

Compressione di immagini tramite DCT

Confronto implementazioni e creazione applicazione grafica

<https://github.com/nicoripa/MCS-Project/>

Lorenzo Roviada, Nicolò Ripamonti
l.rovida1@campus.unimib.it, n.ripamonti@campus.unimib.it
817151, 816171

*Dipartimento di Informatica, Sistemi e Comunicazione, Università degli Studi di
Milano-Bicocca, Milano, Italia*

Abstract

Lo scopo di questo progetto è quello di utilizzare l'implementazione dell'operazione matematica denominata **DCT2** in un ambiente *open source* e di studiare gli effetti di un algoritmo di compressione di tipo jpeg (senza utilizzare una matrice di quantizzazione) su delle immagini in toni di grigio di formato *bitmap*. Il progetto si divide in due macro parti:

Parte 1 Implementare la DCT2 in un ambiente open source a scelta e confrontare i tempi di esecuzione con la DCT2 ottenuta usando la libreria dell'ambiente utilizzato.
In particolare, bisogna utilizzare array quadrati $N \times N$ con N crescente e rappresentare su un grafico i tempi di esecuzione dei due algoritmi.

Parte 2 Scrivere un software in grado di far scegliere all'utente un'immagine in formato bitmap e successivamente comprimere tale immagine utilizzando la DCT2, tagliare le frequenze che l'utente sceglie di eliminare, utilizzare l'inversa della DCT e far visualizzare a schermo l'immagine originale con quella ottenuta dopo aver modificato le frequenze.

Introduzione

Discrete Cosine Transform

La trasformata discreta del coseno o DCT, è la più diffusa funzione che provvede alla compressione spaziale, capace di rilevare le variazioni di informazione tra un'area e quella contigua di un'immagine digitale trascurando le ripetizioni.

È una trasformata simile alla trasformata discreta di Fourier (DFT), ma fa uso solo di numeri reali.

La variante più comune della trasformata discreta del coseno è la DCT tipo II che è spesso chiamata semplicemente DCT; la sua inversa, la DCT tipo III è, in corrispondenza, chiamata spesso DCT inversa o IDCT.

La DCT, e in particolare la DCT-II, è spesso usata nell'elaborazione dei segnali e delle immagini, specialmente per la compressione con perdita (compressione di tipo **lossy**). L'algoritmo JPEG è basato sulla Trasformata discreta del coseno bidimensionale (DCT2), che viene applicata su blocchi di 8x8 pixel, i cui risultati sono poi quantizzati e compressi con tecniche basate sull'entropia (come la Codifica di Huffman o la Codifica aritmetica).

La formula che descrive il funzionamento della DCT bidimensionale è la seguente:

$$c_{kl} = a_{kl} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} f_{ij} \cos \left(k\pi \frac{2i+1}{2N} \right) \cos \left(l\pi \frac{2j+1}{2M} \right)$$

Dove $a_{kl} = a_k^N a_l^M$:

$$a_{00} = \frac{1}{\sqrt{NM}}, \text{ e } a_{k0} = a_{0l} = \sqrt{\frac{2}{NM}}, \text{ } a_{kl} = \frac{2}{\sqrt{NM}}, \text{ } k, l \geq 1.$$

Mentre f_{ij} sono gli indici bidimensionali:

$$\mathbf{f} = (f_{ij}), \text{ } i = 0, \dots, N-1, j = 0, \dots, M-1.$$

Formato bitmap

Windows bitmap è un formato dati utilizzato per la rappresentazione di immagini **raster** sui sistemi operativi Microsoft Windows. Noto soprattutto come formato di file, fu introdotto con Windows 3.0 nel 1990. Le bitmap, come sono comunemente chiamati i file d'immagine di questo tipo, hanno generalmente l'estensione **.bmp**.

Sono state sviluppate tre versioni del formato bitmap. La prima e più comunemente utilizzata è la versione 3: non esistono versioni precedenti. Le versioni successive 4 e 5 si incontrano piuttosto raramente.

Il formato di file Windows bitmap nella versione 3 permette operazioni di lettura e scrittura molto veloci e senza perdita di qualità, ma richiede generalmente una maggior quantità di memoria rispetto ad altri formati analoghi.

