# Metodi del Calcolo Scientifico - Progetto 2

Francesco Trolli
889039

June 5, 2025

## 1 Confronto delle prestazioni tra DCT2 implementata manualmente e versione ottimizzata (fast)

In questa parte della relazione è stata confrontata la dct implementata manualmente, come visto a lezione, e quella implementata dalla libreria `fft`.

Per fare il confronto sono stati generati array casuali $N \times N$ per diversi valori di $N$ (multipli di 12 fino a 124). Per ciascun array, si è misurato il tempo medio di esecuzione di:

- **DCT2 custom**, realizzata con due chiamate alla DCT1D lungo le colonne e poi lungo le righe.
- **DCT2 fast**, ottenuta usando `scipy.fftpack.dct` con tipo 2 e normalizzazione ortogonale.

| N | Custom DCT2 (s) | Fast DCT2 (s) |
|---|---|---|
| 4 | $5.69 \times 10^{-5}$ | $3.40 \times 10^{-5}$ |
| 16 | $2.96 \times 10^{-4}$ | $1.73 \times 10^{-5}$ |
| 28 | $9.04 \times 10^{-4}$ | $2.61 \times 10^{-5}$ |
| 40 | $1.70 \times 10^{-3}$ | $2.99 \times 10^{-5}$ |
| 52 | $3.15 \times 10^{-3}$ | $7.14 \times 10^{-5}$ |
| 64 | $3.95 \times 10^{-3}$ | $4.50 \times 10^{-5}$ |
| 76 | $6.39 \times 10^{-3}$ | $8.45 \times 10^{-5}$ |
| 88 | $7.75 \times 10^{-3}$ | $7.28 \times 10^{-5}$ |
| 100 | $1.11 \times 10^{-2}$ | $8.91 \times 10^{-5}$ |
| 112 | $1.26 \times 10^{-2}$ | $1.19 \times 10^{-4}$ |
| 124 | $1.59 \times 10^{-2}$ | $1.92 \times 10^{-4}$ |

Table 1: Tempi medi di esecuzione della DCT2 per varie dimensioni $N$.

## Analisi della complessità

La differenza nei tempi di esecuzione è spiegabile osservando la complessità computazionale delle due versioni.

### DCT2 Custom (manuale)

La DCT2 viene calcolata eseguendo due trasformate 1D:

1. Una DCT1D su ogni colonna della matrice $N \times N$.

2. Una DCT1D su ogni riga del risultato ottenuto.

Ogni DCT1D implementata manualmente su un vettore di lunghezza $N$ richiede $O(N^2)$ operazioni, poiché per ogni indice $u$ bisogna sommare $N$ prodotti (ciclo annidato).

Poiché vengono effettuate $N$ DCT1D sulle colonne e poi $N$ DCT1D sulle righe, il numero totale di operazioni è:

$$O(N) \times O(N^2) + O(N) \times O(N^2) = O(N^3)$$

### DCT2 Fast (libreria)

La versione ottimizzata utilizza algoritmi basati sulla Fast Fourier Transform (FFT), che riducono la complessità della DCT1D a $O(N \log N)$. Applicandola a tutte le colonne e le righe si ottiene:

$$O(N) \times O(N \log N) + O(N) \times O(N \log N) = O(N^2 \log N)$$

Inoltre, le librerie come `scipy.fftpack` sono scritte in linguaggi compilati (C/Fortran) rendendole molto più veloci anche a parità di complessità teorica.

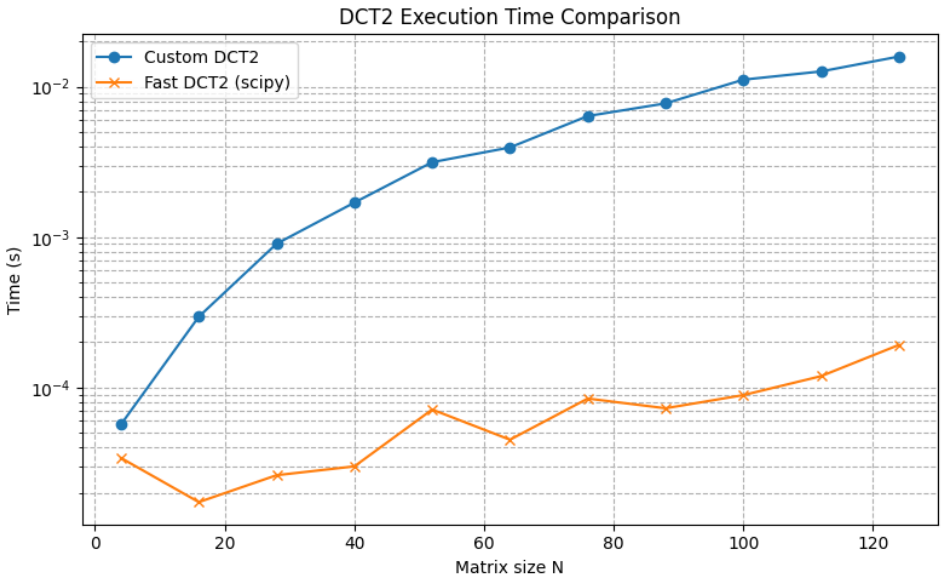


Figure 1: Confronto in scala semi-logaritmica tra i tempi di esecuzione delle due versioni della DCT2.

