Università degli Studi di Milano-Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di Laurea Magistrale in Informatica

2022/2023

Progetto 2

METODI DEL CALCOLO SCIENTIFICO

COMPRESSIONE DI IMMAGINI

Membri del team:

Matteo Rondena 847381

Luca Loddo 844529

Riccardo Moschi 856243

Indice

Parte 1 ..... *3*

1.1. Introduzione 3

1.2. Implementazione DCT/DCT2 3

1.3. Risultati 5

Parte 2 ..... *6*

1. Introduzione 6

2. Implementazione 6

2.1.1. Struttura 6

2.1.2. Interfaccia Grafica 6

3. Processo di compressione 10

4. Risultati 10

Parte 1

1.1. Introduzione

Si vuole confrontare i tempi d'esecuzione della DCT2 ottenuti usando la libreria SciPy, che implementa nativamente la funzione, con la DCT2 di nostra implementazione. In particolare, si richiede l'elaborazione di matrici quadrate N x N, con N crescente, generate casualmente. I risultati ottenuti vengono rappresentati sotto forma di grafico a scala semilogaritmica e tabella.

1.2. Implementazione DCT/DCT2

1.2.1. DCT

DCT (dall'inglese Discrete Cosine Transform) è una funzione che, dato in input un vettore di R^n, costruisce i coefficienti a_k, seguendo la formula:

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

```
def dctFromCode(matA):
    n = len(matA)
    output = np.zeros(n)
    for k in range(0, n):
        tmp = 0
        for i in range(0, n):
            tmp += matA[i] * np.cos(np.pi * k * (2 * i + 1) / (2 * n))
        if k == 0:
            alpha = m.sqrt(1 / (n))
        else:
            alpha = m.sqrt(2 / (n))
        output[k] = alpha * tmp
    return output
```

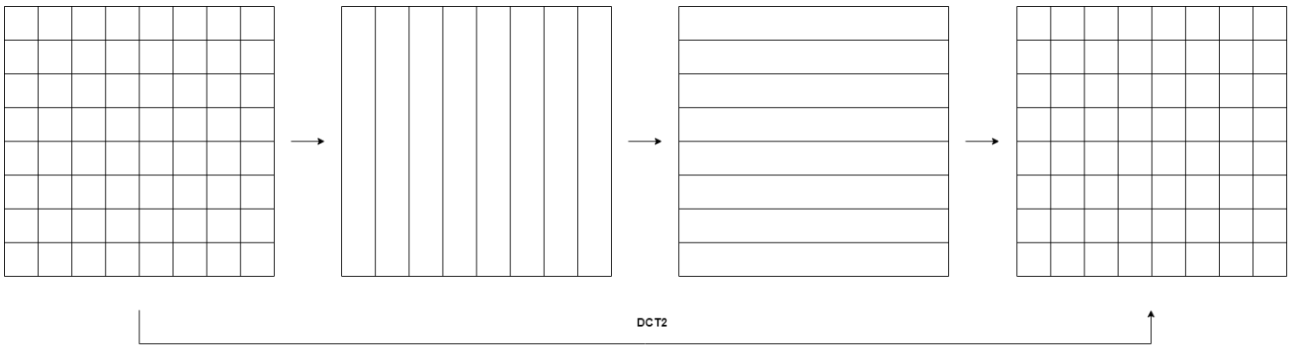
Figura 1: implementazione DCT

Nell'immagine soprastante, Figura 1, viene riportato il metodo per implementare la funzione DCT "customizzata".

Come è possibile notare, si è deciso di implementare lo scorrimento dei singoli valori dell'array quadrato attraverso 2 cicli for, sommando ogni volta il nuovo risultato dell'iterata alla variabile *tmp* (fino a fine riga) e moltiplicandola per il valore di *alpha*.
L'output della funzione restituirà il nuovo array.

