

# **Metodi del calcolo scientifico**

## Compressione di immagini con DCT

Volpato Mattia 866316 \*  
Andreotti Stefano 851596 †

Appello di Giugno 2024

---

\*m.volpato4@campus.unimib.it  
†s.andreotti7@campus.unimib.it

# Contents

<b>1 Prima parte - DCT2 benchmark</b>	<b>3</b>
1.1 Obiettivo . . . . .	3
1.2 Libreria . . . . .	3
1.3 Implementazioni . . . . .	3
1.4 Benchmark . . . . .	5
1.4.1 Matrici e modalità di utilizzo . . . . .	5
1.4.2 Risultati . . . . .	5
<b>2 Seconda parte - Compressione di immagini</b>	<b>7</b>
2.1 Obiettivo . . . . .	7
2.2 Formato JPUG . . . . .	7
2.3 Architettura di sistema . . . . .	9
2.3.1 Architettura . . . . .	9
2.3.2 Utilizzo . . . . .	9
2.4 Esperimenti . . . . .	11
2.4.1 Blocchi di dimensione $F = 8$ . . . . .	11
2.4.2 Blocchi di dimensione $F = 32$ . . . . .	13

# 1 Prima parte - DCT2 benchmark

## 1.1 Obiettivo

In questa prima parte dell'elaborato si richiede di fornire una propria implementazione dell'algoritmo **DCT2** (**Discrete Cosine Transform 2-dimensional**) in ambiente **open source** e di confrontare le performance ottenute (in termini di **tempo di esecuzione**) con un'implementazione fornita da una libreria dell'ambiente open source.

A tal fine si è scelto di utilizzare il linguaggio di programmazione **python** e le librerie **scipy**, dedicata al **calcolo scientifico**, e **numpy**, descritte nel dettaglio nel paragrafo 1.2.

Successivamente, al fine di determinare l'impatto di implementazioni più o meno efficienti sui **tempi di esecuzione**, sono state fornite tre diverse implementazioni personali dell'algoritmo **DCT2**, trattate nel paragrafo 1.3.

Come indicato nelle specifiche, per semplicità verranno trattate solo matrici quadrate.

## 1.2 Libreria

La libreria **scipy** mette a disposizione il modulo **fftpack**, contenente una serie di funzioni che permettono di calcolare diverse varianti della *Discrete Cosine Transform*.

In particolare, nella stesura di questo elaborato è stata utilizzata unicamente la funzione *dctn*, che permette di specificare uno o più assi lungo i quali eseguire la *DCT*. Per ottenere lo scaling indicato nel progetto sono stati usati i parametri:

```
1   fft.dctn(x, axes=(0, 1), type=2, norm='ortho')
```

Implementando la *versione fast* dell'algoritmo, ci si aspetta di ottenere una complessità temporale dell'ordine  $O(n^2 \log(n))$ .

## 1.3 Implementazioni

Come già indicato, al fine di analizzare le differenze di perfomance derivanti dall'utilizzo di implementazioni che variano per efficienza, sono state fornite **tre diverse** versioni della *DCT* 'fatta in casa':

- **dct\_naive**: implementazione diretta della definizione di *DCT* in python, utilizzando due cicli *for* innestati

```
1  def dct_naive(x:np.array) -> np.array:
2      N = len(x)
3      sqrt_N, sqrt_2 = np.sqrt(N), np.sqrt(2)
4      dct_x = np.zeros(N, dtype=np.float64)
5      coeff = np.pi / (2 * N)
6
7      for k in range(N):
8          a_k = 0.0
9          coeff_k = coeff * k
10         for i, x_i in enumerate(x):
11             a_k += np.cos(coeff_k * (2 * i + 1)) * x_i
12         dct_x[k] = a_k / sqrt_N * sqrt_2
13
14     dct_x[0] /= sqrt_2
15
16     return dct_x
```

- **dct\_outer**: sfrutta al massimo le **operazioni tra matrici** ottimizzate messe a disposizione da **numpy**, pre-computando tutti i coefficienti

$$\cos\left(\pi k \frac{2i+1}{2N}\right), 0 \leq k, i \leq N-1$$

attraverso il **prodotto esterno**  $\otimes$  (riga 8)