Le immagini bitmap possono avere una profondità di 1, 4, 8, 16, 24 o 32 bit per pixel. Le bitmap con 1, 4 e 8 bit contengono una tavolozza per la conversione dei (rispettivamente 2, 16 e 256) possibili indici numerici nei rispettivi colori. Nelle immagini con profondità più alta il colore non è indicizzato bensì codificato direttamente nelle sue componenti cromatiche RGB; con 16 o 32 bit per pixel alcuni bit possono rimanere inutilizzati.

Nel caso in esame le immagini in formato *.bmp* sono state scelte con colori in scala di grigi, ovvero immagini con una profondità di 8 bit. Tali bit servono per convertire gli indici numerici nei rispettivi bit, in questo caso da 0, che identifica il colore nero, a 255, che identifica il colore bianco. Tutti i valori tra l'1 e il 254 sono tutte le possibili varianti di grigio.

Calcolatore utilizzato

Per effettuare i calcoli sugli script è stato utilizzato un MacBook Pro 2016 con le seguenti caratteristiche hardware:

Processore 2 GHz Intel Core i5 dual-core

Architettura CPU x64

Memoria 8 GB 1867 MHz LPDDR3

Scheda grafica Intel Iris Graphics 540 1536 MB

Ambiente di sviluppo

L'ambiente di sviluppo scelto per il progetto, sia per la parte 1 che per la parte 2, è stato **Python**.

Versione: 3.7.3

Referenze: Sito - Documentazione



Python è un linguaggio di programmazione di più "alto livello" rispetto alla maggior parte degli altri linguaggi. È orientato a oggetti, ma non in maniera ferrea come ad esempio Java; adatto, tra gli altri usi, a sviluppare applicazioni distribuite, scripting, computazione numerica e system testing.

È un linguaggio multi-paradigma che ha tra i principali obiettivi: dinamicità, semplicità e flessibilità. Supporta il paradigma object oriented, la programmazione strutturata e molte caratteristiche di programmazione funzionale e riflessione.

Le caratteristiche più immediatamente riconoscibili di Python sono le variabili non tipizzate e l'uso dell'indentazione per la definizione delle specifiche.

Librerie

Nello specifico sono state utilizzate diverse librerie sia per l'implementazione del codice della parte 1 del progetto, sia per la parte 2.

Esse nello specifico sono:

TkInter Sito - Versione 3.8.3

È una libreria che permette di creare interfacce grafiche nella programmazione con Python, molto conosciuta e utilizzata per la sua leggerezza e stabilità.

SciPy Sito - Versione 1.19.0

È una libreria open source di algoritmi e strumenti matematici. Contiene moduli per l'ottimizzazione, per l'algebra lineare, elaborazione di segnali ed immagini e altri strumenti comuni nelle scienze e nell'ingegneria. Trova utilizzo in quei programmatori che usano anche MATLAB. All'interno di SciPy sono presenti diversi pacchetti tra cui **fftpack**, che contiene tutti gli algoritmi della trasformata discreta di Fourier e anche classi per l'implementazione della trasformata discreta del coseno (**DCT**);

NumPy Sito - Versione 1.18.4

È un pacchetto per l'elaborazione scientifica con Python. In particolare questo pacchetto è stato utilizzato per effettuare operazioni su array bidimensionali;

Matplotlib Sito - Versione 3.2.1

È una libreria completa per la creazione di visualizzazioni statiche, animate e interattive. Nel nostro interesse è stata usata per stampare le immagini nella conclusione dello script della seconda parte del progetto;

Random Sito - Versione 3.0

Questo modulo implementa generatori di numeri pseudo-casuali per varie distribuzioni. È stata utilizzata questa libreria per la generazione di matrici casuali per lo studio dei tempi di esecuzione delle DCT nella prima parte del progetto;

Time Sito - Versione 3.7

Questo modulo offre varie funzioni correlate al tempo. Utilizzata per la misurazione dei tempi di esecuzione delle CDT nella prima parte del progetto;

CV2 Sito - Versione 4.3.0

È un modulo facente parte della libreria OpenCV. CV2 è stata utilizzata all'interno dell'interfaccia grafica per poter far scegliere all'utente il file .bmp.