1.2.2. DCT2

La trasformazione DCT2 equivale a fare 2 trasformazioni DCT, la prima applicata alle colonne della matrice e la seconda alle righe, come presentato di seguito.




```python
def myDct2(matrice):
    N=matrice.shape[0]
    M=matrice.shape[1]
    matOutput = np.empty([N, M])
    for j in range(M):
        matOutput[:, j] = dct2FromCode(matrice[:, j])

    for i in range(N):
        matOutput[i, :] = dct2FromCode(matOutput[i, :])



    return matOutput
```

Figura 2: implementazione nostra DCT2

Nell'immagine soprastante, Figura 2, riportiamo il metodo all'interno del codice che applica due volte, la prima sulle colonne e la seconda sulle righe, la funzione DCT di nostra implementazione già trattata nelle pagine precedenti. Il metodo prende in input un array quadrato N x N e restituisce un array quadrato N x N; ovviamente, non potendo assegnare lo stesso nome alle grandezze della matrice, abbiamo ridefinito le grandezze nel codice come N x M.

1.2.3. Esecuzione programma

L'esecuzione del programma avviene attraverso il metodo *createMatrix()* che prende in input i seguenti parametri:

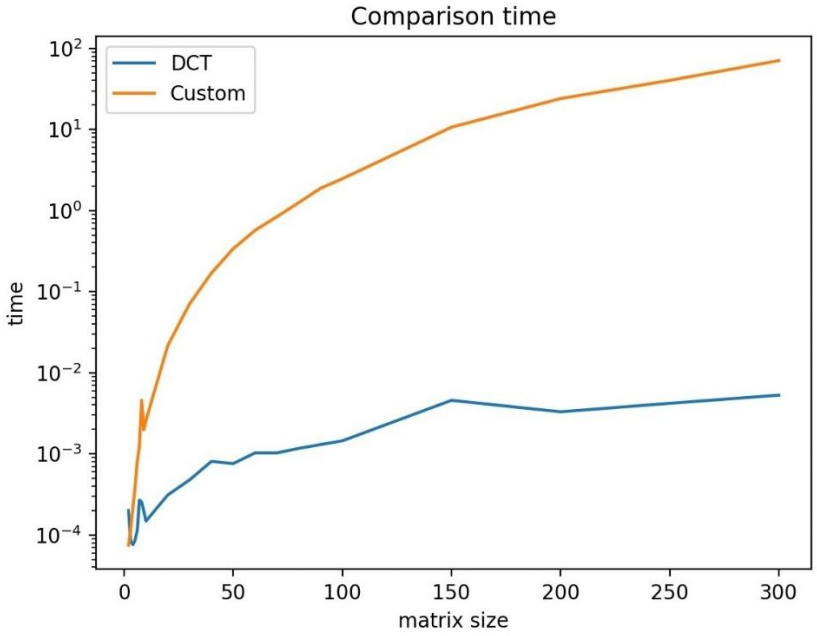
- ***fromStart:*** la dimensione di partenza con la quale generare le matrici
- ***N:*** la dimensione massima delle matrici generate

La complessità di tempo è stata valutata attraverso una serie di applicazioni della DCT2 a matrici di dimensioni crescenti.
La funzione genera matrici con dimensione *i* x *i* crescente da *fromStart* a 10 con uno step di 1. ovvero i viene incrementato di 1 ad ogni iterata. Superato *i* = 10, si passa ad uno step di 10. Superato, infine, *i* = 100, si passa ad uno step di 50, per arrivare a generare l'ultima matrice di dimensione *N* x N, specificato come input.

Il metodo tiene in memoria dei timer per ogni matrice prima e dopo la chiamata ai metodi che implementano DCT2 (custom e Scipy).
Per ogni funzione, quindi, generiamo una lista con i timer.
Successivamente, questi valori vengono utilizzati per creare un dataframe di riepilogo e il grafico.
Il tempo computazionale impiegato dal metodo custom (dctFromCode) è proporzionale a $O(n^3)$, mentre il tempo impiegato per il metodo fornito dalla libreria Scipy è proporzionale a $O(n^2 \log(n))$.

1.3. Risultati

Per provare la corretta esecuzione è stato verificato che la DCT2 utilizzasse lo scaling richiesto dal problema.



Come è possibile notare dal grafico, la DCT con la libreria Scipy è estremamente più veloce rispetto a quella implementata da noi, in quanto questa viene calcolata con una FFT (Fast Fourier Transform). Inoltre, data questa casistica, in cui si ha la creazione casuale di matrici sempre crescenti con *N*=300, è facile osservare un particolare comportamento iniziale, ovvero le due linee si invertono; infatti, la DCT custom alle prime iterazioni è più veloce rispetto a quella di Scipy.

