COMPRESSIONE DI IMMAGINI TRAMITE DCT

METODI DEL CALCOLO SCIENTIFICO

Mario Avolio 880995

Simone Benitozzi 889407

13 giugno 2023

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

L'obiettivo dell'elaborato è principalmente quello di esporre i risultati ottenuti nella compressione di immagini in scala di grigio attraverso la Cosine Discrete Transform bidimensionale.

La prima parte prevede il confronto tra i risultati, in termini di tempo di esecuzione, tra l'implementazione della DCT-2 di una libreria Open Source, e un'implementazione from scratch.

La seconda parte del progetto consiste nell'implementazione di un'interfaccia grafica che permetta ad un utente di effettuare la compressione di un'immagine secondo parametri personalizzabili.

In quanto all'ambiente di programmazione, è stato scelto Python, in particolare con la librearia dctn.
Lo scopo è quello di presentare un'analisi approfondita, facendo assunzioni preliminari sui risultati attesi, per poi analizzare se effettivamente esse corrispondono alla realtà.

Reperibilità del progetto su GitHub Nella trattazione che segue verranno descritte le scelte di progettazione effettuate e i corrispettivi codici Python. Invitiamo il lettore a dare uno sguardo alla Repository GitHub qualora si voglia analizzare più dettagliatamente il progetto.

1 Python e Scipy

L'intero progetto è stato svolto mediante l'utilizzo del linguaggio Python. La scelta del linguaggio è inerente sopratutto alla vastità di librerie *open-source* in ambito scientifico. E' doveroso citare quantomeno quelle maggiormente sfruttate durante lo sviluppo come Scipy Virtanen et al. (2020), Numpy Harris et al. (2020), Pandas pandas development team (2020) e Matplotlib Hunter (2007). In particolare per questo scopo è stato fatto un uso massiccio di Scipy tramite la sottolibreria scipy.fft

La libreria scipy.fft Essa offre la possibilità di implementare la *Discrete Fourier transforms*. Sono state sfruttate principalmente le funzioni **dctn** e **idctn** per la *multidimensional Discrete Cosine Transform*. La libreria inoltre lavora per mezzo della *FFT*, Fast Fourier Transform, con una complessità computazionale asintotica attesa nell'ordine di $\mathcal{O}(n^2 \log n)$. Nel dettaglio la funzione **dctn** (analoga a **idctn**) offre inoltre diversi paramentri personalizzabili: quattro diversi tipi di implementazione di dct e tre differenti norme per il calcolo. Le figure seguenti indicano le quattro possibili tipologie di implementazione per la dct:

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

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

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

$$y_k = 2\sum_{n=0}^{N-1} x_n \cos\left(\frac{\pi(2k+1)(2n+1)}{4N}\right)$$

La norma utilizzata di default dalla funzione è la *backward*, i cui risultati non coincidevano con quelli visti a lezione. Per ottenre lo stesso scaling, necessario per la seconda parte del progetto, è bastato modificare il parametro a *ortho*, per ottenere scaling ortogonale. Seguendo la documentazione riportiamo di seguito la differenza:

> For norm="ortho" both the dct and idct are scaled by the same overall factor in both directions. By default, the transform is also orthogonalized which for types 1, 2 and 3 means the transform definition is modified to give orthogonality of the DCT matrix.
>
> For norm="backward", there is no scaling on dct and the idct is scaled by 1/N where N is the "logical" size of the DCT. For norm="forward" the 1/N normalization is applied to the forward dct instead and the idct is unnormalized.

1.1 Verifica dello Scaling

Sono state effettuate delle verifiche dello scaling in relazione alle norme utilizzate dalla funziona offerta della libreria *scipy.fft*. In particolare si è provata ad eseguire la funzione *dctn* sulla tabella 1 mediante $norm = "ortho"$, il risultato è visibile nella tabella 2.

	0	1	2	3	4	5	6	7
0	231.0	32.0	233.0	161.0	24.0	71.0	140.0	245.0
1	247.0	40.0	248.0	245.0	124.0	204.0	36.0	107.0
2	234.0	202.0	245.0	167.0	9.0	217.0	239.0	173.0
3	193.0	190.0	100.0	167.0	43.0	180.0	8.0	70.0
4	11.0	24.0	210.0	177.0	81.0	243.0	8.0	112.0
5	97.0	195.0	203.0	47.0	125.0	114.0	165.0	181.0
6	193.0	70.0	174.0	167.0	41.0	30.0	127.0	245.0
7	87.0	149.0	57.0	192.0	65.0	129.0	178.0	228.0

Tabella 1: Table Test

	0	1	2	3	4	5	6	7
0	1110.00	44.0	75.900	-138.00	3.50	122.0	195.00	-101.0
1	77.10	114.0	-21.800	41.30	8.77	99.0	138.00	10.9
2	44.80	-62.7	111.000	-76.30	124.00	95.5	-39.80	58.5
3	-69.90	-40.2	-23.400	-76.70	26.60	-36.8	66.10	125.0
4	-109.00	-43.3	-55.500	8.17	30.20	-28.6	2.44	-94.1
5	-5.38	56.6	173.000	-35.40	32.30	33.4	-58.10	19.0
6	78.80	-64.5	118.000	-15.00	-137.00	-30.6	-105.00	39.8
7	19.70	-78.1	0.972	-72.30	-21.50	81.3	63.70	5.9

Tabella 2: Results Table DCTN

2 Prima Parte

2.1 Scelte di Progettazione

La nostra implementazione di DCT 2D prevede i seguenti passi:

1. Computazione della matrice *D* della *Discrete Cosine Transform*
2. dct monodimensionale sulle colonne attraverso *D*
3. dct monodimensionale sulle righe attraverso *D*

```python
def compute_D(N):
    alpha_vect = np.zeros((N, 1))
    D = np.zeros((N, N))

    alpha_vect[0] = N**(-0.5)
    alpha_vect[1:] = N**(-0.5) * math.sqrt(2)

    for k in range(1, N+1):
        for i in range(1, N+1):
            D[k-1, i-1] = alpha_vect[k-1] * math.cos((k-1) * math.pi * (2*i-1) / (2*N))

    return D
```

Oltre alla DCT bidimensionale, riportata di seguito, è stata anche implementata la DCT monodimensionale, e le rispettive inverse per entrambe, di cui si omette il codice per brevità.

```python
def my_dct2(f_mat):
    N = len(f_mat)      #number of rows and columns
    D = compute_D(N)
    c_mat = f_mat

    # DCT1 for columns
    for j in range(N):
        c_mat[:,j] = np.dot(D, c_mat[:,j])

    # DCT1 for rows
    for j in range(N):
        c_mat[j,:] = np.transpose(np.dot(D, np.transpose(c_mat[j,:])))

    return c_mat
```

Trattandosi di un'implementazione molto semplificata, senza particolari ottimizzazioni, non ci si aspetta grandi risultati in termini di efficienza, con una complessità computazionale asintotica attesa nell'ordine di $\mathcal{O}(n^3)$

2.2 Risultati Sperimentali

Punto focale della prima parte del progetto è stato quello dell'analisi dei risultati delle due implementazioni di DCT, per confrontarne i risultati in termini di efficienza computazionale.

Nello specifico sono state testate matrici di dimensioni da *N=8* a *N=10.000*.Su di esse sono state eseguite entrambe le implementazioni di *DCT-2D*, e sui risultati ottenuti, le rispettive i-DCT2, per riottenere la matrice originale. Per tutte e 4 le funzioni sono stati registrati i tempi ottenuti.

