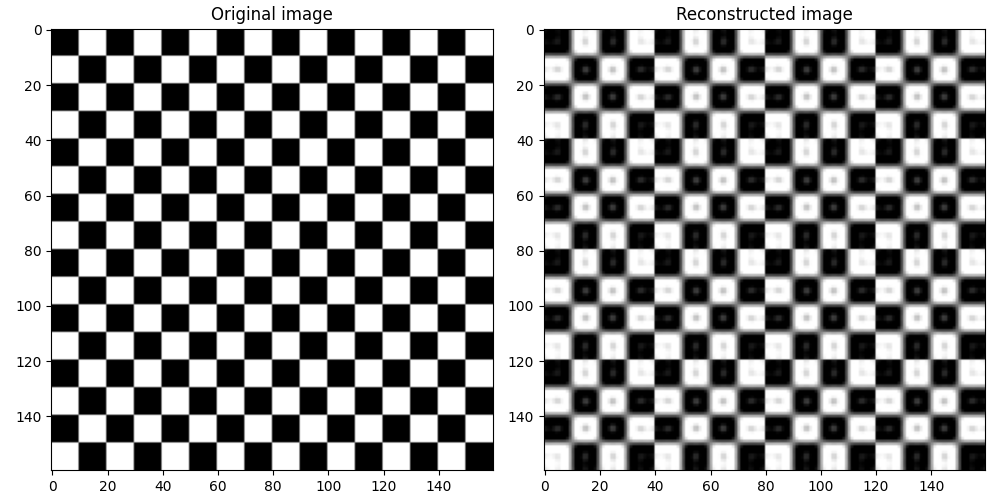
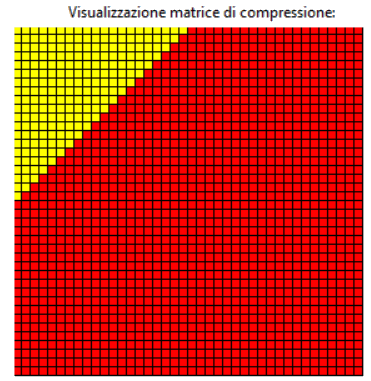
**Compressione con e**  (esempio di buona compressione)

Come è possibile visualizzare, scegliendo una dimensione di che è sottomultipla della dimensione di base ed altezza dell’altezza originale dell’immagine (40 è sottomultiplo di 160) non si verifica alcun crop sull’immagine risultante, e come ci si poteva aspettare, mantenendo un livello relativamente alto di rispetto al valore massimo che si potrebbe assumere (in questo caso 78) si riesce a mantenere una qualità dell’immagine relativamente alta, anche se zoomando sull’immagine compressa è possibile iniziare ad intravedere dei piccoli artefatti.

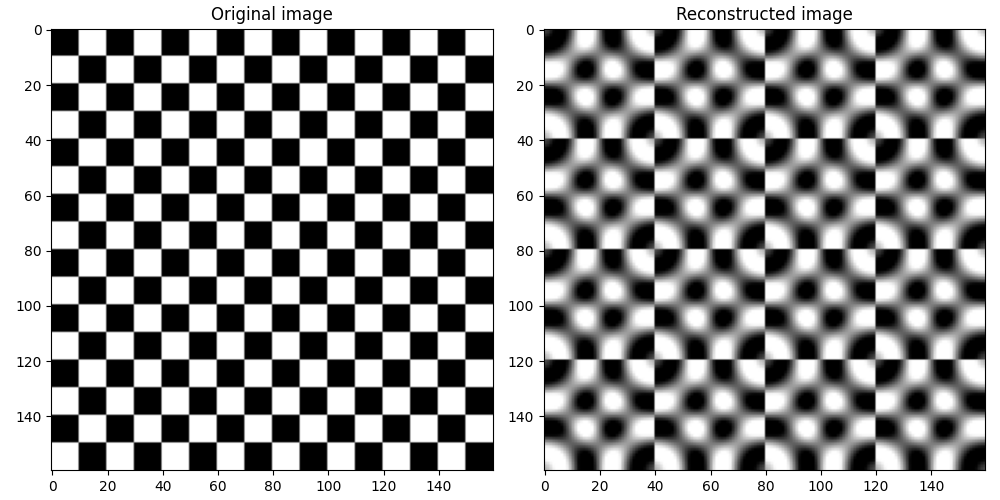
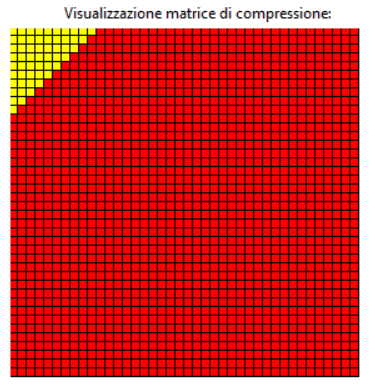