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \otimes \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} = \underline{v_1} \otimes \underline{v_2} = \underline{v_1} \cdot \underline{v_2}^T = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \cdot [b_1 \ b_2 \ \dots \ b_m] = \begin{bmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 & \dots & a_1 \cdot b_m \\ a_2 \cdot b_1 & a_2 \cdot b_2 & \dots & a_2 \cdot b_m \\ \vdots & \vdots & \ddots & \vdots \\ a_n \cdot b_1 & a_n \cdot b_2 & \dots & a_n \cdot b_m \end{bmatrix} \quad (1)$$

e successivamente calcolando gli  $a_k$  tramite il **prodotto vettore-matrice** (riga 10)

```

1 def dct_outer(x:np.array) -> np.array:
2     N = len(x)
3     k = np.arange(N)
4     n = np.arange(N)
5
6     const_coeff = np.pi / (2 * N)
7     var_coeff = np.outer(k, 2 * n + 1)
8     transform_matrix = np.cos(const_coeff * var_coeff)
9
10    result = transform_matrix @ x
11
12    result[0] *= np.sqrt(1 / N)
13    result[1:] *= np.sqrt(2 / N)
14    return result

```

Questo approccio porta al dover salvare una matrice addizionale di dimensione  $(N \times N)$ , portando la complessità spaziale a  $\theta(N^2)$ .

- **dct\_cpp**: implementazione diretta della definizione della *DCT* in **C++**, che rispetto a **python** permette una gestione più efficiente delle **strutture dati** e presenta dei **cicli** molto più **performanti**. Codice disponibile in questa repository.

Per tutte e tre le versioni, la funzione *DCT2* è stata ottenuta applicando la *DCT* prima per righe e poi per colonne come segue:

```

1 def dct2(x:np.array, dct_functor:callable) -> np.array:
2     dct2_temp = np.apply_along_axis(dct_functor, axis=0, arr=x)
3     dct2_x = np.apply_along_axis(dct_functor, axis=1, arr=dct2_temp)
4     return dct2_x

```

Ottenendo:

- **dct2\_naive**:

```
1 dct2_naive = lambda x : dct2(x, dct_naive)
```

- **dct2\_outer**:

```
1 dct2_outer = lambda x : dct2(x, dct_outer)
```

- **dct2\_cpp**

- **dct2\_lib**, l'implementazione della libreria:

```
1 dct2_lib = lambda x : fft.dctn(x, axes=(0, 1), type=2, norm='ortho')
```

Infine, è stata proposta anche una variante di **dct2\_outer** che utilizza la stessa idea ma, anziché iterare su righe e colonne per applicare **dct\_outer**, sfrutta il **prodotto tra matrici** per computare la trasformata su entrambe le dimensioni:

```

1 def dct2_outer_no_call(x: np.array) -> np.array:
2     N, M = x.shape
3     n = np.arange(N)
4     m = np.arange(M)
5
6     const_coeff_N = np.pi / (2 * N)
7     const_coeff_M = np.pi / (2 * M)
8
9     var_coeff_N = np.outer(n_vec, 2 * n + 1)
10    var_coeff_M = np.outer(m_vec, 2 * m + 1)
11
12    transform_matrix = np.cos(const_coeff_N * var_coeff_N)
13    result = transform_matrix @ x
14
15    transform_matrix = np.cos(const_coeff_M * var_coeff_M)
16    result = result @ transform_matrix.T
17
18    result[0, :] *= np.sqrt(1 / N)
19    result[1:, :] *= np.sqrt(2 / N)
20    result[:, 0] *= np.sqrt(1 / M)
21    result[:, 1:] *= np.sqrt(2 / M)
22    return result

```

Anche in questo caso, la **complessità spaziale** sale a  $\theta(\max(N, M)^2)$ .

