Progetto sulla steganografia per nascondere testi ed immagini in una immagine portante

Salemi Andrea Filippo – Progetto di Multimedia e Laboratorio

Introduzione

Nel seguente progetto si è sviluppato un programma in python (estensione Jupyter Notebook per VSCode) sulla steganografia con diverse metodologie testando la robustezza dei rumori, della compressione, delle trasformazioni affini e così via.
La steganografia è una tecnica capace di nascondere un qualunque tipo dato all'interno di un'immagine detta immagine portante senza alterare percettivamente l'immagine o alterandola ma senza che il dato appaia percettivamente visibile.
Le tecniche scelte per questo progetto sono tra quelle più famose:
-LSB: In sostanza immette nei bit meno significativi il dato da nascondere;
-DCT: Usa i coefficienti della trasformata discreta del coseno per nascondere il dato interessato;

Cenni Teorici

Prima di tutto si accennano dei concetti sulla **LSB** e **DCT** e dei vantaggi e degli svantaggi proposti da questi due tecniche:

- Il metodo **Least Significant Bit** sfrutta i bit meno significativi di un'immagine per nascondere dati riservati. L'idea è che modificare il bit meno significativo di un pixel non altera visibilmente l'immagine ma consente di immagazzinare informazioni nascoste.
  Supponiamo di avere un immagine formato da 16 milioni di colori ovvero un immagine usata di correnti utilizzo, se si considera che effettivamente un'immagine è composta da 3 matrici chiamate canali di cui ognuno di questi tre canali rappresenta il modello di colore RGB standard e se si considera il quale ogni canale ha una variazione di colore che varia da 0 a 255 e la tecnica va a modificare i bit meno significativi. Il vantaggio della tecnica è la sua facile implementazione e quindi questo ha la capacità di poter nascondere il messaggio da trasmettere.
- Il metodo, invece della **trasformata discreta del coseno** (DCT) è un metodo che usufruisce dei coefficienti per fare la steganografia di un dato da "nascondere".
  Il suo funzionamento è la seguente:
  1. Conversione in DCT:
     L'immagine viene suddivisa in blocchi 8x8 e ogni blocco viene trasformato in coefficienti DCT che rappresentano le frequenze dell'immagine.
  2. Modifica dei coefficienti:
     I dati segreti vengono nascosti nei coefficienti meno importanti della DCT (escludendo il coefficiente DC, che rappresenta la luminosità generale del blocco) come, per esempio, si può modificare la parità di un coefficiente un bit segreto.

3. Ricostruzione dell'immagine
   Dopo aver modificato i coefficienti DCT, viene applicata l'anti-trasfomata (detta anche IDCT per comodità)

Uno dei principali vantaggi rispetto alla tecnica citata è la sua resistenza superiore alla compressione o ai rumori generati e meno rilevabile tramite le analisi statistiche. Tuttavia, presenta uno svantaggio, ovvero, lo svantaggio principale è la sua difficoltà nell'implementazione e nella sua esecuzione.

Svolgimento del Progetto

Il progetto è stato scritto usando python con l'estensione di VsCode per usufruire del notebook di Jupyter.
In primis, sono state importate delle librerie e dei moduli esterni per fare in modo da usare delle funzioni per semplificare lo scopo del progetto, mentre la scrittura dei vari metodi sono stati messi in un modulo implementato in maniera separata.

Descrizione delle Librerie

```
import os
from PIL import Image
import scipy.ndimage
import cv2
import numpy as np
from skimage.io import imread
import matplotlib.pyplot as plt
from LSBSteg import LSBSteg
from LSBSteg import ImageWrapper
from DCT import DCT
from DCT import Compare
from DCT_image import *
```

Qui di seguito sono riportate le seguenti librerie con una descrizione breve:

os
Libreria standard di Python per l'accesso ai servizi del sistema operativo, che consente di gestire file e directory, manipolare percorsi ed eseguire comandi di sistema.

PIL (Pillow) – from PIL import Image
Fork aggiornato della Python Imaging Library (PIL). permette di gestire le immagini in diversi formati ed il modulo spcecifico **Image** contiene delle funzioni per ridimensionare, ruotare e applicare filtri alle immagini.

scipy.ndimage
Modulo di **SciPy** per l'elaborazione multidimensionale delle immagini che fornisce funzioni per filtrare, trasformare geometricamente, analizzare e manipolare array di immagini e quindi utile per operazioni avanzate di elaborazione.

cv2

una delle più famose librerie usata per la computer vision utile per manipolare non solo le immagini ma anche video.

numpy

Libreria fondamentale per il calcolo numerico in Python. Essa offre array multidimensionali e funzioni matematiche ottimizzate per operazioni vettoriali e matriciali, essenziali per l'elaborazione numerica.

skimage.io – imread

La funzione **imread** permette di leggere immagini da file in vari formati, utile per caricare immagini in progetti di elaborazione.

matplotlib.pyplot

Questo per la creazione di grafici e serve per la visualizzazione di immagini, grafici, istogrammi e altre rappresentazioni grafiche, facilitando l'analisi e la presentazione dei dati.

LSBSteg (LSBSteg, ImageWrapper)

Il modulo contiene la steganografia tramite LSB (Least Significant Bit). Implementando la tecnica per nascondere informazioni all'interno di immagini modificando i bit meno significativi, senza alterare visibilmente l'immagine.

DCT (DCT, Compare)

Modulo che implementa la Trasformata Discreta del Coseno (DCT) ma per nascondere il testo. La classe importata "Compare" implementa dei metodi che consentono di confrontare tramite pnr, mse l'immagine di partenza con quella finale per vedere la qualità effettiva dell'immagine (metrica di qualità).

DCT_image

A differenza dell'altro modulo questo modulo implementa una classe che serve per nascondere attraverso sempre la tecnica della DCT un'immagine invece che un testo.

Analisi delle classi importati

Come accennato in precedenza, sono stati implementati moduli che costituiscono il cuore degli algoritmi utilizzati.
All'interno della classe sono presenti i metodi principali che ne determinano il funzionamento.
Nella classe **LSBSteg** troviamo:

```
    def encode_text(self, txt):
        l = len(txt)
        binl = self.binary_value(l, 16)
        self.put_binary_value(binl)
        for char in txt:
            c = ord(char)
            self.put_binary_value(self.byteValue(c))
        return self.image
```

Questo metodo nasconde un testo all'interno dell'immagine utilizzando la tecnica LSB ed esegue i seguenti passaggi:

1) Il testo viene prima convertito in una rappresentazione binaria.
2) La lunghezza del testo viene codificata nei primi 16 bit dell'immagine.
3) Successivamente, ogni carattere viene trasformato in un valore binario di 8 bit e inserito nei pixel dell'immagine.
4) Infine , l'immagine modificata con il testo nascosto viene restituita.

```
def decode_text(self):
        ls = self.read_bits(16)
        l = int(ls,2)
        i = 0
        unhideTxt = ""
        while i < l:
            tmp = self.read_byte()
            i += 1
            unhideTxt += chr(int(tmp,2))
        return unhideTxt
```

Invece qui viene estratto il testo nascosto in un'immagine eseguendo i seguenti passaggi:

1) Legge i primi 16 bit per determinare la lunghezza del messaggio.
2) Estrae i successivi gruppi di 8 bit (un byte alla volta) e li converte in caratteri.
3) Ricompone il testo e lo restituisce.

```
    def encode_image(self, imtohide):
        w = imtohide.width
        h = imtohide.height
        if self.width*self.height*self.nbchannels < w*h*imtohide.channels:
            raise SteganographyException("Carrier image not big enough to hold all
the datas to steganography")
        binw = self.binary_value(w, 16)
        binh = self.binary_value(h, 16)
        self.put_binary_value(binw)
        self.put_binary_value(binh)
        for h in range(imtohide.height):
            for w in range(imtohide.width):
                for chan in range(imtohide.channels):
                    val = imtohide[h,w][chan]
                    self.put_binary_value(self.byteValue(int(val)))
        return self.image
```