**Compressione con e**  (esempio di troppa compressione)

Se iniziamo ad aumentare la soglia di coefficienti da tagliare (diminuendo ) come in questo caso, è possibile vedere che la presenza degli artefatti diventa molto più accentuata, si hanno infatti che oltre alle macchie grigie presenti all’interno dei quadrati bianchi inizia anche a diventare evidente la discontinuità sul bordo delle finestre, dove, in questo caso, si vanno a formare degli artifact di forma quadrata che rispecchiano il bordo delle finestre usate per la compressione. La qualità dell’immagine risultante risulta comunque buona, anche se distorta, ma almeno il contenuto informativo dell’immagine originale (la scacchiera) è individuabile anche nella seconda immagine.

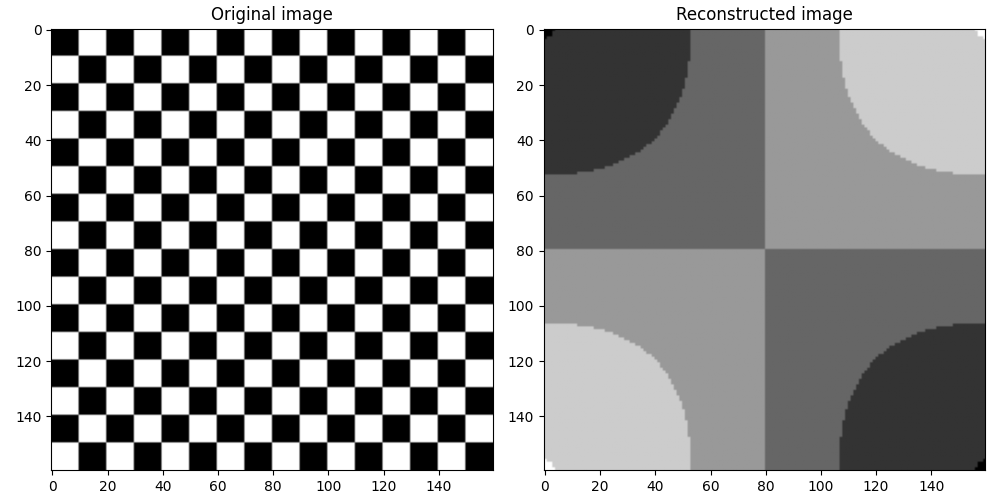


**Compressione con e**  (esempio di troppa compressione)



Se vogliamo spingere l’empio precedente ad avere un risultato ancora più estremo, possiamo rendere ancora più evidente la discontinuità sul bordo della finestra diminuendo ancora di più il coefficiente , in questo caso oltre alle discontinuità che si formano sui bordi delle finestre possiamo anche osservare che quelli che prima erano i quadratini che componevano la scacchiera diventare pesantemente “blurrati”. Questo effetto è dovuto alla grossa rimozione delle alte frequenze, che contengono le informazioni sui dettagli netti dell'immagine. L'effetto osservato all'interno dei blocchi è dunque causato dalla somma di poche componenti sinusoidali dominanti, che generano il tipico aspetto sfocato e ondulato.