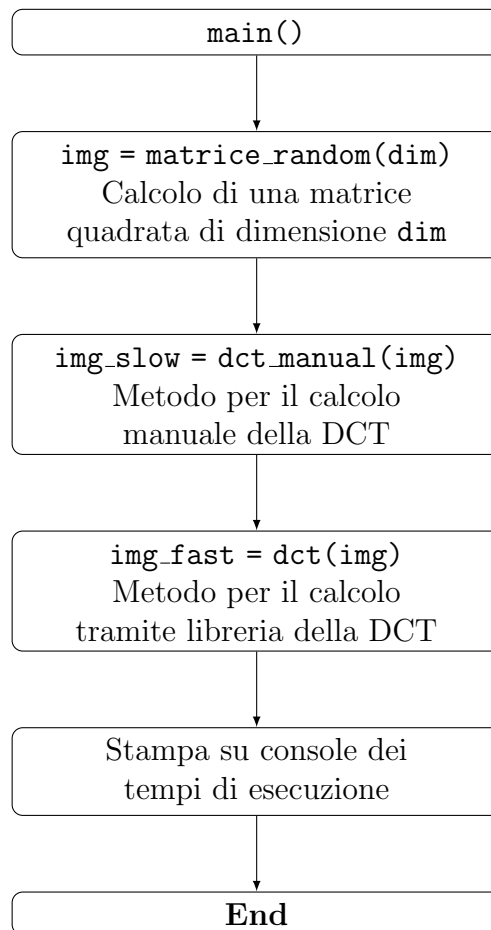
Confronto implementazioni DCT

Parte 1

Nella prima parte del progetto il compito è quello di creare un algoritmo di compressione del tipo DCT e confrontare i tempi di esecuzione con un algoritmo DCT implementato da una libreria (verrà utilizzata Scipy).

Diagramma

Viene presentato un diagramma a blocchi che riassume lo svolgimento dell'elaborazione.



Implementazione

Di seguito viene spiegato approfonditamente come opera lo script.

1. Nella funzione `main()` viene implementato un ciclo `while` dove avvengono tutte le operazioni. È stato scelto questo metodo per poter implementare uno script che faccia in automatico tutte le operazioni su matrici quadrate di dimensione sempre maggiore, ovvero la dimensione è di $100 \cdot n$, dove $n = 1, 2, \dots, 10$;

```
dim = 100
while (dim <= 1000):
    img = matrice_random(dim)

    time1 = time.time()
    dct_fftpack_2d(img)
    time2 = time.time()

    tempi_scipy.append(time2 - time1)

    time1 = time.time()
    dct_manual_2d(img)
    time2 = time.time()

    tempi_manual.append(time2 - time1)

    dimensioni.append(dim)

    dim += 100
```

2. Viene chiamata la funzione `matrice_random()` che crea un array bidimensionale di valori random compresi tra 0 e 255, ovvero i valori che corrispondono alla scala di grigi in formato bitmap. Questa funzione prende in ingresso il valore che corrisponde alla dimensione che si vuole dare alla matrice;

```
def matrice_random(dim):
    img = np.arange(dim * dim).reshape(dim, dim)

    m = img.shape[0]

    for i in range(0, m):
        for j in range(0, m):
            img[i][j] = random.randrange(256)

    return img
```

3. Viene calcolata la DCT della libreria Scipy e i tempi di esecuzione;

```
def dct_fftpack_2d(x):  
    return dct(np.transpose(dct(np.transpose(x), type =  
2, norm = 'ortho')), \  
                type = 2, norm='ortho')
```

4. Viene calcolata la DCT manuale e i tempi di esecuzione. La DCT viene implementata come segue;

```
def dct_manual_2d(x):  
    return dct_manual(np.transpose(dct_manual(np.  
transpose(x))))  
  
def dct_manual(x):  
    N = len(x)  
  
    print(type(x[0]))  
    c = []  
  
    for k in range(0, N):  
        somma = 0  
        for i in range(0, N):  
            somma = somma + (x[i] * math.cos(math.pi * k  
* (2 * i + 1) / (2 * N)))  
  
        if k == 0:  
            alpha = math.sqrt(1 / N)  
        else:  
            alpha = math.sqrt(2 / N)  
  
        c.append(somma * alpha)  
  
    return c
```

5. Vengono stampati in console tutti i tempi di esecuzione delle due DCT per ogni dimensione dell'array bidimensionale e viene creato un semplice plot;

```
plot(dimensioni, tempi_manual, tempi_scipy)  
print('Tempi Scipy: ')  
print(tempi_scipy)  
print('Tempi manual: ')  
print(tempi_manual)
```


Test e risultati

I risultati della prima parte del progetto riguardano il confronto dei tempi d'esecuzione della DCT2. In particolare il confronto è stato fatto tra una DCT implementata a mano e la funzione DCT del pacchetto `scipy.fftpack`.