Nasconde un'altra immagine dentro l'immagine principale e fa la seguente:

1) Codifica larghezza e altezza dell'immagine da nascondere nei primi 32 bit (16 bit per ciascuna dimensione).
2) Successivamente, scansiona ogni pixel dell'immagine da nascondere e inserisce i valori dei canali R, G e B nei bit meno significativi dell'immagine contenitore.
3) Restituisce l'immagine con l'immagine nascosta all'interno.

```
    def decode_image(self) :
        width = int(self.read_bits(16), 2)
        height = int(self.read_bits(16), 2)
        unhideimg = np.zeros((height, width, 3), np.uint8)
        wrapped_unhideimg = ImageWrapper(unhideimg)
        for h in range(height):
            for w in range(width):
                for chan in range(3):  # Usiamo direttamente 3 per i canali RGB
                    val = int(self.read_byte(), 2)
                    unhideimg[h, w, chan] = val
        return unhideimg
```

Questo metodo estrae un'immagine nascosta all'interno dell'immagine principale e fa la seguente:

1) Legge i primi 32 bit per ottenere larghezza e altezza dell'immagine nascosta.
2) Ricostruisce i pixel leggendo i valori codificati nei bit meno significativi.
3) Restituisce l'immagine estratta.

Qui terminano la descrizione dei metodi essenziali per poter lavorare sulla steganografia con la tecnica del **LSB**. Tuttavia, c'è un metodo aggiuntivo che è usata per la generazione di un rumore casuale gaussiano:

```
def gaussian_noisy(self, mean, sigma): #mean = media, sigma=dev.std.
        x1, y1, z1 = self.shape
        gauss_noise = np.random.normal(mean, sigma, (x1, y1,
z1)).astype(np.float32)
        noisy_image = self.astype(np.float32) + gauss_noise
        gn_img = np.clip(noisy_image, 0, 255).astype(np.uint8)
        return gn_img
```

Questa funzione, in parole povere, va ad aggiungere un rumore gaussiano all'immagine i cui prende come due valori come parametri:

- **mean**: si intende la media con cui dobbiamo generare il rumore
- **sigma**: determina l'intensità del rumore.
  Un piccolo appunto su **sigma:**
- Da 10 a 25 rumore addizionato leggero.
- Da 25 a 50 rumore addizionato medio.
- Da 50 a 100 rumore addizionato sarà alto e questo compromette la percezione dell'immagine totalmente.

Adesso, si andranno a descrivere i metodi principali della **DCT** e della **DCT_image**:
La dct e la dct_image sono due moduli python di cui sono state implementate due soluzioni per la steganografia con la tecnica della trasformata discreta del coseno:
La DCT nasconde nell'immagine portante una stringa, mentre la DCT_image consente ad un'immagine di nascondere un'altra immagine.

Un piccolo appunto sulla trasformata discreta del coseno

La trasformata discreta del coseno è una trasformata a simile a quella di Fourier ed è usata per attenuare le ridondanze spaziali presenti nell'immagine per comprimere. Inoltre, è assai usata anche in diversi formati multimediali.
La formula usata è la seguente:

$$F(u,v) = \frac{1}{4}C(u)C(v)\left[\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\ \cos\frac{(2x+1)u\boldsymbol{\pi}}{16}\cos\frac{(2y+1)v\pi}{16}\right]$$

$$f(x,y) = \frac{1}{4}\left[\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v)\cos\frac{(2x+1)u\boldsymbol{\pi}}{16}\cos\frac{(2y+1)v\boldsymbol{\pi}}{16}\right]$$

Dove:

$$C(u), C(v) = \frac{1}{\sqrt{2}}\ per\ u, v = 0$$
$$C(u), C(v) = 1\ altrimenti$$

```
    def encode_image(self,img,secret_msg):
        secret=secret_msg
        self.message = str(len(secret))+'*'+secret
        self.bitMess = self.toBits()
        row,col = img.shape[:2]
        self.oriRow, self.oriCol = row, col
        if((col/8)*(row/8)<len(secret)):
            print("Error: Message too large to encode in image")
            return False
        if row%8 != 0 or col%8 != 0:
            img = self.addPadd(img, row, col)
        row,col = img.shape[:2]
        bImg,gImg,rImg = cv2.split(img)
        bImg = np.float32(bImg)
        imgBlocks = [np.round(bImg[j:j+8, i:i+8]-128) for (j,i) in
itertools.product(range(0,row,8),
                                                                range(0,col,
8))]
        dctBlocks = [np.round(cv2.dct(img_Block)) for img_Block in imgBlocks]
        quantizedDCT = [np.round(dct_Block/quant) for dct_Block in dctBlocks]
        messIndex = 0
        letterIndex = 0
        for quantizedBlock in quantizedDCT:
            DC = quantizedBlock[0][0]
            DC = np.uint8(DC)
            DC = np.unpackbits(DC)
            DC[7] = self.bitMess[messIndex][letterIndex]
            DC = np.packbits(DC)
            DC = np.float32(DC)
            DC= DC-255
            quantizedBlock[0][0] = DC
            letterIndex = letterIndex+1
            if letterIndex == 8:
                letterIndex = 0
                messIndex = messIndex + 1
                if messIndex == len(self.message):
                    break
        sImgBlocks = [quantizedBlock *quant+128 for quantizedBlock in quantizedDCT]
        sImg=[]
        for chunkRowBlocks in self.chunks(sImgBlocks, col/8):
            for rowBlockNum in range(8):
                for block in chunkRowBlocks:
                    sImg.extend(block[rowBlockNum])
        sImg = np.array(sImg).reshape(row, col)
        sImg = np.uint8(sImg)
        sImg = cv2.merge((sImg,gImg,rImg))
        return sImg
```

Il metodo presente nella classe **DCT** presenta diversi metodi come detto in precedenza ha differenti metodi ma quelli che implementano gli algoritmi principali sono : **encode_image** che consente di nascondere un testo usando la trasformata del discreta del coseno. L'operazione inversa ovvero di decodifica per estrarre il testo all'interno dell'immagine la fa il **decode_image.**
Parlando nel dettaglio del primo metodo si avranno diversi passaggi per parlare.

1) **Preparazione del messaggio:**
   Il messaggio viene convertito in una stringa speciale ovvero viene immessa nel messaggio una stringa che contiene la lunghezza del messaggio la "*" come divisore e il resto del messaggio.
   Quindi, se si prende per esempio la stringa "ciao" si calcola la dimensione totale della stringa che in questo caso ha la lunghezza di 4 caratteristi e si va a convertire e il risultato sarà "4*ciao" poiché questa cosa è fatta per avvisare che il messaggio è formato di 4 caratteri partendo come punto di inizio il carattere che viene dopo la "*" e infine, viene trasformata in una sequenza di bit (tramite la funzione **toBits()**.