**Compressione con e**  (esempio di perdita tutto valore informativo)



In questo caso, per dimostrare quanto disastrosi possano essere gli effetti della distorsione sull’immagine originale, si sceglie di utilizzare un’unica finestra di dimensione uguale alla dimensione originale dell’immagine. Avremo quindi una sola finestra molto grande. Si sceglie inoltre di usare un coefficiente molto basso. In questo modo possiamo osserva che la compressione ha rimosso quasi tutte le informazioni di dettaglio dell'immagine. Poiché sono state mantenute solo poche frequenze (). Abbiamo infatti che la ricostruzione è composta solo da una combinazione di pochissime sinusoidi a bassa frequenza. Questo ci porta a perdere completamente il contenuto dell’immagine originale.

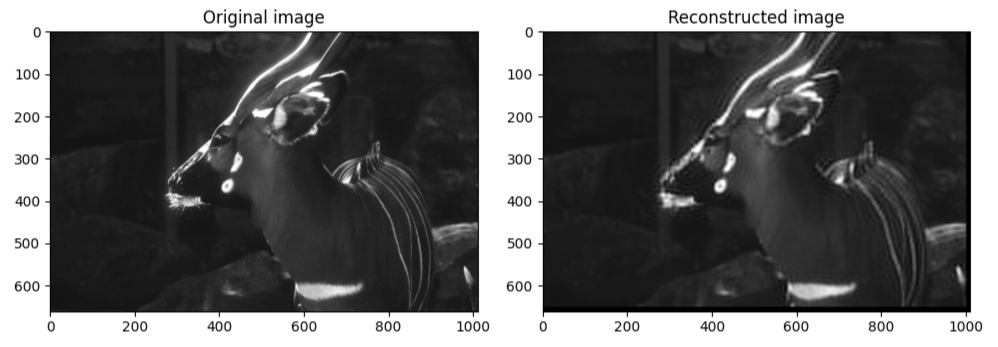
**NOTA SULLA RIPETIZIONE**: matrice di compressione

Per mostrare più nel dettaglio (ossia più grandi) le immagini relative alle prossime elaborazioni (e anche per evitare ripetizioni) non verrà più illustrata la matrice di compressione nei prossimi esempi, anche se questa, da applicazione, funziona e viene mostrata correttamente per ognuno degli esempi che mostriamo di seguito.

###### **2.8.2. Immagine 2:**

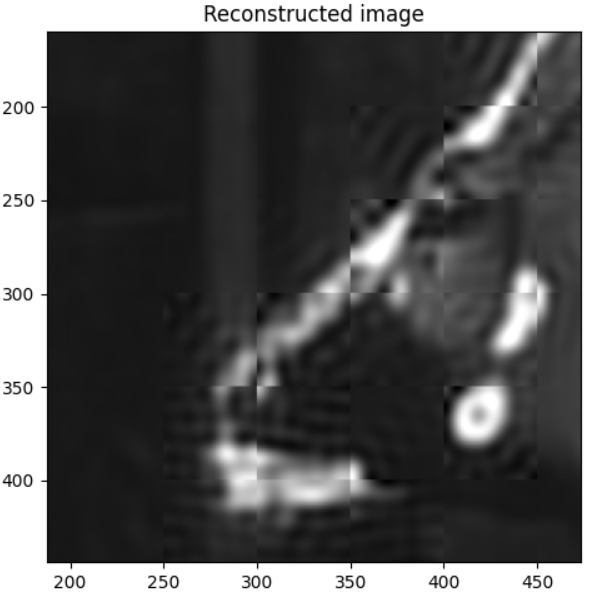


**Compressione con e**  (esempio di crop e fenomeno di Gibbs)



La qualità della ricostruzione di questa immagine è relativamente buona (finché non si zooma nei dettagli), ma si vuole presentare questa elaborazione come chiaro esempio dei seguenti problemi:

* Crop dell’immagine: è un fenomeno avevamo predetto sarebbe accaduto in base a come è strutturato il codice. In questo caso abbiamo che la dimensione originale dell’immagine è utilizzando una finestra di significa lasciare 11 pixel di bordo nero alle estremità dell’immagine, in quanto in questi pixel l’immagine non viene elaborata e si lascia dunque il valore dei pixel a 0, ossia li si lascia completamente neri.
* Discontinuità sui bordi delle finestre + artifact di forma quadrata + Presenza del fenomeno di Gibbs: In particolare se zoomiamo sull’immagine questi problemi diventano particolarmente evidenti:



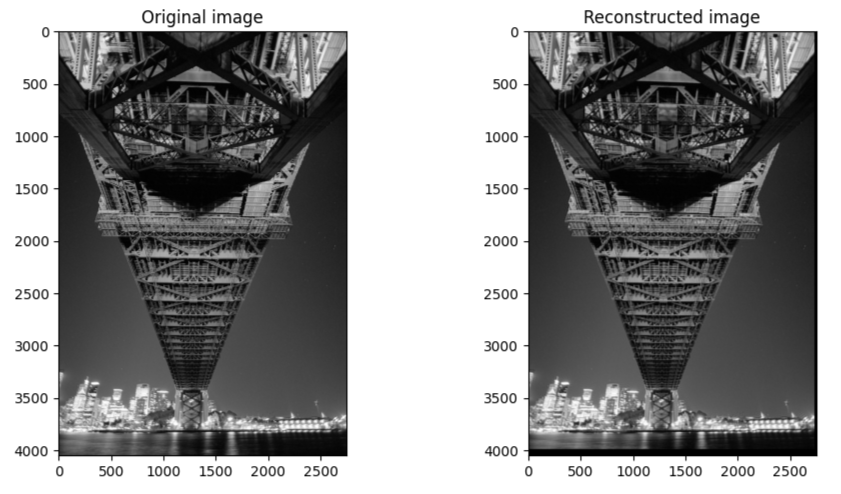
Il fenomeno di Gibbs è un effetto tipico della troncatura delle alte frequenze nella DCT. Si manifesta come oscillazioni e sovra/sotto-shooting vicino ai bordi netti dell'immagine. Il fenomeno è particolarmente visibile vicino ai contorni dell'animale, dove si notano strisce o bande luminose attorno ai bordi. Questo avviene appunto perché la brusca transizione di frequenze tra due regioni (ad esempio tra il corpo dell'animale e lo sfondo) richiede componenti ad alta frequenza per essere descritta con precisione. Eliminando noi gran parte delle frequenze con questa compressione diventa facile prevedere la comparsa di questo tipo di artefatto.