Di seguito è mostrato il pattern di creazione delle matrice testate, che è lo stesso visto a lezione. La scelta di avere un pattern definito per tutte le matrici, anche al crescere delle dimesioni, può essere motivata dal fatto che volendo effetturare un'analisi quanto più possibile precisa e rappresentativa dei risultati, abbiamo preferito evitare di inserire una componente randomica all'interno del processo

```
for j in range(1, N+1):
    for ell in range(1, N+1):
        x_val = (2*j-1) / (2*N)
        y_val = (2*ell-1) / (2*N)
        f_mat[j-1, ell-1] = f(x_val, y_val)
```

Un esempio di matrice costruita in questo modo attraverso il parametro *N=8* è il seguente

$$\begin{matrix} 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \end{matrix}$$

I risultati ottenuti sono dunque mostrati nella tabella 3, suddivisi per algoritmo applicato e dimensione della matrice.
Si nota subito come le esecuzioni più pesanti abbiano impiegato tempi considerevoli, soprattuto per l'implementazione fatta da noi, in particolare per *N=10.000*, il cui tempo complessivo di esecuzione è stato di 53 minuti, mentre l'intera esecuzione ha richiesto 2 ore e 45 minuti totali.

N	my_dct2	scipy_dct2	my_idct2	scipy_idct2
8.0	0.000000	0.000958	0.000000	0.000000
16.0	0.000000	0.000000	0.002004	0.000000
32.0	0.003052	0.000000	0.001994	0.000000
64.0	0.016999	0.000000	0.040002	0.000000
100.0	0.048003	0.000995	0.038997	0.001000
250.0	0.171372	0.001997	0.191005	0.000997
500.0	0.585959	0.009003	0.728988	0.003996
750.0	1.757999	0.017997	1.682003	0.016996
1000.0	3.473949	0.041000	4.368999	0.029999
1500.0	8.694001	0.079000	10.939000	0.073998
2000.0	17.948997	0.140060	22.092954	0.169000
2500.0	33.305009	0.238048	34.149954	0.311993
3000.0	54.187999	0.512000	56.753988	0.408993
3500.0	80.074001	0.511000	90.720990	0.604976
4000.0	112.808998	0.641059	128.283914	0.608915
4500.0	151.504001	0.851001	149.634990	0.964912
5000.0	198.861000	1.153052	199.803908	1.216013
6000.0	324.213001	1.609000	319.287986	1.787985
7000.0	528.754846	2.923000	514.121990	2.724993
8000.0	733.791951	3.921043	742.012907	3.891963
9000.0	1053.912169	4.901043	1098.505610	4.820091
10000.0	1550.146987	5.979995	1631.578991	7.541971

Tabella 3: DCT Times

2.2.1 DCT comparison

Si parte dal confronto della *DCT2* implementata con quella della libreria *scipy*. Le figure 1 e 2 mostrano i tempi di esecuzione al crescere della dimensione della matrice. Oltre ai tempi di esecuzione, sono stati tracciate sul grafico anche le forme delle funzioni n^3 e $n^2 \log n$, divise per un coefficiente di normalizzazione per fare in modo che solo l'ultimo valore coincidesse con i tempi effettivi. Si nota come in entrambi i casi le funzioni assumano la forma attesa, se non per alcune differenze negli stadi di osservazione iniziali.

2.2.2 iDCT comparison

Lo stesso paragone è stato fatto per le *inverse-DCT 2D*, confrontando la nostra implementazione con quella di *scipy*. I risultati mostrati nelle figure 3 e 4 seguono la stessa tendenza di quelli visti per *DCT2*. Anche in questo caso la forma assunta dal grafico dei risultati segue precisamente quella della funzione attesa.