2) **Pre-elaborazione dell'immagine:**
   L'immagine viene divisa nei tre canali RGB e si lavora solo sul canale blu in questa implementazione con questo modello di colori (NB: Se si fosse lavorato con il sistema YCbCr si sarebbe scelto di nascondere il testo nella Y) e se le dimensioni non sono multipli di 8 viene aggiunto il padding.
3) **Conversione in DCT:**
   L'immagine viene suddivisa in blocchi 8x8 e in ogni blocco, si applica la DCT per convertire i valori spaziali in frequenze.
4) **Modifica dei coefficienti DCT**
   Nel **coefficiente DC** (il primo valore di ogni blocco), si modifica il **bit meno significativo (LSB)** per inserire i bit del messaggio.
5) **Ricostruzione immagine:**
   Si applica la quantizzazione inversa e la DCT inversa (IDCT) per ricostruire l'immagine con il messaggio nascosto e l'immagine viene restituita.

```
   def decode_image(self,img):
        row,col = img.shape[:2]
        messSize = None
        messageBits = []
        buff = 0
        bImg = np.float32(bImg)
        imgBlocks = [bImg[j:j+8, i:i+8]-128 for (j,i) in
itertools.product(range(0,row,8),
                                                                range(0,col,
8))]
        quantizedDCT = [img_Block/quant for img_Block in imgBlocks]
        i=0
        for quantizedBlock in quantizedDCT:
            DC = quantizedBlock[0][0]
            DC = np.uint8(DC)
            DC = np.unpackbits(DC)
            if DC[7] == 1:
                buff+=(0 & 1) << (7-i)
            elif DC[7] == 0:
                buff+=(1&1) << (7-i)
            i=1+i
            if i == 8:
                messageBits.append(chr(buff))
                buff = 0
                i =0
                if messageBits[-1] == '*' and messSize is None:
                    try:
                        messSize = int(''.join(messageBits[:-1]))
                    except:
                        pass
            if len(messageBits) - len(str(messSize)) - 1 == messSize:
                return ''.join(messageBits)[len(str(messSize))+1:]
        sImgBlocks = [quantizedBlock *quant+128 for quantizedBlock in quantizedDCT]
        sImg=[]
        for chunkRowBlocks in self.chunks(sImgBlocks, col/8):
            for rowBlockNum in range(8):
                for block in chunkRowBlocks:
                    sImg.extend(block[rowBlockNum])
        sImg = np.array(sImg).reshape(row, col)
        sImg = np.uint8(sImg)
        sImg = cv2.merge((sImg,gImg,rImg))
        return ''
```

Il metodo **decode_image()** è un metodo che l'opposto di quella vista in precedenza ovvero prende in input l'immagine codificata e ne estrapola il testo salvato ne restituisce una stringa e per realizzarla esegue questi passaggi:

1) **Pre-elaborazione dell'immagine:**
   L'immagine viene divisa nei tre canali RGB e si lavora solo sul canale blu e viene convertita in formato float32 per la DCT.

2) **Estrazione dei blocchi 8x8:**
   L'immagine viene suddivisa in blocchi 8x8 applicando la quantizzazione per ottenere i coefficienti DCT.

3) **Recupero del messaggio dai coefficienti DC:**
   Si leggono i bit meno significativi (LSB) dei coefficienti DC per ricostruire il messaggio bit per bit convertendo i bit in caratteri.

4) **Decodifica del messaggio:**
   - Si estrae la lunghezza del messaggio dal formato "lunghezza*messaggio", ad esempio: si estrapola la stringa "4*ciao" e lui sa che la stringa contiene 4 caratteri ed estrapola "**ciao**" restituendolo a fine metodo.

All'interno sempre questo modulo c'è un ulteriore classe con dei metodi che per confrontare due immagini:

```
class Compare():
    def correlation(self, img1, img2):
        return signal.correlate2d (img1, img2)
    def meanSquareError(self, img1, img2):
        error = np.sum((img1.astype('float') - img2.astype('float')) ** 2)
        error /= float(img1.shape[0] * img1.shape[1]);
        return error
    def psnr(self, img1, img2):
        mse = self.meanSquareError(img1,img2)
        if mse == 0:
            return 100
        PIXEL_MAX = 255.0
        return 20 * math.log10(PIXEL_MAX / math.sqrt(mse))
```

- Molto semplicemente la **correlation()** è un metodo che semplicemente calcola la correlazione tra le due immagini.
- Il metodo **meanSquareError()** prende in input due immagini e calcola la differenza tra queste due immagini.
- il **psnr()** prende in input due immagini e ne calcola il psnr ovvero la qualità complessiva dell'immagine.

Osservazione:

$MSE = \frac{1}{MN}\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}||I(i,j) - K(i,j)||^2$ più tende a 0 e meno differenza c'è tra le due immagini poiché misura l'errore quadratico medio.

$PSNR = 20 \cdot \log_{10}\left(\frac{\max\{I\}}{\sqrt{MSE}}\right)$ tende a infinito se la qualità è alta, più tende a 0 e più errori contiene l'immagine.

Adesso vengono descritte i metodi più importanti contenuti nel modulo **DCT_image** il quale come principio è assai simile al modulo precedente ma per comodità sono state implementate in una classe diversa per una questione di comodità.

```
def encode_image_with_image(self, host_img, target_img):
      # Ridimensiona target per avere le stesse dimensioni di host
      target_img = cv2.resize(target_img, (host_img.shape[1], host_img.shape[0]))
      encoded_r = self._encode_channel(host_img[:,:,0], target_img[:,:,0])
      encoded_g = self._encode_channel(host_img[:,:,1], target_img[:,:,1])
      encoded_b = self._encode_channel(host_img[:,:,2], target_img[:,:,2])
      # Convertiamo in uint8 prima di salvare o visualizzare
      encoded = cv2.merge([encoded_r, encoded_g, encoded_b])
      return np.uint8(np.clip(encoded, 0, 255))
```

Il metodo **encode_image_with_image()** Questo metodo nasconde un'immagine all'interno di un'altra immagine, sono state distinte le due immagini:
host_image : l'immagine portante.
target_image: l'immagine da nascondere.
Il funzionamento è il seguente:

1) **Ridimensionamento**
   L'immagine target (target_img) viene ridimensionata per avere le stesse dimensioni dell'immagine host (host_img) tramite cv2.resize. Questo in modo da non creare problemi durante la conservazione dell'immagine, cioè, serve per facilitare il suo recupero.
2) **Codifica dei canali RGB separatamente**

- L'immagine viene suddivisa nei suoi tre canali di colore: Rosso (R), Verde (G), Blu (B) e Per ogni canale, viene chiamato **_encode_channel**, che applica la DCT per incorporare informazioni dell'immagine target all'interno dell'immagine host.

3) **Ricostruzione dell'immagine codificata**
   I tre canali codificati vengono combinati nuovamente in un'unica immagine usando cv2.merge. L'immagine risultante viene convertita in formato **uint8** e limitata ai valori validi [0, 255] usando **np.clip** per evitare artefatti restituendo in fine un'immagine output contente l'immagine codifica.

```
  def decode_image_with_image(self, encoded_img, host_img):
        decoded_r = self._decode_channel(encoded_img[:,:,0], host_img[:,:,0])
        decoded_g = self._decode_channel(encoded_img[:,:,1], host_img[:,:,1])
        decoded_b = self._decode_channel(encoded_img[:,:,2], host_img[:,:,2])
        decoded = cv2.merge([decoded_r, decoded_g, decoded_b])
        return np.uint8(np.clip(decoded, 0, 255))
```

il metodo **decode_image_with_image()** fa l'opposto del metodo precedente, ovvero, decodifica l'immagine nascosta dentro l'altra immagine e ne trae l'immagine nascosta.
Il suo funzionamento è la seguente:

1) **Decodifica dei canali RGB separatamente**
   Ogni canale (R, G, B) viene elaborato separatamente. Ed inoltre, in ogni canale, viene chiamato **_decode_channel**, che usa la DCT per estrarre l'informazione dell'immagine target nascosta nell'immagine codificata.
2) **Ricostruzione dell'immagine target**
   I tre canali decodificati vengono combinati nuovamente in un'unica immagine tramite cv2.merge. L'immagine risultante viene convertita in formato uint8 e limitata ai valori validi [0, 255] usando **np.clip** restituendo l'immagine nascosta.

```
def compress_image(img, quality):
    encode_param = [int(cv2.IMWRITE_JPEG_QUALITY), quality]
    _, encoded_img = cv2.imencode('.jpg', img, encode_param)
    decoded_img = cv2.imdecode(encoded_img, cv2.IMREAD_COLOR)
    return decoded_img
```

Questa funzione "simula" la compressione jpeg e prende in input l'immagine e la qualità, chiaramente, la qualità più sarà alta e meno invasiva sarà la compressione. Questa funzione è stata sviluppata per fare in modo da testare la robustezza della DCT.