Per le tre implementazioni *custom* basate su iterazioni ci si aspetta una **complessità temporale**  $T(n) = \theta(n^3)$ ; al contrario, la complessità di **DCT2\_outer\_no\_call** dipende interamente dalle *ottimizzazioni* applicate da **numpy** al **prodotto tra matrici**, che ci si aspetta rendano l'andamento instabile. Tutto il codice è disponibile in questa repository.

## 1.4 Benchmark

### 1.4.1 Matrici e modalità di utilizzo

Il **benchmark** è stato eseguito utilizzando matrici  $N \times N$  di dimensione crescente, con il fine di individuare un *andamento asintotico regolare* nei **tempi di esecuzione**.

Partendo da una dimensione minima di  $25 \times 25$ , si sono ripetutamente raddoppiate le singole dimensioni (e quindi quadruplicate le dimensioni totali delle matrici), fermandosi quando le diverse implementazioni iniziavano a richiedere troppo tempo.

Tutte le matrici sono state inizializzate casualmente con valori interi compresi tra 0 e 255, in maniera da simulare uno dei principali scenari applicativi della *DCT2*; inoltre, al fine di ridurre la varianza dei risultati, tutte le esecuzioni sono state ripetute tre volte e se ne è considerato il **valore medio**.

### 1.4.2 Risultati

A seguire vengono riportati i risultati ottenuti, prima in forma tabellare (tabella 1) e poi come grafico (in scala logaritmica, figura 1); i **tempi** riportati *in rosso* nella tabella (e *tratteggiati* nel grafico) sono solo stimati<sup>1</sup>, utilizzando un **modello di regressione polinomiale** (per un polinomio di grado 3).

Dal grafico 1 si nota come i risultati della sperimentazione confermino l'attesa teorica di una **complessità temporale**  $T(n) = \theta(n^3)$  per le implementazioni *custom*, mentre l'implementazione *fast* della libreria e **dct2\_outer\_no\_call** (che sfrutta i **prodotti matriciali ottimizzati**) mostrano una capacità di scalare decisamente superiore, sebbene non sia immediato riconoscere l'andamento asintotico.

---

<sup>1</sup>I tempi di *dct2\_outer\_no\_call* non sono stati stimati in quanto, con ogni probabilità, la **complessità temporale** derivante dall'utilizzo di **prodotti matriciali ottimizzati** è inferiore a  $O(N^3)$ .

N	Times (s)				
	dct_naive	dct_outer	dct_cpp	dct_outer_no_call	dct_lib
25	0.0403	0.0014	0.0003	0.0001	0.0001
50	0.2530	0.0097	0.0037	0.0029	0.0002
100	1.9649	0.0646	0.0267	0.0006	0.0007
200	16.2444	0.3382	0.1820	0.0019	0.0015
400	133.90	2.2810	1.4267	0.0145	0.0033
800	1.09 · 10 <sup>3</sup>	15.673	11.636	0.0419	0.0083
1600	8.82 · 10 <sup>3</sup>	113.76	95.239	0.2369	0.0411
3200	7.09 · 10 <sup>4</sup>	860.55	772.96	1.7182	0.1867
6400	5.69 · 10 <sup>5</sup>	6.68 · 10 <sup>3</sup>	6.2 · 10 <sup>3</sup>	10.586	1.0082
12800	4.56 · 10 <sup>6</sup>	5.26 · 10 <sup>4</sup>	5.0 · 10 <sup>4</sup>	-	5.0009

Table 1: Risultato del benchmark (i tempi in rosso sono solo stimati)

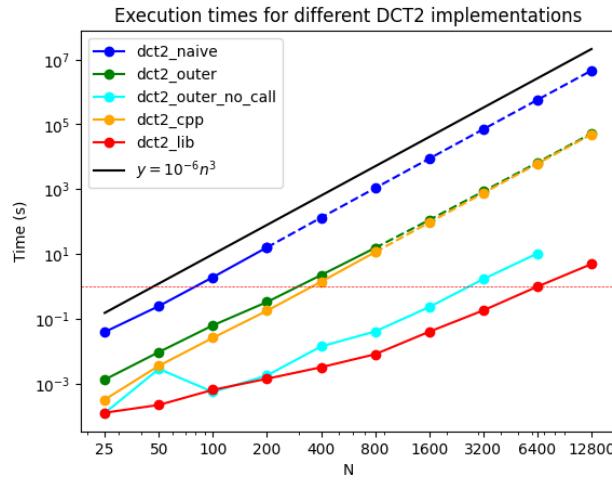


Figure 1: Risultato del benchmark su scala logaritmica (i tempi tratteggiati sono solo stimati)