Matrix Dimension	Time Custom	Time Default
2	0.000176	0.000402
3	0.000183	0.000200
4	0.000495	0.000449
5	0.000772	0.000209
6	0.000767	0.000099
7	0.001018	0.000126
8	0.001469	0.000120
9	0.002510	0.000436
10	0.002764	0.000145

Parte 2

1. Introduzione

Si vuole implementare un programma che permetta, a partire dalla scelta di un immagine bitmap in toni di grigio da file system, la compressione basata sulla DCT e la sua visualizzazione affiancata all'originale.

2. Implementazione

2.1.1. Struttura

Il progetto impiega varie librerie tra cui Numpy per la gestione delle strutture dati, quali matrici e vettori, Pandas per la gestione dei dataframe, Scipy e la sua funzione DCT2 implementata in modo nativo, Qt per la creazione di widgets e gestione dei layout nell'ambito dello sviluppo della GUI.

Il progetto si compone di diverse directories e files:

- ***images:*** directory contenente le immagini caricate da file system, al cui interno si possono già trovare le immagini rese disponibili dai docenti su e-learning
- ***imagesExported:*** directory contenente le immagini compresse, risultato dell'utilizzo dell'applicativo. In particolare, il nome del file esportato, o meglio dell'immagine compressa, si compone col nome dell'immagine concatenata con il valore di *F* e *D,* rispettivamente F utilizzata come dimensione dei blocchi *F* x *F* e *D* utilizzato come parametro per il taglio delle frequenze, inseriti dall'utente (ad esempio per l'immagine *bridge F*=100 e *D*=198 -> bridge_F100_D198.jpg)
- ***InterfaceGUI.py:*** file che implementa un'interfaccia grafica per l'utilizzo della libreria, in modo da fornire una buona user experience all'utente
- ***comprimeIMG.py:*** file contenente la vera e propria logica di compressione delle immagini e il relativo salvataggio nella cartella di output *imagesExported*

2.1.2. Interfaccia Grafica

Per quanto riguarda lo sviluppo dell'interfaccia grafica (GUI) abbiamo deciso di utilizzare la libreria **Qt** attraverso **PyQt5**, che è un porting delle suddette librerie per il linguaggio Python. In particolare, sono state importate:

- **QtCore**: classi di base
- **QtWidgets**: classi per la gestione dei Widgets
- **QtGui**: classi riguardanti la GUI e la sua gestione

Il codice che genera la GUI e gestisce la creazione di bottoni, box, layout e altri vari widget è interamente contenuto nel file denominato **InterfaceGUI.py**.

Nella classe sono stati implementati i seguenti metodi:

- *initializeGUI()*, metodo che genera e mostra l'interfaccia, tramite una griglia in cui vengono posizionati i vari widget
- *__init__(),* costruttore della classe
- *selectUpload()*, metodo che crea il layout all'interno della GUI con cui caricare la propria matrice da file system.
- *createOriginalImage(),* metodo che permette di visualizzare l'immagine caricata nella GUI
- *createFinalImage(),* metodo che permette di visualizzare l'immagine compressa nella GUI
- *getImage(),* metodo che svolge un controllo preliminare sull'immagine per verificare che sia in scala di grigi per mezzo del metodo *isGrayscale() e* la restituisce a *createOriginalImage()*
- *calculate(),* metodo che gestisce il path dell'immagine, che passa come parametro assieme ai valori di *F* e *D* per chiamare il metodo *solve(),* e la resituisce a *createFinalImage()*

Di seguito presentiamo la visualizzazione dell'interfaccia grafica in Figura 3 :