## Conclusioni

Il confronto mostra come l'implementazione manuale della DCT2 diventi rapidamente inefficiente per matrici di dimensione medio-grande, con tempi che crescono in modo cubico. Al contrario, la versione ottimizzata di `scipy` mantiene tempi estremamente contenuti grazie a una complessità inferiore e all'uso di tecniche avanzate di ottimizzazione.

Questi risultati confermano l'importanza di affidarsi a librerie ottimizzate per applicazioni pratiche che richiedono elevate prestazioni, come la compressione delle immagini con JPEG.

# 2 Compressione di immagini

## Introduzione

In questa seconda parte del progetto è stato realizzato un software in Python per simulare un algoritmo di compressione JPEG basato sulla trasformata discreta del coseno bidimensionale (DCT2), senza utilizzare una matrice di quantizzazione.

L'obiettivo è permettere all'utente di caricare un'immagine in scala di grigi, impostare due parametri $F$ (dimensione dei blocchi) e $d$ (soglia sulle frequenze), e osservare gli effetti della compressione sul contenuto dell'immagine.

## Interfaccia grafica

L'interfaccia è stata implementata con la libreria `tkinter`. L'utente può:

- Caricare un'immagine BMP in toni di grigio.
- Inserire la dimensione dei blocchi $F$, che deve essere un intero positivo.
- Inserire la soglia $d$, intero tra 0 e $2F-2$, che rappresenta il taglio diagonale sulle frequenze.
- Visualizzare l'immagine originale e quella compressa in una finestra di confronto.
- Salvare l'immagine compressa su file.

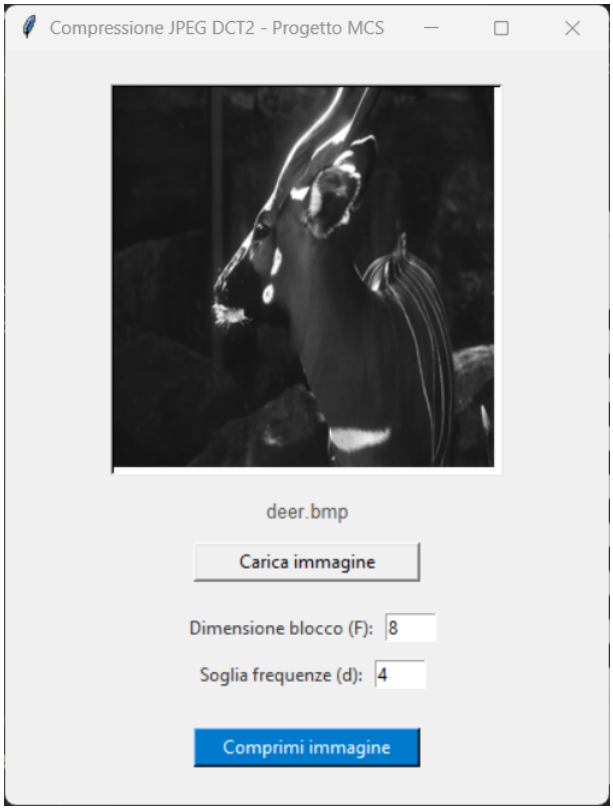


Figure 2: Mockup dell'interfaccia grafica realizzata in Python

## Algoritmo di compressione

L'immagine viene suddivisa in blocchi $F \times F$ (partendo dall'angolo in alto a sinistra, scartando eventuali avanzi). Per ciascun blocco vengono eseguite le seguenti operazioni:

1. Calcolo della DCT2 (con `scipy.fftpack.dct`, tipo 2 e normalizzazione ortogonale).
2. Eliminazione dei coefficienti $c_{k\ell}$ per cui $k + \ell \geq d$.
3. Applicazione della IDCT2 per ricostruire il blocco.
4. Clipping dei valori: tutto ciò che è $< 0$ viene portato a 0, tutto ciò che è $> 255$ viene portato a 255.

I blocchi compressi vengono poi ricomposti nell'ordine originale per formare l'immagine finale.

## Osservazioni e risultati

Il parametro $d$ controlla quanta parte delle alte frequenze viene eliminata:

- Se $d = 0$, tutti i coefficienti vengono azzerati (immagine completamente nera).
- Se $d = 2F - 2$, solo il coefficiente più alto viene eliminato.
- Valori intermedi permettono un buon compromesso tra qualità visiva e compressione.