Confrontando invece solo le tre versioni *custom*, risulta evidente che, a parità di complessità computazionale, le implementazioni *dct\_outer* e *dct\_cpp* siano nettamente superiori a *dct\_naive*, al punto da riuscire a trattare matrici 16 volte più grandi con tempi dagli stessi ordini di grandezza.

Inoltre, si evince come sia possibile ottenere ottime prestazioni da un linguaggio di programmazione *interpretato*, orientato alla *portabilità* e alla *semplicità di utilizzo* come **python**: nonostante i limiti dovuti alla sua struttura, è possibile raggiungere prestazioni molto vicine a quelle del C/C++, a patto di fare ampio utilizzo di librerie specializzate ed efficienti (e sviluppate in linguaggi performanti) come **numpy**.

## 2 Seconda parte - Compressione di immagini

### 2.1 Obiettivo

In questa seconda parte la richiesta è lo sviluppo di un piccolo *sistema software* che implementi una versione semplificata della *compressione con perdita di qualità JPEG*, facendo utilizzo della versione fast della **DCT2**.

Anche in questo caso si è scelto di utilizzare il linguaggio **python** insieme ad un'architettura **MVC (Model-View-Controller)**, descritta nel dettaglio nella sezione 2.3.

Inoltre, al fine di avvicinarsi il più possibile al vero formato **JPEG**, si è scelto di trattare sia immagini in **toni di grigi** che a colori (**RGB**) e di creare un proprio (banale) *formato di serializzazione* (sezione 2.2);

Infine, sono stati effettuati alcuni esperimenti su immagini in formato **bmp** di varie dimensioni (sezione 2.4).

Tutto il codice è disponibile in questa repository.

### 2.2 Formato JPUG

Come già detto, si è deciso di definire un formato personalizzato che andasse a simulare il formato **JPEG**, chiamato (in maniera fantasiosa) **JPUG**. Il formato salva banalmente in binario e in maniera sequenziale le tre componenti necessarie a ricostruire l'immagine compressa:

- il coefficiente  **$F$** , rappresentante la dimensione dei blocchi in cui è suddivisa l'immagine;
- il coefficiente  **$d$** , rappresentante la prima antidiagonale delle entrate da eliminare;
- il valore delle **entrate** (nella base dei coseni) da mantenere, che vengono linearizzate per righe in un **vettore monodimensionale**, come nel seguente esempio (per  $F = 5$  e  $d = 4$ ):

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & - \\ a_{21} & a_{22} & a_{23} & - & - \\ a_{31} & a_{32} & - & - & - \\ a_{41} & - & - & - & - \\ - & - & - & - & - \end{bmatrix} \implies [a_{11} \ a_{12} \ a_{13} \ a_{14} \ a_{21} \ a_{22} \ a_{23} \ a_{31} \ a_{32} \ a_{41}]$$

Il **numero  $n$  di entrate** della matrice **da mantenere (dimensione del vettore linearizzato)** è dato da:

- Per  $0 \leq d \leq F$ :

$$n = \sum_{i=0}^{d-1} i + 1 = \sum_{i=1}^d i = \frac{d(d+1)}{2} \quad (2)$$

- Per  $F < d \leq 2F - 1$ :