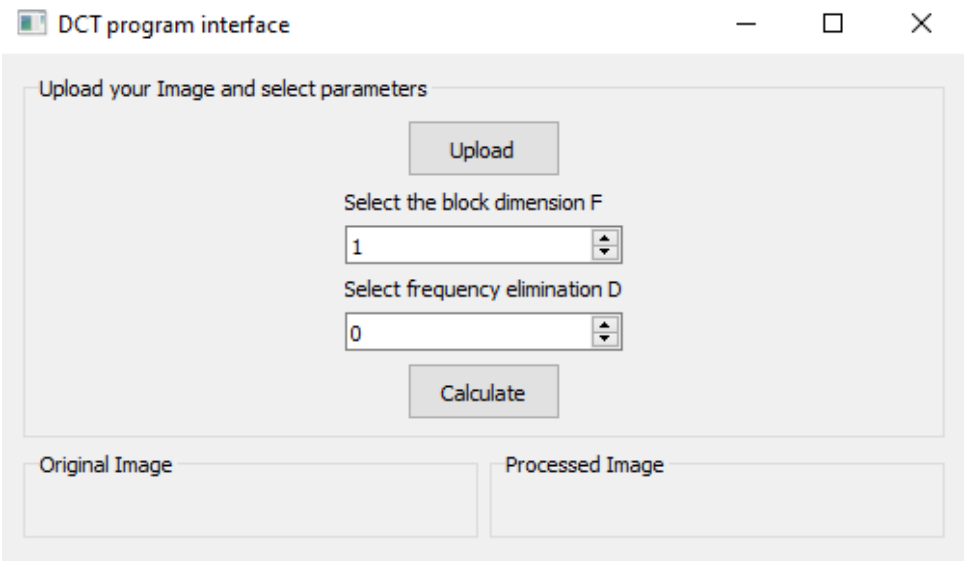


Figura 3: GUI dell'applicazione

L'interfaccia è costituita da una griglia 2 x 2, in cui i due blocchi superiori sono occupati da un singolo layout che visualizza tutti i bottoni e gli spinbox, che svolgono le seguenti funzioni:

- **Bottone Upload:** quando viene premuto il bottone Upload si apre una nuova finestra con la directory di file explorer per selezionare l'immagine da caricare
- **Spinbox di selezione di F:** con questo box è possibile incrementare/decrementare il valore di F, oppure direttamente digitare il valore desiderato all'interno del box da tastiera
- **Spinbox di selezione di D:** con questo box è possibile incrementare/decrementare il valore di D, oppure direttamente digitare il valore desiderato all'interno del box da tastiera
- **Bottone Calculate:** quando viene premuto il bottone Calculate si lancia la compressione dell'immagine caricata secondo i parametri.

È stato effettuato un controllo per quanto concerne i valori F e D, selezionabili per la compressione, dove il valore massimo del parametro D inseribile è pari a (2 x F) - 2.

Nei due blocchi inferiori, è presente in basso a sinistra l'immagine caricata da file system, si veda Figura 4, mentre, in basso a destra, l'immagine compressa dal metodo *solve(),* in Figura 5.