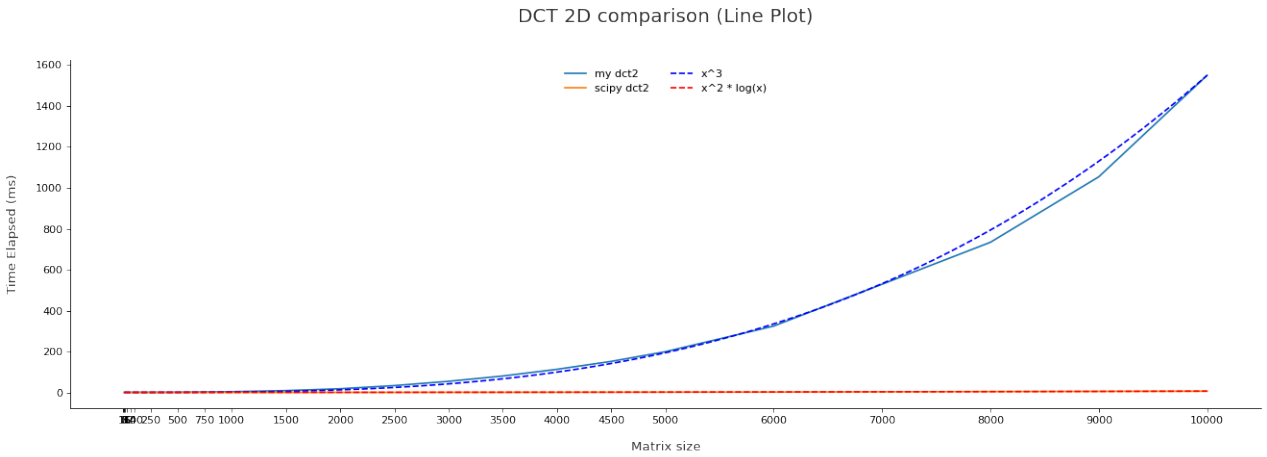


Figura 1: DCT2 comparison

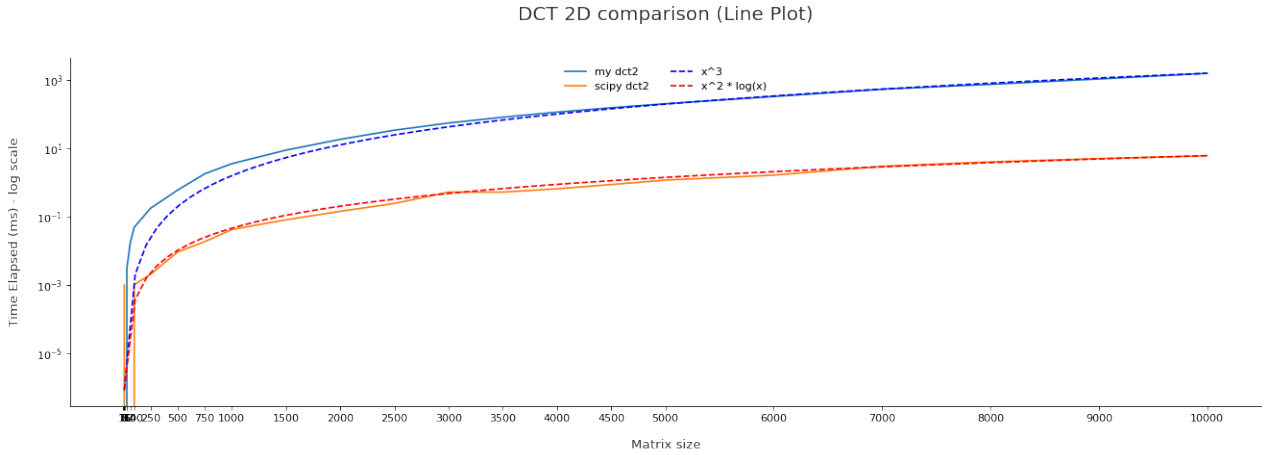


Figura 2: DCT2 semi-log comparison

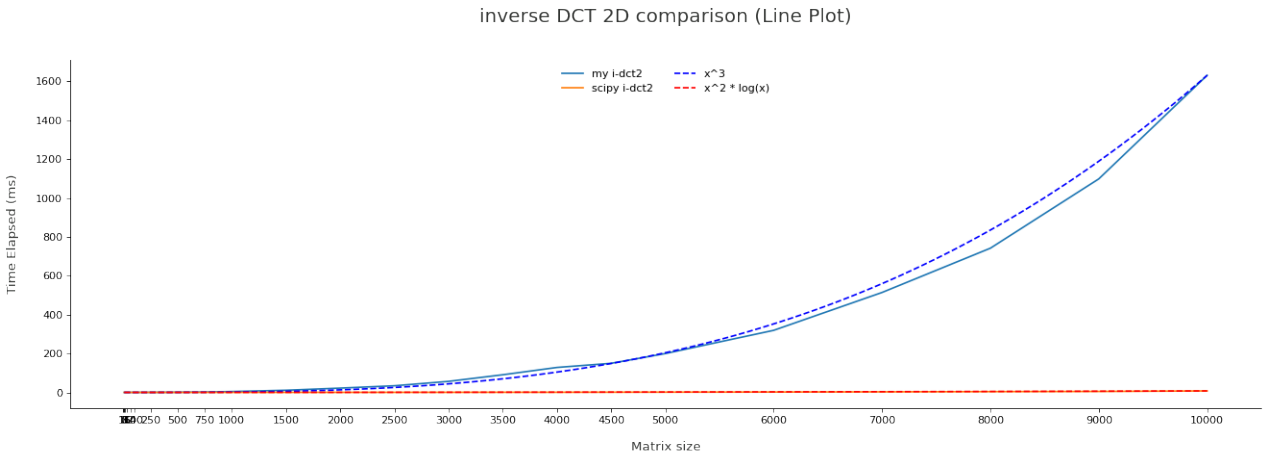


Figura 3: inverse DCT2 comparison

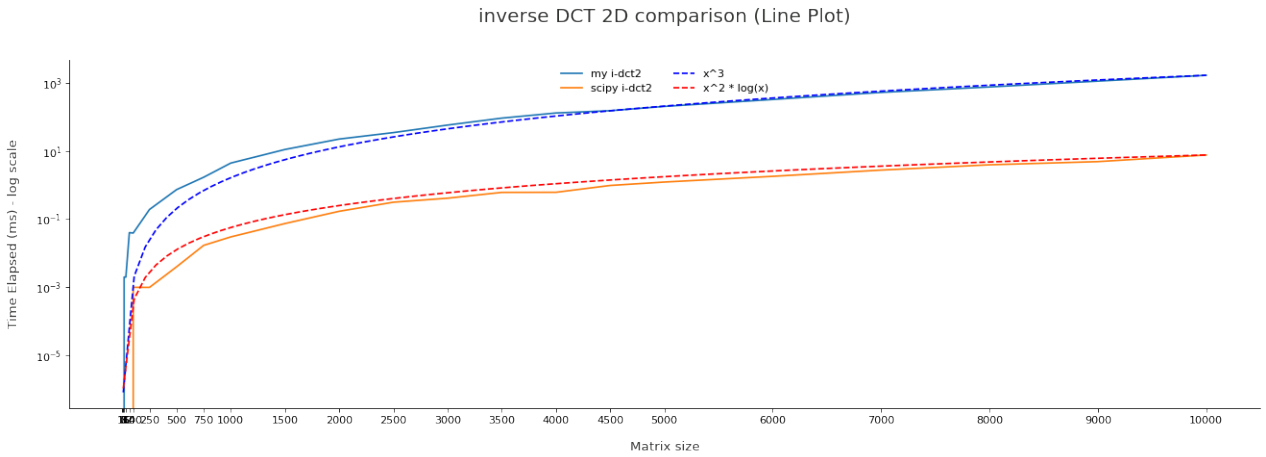


Figura 4: inverse DCT2 semi-log comparison

2.2.3 DCT vs iDCT

Un'ultima breve analisi eseguita è stata quella per confrontare la nostra le *DCT2* e le *iDCT2* delle rispettive implementazioni tra di loro.

La figura 5 mostra come le differenze tra le due funzioni siano praticamente nulle, con la *DCT2* che in media impiega *6 secondi* in meno della *iDCT2*, con una differenza massima di *81 secondi* per *N=10.000*. Tuttavia in alcuni casi la *iDCT* risulta addirittura più veloce della sua controparte, nonostante il suo calcolo richieda il calcolo dell'inversa della matrice da trasformare

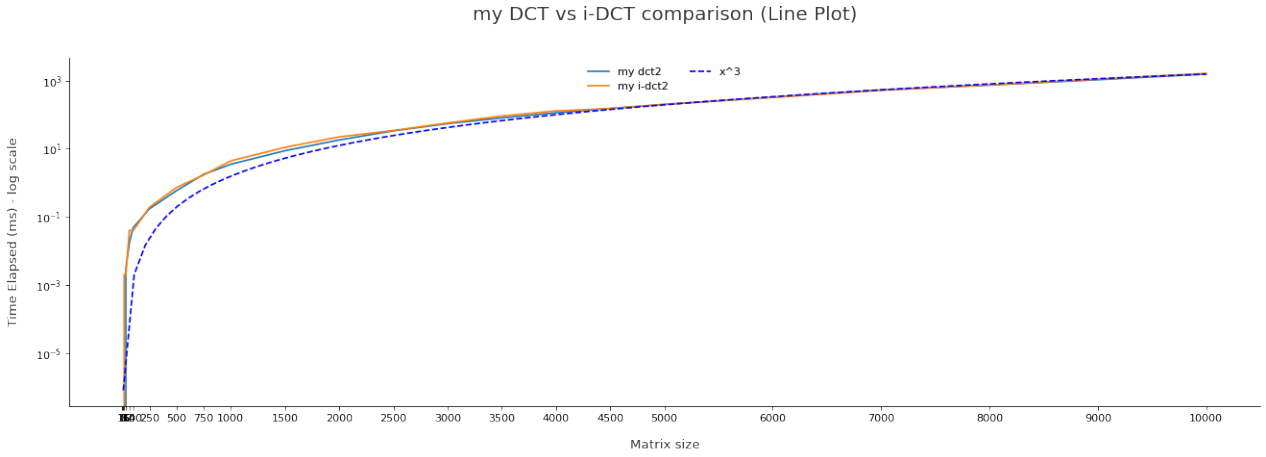


Figura 5: my DCTs semi-log comparison

Anche per l'implementazione delle funzioni in *scipy* la tendenza è la stessa. Qui troviamo che in media *DCT2* impiega *0.075 secondi* in meno, con un picco di *1.5 secondi* di vantaggio per *N=10.000*.