$$n = \frac{F(F+1)}{2} + \sum_{k=1}^{d-F} F - k \quad (3)$$

dove

$$\begin{aligned} \sum_{k=1}^{d-F} F - k &= \sum_{k=1}^{d-F} F - \sum_{k=1}^{d-F} k = \\ &= F(d-F) - \frac{(d-F)(d-F+1)}{2} = \\ &= Fd - F^2 - \frac{1}{2}d^2 + \frac{1}{2}Fd - \frac{1}{2}d + \frac{1}{2}Fd - \frac{1}{2}F^2 + \frac{1}{2}F = \\ &= -\frac{3}{2}F^2 + \frac{1}{2}F + 2Fd - \frac{1}{2}d^2 - \frac{1}{2}d \end{aligned}$$

e di conseguenza

$$\begin{aligned} n &= \frac{1}{2}F^2 + \frac{1}{2}F - \frac{3}{2}F^2 + \frac{1}{2}F + 2Fd - \frac{1}{2}d^2 - \frac{1}{2}d = \\ &= 2Fd - \frac{1}{2}d^2 - F^2 - \frac{1}{2}d + F \end{aligned}$$

Riassumendo:

$$n(F, d) = \begin{cases} \frac{d(d+1)}{2} & \text{se } 0 \leq d \leq F \\ 2Fd - \frac{1}{2}d^2 - F^2 - \frac{1}{2}d + F & \text{se } F < d \leq 2F - 1 \end{cases} \quad (4)$$

La percentuale di entrate salvate (o tasso di compressione)  $\delta$  è invece data da:

$$\delta(F, d) = \frac{F^2 - n}{F^2} = 1 - \frac{n}{F^2} \quad (5)$$

che vale

- $0 \leq d \leq F$ :

$$\delta(F, d) = 1 - \frac{d(d+1)}{2F^2} \quad (6)$$

- $F < d \leq 2F - 1$ :

$$\delta(F, d) = 1 - \frac{4Fd - d^2 - 2F^2 - d + 2F}{2F^2} = 2 - \frac{4Fd - d^2 - d + 2F}{2F^2} \quad (7)$$

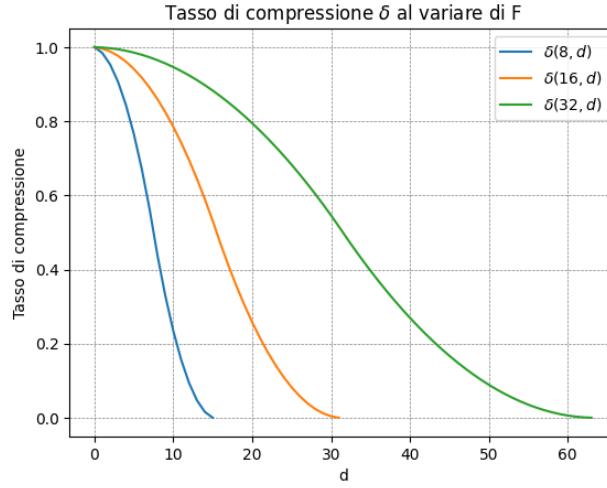


Figure 2: Tasso di compressione  $\delta$  per diversi valori di  $F$

Si noti che sia  $n$  sia  $\delta$  fanno riferimento al *numero di elementi* dei blocchi ma **non direttamente** alla *quantità di spazio effettivo (in byte)*: infatti, data l'assenza di utilizzo di una **matrice di quantizzazione**, questo formato 'naive' salva gli elementi dei vettori compressi come **numeri a virgola mobile**, che quindi occupano **2, 4 o 8 bytes** (a seconda della precisione richiesta), a differenza dei singoli **byte** delle immagini originali. Questo comporta che, per valori di  $d$  non abbastanza piccoli, l'effetto sia quello di **aumentare la dimensione dell'immagine** anziché diminuirla (nonostante ci sia comunque perdita di qualità).

Nel caso di immagini **RGB**, vengono compresse e salvate tutte e tre le matrici **R**, **G** e **B** in maniera indipendente.

## 2.3 Architettura di sistema

### 2.3.1 Architettura

Come **architettura di sistema** si è scelto di adottare un modello **MVC**, le cui componenti più importanti sono state riportate nel diagramma 4.