###### **2.8.3. Immagine 3:**

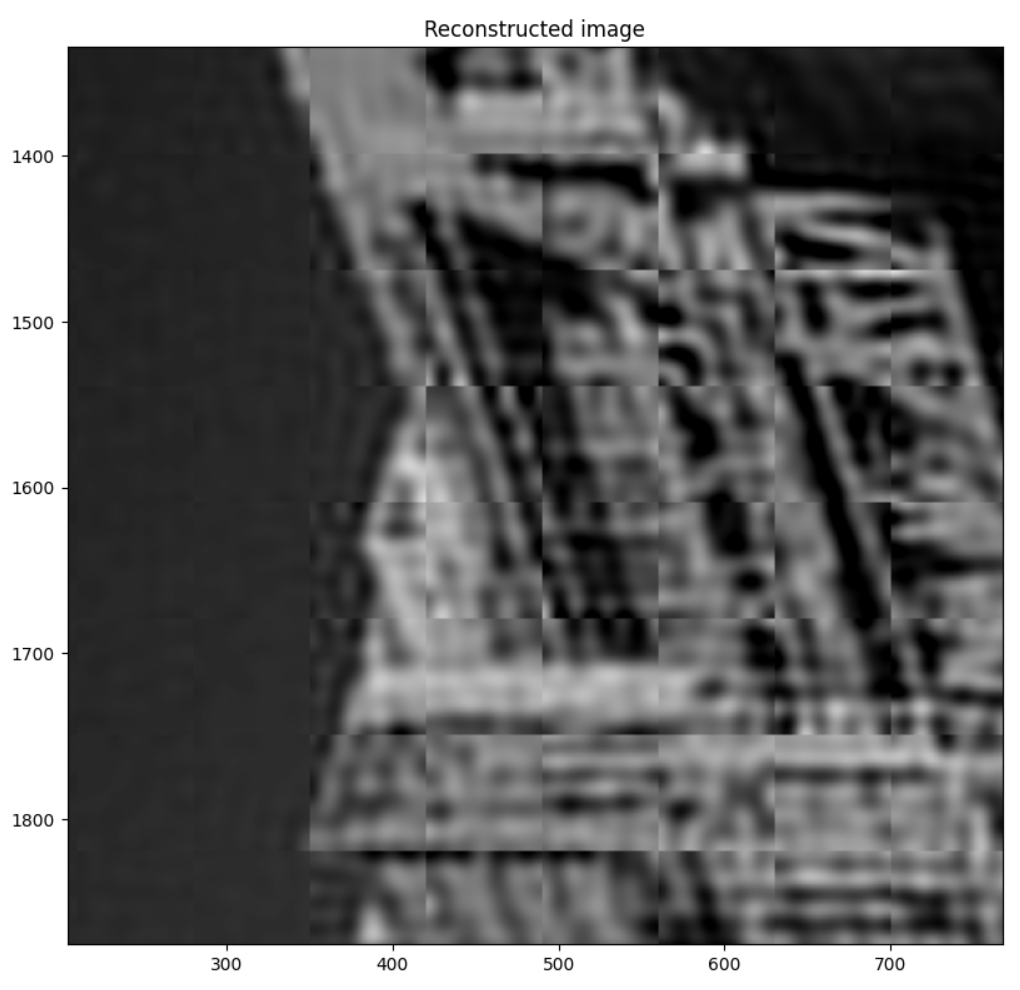


**Compressione con e**  (esempio di tempi di computazione lunghi)

Come possiamo vedere da quest’altro esempio anche se si tiene una soglia d molto bassa () su una finestra di dimensioni abbastanza grandi () il prodotto della compressione rimane, in questo caso, comunque di ottima qualità. Questo succede perché l’immagine originale ha dimensioni MOLTO grandi (), quindi usando una finestra di dimensione 70, stiamo in verità usando una finestra molto piccola rispetto la dimensione totale dell’immagine, dandoci l’illusione che il risultato di compressione sia stato ottimo. Se però andiamo a zoomare nelle regioni più interne dell’immagine compressa possiamo notare gli stessi identici problemi che abbiamo notato nell’immagine precedente (Discontinuità sui bordi delle finestre + artifact di forma quadrata + Presenza del fenomeno di Gibbs):



In questa sezione si vuole però esaminare un altro aspetto fondamentale: la scalabilità del tempo di computazione. Diversamente dal progetto-1 in questo caso non è stato misurato da codice la durata dei tempi di elaborazione, ma sappiamo dall’analisi teorica che la scalabilità dei tempi dovrebbe crescere significativamente () aumentano la dimensione dell’immagine, rendendo il metodo poco scalabile. Questo si riscontra con i tempi di computazione ottenuti per questa immagine. Essendo questa immagine molto più grande abbiamo che richiede tempi di elaborazione nella scala di qualche secondo, quando invece le altre immagini richiedevano elaborazioni in tempi di frazione di secondo. Possiamo dunque dedurre, che anche il principio di scalabilità bassa che ci aspettavamo è rispettato (anche se stiamo basando questa affermazione su osservazioni puramente euristiche).