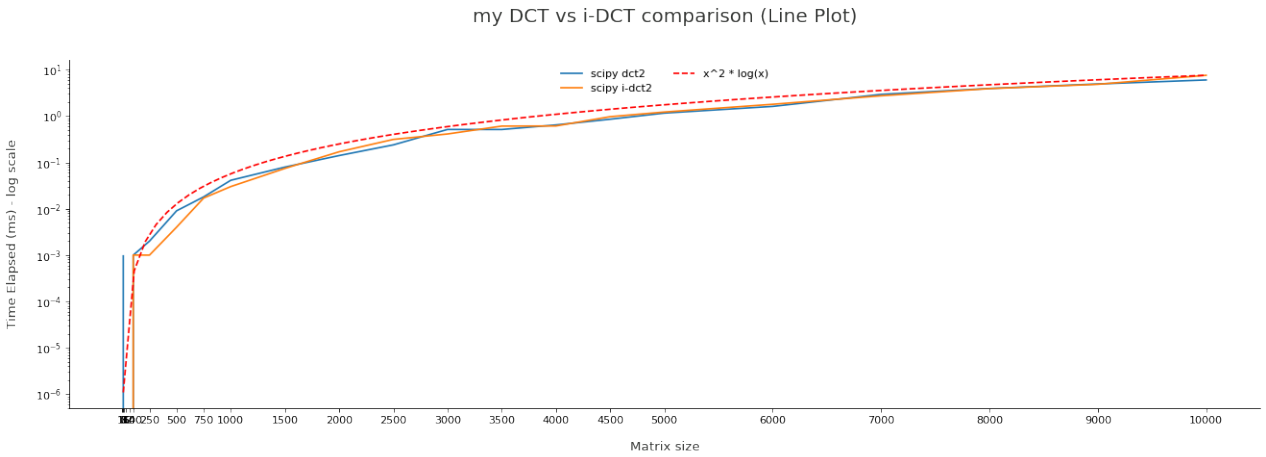


Figura 6: scipy DCTs semi-log comparison

3 Seconda Parte

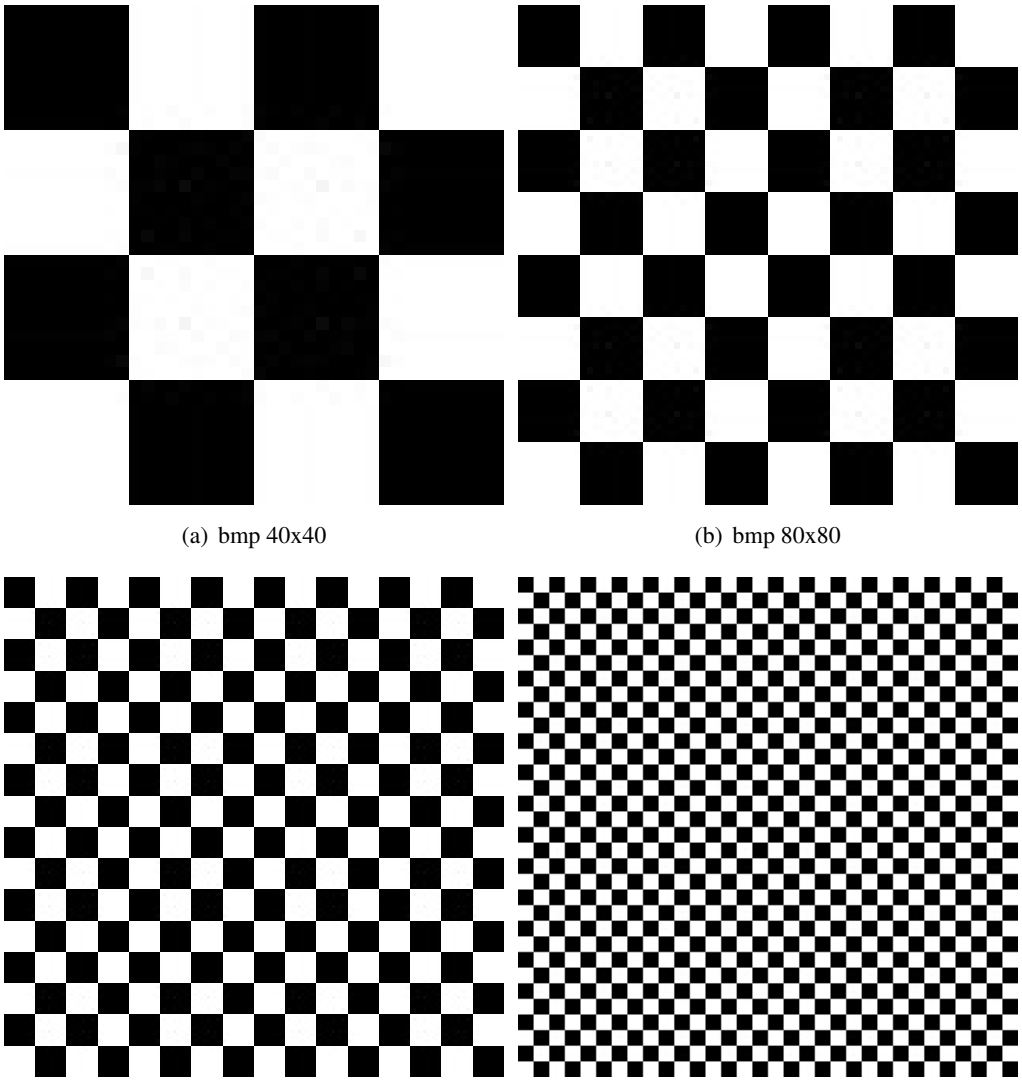
3.1 Descrizione del dominio di riferimento e obiettivi della sperimentazione

In questa trattazione verranno esposti i moventi e i modi per la creazione di un software, compreso di interfaccia grafica, in grado di eseguire la Compressione JPEG su immagini prendendo in input alcuni parametri che ne caratterizzassero la procedura. In particolare si è voluto analizzare il *fenomeno di Gibbs* su alcune figure che presentassero un determinato grado di contrasto. I passi che il software esegue sono rappresentabili graficamenti dalla figura 7.

Figura 7: Schema Compressione JPEG

3.2 Descrizione dei Dati

I dati su cui verte la sperimentazione sono una serie di immagini *.bmp* in toni di grigio. Le immagini nella figura 8 ne mostrano un esempio.



(a) bmp 40x40 (b) bmp 80x80

(c) bmp 160x160 (d) bmp 320x320

Figura 8: Immagini bmp in toni di grigio

Il formato bmp Il formato bitmap Wikipedia contributors (2022) è usato per salvare in memoria *bitmap digital images* indipendentemente dal dispositivo utilizzato. Tale formato è in grado di salvare immagini digitali a due dimensioni sia in versione monocromatica che in versione a colori. Un file bitmap consiste in diverse strutture dimensionalmente fissate:

- Bitmap File Header
- DIB header
- Extra bit masks
- Color Table
- Gap1
- Pixel Array

- Gap2
- ICC color profile

3.3 Scelte di Progettazione

Come già accennato precedentemente si è optato per l'utilizzo di Python anche nella seconda parte del progetto. Si è sfruttata principalmente la libreria *Scipy* mediante le funzioni **dctn** e **idctn**. Di seguito viene proposta la sfruttura del progetto:

```
Part2
└── bin
    ├── Analyze
    │   ├── __init__.py
    │   └── analyze.py
    ├── Compression
    │   ├── __init__.py
    │   └── compression.py
    ├── GUI
    │   ├── __init__.py
    │   ├── GUI.py
    │   └── helper.py
    └── Core
        ├── __init__.py
        ├── helper.py
        ├── costants.py
        └── core.py
```

L'uso dei Thread Il package Analyze implementa al suo interno alcune funzionalità implemenate oltre ciò che è stato richiesto dalla traccia. In particolare si occupa di instanziare una serie di thread preposti ognuno a comprimere tramite DCT2 una singola immagine. Il codice 1 ne mostra un esempio di implementazione.

```
1 class Analyze(Thread):
2     def __init__(self, file_path, f:int, d:int):
3         super().__init__()
4         self.__file_path = file_path
5         self.__f = f
6         self.__d = d
```

```
7
8       def run(self) -> None:
9           img = read_img(SHOW=False, FILE_PATH=os.path.join(self.__file_path))
10          matrix_list: list[ndarray] = splitting_img(img, self.__f)
11          c = Compression(matrix_list, self.__d)
12          matrix_compressed_list = c.start()
13          reassemble = img_reassemble(matrix_compressed_list, self.__f, img.shape
    [0])
14          show_img(img, reassemble)
```

Listing 1: Classe Analyze

La GUI Il software implementa una piccola interfaccia grafica implementata nel package GUI mediante la libreria **Tkinter**. L'interfaccia è molto semplice ed intuitiva ed ha il compito di prendere in input i parametri che caratterizzano la compressione e di andare a scegliere l'immagine da comprimere. La figura 9 mostra l'interfaccia descritta.