Ci sono altre funzioni che servono per lavorare con le trasformazioni affini:

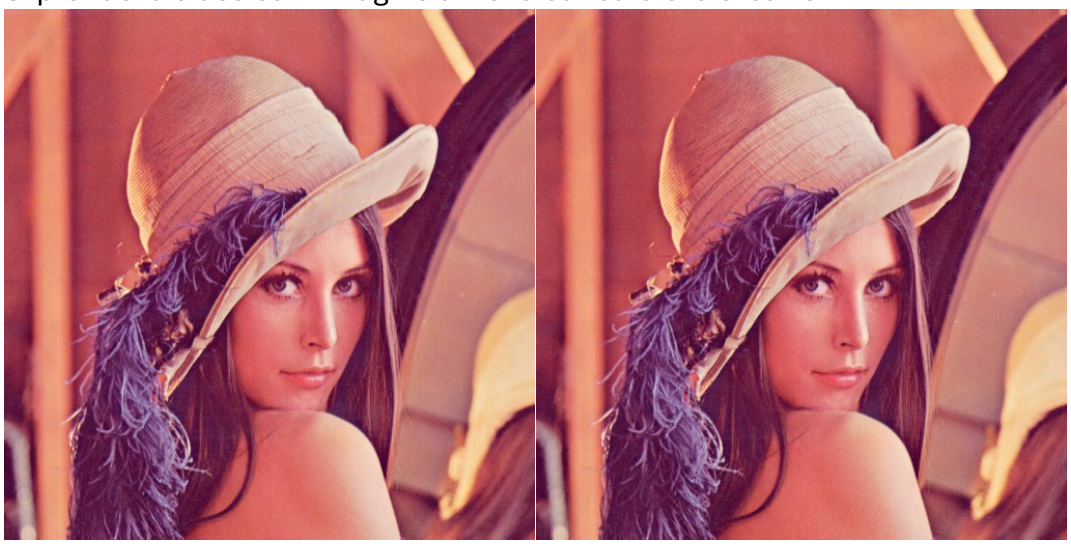
- **manual_translation(img, dx, dy)**
- **inverse_translation(img, M)**
- **manual_rotation(img, angle)**
- **inverse_rotation(img, M)**

Per testare sempre la resistenza delle DCT.

Esecuzione del progetto

Parte LSB

Durante l'esecuzione del progetto è stata caricata l'immagine tramite la funzione imread del modulo cv2 (libreria OpenCV) attraverso la funzione imread per poter caricare l'immagine.
Per comodità viene scelta di conservare una stringa breve chiamata "HelloWorld!!!"
Si prende la classica immagine di Lena caricata e la si salva



(a sinistra l'immagine originale senza alternazioni a destra l'immagine con la stringa interessata di seguito troviamo il codice.

```
TextString="HelloWorld!!!"
PathOriginalImage = "images/lena.png"
PathNewImage = "images/HelloWorldLena.png"
ims=cv2.imread(PathOriginalImage)
steg = LSBSteg(ims)
img_enc=steg.encode_text(TextString)
cimd = cv2.imread(PathNewImage)
if imd is None:
    print("il file non è stato trovato")
steg = LSBSteg(imd)
print("Il  testo nascosto nell'immagine è ", steg.decode_text())
v2.imwrite(PathNewImage,img_enc)
```

L'output sarà il seguente:

Il testo nascosto nell'immagine è HelloWorld!!!

Come l'algoritmo descritto in seguito andrà a mostrare la scritta che era stata conservata in precedenza.

Adesso, continuando si analizzerà cosa succede se si conserva l'immagine con LSB in un'immagine.

```
PathHidden = "images/133.png"
ims = cv2.imread(PathOriginalImage)
steg = LSBSteg(ims)
imf = cv2.imread(PathHidden)
x, y, z= imf.shape
x1, y1, z1= ims.shape
#perfetto, ho fatto un controllo in cui se da nascondere è più grande dell'immagine
portante allora lui ridimensiona l'immagine da nascondere.
if x> x1 or y>y1 :
    imf = cv2.resize(imf, (int(x/4),int(y/4)))
# Wrappiamo l'immagine imf in modo da fornire gli attributi width e height
wrapped_image = ImageWrapper(imf)

# Passiamo l'immagine wrappata al metodo encode_image
img_enc = steg.encode_image(wrapped_image)  # Nascondi l'immagine qui
cv2.imwrite("images/LenaNew.png", img_enc)

imss = cv2.imread("images/LenaNew.png")
steg = LSBSteg(imss)
unhideimg = LSBSteg.decode_image(steg)
# Salviamo l'immagine recuperata
cv2.imwrite("recovered.png", unhideimg)
# Visualizzazione
fig = plt.figure(figsize=(10,7))
rows = 1
columns = 3

fig.add_subplot(rows, columns, 1)
plt.imshow(cv2.cvtColor(ims, cv2.COLOR_BGR2RGB))
plt.axis('on')
plt.title('immagine originale')

fig.add_subplot(rows, columns, 2)
plt.imshow(cv2.cvtColor(imss, cv2.COLOR_BGR2RGB))
plt.axis('on')
plt.title('immagine con un immagine criptata')


fig.add_subplot(rows, columns, 3)
plt.imshow(cv2.cvtColor(unhideimg, cv2.COLOR_BGR2RGB))
plt.axis('on')

plt.title('immagine recuperata')
plt.tight_layout()
plt.show()
```

1) **Caricamento delle immagini**

- Viene definito il percorso dell'immagine da nascondere con PathHidden = "images/133.png".
- ims = cv2.imread(PathOriginalImage) carica l'immagine portante (quella in cui nascondere il dato).
- imf = cv2.imread(PathHidden) carica l'immagine che si vuole nascondere.

2) **Controllo e ridimensionamento**

- Le dimensioni delle immagini vengono ottenute tramite imf.shape (per l'immagine nascosta) e ims.shape (per quella portante).
- Se l'immagine da nascondere è più grande, allora l'immagine da nascondere viene ridimensionata usando cv2.resize (in questo caso, le dimensioni vengono scalate dividendo per garantire che possa essere incorporata correttamente nella portante.

3) **Preparazione dell'immagine da nascondere**

L'immagine ridimensionata viene "wrappata" in un oggetto ImageWrapper, che fornisce gli attributi necessari (come width e height) richiesti dal metodo di encoding.

4) **Codifica (nascondi l'immagine)**

- Viene creato un oggetto LSBSteg passando l'immagine portante ims. Con steg.encode_image(wrapped_image) si nasconde l'immagine wrappata all'interno dell'immagine portante utilizzando la tecnica LSB. Il risultato, cioè l'immagine steganografata, viene salvato su disco con cv2.imwrite("images/LenaNew.png", img_enc).

5) **Decodifica (recupera l'immagine nascosta)**

- Si ricarica l'immagine steganografata (imss) dal file.
- Viene usato un nuovo oggetto LSBSteg per decodificare e recuperare l'immagine nascosta con LSBSteg.decode_image(steg).
- L'immagine recuperata viene salvata come "recovered.png".

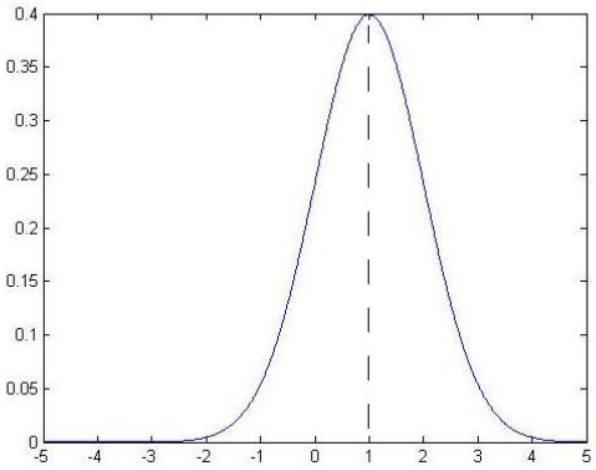
6) **Visualizzazione dei risultati**

- Utilizzando Matplotlib, il codice crea una figura con 1 riga e 3 colonne per mostrare:
- L'immagine originale (portante).
- L'immagine steganografata (contenente il dato nascosto).
- L'immagine recuperata (estratta dal file steganografato).
- Le immagini vengono convertite dal formato BGR (utilizzato da OpenCV) a RGB (per una corretta visualizzazione) e quindi mostrate a schermo.