## 2.9. Problemi riscontrati con l’applicazione:

Un problema riscontrato durante l’utilizzo dell’applicazione è che dalla quarta elaborazione in poi compare questo errore in console di debug:

"C:\Users\admin\AppData\Local\Programs\Python\Python310\lib\tkinter\_init.py", line 388, **in** **del** **if** self.\_tk.getboolean(self.\_tk.call("info", "exists", self.name)): RuntimeError: main thread **is** **not** **in** main loop Exception ignored **in**: <function Image.\_\_del\_\_ at 0x000001BED14C9870> Traceback (most recent call last): File "C:\Users\admin\AppData\Local\Programs\Python\Python310\lib\tkinter\_init.py", line 4046, **in** **del** self.tk.call('image', 'delete', self.name) RuntimeError: main thread **is** **not** **in** main loop

Questo stesso messaggio viene stampato in console 4/5 volte, è causato dall’utilizzo della libreria matplotlib.pyplot (plt) in congiunzione con il multithreading, più nello specifico il messaggio di errore è dovuto all’utilizzo della funzione diplay\_images in un thread separato. La non compatibilità delle librerie di threading di sistema e plt causa questa eccezione.

Si vuole fare però notare che questo non è un breaking bug in quanto l’eccezione è gestita come possibile nel codice dell’applicazione, però la soluzione del bug causa delay nel mostrare il paragone delle immagini (dal quarto tentativo in poi). Alternativamente per aggirare questo problema è semplicemente possibile riavviare l’applicazione, in questo modo l’eccezione non verrà più rilanciata.

## 2.10. Conclusioni parte-2:

I risultati ottenuti confermano che la DCT2 è uno strumento molto efficace per la compressione, ma mostrano anche quanto sia fondamentale scegliere con attenzione i parametri ed necessari per trovare il giusto compromesso tra qualità visiva e riduzione dei dati. Possiamo ritenerci soddisfatti dello sviluppo anche di questo progetto in quanto abbiamo dimostrato sia che tutti i requisiti tecnici ([2.1](#_2.1._Requi)) sono stati rispettati (mostrando una descrizione dettagliata per ogni metodo che implementa la compressione), sia che tutte le nostre aspettative ([2.2](#_2.2._Le_ba)) si sono dimostrate corrette. Abbiamo infatti che tramite l’analisi dei risultati abbiamo dimostrato che:

* **Compressione efficace delle immagini**: scegliendo una soglia adeguata di ed è possibile ottenere un buon compromesso.
* **Immagini in cui si perde tutta l’informazione**: al contrario del caso precedente, se si scelgono valori inadatti per le variabili di compressione (come un troppo piccolo) si corre il rischio di perdere tutto il valore informativo delle immagini.
* **Immagini con crop**: se non si sta attenti al valore inserito di , per come è strutturato il codice, è possibile ottenere in output un immagine croppata (esattamente come ci aspettavamo)
* **Presenza di artifact nell’immagine compressa**: In linea con le nostre aspettative, operando con questo tipo di compressione è possibile fare emergere diversi tipi di artefatti tra cui:
  + Artifact quadrati che rappresentano il bordo della finestra
  + Discontinuità tra bordi di due blocchi adiacenti
  + Presenza del fenomeno di Gibbs nei punti di edges dell’immagine
* **Scalabilità della compressione**: Seppur basato su osservazioni puramente euristiche è stato comunque possibile verificare che il metodo di compressione implementato con FFT (tramite funzioni di libreria) abbia tempi di compressione che aumentano significamene all’aumentare della dimensione dell’immagine. Cosa che ci aspettavamo in quanto l’applicazione di DCT2 / IDCT2 che applichiamo hanno complessità

Concludendo, siamo perfettamente soddisfatti dei risultati raggiunti, in quanto perfettamente in linea con le nostre aspettative.