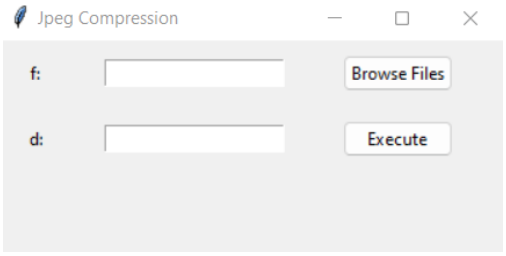


Figura 9: GUI

La GUI richiama **da altre funzioni di utilità** le seguenti operazioni implementate nel codice 2:

1. Lettura dell'immagine
2. Suddivisione dell'immagine in blocchi $F \times F$
3. Compressione
4. Riassemblamento Immagine
5. Visualizzazione dell'immagine originale e dell'immagine compressa

```
1 def execute(file_path, f, d):
2     img = read_img(FILE_PATH=file_path, SHOW=True)
3     matrix_list: list[ndarray] = splitting_img(img, f)
4     c = Compression(matrix_list, d)
5     matrix_compressed_list = c.start()
6     reassemble = img_reassemble(matrix_compressed_list, f, img.shape[0])
7     show_img(img, reassemble)
```

Listing 2: Esecuzione Compressione

Il Package Core Il package Core contiene al suo interno tutta la logica implementativa inerenti alle fasi adiacenti alla compressione stessa, molte delle quali sfruttate dall'interfaccia precedentemente descritta. In particolare offre diverse funzioni di utilità come:

- Lettura dell'immagine
- Suddivisione dell'immagine in blocchi $F \times F$
- Riassemblamento dell'immagine originale
- Visualizzazione dell'immagine originale e dell'immagine compressa

Il Package Compression Il package Compression implementa le funzionalità di compressione JPEG richiamate dalla GUI. Come si può notare dal codice 3, l'algoritmo esegue le seguenti operazioni in ordine:

1. Esecuzione della DCT2 sul blocco $F \times F$
2. Rimozione delle frequenze
3. Esecuzione della IDCT2
4. Arrotondamento

```
    def start(self) -> list[ndarray]:
        compressed_matrix_list: list[ndarray] = []
        for matrix in self.__matrix_list:
            c = self.dct2(matrix)
            c = self.remove_frequencies(c)
            ff = self.idct2(c)
            ff = self.round(ff)
            compressed_matrix_list.append(ff)

        return compressed_matrix_list
```

Listing 3: Classe Compression

3.4 Risultati Sperimentali

Di seguito verranno proposte le principali analisi effettuate per mostrare in maniera marcata gli effetti della compressione. In particolare si è optato per una compressione con parametri $f = 8$ e $d = 1, 8, 16$.

3.4.1 Deer Image

Dalle figure 11 e 12 si può notare che la mancanza di un forte contrasto minimizza il *fenomeno di Gibbs* nei casi si usasse un valore di d alto.

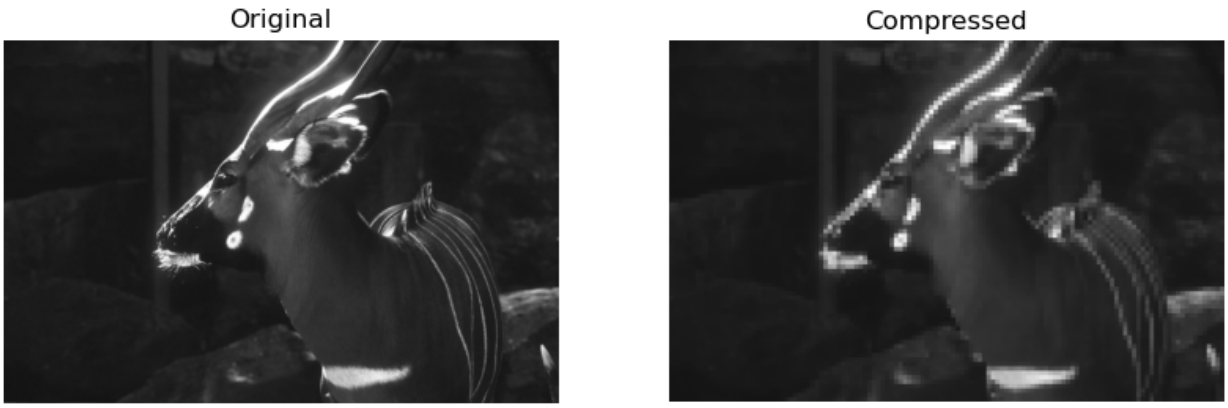


Figura 10: Deer f=8, d=1

Figura 11: Deer f=8, d=8

Figura 12: Deer f=8, d=14

3.4.2 Bridge Image

Dalle figure 13,14 e 15 si può notare come il *fenomeno di Gibbs* sia quasi del tutto assente anche nel caso di *d=1*. Si può dedurre che questo sia divuto alla mancanza di contrasto nell'immagine.

