Libraries

In [1]:

```
import numpy as np
import random as rd
import cv2
from matplotlib import pyplot as plt
import os
import math
```

Formulas

In [129…

```
# MSE and PSNR

MN = I.shape[0] * I.shape[1]
for x in range(I.shape[0]):
    for y in range(I.shape[1]):
        MSE = (sum(rep_BP[x, y] - I[x, y])**2) // MN

PSNR1 = -10 * math.log10(MSE // np.max(I))
PSNR2 = 20 * math.log10(np.max(I) // math.sqrt(MSE))
PSNR3 = 10 * math.log10((np.max(I))**2 // MSE)
```

In [ ]:

```
# Pixel normalization

v_new = (MAX - MIN) * ((v_old - obs_MIN) // (obs_MAX - obs_MIN)) + MIN
```

In [ ]:

```
# Convolution: common arrays

median = np.median(subpatch)
mean = np.mean(subpatch)
max_f = np.max(subpatch)
min_f = np.min(subpatch)
box3 = np.array([[1, 1, 1], [1, 1, 1], [1, 1, 1]]) * (1/9)
box5 = np.array([[1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1],
                 [1, 1, 1, 1, 1], [1, 1, 1, 1, 1]]) * (1/25)
binomial3 = np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]]) * (1/16)
binomial5 = np.array([[1, 4, 6, 4, 1], [4, 16, 24, 16, 4],
                      [6, 24, 36, 24, 6], [4, 16, 24, 16, 4],
                      [1, 4, 6, 4, 1]]) * (1/256)
sobel_x = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
prewitt_x = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]])
sobel_y = np.array([[-1, 0, 1], [-2, 0, 2], [- 1, 0, 1]])
prewitt_y = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
laplacian = np.array([[-1, 0, -1], [0, 4, 0], [-1, 0, -1]])
```

In [ ]:

```
# YMC space
yellow_channel = 1 - blue_channel
magenta_channel = 1 - green_channel
cyan_channel = 1 - red_channel

# YUV space
Y = 0.299 * R + 0.587 * G + 0.114 * B
U = 0.564 * (B - Y)
V = 0.713 * (R - Y)

# YCbCr space
Cb = U + 128
Cr = V + 128
```

In [ ]:

```
# Open an image on OpenCV

img = cv2.imread(r"path")
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
imgYCC = cv2.cvtColor(img, cv2.COLOR_BGR2YCR_CB)
plt.imshow(img)
plt.axis('on')  # Nascondi gli assi
plt.show()

#SAVE
cv2.imwrite('output_image.jpg', img)  # Replace with your desired file name and ext
```

In [ ]:

```
# Show more than one image on OpenCV

plt.figure()
plt.imshow(img1, cmap='gray')
plt.title("img1")

plt.figure()  # Crea una nuova figura
plt.imshow(img2, cmap='gray')
plt.title("img2")
```

In [ ]:

```
# Interpolation methods with OpenCV

methods = {
    'Nearest': cv2.INTER_NEAREST,
    'Linear': cv2.INTER_LINEAR,
    'Cubic': cv2.INTER_CUBIC,
    'Lanczos': cv2.INTER_LANCZOS4

}

image = cv2.imread('input_image.jpg')

plt.figure(figsize=(10, 8))

for i, (name, method) in enumerate(methods.items(), 1):
    resized = cv2.resize(image, (300, 300), interpolation=method)
    plt.subplot(2, 2, i)
    plt.imshow(cv2.cvtColor(resized, cv2.COLOR_BGR2RGB))
    plt.title(name)
    plt.axis('off')

plt.tight_layout()
plt.show()


laplacian = cv2.Laplacian(img, cv2.CV_64F)
```

In [ ]:

```
# Saving results in a folder

filename = input("Filename: ")
mg_filename = f"{filename}.png"
output_path = os.path.join(ptOut, mg_filename)
cv2.imwrite(output_path, merged)
```

Numpy exercises

Array creation

In [39]:

```python
def arr_creation():
    N = int(input("Rows: "))
    M = int(input("Columns: "))
    arr = np.random.randint(0, 255, size = (N, M, 3))
    return arr
```

Exercise 1 - Rows/columns swapping

In [20]:

```python
def swap_columns(I):
    new_I = np.zeros_like(I)
    for i in range(I.shape[0]):
        for j in range(0, I.shape[1], 2):
            if j + 1 < I.shape[1]:
                new_I[i, j] = I[i, j + 1]
                new_I[i, j + 1] = I[i, j]
            else:
                new_I[i, j] = I[i, j]
    return new_I

def swap_rows(I):
    new_I = np.zeros_like(I)
    for i in range(0, I.shape[0], 2):
        for j in range(I.shape[1]):
            if i + 1 < I.shape[0]:
                new_I[i, j] = I[i + 1, j]
                new_I[i + 1, j] = I[i, j]
            else:
                new_I[i, j] = I[i, j]
    return new_I

def set_zero(I):
    new_I = I.copy()
    X = int(input("Zero input: "))
    for i in range(I.shape[0]):
        for j in range(I.shape[1]):
            for c in range(I.shape[2]):
                if I[i, j, c] == X:
                    new_I[i, j, c] = 0
    return new_I
```