Gli array bidimensionali utilizzati di dimensione $N \times N$ sono stati creati in maniera casuale e sono state scelte dimensioni che variano da 100×100 a 1000×1000 . Non sono state scelte dimensioni maggiori, sia per motivi di eccessivo tempo di calcolo, sia perché un'array 1000×1000 rispecchia un'immagine di 1000×1000 pixel, definizione già abbastanza alta, paragonabile in termini di numero di pixel al formato HD a 720p.

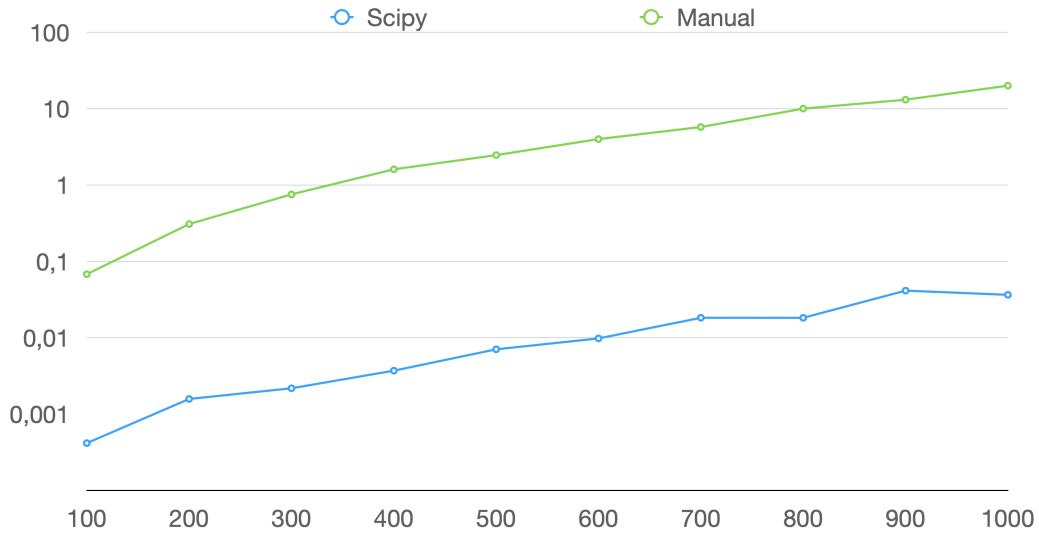
Durante l'implementazione abbiamo prestato molta attenzione al fatto che il nostro algoritmo venisse implementato correttamente confrontando sempre i risultati degli array dopo il calcolo di entrambe le DCT.

In particolare è stata effettuata una verifica basata su questo risultato (esatto per ipotesi):

231	32	233	161	24	71	140	245
↓							
4.01e+02	6.60e+00	1.09e+02	-1.12e+02	6.54e+01	1.21e+02	1.16e+02	
			2.88e+01				

Il programma calcola la DCT in modo corretto (assumiamo come 'accettabili' differenze con la soluzione dell'ordine di $0.01e+02$, ovvero 1):

```
Test 1D - Implementazione manuale:
4.02e+02
6.60e+00
1.09e+02
-1.13e+02
6.54e+01
1.22e+02
1.17e+02
2.88e+01
```



Nella figura sopra si possono notare come i tempi della DCT implementata a mano (`dct_manual()`) siano nettamente superiori rispetto alla DCT implementata in `scipy.fftpack.dct`. Questo è del tutto normale in quanto le librerie sono sicuramente implementate meglio e più efficienti.

Nella figura sotto sono riportati in una tabella tutti i tempi di esecuzione della DCT.