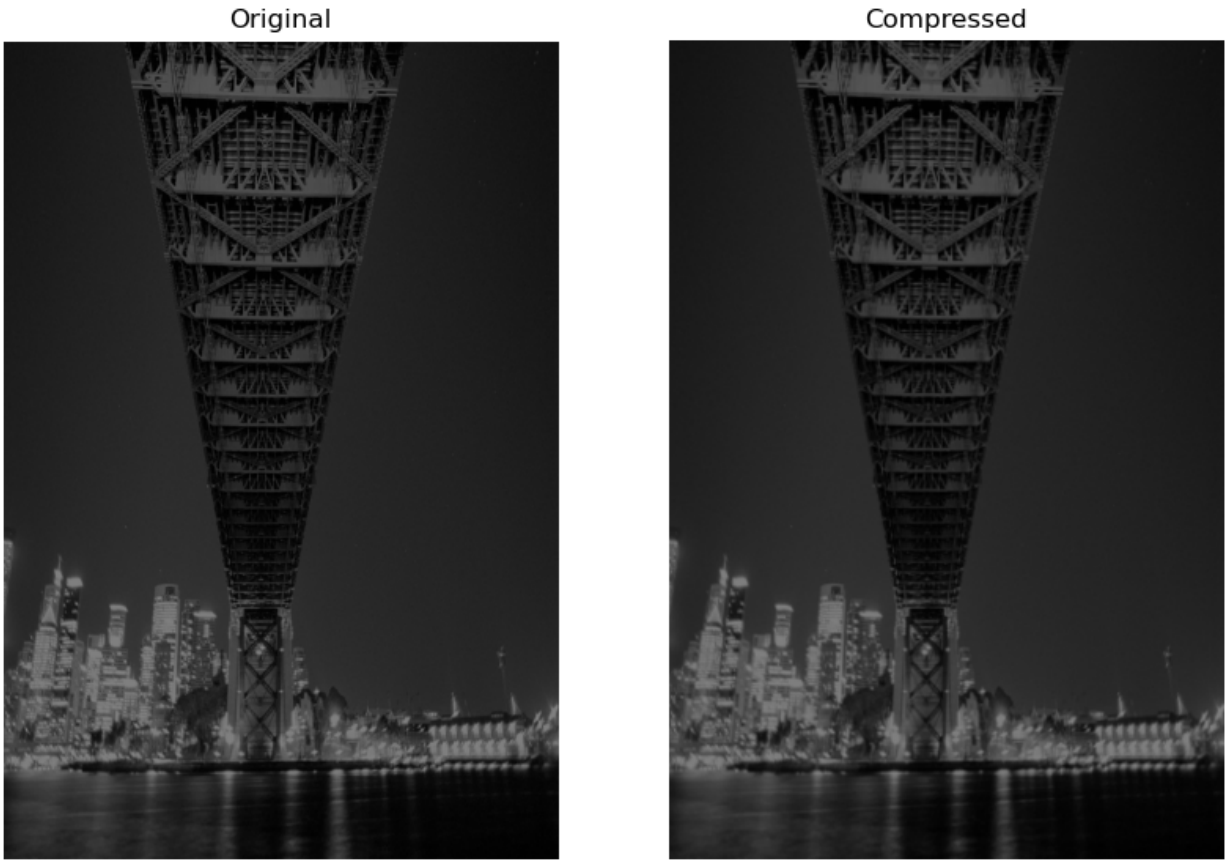


Figura 13: bridge f=8, d=1

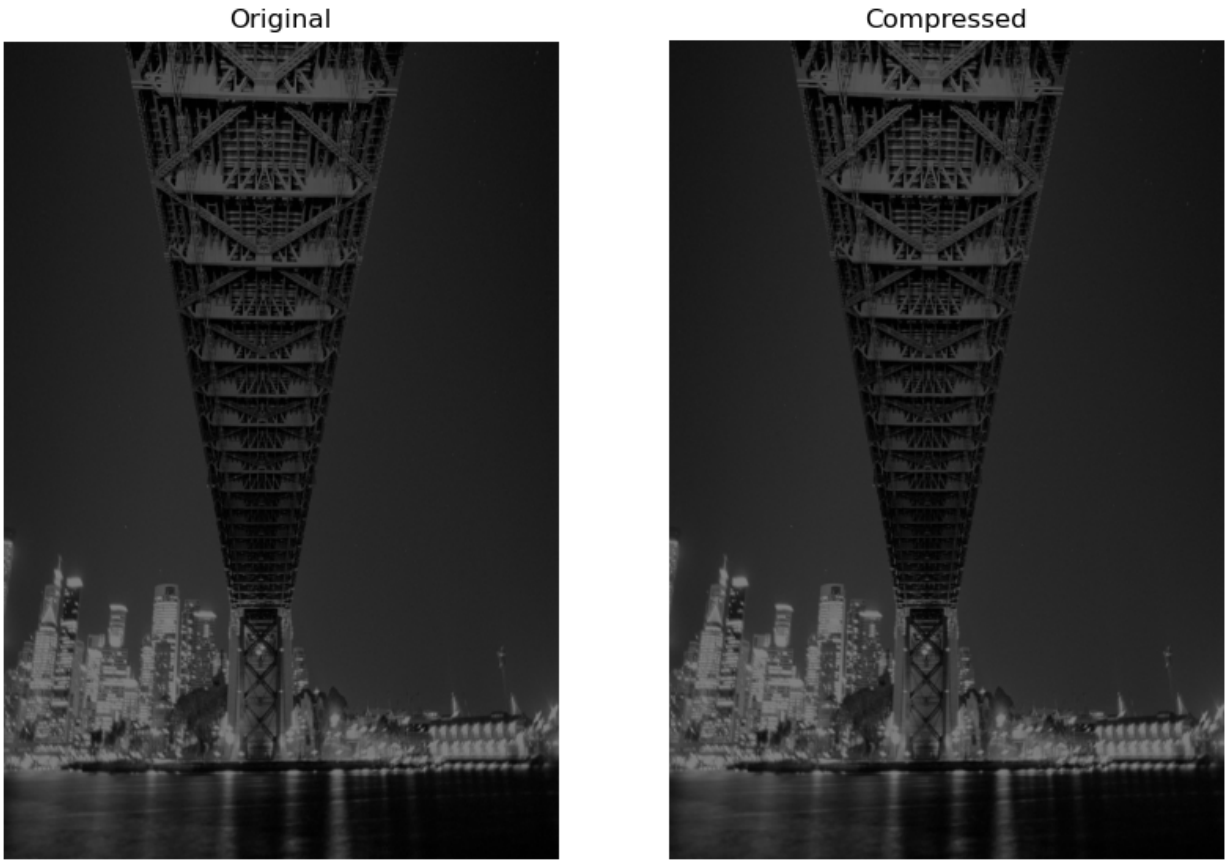


Figura 14: bridge f=8, d=8

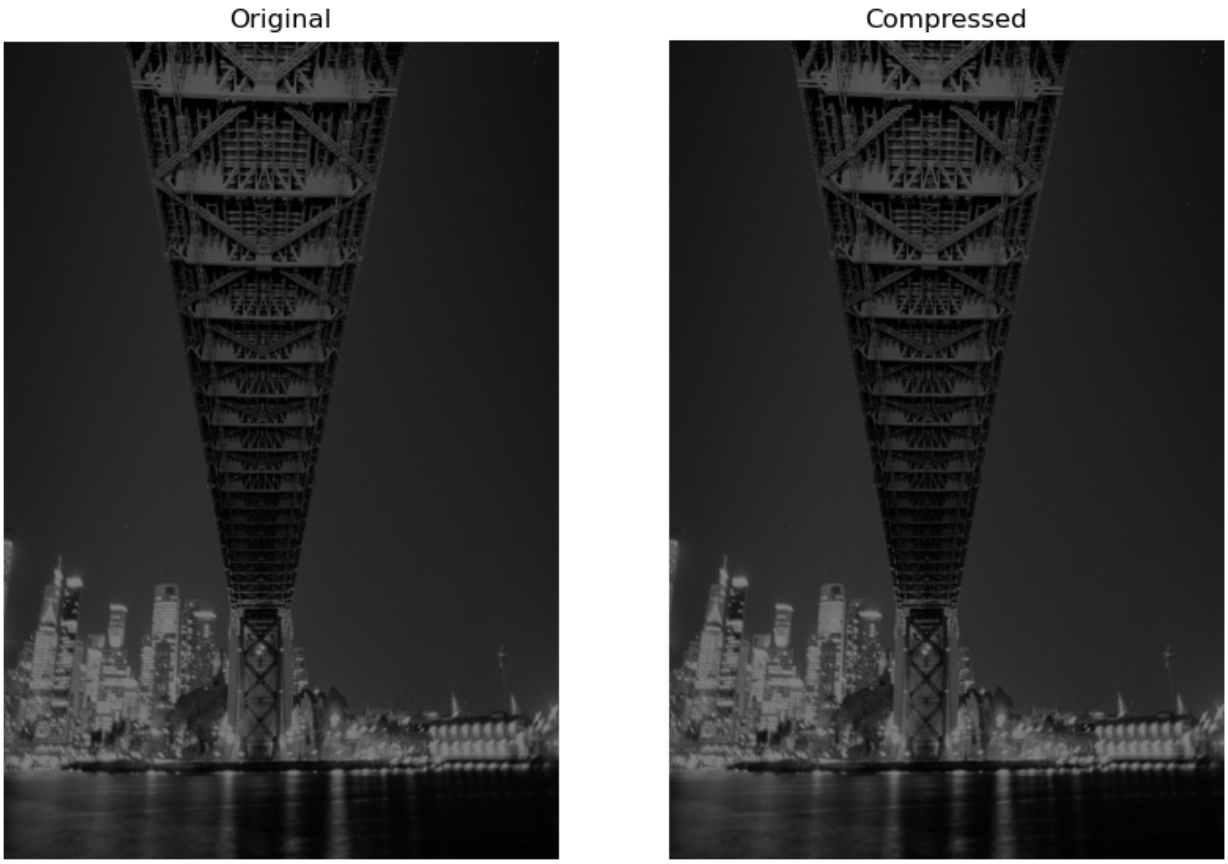


Figura 15: bridge f=8, d=14

3.4.3 Prova image

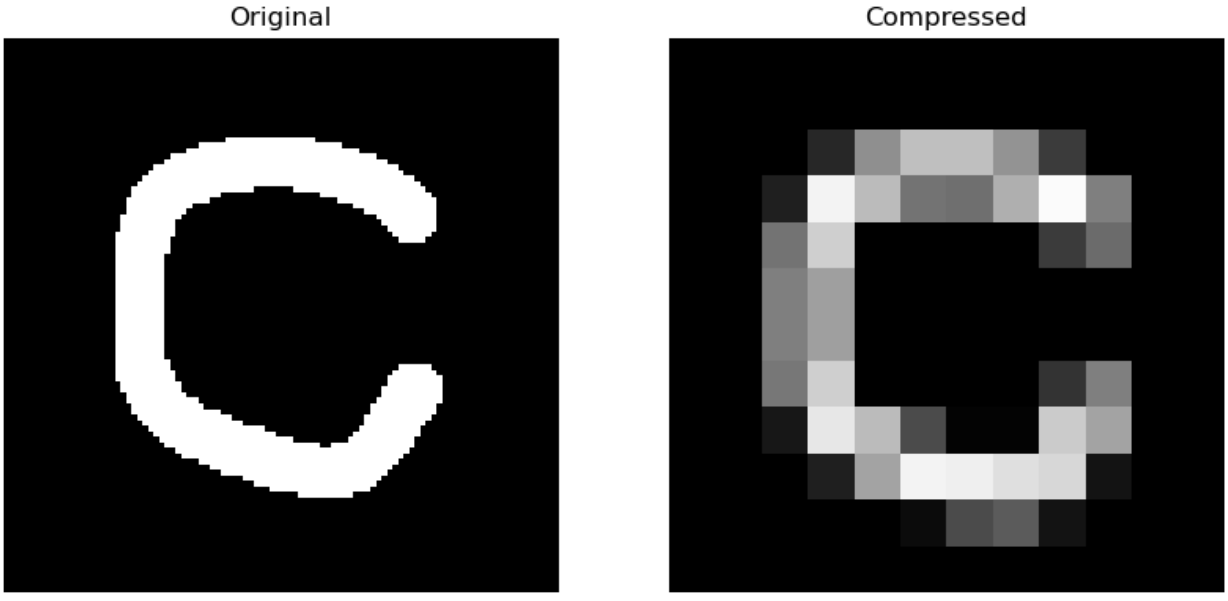


Figura 16: Prova f=8, d=1

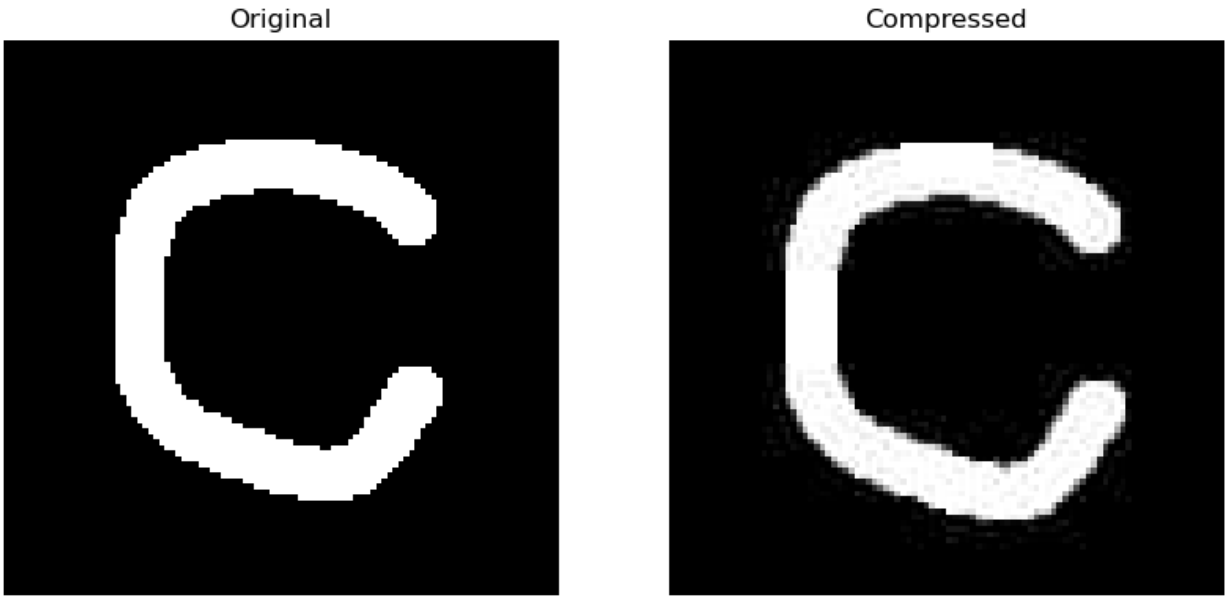


Figura 17: Prova f=8, d=8

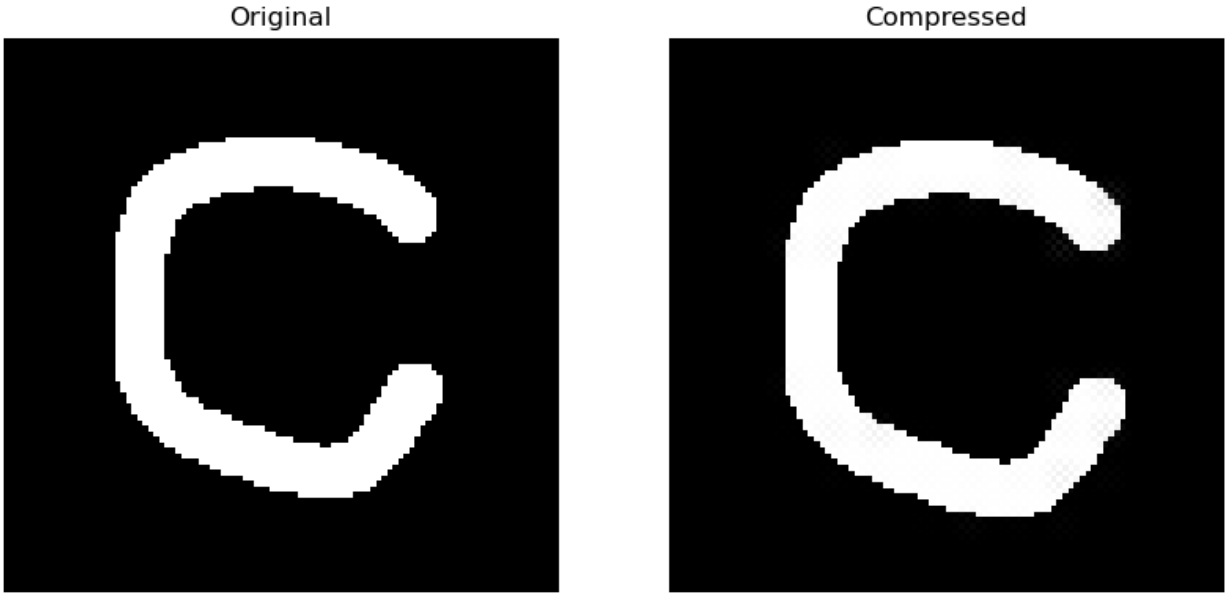


Figura 18: Prova f=8, d=14

3.4.4 80x80 Image

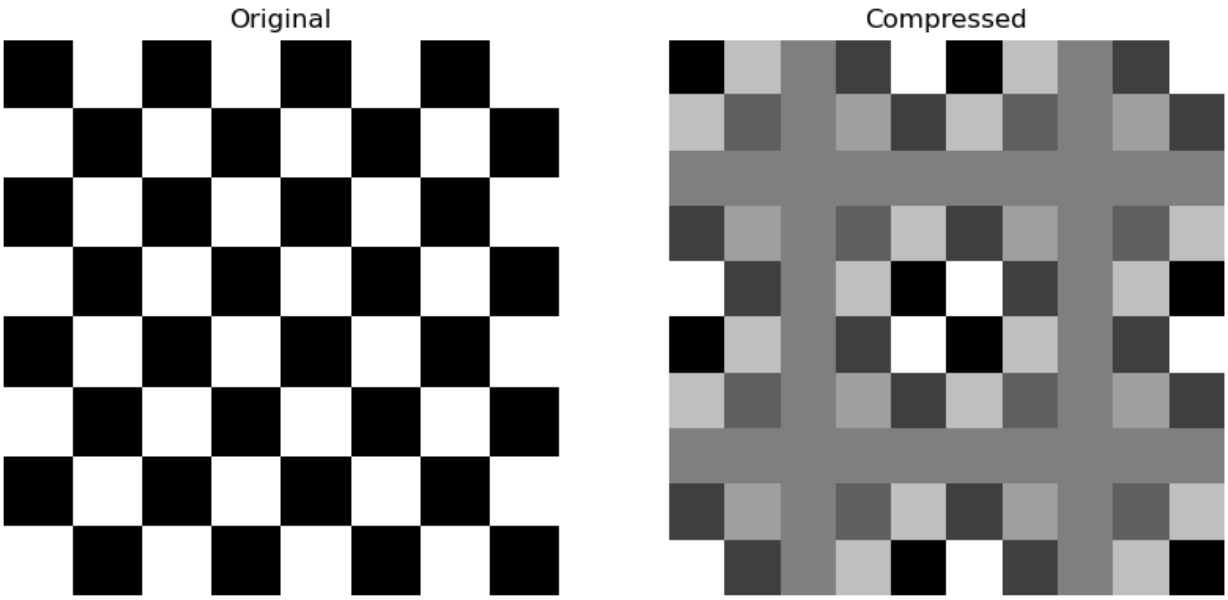


Figura 19: 80x80 f=8, d=1

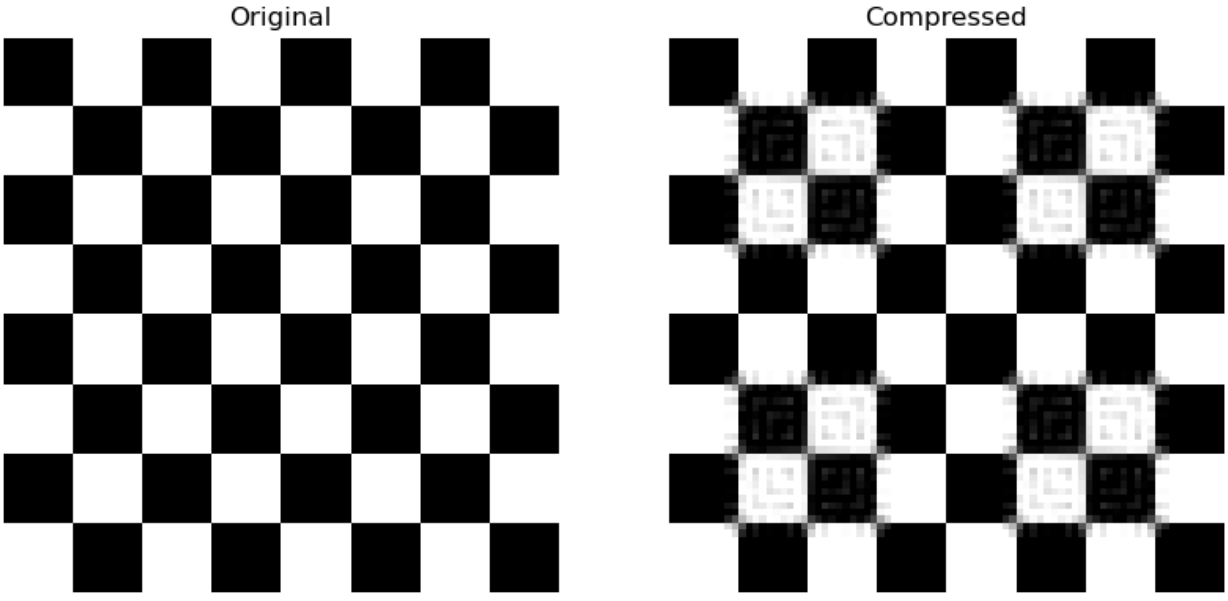


Figura 20: 80x80 f=8, d=8

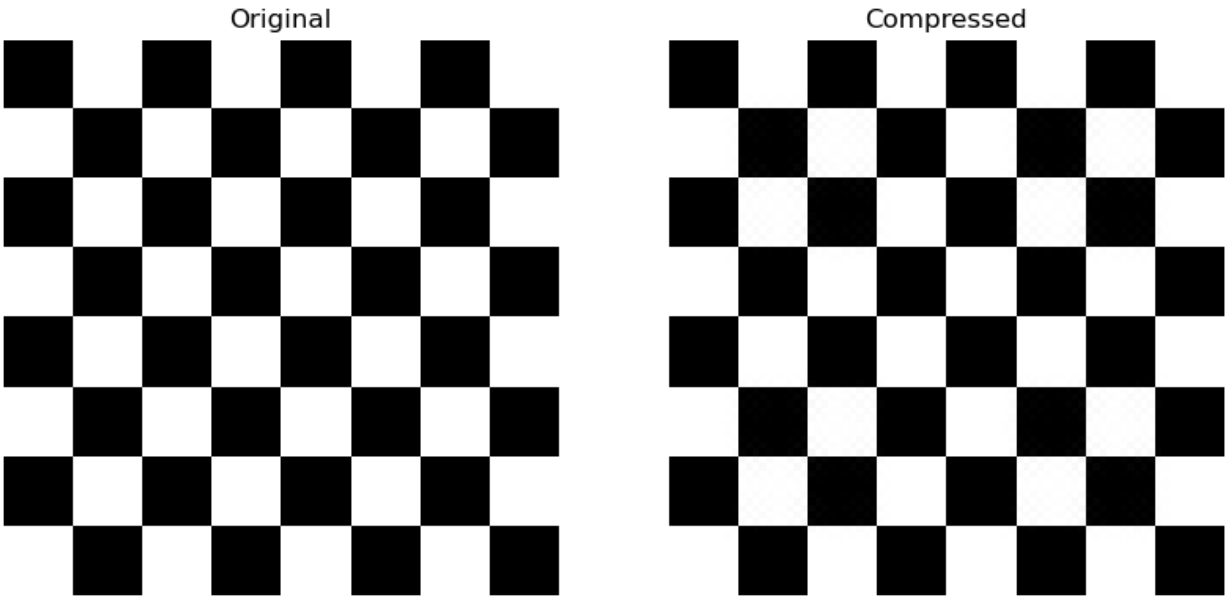


Figura 21: 80x80 f=8, d=14

Nelle immagini che vengono numerate da 16 a 21 si può notare che il fenomeno sopra citato è molto più evidente. Questo, come già detto, è dovuto al forte contrasto nell'immagine.

4 Conclusioni

Prima Parte La prima parte dell'analisi consisteva nel confrontare un'implementazione homemade della dct2, con quella di una libreria open source che facesse uso della Fast Fourier Transform.

I risultati come hanno confermato le supposizioni iniziali, per cui la nostra implementazione segue la tendenza di una curva rappresentata dalla funzione $\mathcal{O}(n^3)$, contro $\mathcal{O}(n^2 \log n)$ della FFT.