Come si può dimostrare l'immagine viene recuperata perfettamente.

Applicazione Rumore Gaussiano:
Adesso, si prova a vedere cosa accade se si introduce un rumore. Per semplicità si introduce il rumore gaussiano (o normale).
Per cui, a livello di codice dovrà aggiungere all'immagine iniziale un rumore con distribuzione a campana generata dalla seguente funzione matematica:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)}$$



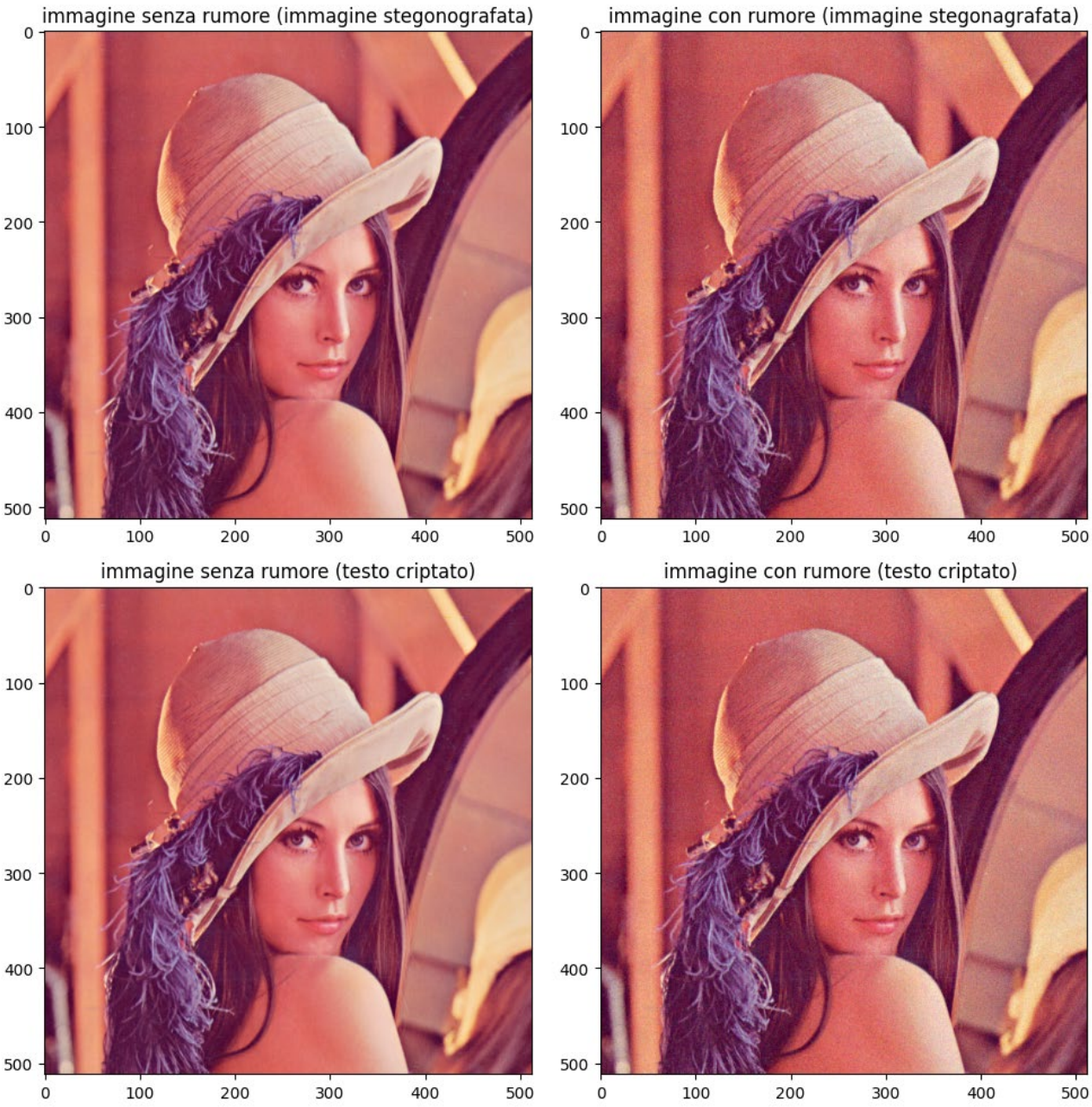

```
m= 0
s= 10
gn_image=LSBSteg.gaussian_noisy(imss, m, s)
gn_image_text=LSBSteg.gaussian_noisy(imd, m, s)
cv2.imwrite("images/image_noisy.png", gn_image)
cv2.imwrite("images/image_noisy_text.png", gn_image_text)
# se noi provassimo a eseguire il recupero dei bit l'algoritmo darà un errore
poiché la maggiorparte deio bit andranno persi
# poiché l'immagine a cui facciamo riferimento

#dopo averla salvata, proviamo a capire se effettivamente l'immagine presenta
ancora l'immagine criptata
try:
    imr = cv2.imread("images/image_noisy.png")
    steg1 = LSBSteg(imr)
    unhideimg1 = LSBSteg.decode_image(steg1)

except Exception as e:
    print("L'immagine nascosta sarà stata distrutta dal rumore")
try:
    imt=cv2.imread("images/image_noisy_text.png")
    steg1= LSBSteg(imt)
    decriptText = LSBSteg.decode_text(steg1)
    if TextString != decriptText :
        print(" Non sono la stessa stringa, poiché qualche rumore avrà alterato il
contenuto della stringa. ")
except Exception as e:
    print(" il testo sarà stata distrutta dal rumore.")
```

Output:
L'immagine nascosta sarà stata distrutta dal rumore
Non sono la stessa stringa, poiché qualche rumore avrà alterato il contenuto della stringa.
(questo è per dimostrare che il rumore anche minimo distrugge il testo nella LSB).

all'interno del codice m ed s sono rispettivamente il rumore e la deviazione standard, la media è 0 e la deviazione è 10.
LSBSteg.gaussian_noisy(imss, m, s) genera un'immagine (gn_image) aggiungendo rumore gaussiano all'immagine imss (che è già stata steganografata con un'immagine nascosta) ed LSBSteg.gaussian_noisy(imd, m, s) fa la stessa cosa, ma sull'immagine imd (quella che contiene testo nascosto). Infine, entrambe le immagini vengono salvate per poi essere plottate del codice successivo:

```
# per comodità plottiamo le due immagini : prima del rumore e dopo il rumore
(gaussiano) in modo tale vediamo il lato percettivo.
# Visualizzazione
fig = plt.figure(figsize=(10,10))
rows = 2
columns = 2

fig.add_subplot(rows, columns, 1)
plt.imshow(cv2.cvtColor(imss, cv2.COLOR_BGR2RGB))
plt.axis('on')
plt.title('immagine senza rumore (immagine stegonografata)')

fig.add_subplot(rows, columns, 2)
plt.imshow(cv2.cvtColor(gn_image, cv2.COLOR_BGR2RGB))
plt.axis('on')
plt.title('immagine con rumore (immagine stegonagrafata)')


fig.add_subplot(rows, columns, 3)
plt.imshow(cv2.cvtColor(imd, cv2.COLOR_BGR2RGB))
plt.axis('on')
plt.title('immagine senza rumore (testo criptato)')

fig.add_subplot(rows, columns, 4)
plt.imshow(cv2.cvtColor(gn_image_text, cv2.COLOR_BGR2RGB))
plt.axis('on')
plt.title('immagine con rumore (testo criptato) ')

plt.tight_layout()
plt.show()
```

Invece, il plot sarà la seguente:



Come si può vedere avremo un'immagine con un leggero disturbo che a causa della poca resistenza dell'LSB l'immagine e il testo non vengono recuperate.