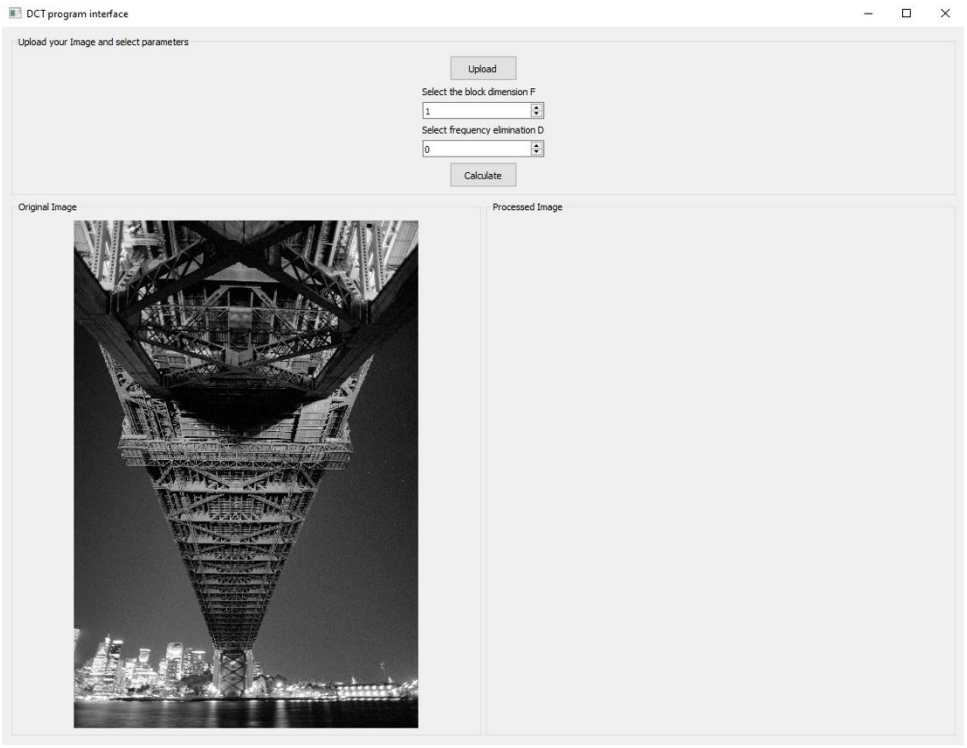


Figura 4: Caricamento dell'Immagine

Durante il caricamento dell'immagine, successivamente alla selezione della stessa, abbiamo svolto un controllo dall'interfaccia grafica che va ad analizzare le proprietà dell'immagine per verificare che sia una immagine *.bmp* in scala di grigi. Se così non fosse, l'interfaccia genera un messaggio d'errore che ricorda all'utente di caricare un'immagine nel formato corretto, ciò avviene anche in caso di mancato caricamento dell'immagine.

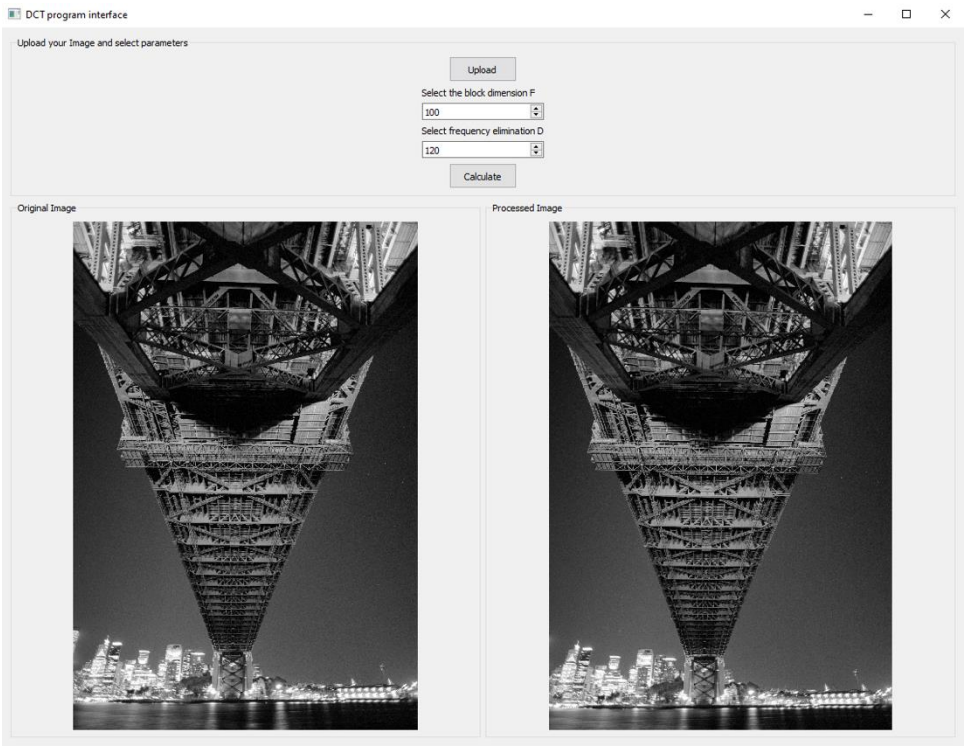


Figura 5: Compressione dell'immagine

In Figura 6 mostriamo il flowgraph che rappresenta il funzionamento della navigazione dell'applicativo.