A parità di immagine, un blocco $F$ più piccolo introduce meno perdita di dettaglio locale, ma può risultare in compressione meno efficace. Viceversa, blocchi grandi amplificano la perdita di frequenze alte e rendono l'immagine più sfocata.

(a) Immagine originale (b) Immagine compressa

Figure 3: Confronto tra immagine originale e compressa con $F = 8$, $d = 4$

Un aspetto importante emerso durante le prove è che immagini con variazioni molto brusche di intensità (come quelle composte da pattern a blocchi bianchi e neri alternati) risultano molto più difficili da comprimere efficacemente. In questi casi, infatti, la DCT2 produce coefficienti significativi anche nelle frequenze alte, che vengono tagliate dall'algoritmo quando $k + \ell \geq d$. Il risultato è una perdita evidente di dettaglio e contrasto, con artefatti visivi marcati (Fig. 4).

Al contrario, immagini con transizioni più dolci, come nel caso dell'immagine del *cervo* (Fig. 3), si prestano meglio alla compressione: anche con valori moderati di $d$, la qualità visiva rimane ottima. In particolare, con $F = 8$ e $d = 4$, la compressione ha mantenuto un buon livello di dettaglio pur riducendo significativamente le frequenze alte.

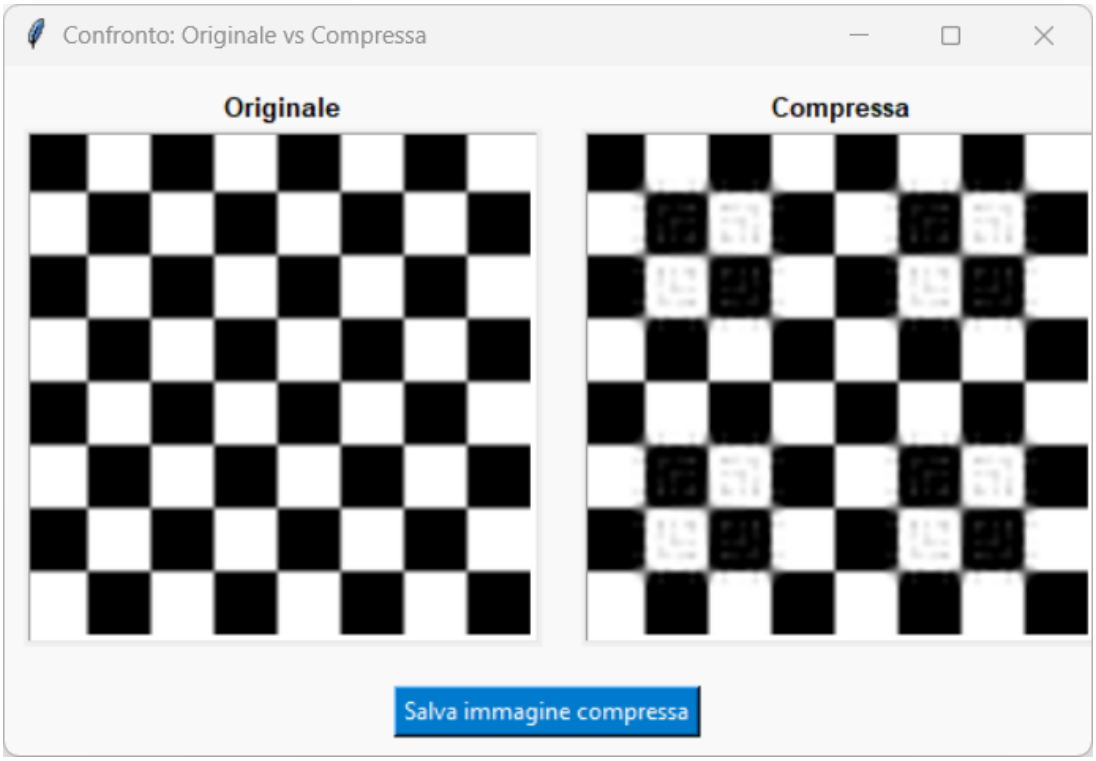


Figure 4: Confronto immagine scacchiera con valori $F = 8$, $d = 8$

## Tecnologie utilizzate

- **Linguaggio**: Python 3.10
- **GUI**: `tkinter`
- **Elaborazione immagini**: `Pillow`
- **Trasformate DCT**: `scipy.fftpack`

Il codice è strutturato in una classe `JPEGGui` per separare interfaccia e logica, ed è facilmente estendibile.

## Conclusioni

Il progetto ha mostrato come sia possibile realizzare un algoritmo di compressione in stile JPEG operando direttamente sulle frequenze ottenute dalla DCT2. È stato interessante osservare come i parametri influenzino la qualità dell'immagine compressa.

L'interfaccia sviluppata è funzionale, flessibile e utilizzabile su immagini qualsiasi in scala di grigi. Il progetto è stato sviluppato interamente con strumenti open-source.