Metodo DCT

La seconda fase del progetto prevede l'impiego di un ulteriore metodo di steganografia, basato sulla Trasformata Discreta del Coseno (DCT, dall'inglese Discrete Cosine Transform). Analogamente a quanto svolto in precedenza, verrà illustrato come implementare e testare in pratica questa tecnica, evidenziandone le differenze rispetto al metodo LSB e valutandone le prestazioni.

```
img2=cv2.imread("images/lena.png")
stegDCT = DCT()
#codifichiamo il messaggio
imageDCT = stegDCT.encode_image(img2,"Hello World!")
cv2.imwrite("images/DCTLena.png", imageDCT)
```

Prima di tutto viene caricata l'immagine e si applica la codifica con il testo "Hello World!" Che per l'appunto verrà nascosta nell'immagine interessata quella stringa.
Successivamente, se si prende l'immagine codificata salvata in **png** nell'image col nome "**DCTLena.png**" e poi passiamo alla decodifica, avremo questo codice:

```
imgEnc = cv2.imread("images/DCTLena.png")
textDec = stegDCT.decode_image(imgEnc)
print("ecco qui: ",textDec)
```

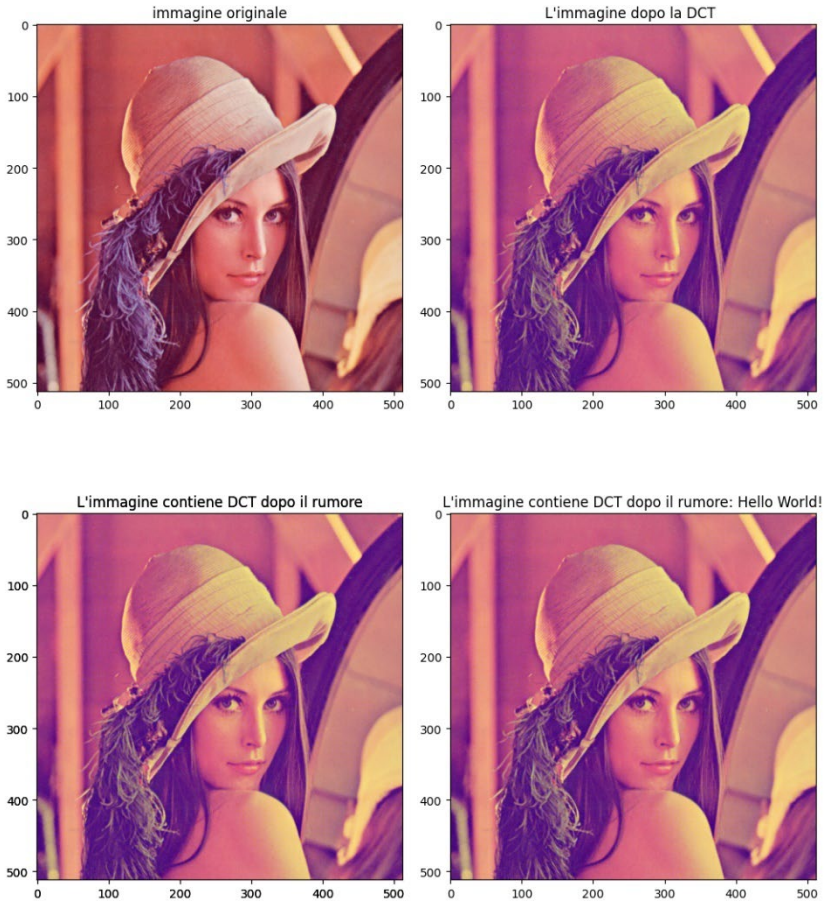
il metodo **decode_image** estrae il testo da quella immagine codificata in precendenza. Infatti, una volta estratta la stringa da quella immagine il suo output sarà il seguente:
Output:
ecco qui: Hello World!

Adesso usiamo il seguente codice per testare sia la robustezza del metodo e sia la plottiamo i risultati:

```
imgNoisy=LSBSteg.gaussian_noisy(imgEnc, 0, 0.000001)
fig = plt.figure(figsize=(10,12))
rows = 2
columns = 2
fig.add_subplot(rows, columns, 1)
plt.imshow(cv2.cvtColor(img2, cv2.COLOR_BGR2RGB))
plt.axis('on')
plt.title('immagine originale')
fig.add_subplot(rows, columns, 2)
plt.imshow(cv2.cvtColor(imgEnc, cv2.COLOR_BGR2RGB))
plt.axis('on')
plt.title("L'immagine dopo la DCT ")
fig.add_subplot(rows, columns, 3)
plt.imshow(cv2.cvtColor(imgNoisy, cv2.COLOR_BGR2RGB))
plt.axis('on')
plt.title("L'immagine contiene DCT dopo il rumore")
fig.add_subplot(rows, columns, 3)
plt.imshow(cv2.cvtColor(imgNoisy, cv2.COLOR_BGR2RGB))
plt.axis('on')
plt.title("L'immagine contiene DCT dopo il rumore")
textDec = stegDCT.decode_image(imgNoisy)
fig.add_subplot(rows, columns, 4)
plt.imshow(cv2.cvtColor(imgNoisy, cv2.COLOR_BGR2RGB))
plt.axis('on')
plt.title("L'immagine contiene DCT dopo il rumore: " + textDec)
plt.tight_layout()
plt.show()
print(textDec)
```

Applicando il rumore gaussiano applicato attraverso l'invocazione del metodo **"guassian_noisy()"** con una deviazione standard bassa e una media pari allo 0.

Quello che plotta sono le seguenti:



Vedendo l'immagine originale e le altre notiamo un leggero cambiamento di distorsione del colore, ci sono quattro quadranti:

1) Quella alto a sinistra rappresenta l'immagine originale
2) La seconda in alto a destra rappresenta l'immagine dopo aver applicata la DCT nascondendo il testo.
3) In basso a sinistra rappresenta l'immagine contente il rumore
4) L'ultima in basso a destra c'è il messaggio di quello che riesce ad estrarre.

In conclusione, si può dire che rispetto alla tecnica della **LSB** riesce a resistere meglio agli attacchi dei rumori.

In questa parte, si vede analogamente per il nascondere un'immagine all'interno di un'altra applicando la DCT.

```
dct_image = DCT_image()

host_img_path = "images/lena.png"
target_img_path = "images/133.png"
# Carica le immagini host e target
host_img = cv2.imread(host_img_path, cv2.IMREAD_COLOR)
target_img = cv2.imread(target_img_path, cv2.IMREAD_COLOR)
target_img = cv2.resize(target_img, (host_img.shape[1], host_img.shape[0]))

# Fase di encoding: nascondi target dentro host tramite DCT
encoded_img = dct_image.encode_image_with_image(host_img, target_img)
cv2.imwrite("encoded_image.png", encoded_img)

# Verifica della decodifica senza trasformazioni
decoded_img = dct_image.decode_image_with_image(encoded_img, host_img)
cv2.imwrite("decoded_image.png", decoded_img)


# Comprimiamo l'immagine encoded

quality_image = 50
compressed_image = compress_image(encoded_img, quality_image)
cv2.imwrite("compress_image.jpeg", compressed_image)

decoded_img_after_compression = dct_image.decode_image_with_image(compressed_image,
host_img)
cv2.imwrite("decoded_compres_image.jpeg", decoded_img_after_compression)

affine_corrected = create_translation(encoded_img)
cv2.imwrite("affine_corrected.jpeg", affine_corrected)


decoded_affine = dct_image.decode_image_with_image(affine_corrected, host_img)
cv2.imwrite("decoded_aff.jpeg", decoded_affine)
```

Il seguente codice ha diversi punti da commentare e di seguito verranno trattate diversi punti in presenti nell'immagine e suddividendo il codice in più parti:

1) Inizializzazione e Caricamento delle Immagini

Il codice inizia creando un'istanza della classe DCT_image, che fornisce i metodi per eseguire l'encoding e il decoding tramite la Trasformata Discreta del Coseno. Successivamente, vengono definiti i percorsi per l'immagine host (contenuta in "images/lena.png") e per l'immagine target (contenuta in "images/133.png"). Entrambe le immagini vengono caricate in modalità colore tramite cv2.imread. Per garantire la compatibilità e l'adeguata integrazione durante il processo di embedding, l'immagine target viene ridimensionata in modo da avere le stesse dimensioni dell'immagine host, utilizzando cv2.resize.