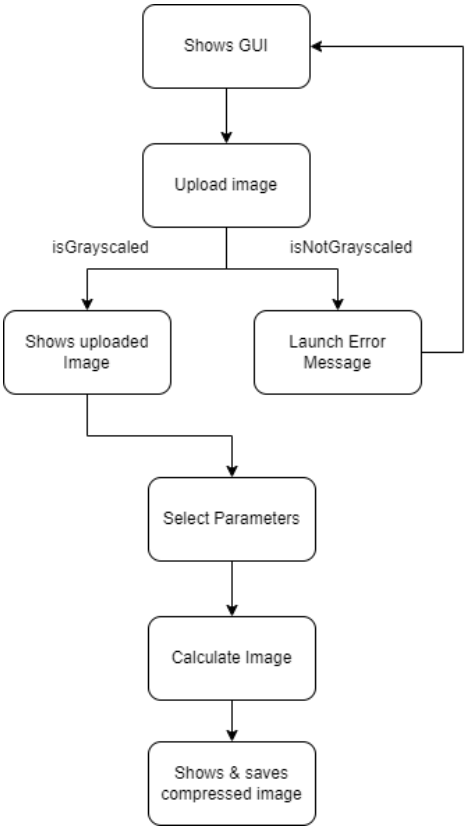


Figura 6: Use case flowgraph

3. Processo di compressione

Come già anticipato, il programma prende in input 3 valori:

- Immagine in formato bitmap
- F, ovvero la dimensione della matrice (*F* x *F*) per ogni blocco in cui verrà divisa l'immagine
- D, ovvero un parametro che permette di rimuovere parte delle informazioni di ogni blocco
    - Deve essere compreso tra 0 e 2F-2
    - Elimina i valori del singolo blocco se l'indice (riga + colonna) della cella è >=D

La compressione avviene secondo un processo ben preciso, e prevede in particolare diverse fasi:

- Splitting dell'immagine in blocchi *F* x *F*
- Per ogni blocco:
    - Si applica DCT2
    - Si svolge un'eliminazione della frequenza all'indice sse indice(riga) + indice(colonna)>=D
    - Si applica IDCT2
    - Normalizzazione dei valori tra 0 e 255
- Ricostruzione blocchi per composizione della matrice finale

4. Risultati

In questo capitolo mostriamo i risultati derivanti dalla compressione delle immagini caricate, in particolar modo andiamo ad analizzare la variazione del parametro *D*, ovvero la soglia di taglio delle frequenze per ogni blocco.

Eseguendo una compressione, mantenendo fisso il parametro *F*, notiamo che diminuendo il parametro *D* si riduce l'informazione. Tale situazione è evidente confrontando le Figure 7 e 8.

Figura 7: Immagine originale e immagine compressa con parametri F=132 e D=100

In Figura 7 notiamo che nonostante ci sia stato un taglio di più della metà delle frequenze, l'immagine risulta praticamente indistinguibile ad un esame preliminare; infatti, visivamente non

notiamo un estremo cambiamento nell'immagine, la quale sembra essere perfettamente identica ed indistinguibile.

Figura 8: Immagine originale e immagine compressa con parametri F=132 e D=10

In Figura 8, invece, è osservabile, ad un primo sguardo, una grande differenza dettata dalla scelta del parametro *D* molto bassa e ciò comporta l'eliminazione della maggior parte dei coefficienti in frequenza a destra della diagonale individuata dall'intero *D.*
Un parametro *D* basso, in relazione al valore di *F*, comporta una compressione massiccia dell'immagine e di conseguenza sarà particolarmente visibile il delinearsi dei blocchi *F* x *F* nell'immagine risultante, si veda immagine di sinistra in Figura 8.

Un'altra osservazione interessante, visibile nei risultati di compressione, è quella del fenomeno di Gibbs, che si nota molto bene nella Figura 9. Il fenomeno di Gibbs è il manifestarsi di aloni che seguono i contorni delle immagini in prossimità di un passaggio netto tra il bianco e il nero. Come possiamo vedere nell'immagine del cervo, riscontriamo la presenza di aloni che sono molto evidenti nella zona dove il volto e le corna del cervo sono illuminate e sono anche, allo stesso tempo, in forte contrasto con lo sfondo più scuro.

Figura 9: Immagine originale e immagine compressa con parametri F=132 e D=40

La propagazione del fenomeno di Gibbs viene attenuata, nella classica compressione JPEG, applicando le trasformate DCT2 e IDCT2 a blocchi di pixel di dimensioni 8 x 8, che sono solitamente molto più piccoli in proporzione alle dimensioni medie delle immagini.
In Figura 10 è riproposta una compressione con blocchi 8 x 8 che mostra nella pratica come l'effetto di Gibbs sia notevolmente attenuato rispetto all'immagine precedente.

Figura 10: Immagine compressa con blocchi 8 x 8, F=8 e D=4