		Dimensione array bidimensionale [Pixel]									
		100	200	300	400	500	600	700	800	900	1000
Tempo [Sec]	Scipy	0,00041484	0,00157308	0,00217103	0,0036931	0,00703287	0,00976991	0,01821494	0,01818513	0,04131293	0,03638101
	Manual	0,06779313	0,30789494	0,75378012	1,6057541	2,46435403	3,99684786	5,74928379	10,0226428	13,1171091	20,0206999

Si ricorda che i tempi di un algoritmo per DCT2 implementato da una libreria sono di circa $O(N^2 \log N)$, mentre i tempi di esecuzione di una DCT2 manuale si aggirano intorno a $O(N^3)$, dove con N si intende ovviamente la dimensione della matrice.

Detto ciò, i risultati finali dei tempi rispecchiamo a pieno tale regola, quindi possiamo tenerci soddisfatti circa la prima parte del progetto.

GUI per la compressione

Parte 2

Come spiegato all'inizio dell'elaborato, la seconda parte del progetto si occupa di implementare un interfaccia grafica che permetta all'utente di effettuare una compressione di un file .bmp

Come prima cosa, il programma si preoccupa di fare scegliere all'utente un immagine .bmp e successivamente di scegliere due valori, F e d , che saranno il fulcro di tutta l'implementazione del programma.

In particolare, i valori F e d scelti dall'utente servono per l'elaborazione e la compressione dell'immagine nel seguente modo:

F è un intero che verrà utilizzato per la divisione in blocchi dell'immagine.

Tali blocchi avranno dimensione $F \times F$ ed è proprio su di essi che verrà calcolata la DCT.

d è il valore intero che serve per eliminare le frequenze all'interno dei blocchi una volta calcolata la DCT. Tale valore deve essere un numero intero compreso tra 0 e $2F - 2$, perché le frequenze vengono eliminate sui blocchi $F \times F$, dove il valore che ha indici k ed l del blocco viene eliminato se $k + l \geq d$.

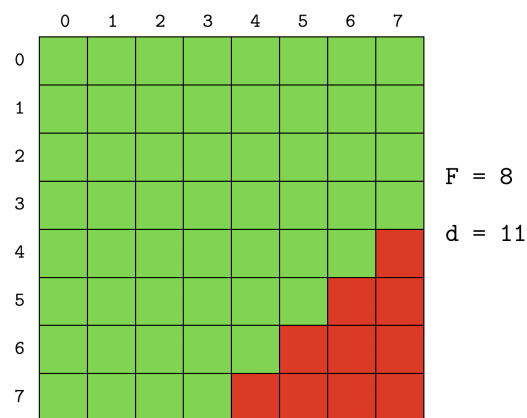
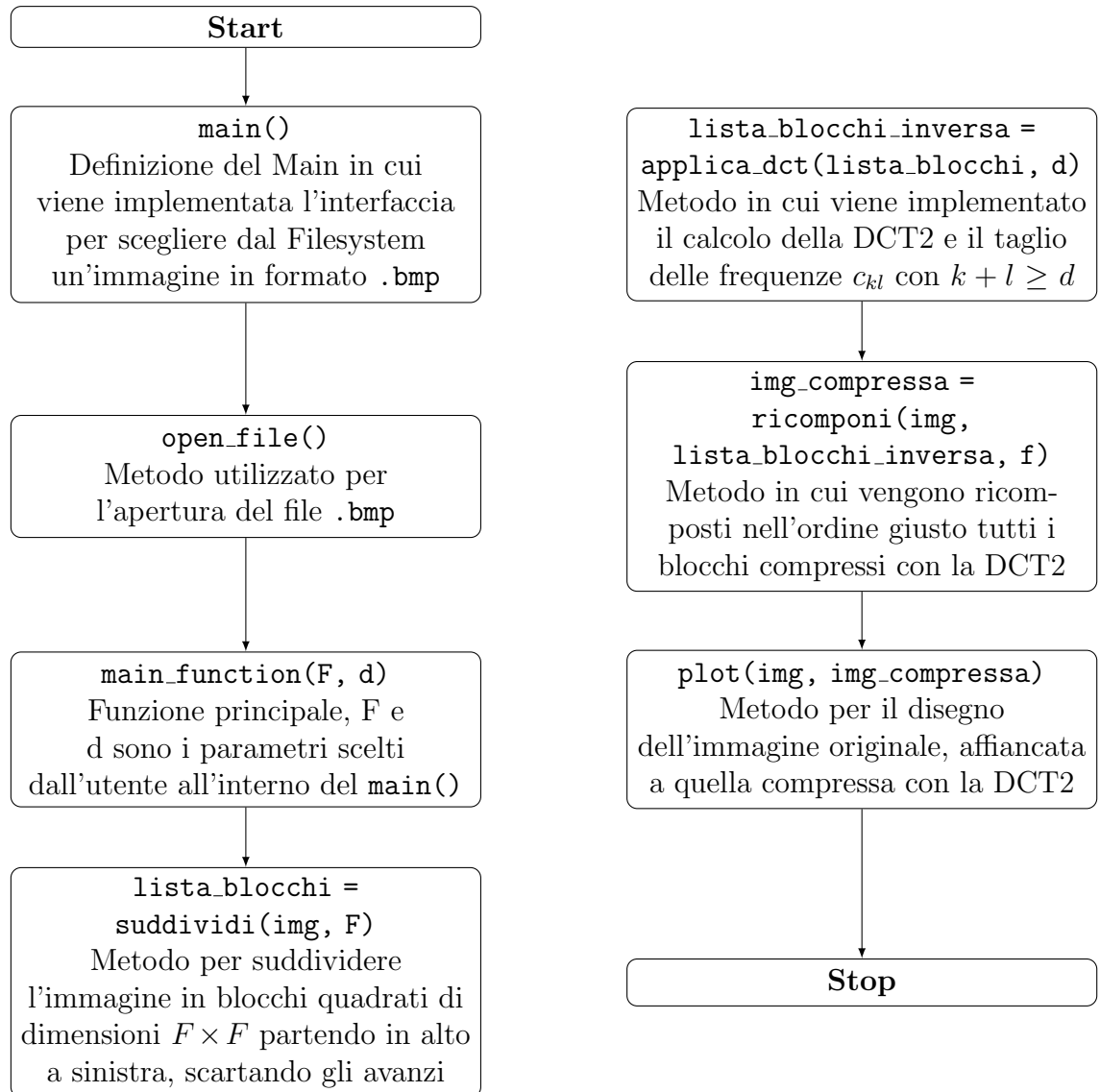