2) Processo di Encoding con DCT

In questa fase viene applicato il metodo encode_image_with_image, il quale nasconde l'immagine target all'interno dell'immagine host. Il risultato di questa operazione è un'immagine codificata che incorpora il contenuto nascosto. L'immagine risultante viene poi salvata su disco con il nome "encoded_image.png". Questo passaggio serve fondamentalmente per dimostrare che tale tecnic possa integrare dati all'interno di una portante (immagine che nasconde un'altra immagine), mantenendo intatta la qualità percepita dall'osservatore.

3) Verifica dell'Encoding in Condizioni Ideali

Per assicurarsi che il processo di codifica sia stato eseguito correttamente, viene effettuata una prima decodifica dell'immagine codificata senza alcuna trasformazione applicata successivamente. Il metodo decode_image_with_image, utilizzando l'immagine host originale come riferimento, estrae il contenuto nascosto. Il risultato della decodifica viene salvato come "decoded_image.png", confermando l'efficacia del metodo in condizioni ideali e controllate.

4) Test di Robustezza mediante Compressione JPEG

Per valutare la resistenza del metodo DCT in presenza di perturbazioni, il codice simula un attacco di compressione JPEG. Viene definito un parametro di qualità pari a 50 e l'immagine portante viene compressa tramite la funzione compress_image. L'immagine una volta compressa, sarà salvata come "compress_image.jpeg", subendo poi una nuova fase di decodifica per estrarre il contenuto nascosto. Il risultato di questa operazione viene memorizzato in "decoded_compres_image.jpeg". Questo test permette di verificare se la compressione JPEG comprometta l'integrità del contenuto nascosto, evidenziando così la robustezza della tecnica in scenari reali.

5) Valutazione della resistenza alle Trasformazioni Affini

Infine, Viene applicata una traslazione all'immagine codificata attraverso la funzione **create_translation(),** ottenendo un'immagine corretta rispetto alla trasformazione e salvata come "**affine_corrected.jpeg**". Successivamente, viene eseguita la decodifica sul risultato della trasformazione utilizzando nuovamente il metodo **decode_image_with_image(),** e il contenuto nascosto viene estratto e salvato in "**decoded_aff.jpeg**". Questa fase finale serve a dimostrare che si riesca a preservare l'integrità dei dati anche quando l'immagine portante viene sottoposta a trasformazioni geometriche.

Ogni parte del codice è finalizzata a valutare diversi aspetti della robustezza della steganografia basata sulla DCT, verificando sia la correttezza dell'encoding in condizioni

ideali che la capacità del metodo di resistere a perturbazioni come la compressione JPEG e le trasformazioni affini e facendo un plotting con questo codice:

```
fig = plt.figure(figsize=(10,12))
rows = 2
columns = 2

plt.title("Stegonografia con immagine portante usando la DCT")
fig.add_subplot(rows, columns, 1)
plt.title("immagine originale")
plt.imshow(cv2.cvtColor(host_img, cv2.COLOR_BGR2RGB))
plt.axis('on')

fig.add_subplot(rows, columns, 2)
plt.title("immagine da nascondere")
plt.imshow(cv2.cvtColor(target_img, cv2.COLOR_BGR2RGB))
plt.axis('on')


fig.add_subplot(rows, columns, 3)
plt.title("immagine codificata : ")
plt.imshow(cv2.cvtColor(encoded_img, cv2.COLOR_BGR2RGB))
plt.axis('on')



fig.add_subplot(rows, columns, 4)
plt.title("immagine decodificata: ")
plt.imshow(cv2.cvtColor(decoded_img, cv2.COLOR_BGR2RGB))
plt.axis('on')
plt.tight_layout()
plt.show()
```

Come si può vedere dal grafico che la trasformata discreta del coseno funziona anche se presenta alcuni piccoli artefatti grafici.

Adesso, applicando una compressione jpeg e plottando con il seguente codice:

```
from DCT import Compare
quality_image = 100
compressed_image = compress_image(encoded_img, quality_image) #compressa l'immagine
emulando una compressione jpeg
cv2.imwrite("compress_image.jpeg", compressed_image)
decoded_img_after_compression = dct_image.decode_image_with_image(compressed_image,
host_img)
cv2.imwrite("decoded_compres_image.jpeg", decoded_img_after_compression)

fig = plt.figure(figsize=(10,12))
rows = 2
columns = 2

plt.title("Stegonografia con immagine portante usando la DCT: Compressione")
fig.add_subplot(rows, columns, 1)
plt.title("immagine originale")
plt.imshow(cv2.cvtColor(host_img, cv2.COLOR_BGR2RGB))
plt.axis('on')

fig.add_subplot(rows, columns, 2)
plt.title("immagine da nascondere")
plt.imshow(cv2.cvtColor(target_img, cv2.COLOR_BGR2RGB))
plt.axis('on')


fig.add_subplot(rows, columns, 3)
plt.title("immagine compressa : ")
plt.imshow(cv2.cvtColor(compressed_image, cv2.COLOR_BGR2RGB))
plt.axis('on')



fig.add_subplot(rows, columns, 4)
plt.title("immagine compressa decodificata: ")
plt.imshow(cv2.cvtColor(decoded_img_after_compression, cv2.COLOR_BGR2RGB))
plt.axis('on')
plt.tight_layout()
plt.show()
cmp=Compare()
print("MSE e PSNR immagine portante prima della codifica e dopo la codifica:")
print("l'MSE:",cmp.meanSquareError(host_img,compressed_image))
print("psnr:",cmp.psnr(host_img,compressed_image))

print("MSE e PSNR immagine da nascondere prima della codifica e dopo la codifica:")
print("l'MSE:",cmp.meanSquareError(target_img,decoded_img_after_compression))
print("psnr:",cmp.psnr(decoded_img_after_compression,target_img))
```

Si può notare come l'immagine compressa in formato jpeg con il metodo **compress_image()**
E all'interno prende due parametri uno che indica l'immagine e l'altra la qualità a cui desideriamo effettuare il livello di compressione (100= leggera compressione converte solo il formato inferiore inizia a comprimere i dati).