La parte più interessante, in cui avviene il processo di *compressione*, è contenuta nel package *model*, suddiviso in tre componenti:

- package *encoder*, contenente le classi che effettuano la compressione;
- package *serialization*, contenente le classi per la definizione del formato **JPUG**;
- classe *Parser*, che si occupa di caricare/salvare le immagini dal file system (in formato **BMP** o **JPUG**).

In particolare, nel package *encoder* è stata definita una classe base *Encoder* con la responsabilità di effettuare le operazioni di compressione e decompressione indicate nelle specifiche; due ulteriori classi (*L\_Encoder* e *RGB\_Encoder*) *specializzano* la classe *Encoder*, in maniera da trattare direttamente immagini del formato corrispondente.

Una struttura analoga è stata applicata anche al modulo **serialization**.

### 2.3.2 Utilizzo

È possibile utilizzare il programma in due diverse modalità:

- Interagendo con una semplice **CLI** (**Command Line Interface**, figura 3), dopo aver lanciato il programma *senza argomenti*:

```
1 python Main.py
```

```
PS C:\Users\Fox\OneDrive\Desktop\calc_scientifico\Jpug\jpub> python Main.py
-----
Active mode: RGB
Active parameters: F = 8, d = 8
-----
Choose an operation:
 0. Switch mode
 1. Change parameters
 2. Show an image
 3. Encode
 4. Decode
 5. Show statistics
 6. Exit
<
```

Figure 3: Command Line Interface

- Specificando i **parametri** (*percorso, F, d, modalità*) come *argomenti del programma*, seguendo la seguente sintassi (i parametri tra [] sono opzionali):

```
1 python Main.py path [F d [mode]]  
1 python Main.py path [mode]
```

Il parametro *mode* può assumere valori appartenenti a {*L, RGB*}.

Se non specificati, vengono utilizzati i seguenti valori default:

- *F* = 8
- *d* = 8

- Mode = RGB

Per l'operazione di *decodifica*, è necessario indicare soltanto il *percorso* del file:

1            `python Main.py path`

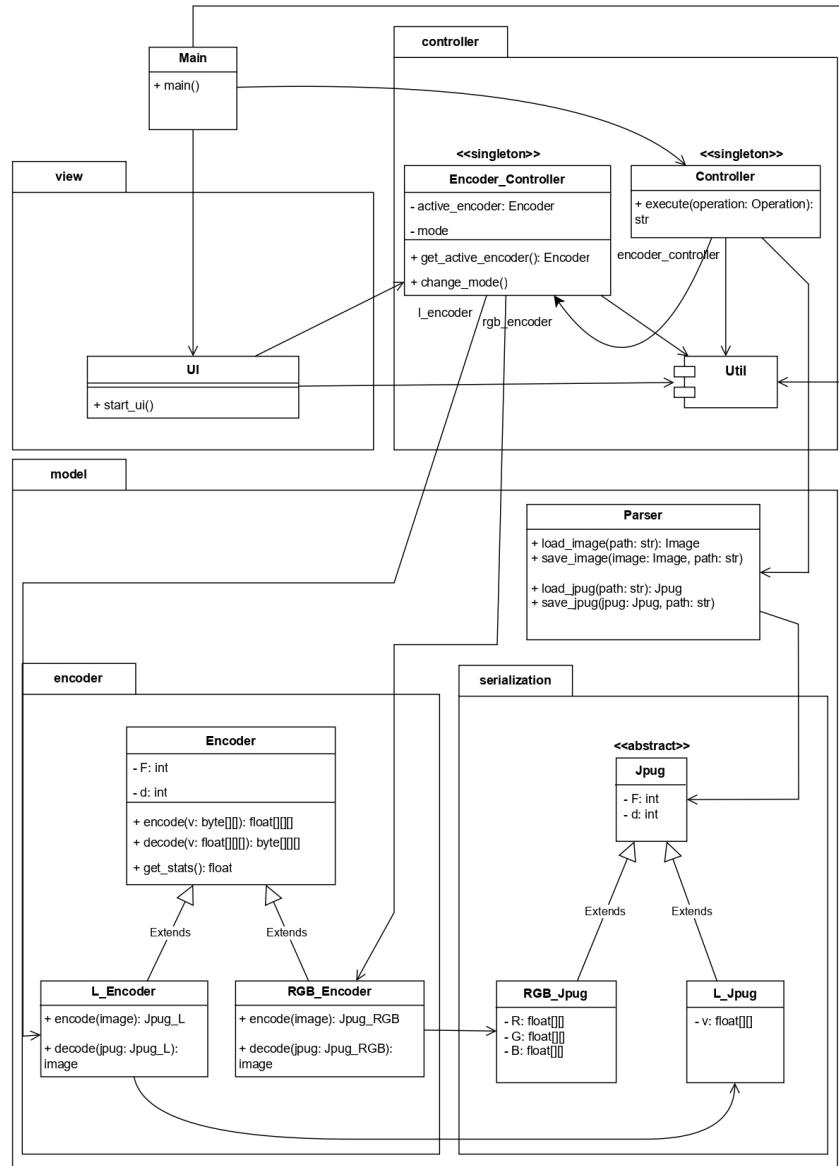


Figure 4: Diagramma delle classi

## 2.4 Esperimenti

In questa sezione si riportano alcuni esperimenti<sup>2</sup> effettuati per testare la compressione proposta. In particolare, si sono considerati blocchi di dimensione  $F \in \{8, 32\}$  e, fissato il valore, si è fatto variare  $d$  in maniera da ottenere diversi **tassi di compressione**. Di seguito i risultati ottenuti.

### 2.4.1 Blocchi di dimensione $F = 8$

Fissando la dimensione dei blocchi a  $F = 8$  (figura 9), si ottiene un'ottima qualità di immagine per un **tasso di compressione**  $\delta$  di poco inferiore a 0.5, al punto che è difficile individuare differenze con l'immagine originale anche entrando nei dettagli (figura 11).

Aumentando  $\delta$  a poco più di 0.75, si continua a ottenere una buona qualità sull'immagine completa (figura 7), sebbene sia possibile individuare i primi segni di *perdita di informazione* zoomando su regioni critiche (figura 12).

Infine, volendo esagerare e portando il **tasso di compressione** a 0.9, risultano evidenti i blocchi in cui è stata suddivisa l'immagine compressa (figura 13).



Immagine originale



$F = 8, d = 8, \delta = 0.4375$



$F = 8, d = 5, \delta = 0.7656$



$F = 8, d = 3, \delta = 0.9062$

Figure 9: Esempi di compressione per blocchi di dimensione  $F = 8$

<sup>2</sup>Immagini raffiguranti *Richard Hendricks*, protagonista della serie tv *Silicon Valley*.



Dettaglio dell'immagine originale



$F = 8, d = 8, \delta = 0.4375$



$F = 8, d = 5, \delta = 0.7656$



$F = 8, d = 3, \delta = 0.9062$

Figure 14: Dettagli di compressione per blocchi di dimensione  $F = 8$

### 2.4.2 Blocchi di dimensione $F = 32$

Anche fissando  $F = 32$  (figura 19), si ottengono risultati molto simili a quelli ottenuti per  $F = 8$ :

- per  $\delta \approx 0.5$ , l'immagine compressa presenta un'ottima qualità, anche nei dettagli (figura 21);
- portando il **tasso di compressione** a circa 0.75, si iniziano a individuare problematiche nei dettagli dell'immagine, pur non avendo ancora evidenza dei blocchi (figura 22);
- infine, per  $\delta \approx 0.9$ , diventano evidenti i blocchi in cui è stata suddivisa l'immagine (figura 23).



Immagine originale



$F = 32, d = 32, \delta = 0.4844$



$F = 32, d = 22, \delta = 0.7529$



$F = 32, d = 14, \delta = 0.8975$

Figure 19: Esempi di compressione per blocchi di dimensione  $F = 32$



Dettaglio dell'immagine originale



$F = 32, d = 32, \delta = 0.4844$



$F = 32, d = 22, \delta = 0.7529$



$F = 32, d = 14, \delta = 0.8975$

Figure 24: Dettagli di compressione per blocchi di dimensione  $F = 32$