Figure 1: Blocco dove i quadrati rossi indicano le frequenze eliminate e i quadrati verdi indicano le frequenze che non vengono eliminate

Diagramma



Implementazione

Di seguito viene spiegato approfonditamente come opera il programma.

1. Crea una lista di blocchi $F \times F$ dell'immagine importata partendo dall'alto a sinistra, eliminando gli scarti sotto e a destra qualora ce ne siano;

```
while (i + f < img.shape[0]):  
    while(j + f < img.shape[1]):  
        blocco = img[i:i+f,j:j+f]  
        lista_blocchi.append(blocco)  
        j = j + f  
    i = i + f  
    j = 0
```

2. Viene applicata la DCT ad ogni blocco;

```
for f in lista_blocchi:  
    c = dct(np.transpose(dct  
        (np.transpose(f), norm='ortho')), norm='ortho  
    ')
```

3. Vengono eliminate le frequenze utilizzando come valore d ;

```
for k in range(0, c.shape[0]):  
    for l in range(0, c.shape[1]):  
        if k + l >= d:  
            c[k, l] = 0
```

4. Viene applicata la DCT inversa ad ogni blocco;

```
ff = idct(np.transpose(idct  
    (np.transpose(c), norm='ortho')), norm='ortho')
```

5. Vengono arrotondati i valori di ogni blocco all'intero più vicino e, i valori minori di 0 vengono resi uguali a 0, mentre i valori maggiori di 255 vengono resi uguali a 255;

```
for i in range(0, ff.shape[0]):  
    for j in range(0, ff.shape[1]):  
        ff[i,j] = int(ff[i,j])  
  
        if ff[i, j] < 0:  
            ff[i,j] = 0  
  
        elif ff[i,j] > 255:  
            ff[i,j] = 255
```

6. Viene ricomposta l'immagine;

```
i = 0
j = 0
index = 1
while (i + f < img.shape[0]):
    while(j + f < img.shape[1]):
        j = j + f
        if j + f < img.shape[1]:
            col = np.hstack((col,
lista_blocchi_inversa[index]))
            index = index + 1

        i = i + f
        j = 0
        colonne.append(col)
        if index < len(lista_blocchi_inversa):
            col = lista_blocchi_inversa[index]
            index = index + 1

img_compressa = colonne[0]

for i in range(1, len(colonne)):
    img_compressa = np.vstack((img_compressa, colonne
[i]))
```

7. Viene visualizzata a schermo l'immagine originale affiancata dall'immagine ottenuta dopo aver eliminato le frequenze.

```
fig = Figure(figsize=(5, 4), dpi=100)
t = np.arange(0, 3, .01)
fig.add_subplot(121).imshow(img)
fig.add_subplot(122).imshow(img_compressa)
```

Risultati

Il software creato permette di utilizzare la formula della DCT su immagini in formato bitmap in scala di grigi.