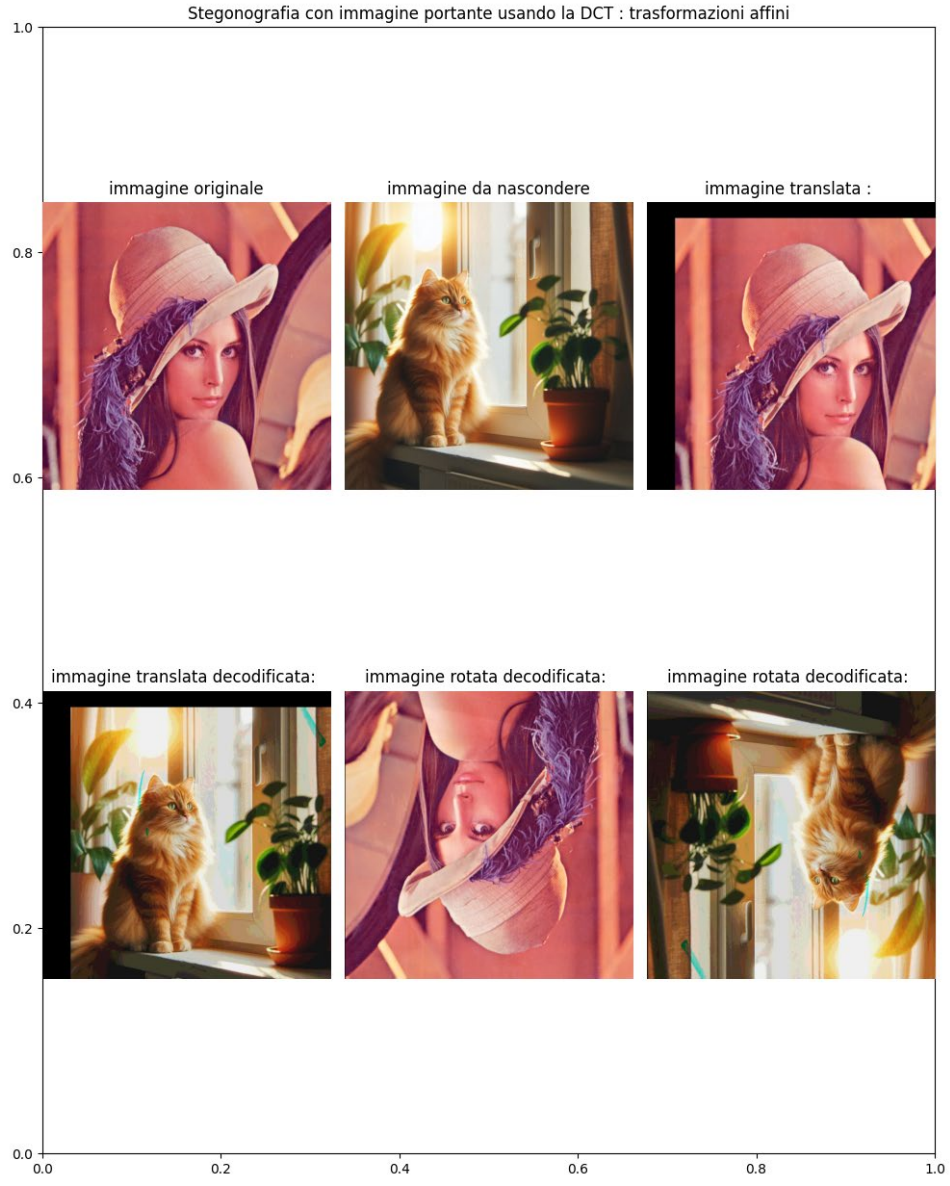
Come si può notare l'immagine nascosta ha resistito alla compressione e conversione di file anche se presenta dei rumori molto evidenti.
Output del mse e Psnr:

MSE e PSNR immagine portante prima della codifica e dopo la codifica:
l'MSE: 173.15723419189453
psnr: 25.746397205758974
MSE e PSNR immagine da nascondere prima della codifica e dopo la codifica:
l'MSE: 8328.54097366333
psnr: 8.925114342013522

dagli output capiamo che effettivamente c'è una grossa differenza nella qualità dell'immagine anche in maniera oggettiva.
A questo punto, si applica all'ultima parte una trasformazione affine per vedere l'efficacia della DCT:

```
dx,dy=50,30
affine_corrected, M = manual_translation(encoded_img,dx,dy)
cv2.imwrite("affine_corrected.jpeg", affine_corrected)
host_img2, M2 = manual_translation(host_img,dx,dy)
decoded_affine = dct_image.decode_image_with_image(affine_corrected,host_img2)
cv2.imwrite("decoded_aff.jpeg", decoded_affine)
#ruotato:
rotated_img, M3 = manual_rotation(encoded_img, angle=180)
cv2.imwrite("rotated_img.jpeg", rotated_img)
host_img3, M3 = manual_rotation(host_img, angle=180)
decode_rotate = dct_image.decode_image_with_image(rotated_img,host_img3)
fig = plt.figure(figsize=(10,12))
rows = 2
columns = 3

plt.title("Stegonografia con immagine portante usando la DCT : trasformazioni
affini")
fig.add_subplot(rows, columns, 1)
plt.title("immagine originale")
plt.imshow(cv2.cvtColor(host_img, cv2.COLOR_BGR2RGB))
plt.axis('off')

fig.add_subplot(rows, columns, 2)
plt.title("immagine da nascondere")
plt.imshow(cv2.cvtColor(target_img, cv2.COLOR_BGR2RGB))
plt.axis('off')
fig.add_subplot(rows, columns, 3)
plt.title("immagine translata : ")
plt.imshow(cv2.cvtColor(affine_corrected, cv2.COLOR_BGR2RGB))
plt.axis('off')
fig.add_subplot(rows, columns, 4)
plt.title("immagine translata decodificata: ")
plt.imshow(cv2.cvtColor(decoded_affine, cv2.COLOR_BGR2RGB))
plt.axis('off')
#
fig.add_subplot(rows, columns, 5)
plt.title("immagine rotata decodificata: ")
plt.imshow(cv2.cvtColor(rotated_img, cv2.COLOR_BGR2RGB))
plt.axis('off')
fig.add_subplot(rows, columns, 6)
plt.title("immagine rotata decodificata: ")
plt.imshow(cv2.cvtColor(decode_rotate, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.tight_layout()
plt.show()
```

Il codice visto applica all'immagine **codificata** e all'immagine host per verificare la robustezza della steganografia. In primo luogo, si definiscono gli spostamenti orizzontale e verticale (dx=50, dy=30) e si applica la funzione manual_translation sia all'immagine **encoded** (ottenendo affine_corrected) che all'immagine host (host_img2). Queste immagini traslate vengono poi usate per decodificare il contenuto nascosto tramite il metodo decode_image_with_image e il risultato viene salvato come "decoded_aff.jpeg". Successivamente, viene eseguita una rotazione di 180 gradi sia su encoded_img che su **host_img** tramite la funzione manual_rotation, ottenendo rispettivamente rotated_img e host_img3. Utilizzando l'immagine host ruotata come riferimento, il contenuto nascosto viene decodificato dall'immagine ruotata, con il risultato memorizzato in **decode_rotate**. Infine, viene creata una figura con **Matplotlib** in una griglia 2×3 per mostrare: l'immagine **host** originale, l'immagine target, l'immagine traslata, l'immagine decodificata dopo la traslazione, l'immagine ruotata e quella decodificata dopo la rotazione. La conversione da BGR a RGB e la disattivazione degli assi garantiscono una visualizzazione corretta e ordinata.
Infine, in seguito il risultato del plotting:



Vedendo un po’ quello che accade è che effettivamente la DCT riesce a recuperare l’immagine anche se effettuiamo delle trasformazioni affini che in questo caso per comodità si è deciso di usare la rotazione e la traslazione.

Conclusione

In conclusione, si può osservare come effettivamente questi due metodi riescono a fare la stessa cosa ma cambia il come è fatta l’LSB al livello computazionale è più diretto ma questo ha il grosso svantaggio che a una minima compressione o a un minimo disturbo di un rumore si perde l’informazione da recuperare. Mentre la DCT ha il grosso vantaggio di essere più resistente ma ha lo svantaggio di avere una complessità più alta. Inoltre, si aggiunge che esistono altre tecniche ma che non sono assai dissimili da questi due e che sono assai simili cambia solo qualche dettaglio, facendo esempio si può invece di usare la DCT usare la DWT ma si ottiene un risultato molto simile. Si conclude, che, la steganografia è un processo che non solo è applicata all’interno di un’immagine ma anche all’interno di qualunque altro mezzo, come per esempio, all’interno di un video e all’interno di un file musicale.