Gli stessi risultati sono stati ottenuti dalle rispettive inverse-dct2, per le quali l'analisi è stata spinta oltre, per analizzare se il calcolo delle matrici trasposte influisse sui tempi di esecuzione totali. Nonostante il leggero vantaggio delle dct2, i tempi restano comunque in linea con le attese e l'esecuzione è interamente dominato dal calcolo della Discrete Cosine Transform.

Seconda Parte La seconda parte del progetto è stata quella che più si avvicina ad un esempio di applicazione reale, pertanto i risultati ottenuti permettono maggiori spunti di riflessione, essendo valutabili ed interpretabili anche da utenti non esperti nel dominio di applicazione.

Non ci sono state problematiche relative all'implementazione della compressione di immagini in formato jpeg, che ha permesso di portare a termine quanto richiesto: il focus dell'analisi è stato quindi spostato sull'emergere del fenomeno di Gibbs al variare dei parametri di compressione.

Ne è risultato che tale problematica veniva fuori in particolar modo nell'esecuzione su immagini con alto contrasto, in particolare quelle in bianco e nero a scacchiera, di dimensioni molto piccole, che in ogni caso non rappresentano immagini realmente utilizzate all'atto pratico. Sulle altre immagini, quali *cathedral* e *bridge*, il fenomeno si è notato molto meno, dal momento che il contrasto nella scala di grigi risultava molto minore.
Resta da tenere in considerazione che le immagini su cui sono stati fatti i test non erano a colori, quindi non rappresentano un caso d'uso reale in tempi moderni, ma non sarebbero poi molte le migliorie da implementare per generalizzare il software in maniera tale che possa supportare anche immagini di uso quotidiano. Ci si può considerare quindi soddisfatti del risultato ottenuto.

Riferimenti bibliografici

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi:10.1038/s41592-019-0686-2.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi:10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

The pandas development team. pandas-dev/pandas: Pandas, February 2020. URL https://doi.org/10.5281/zenodo.3509134.

J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

Wikipedia contributors. Bmp file format — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=BMP_file_format&oldid=1087715076, 2022. [Online; accessed 26-June-2022].