Tale programma di presenta con una semplice interfaccia grafica che facilita l'utente nella scelta dell'immagine .bmp da scegliere tra i propri file.



Figure 2: GUI principale dell'applicazione

Una volta caricato il file e scelti i valori F e d , l'utente può premere il tasto *Avvia* e aspettare che il programma elabori il risultato, per poi farlo visualizzare all'utente.

Se d non è compreso tra 0 e $2F - 2$, il programma mostra un messaggio di errore e termina l'esecuzione.



Figure 3: Messaggio di errore in caso di d incostistente

Verrà visualizzato un errore anche nel caso in cui l'utente scelga un valore di F maggiore di almeno una delle due dimensioni dell'immagine (righe e/o colonne).

Se i dati inseriti sono consistenti e l'immagine viene letta correttamente, il risultato sarà stampato su una schermata apposita in cui verrà visualizzata anche l'immagine originale. Le due immagini (originale a sinistra, compressa a destra) saranno quindi confrontabili a "occhio".

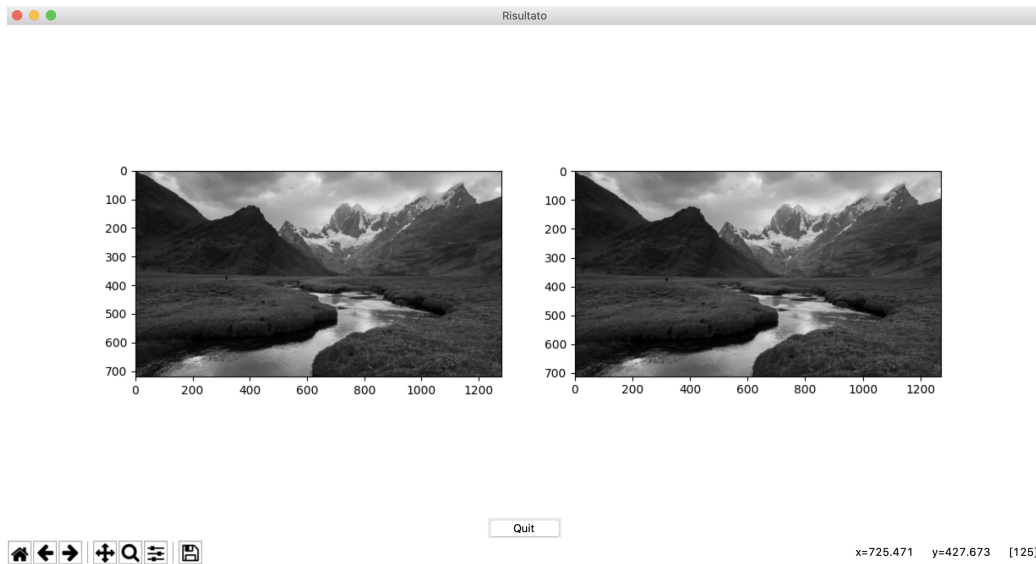


Figure 4: Risultato finale dell'elaborazione, presentato su due plot

Conclusioni

Il programma opera come previsto, l'immagine compressa avrà un peso minore (in termini di archiviazione) e la sua qualità sarà ovviamente ridotta. Le differenze non sono immediatamente visibili, solo con un'attenta osservazione si possono cogliere le sottili differenze fra le due immagini. Questo ovviamente ha i suoi vantaggi e i suoi svantaggi, di sicuro se non si lavora in ambiti in cui la qualità di un'immagine dev'essere pari all'originale (ad esempio in fotografia), questo tipo di compressione è molto vantaggiosa.

Questo è un esempio di compressione su un'immagine .bmp monocromatica recuperata dal web, in particolare sono stati usati i seguenti valori:
 $F = 8$, $d = 10$