Exercise 2 - Diagonals exchange

In [33]:

```python
def diagonals_exchange(I):
    new_I = I.copy()
    for i in range(I.shape[0]):
        for j in range(I.shape[1]):
            for c in range(I.shape[2]):
                new_I[i, i, c] = I[i, I.shape[1] - i - 1, c]
                new_I[i, I.shape[1] - i - 1, c] = I[i, i, c]
    return new_I

def secondary_zero(I):
    new_I = I.copy()
    for i in range(I.shape[0]):
        for j in range(I.shape[1]):
            for c in range(I.shape[2]):
                new_I[i, I.shape[1] - i - 1, c] = 0
    return new_I

def binarization(I):
    new_I = I.copy()
    S = int(input("Threshold in 0-255: "))
    if S < 0 or S > 255:
        raise ValueError("Insert valid threshold.")
    for i in range(I.shape[0]):
        for j in range(I.shape[1]):
            for c in range(I.shape[2]):
                if I[i, j, c] <= S:
                    new_I[i, j, c] = 0
                else:
                    new_I[i, j, c] = 255
    return new_I
```

Exercise 3 - Patch extraction

In [44]:

```
def patch_extraction(I):
    new_I = I.copy()
    x = int(input("x coordinate: "))
    if x < 0 or x > I.shape[0]:
        raise ValueError("Invalid coordinate.")
    y = int(input("y coordinate: "))
    if y < 0 or y > I.shape[1]:
        raise ValueError("Invalid coordinate.")
    w = int(input("Width: "))
    if x + w < 0 or x + w > I.shape[0]:
        raise ValueError("Invalid coordinate.")
    h = int(input("Height: "))
    if y + h < 0 or y + h > I.shape[1]:
        raise ValueError("Invalid coordinate.")
    M1 = new_I[x:x + w, y:y + h, 0]
    M2 = new_I[x:x + w, y:y + h, 1]
    M3 = new_I[x:x + w, y:y + h, 2]
    new_I[x:x + w, y:y + h, 0] = 0
    new_I[x:x + w, y:y + h, 1] = 0
    new_I[x:x + w, y:y + h, 2] = 0
    return new_I
```

Exercise 4 - Neighbors

In [54]:

```
# Ricorda di porre uguali a 0 i neighbours se le condizioni di if-else non vengono

def neighbours(I):
    new_I = I.copy()
    for i in range(I.shape[0]):
        for j in range(I.shape[1]):
            for c in range(I.shape[2]):
                # Central pixels
                if i - 1 >= 0 and i + 1 < I.shape[0] and j - 1 >= 0 and j + 1 < I.s
                    P1 = I[i - 1, j, c]
                    P2 = I[i, j + 1, c]
                    P3 = I[i + 1, j, c]
                    P4 = I[i, j - 1, c]
                    neighborhood = [P1, P2, P3, P4]
                    # Even coordinates, odd channel
                    if i % 2 == 0 and j % 2 == 0 and c == 1:
                        new_I[i, j, c] = sum(neighborhood) / len(neighborhood)
                    # Odd coordinates, even channel
                    else:
                        difference = neighborhood[0]
                        for x in range(len(neighborhood)):
                            difference -= neighborhood[x]
                        new_I[i, j, c] = abs(difference)

                # Top-left corner
                if i == 0 and j == 0:
                    P1 = I[i, j + 1, c] if j + 1 < I.shape[1] else 0
                    P2 = I[i + 1, j, c] if i + 1 < I.shape[0] else 0
                    neighborhood = [P1, P2]
                    if c == 1:
                        new_I[i, j, c] = sum(neighborhood) / len(neighborhood)
                    else:
                        difference = neighborhood[0]
                        for x in range(len(neighborhood)):
                            difference -= neighborhood[x]
                        new_I[i, j, c] = abs(difference)

                # Bottom-left corner
                if i == I.shape[0] - 1 and j == 0:
                    P1 = I[i - 1, j, c] if i - 1 >= 0 else 0
                    P2 = I[i, j + 1, c] if j + 1 < I.shape[1] else 0
                    neighborhood = [P1, P2]
                    if c == 1:
                        new_I[i, j, c] = sum(neighborhood) / len(neighborhood)
                    else:
                        difference = neighborhood[0]
                        for x in range(len(neighborhood)):
                            difference -= neighborhood[x]
                        new_I[i, j, c] = abs(difference)

                # Top-right corner
                if i == 0 and j == I.shape[1] - 1:
                    P1 = I[i, j - 1, c] if j - 1 >= 0 else 0
                    P2 = I[i + 1, j, c] if i + 1 < I.shape[0] else 0
                    neighborhood = [P1, P2]
                    if c == 1:
                        new_I[i, j, c] = sum(neighborhood) / len(neighborhood)
                    else:
                        difference = neighborhood[0]
                        for x in range(len(neighborhood)):
                            difference -= neighborhood[x]
                        new_I[i, j, c] = abs(difference)

                # Bottom-right corner
                if i == I.shape[0] - 1 and j == I.shape[1] - 1:
                    P1 = I[i, j - 1, c] if j - 1 >= 0 else 0
                    P2 = I[i - 1, j, c] if i - 1 >= 0 else 0
                    neighborhood = [P1, P2]
                    if c == 1:
                        new_I[i, j, c] = sum(neighborhood) / len(neighborhood)
                    else:
                        difference = neighborhood[0]
                        for x in range(len(neighborhood)):
                            difference -= neighborhood[x]
                        new_I[i, j, c] = abs(difference)

                # First row
                if i == 0:
                    P1 = I[i, j - 1, c] if j - 1 >= 0 else 0
                    P2 = I[i + 1, j, c] if i + 1 < I.shape[0] else 0
                    P3 = I[i, j + 1, c] if j + 1 < I.shape[1] else 0
                    neighborhood = [P1, P2, P3]
                    if c == 1:
                        new_I[i, j, c] = sum(neighborhood) / len(neighborhood)
                    else:
                        difference = neighborhood[0]
                        for x in range(len(neighborhood)):
                            difference -= neighborhood[x]
                        new_I[i, j, c] = abs(difference)
                # Last row
                if i == I.shape[0] - 1:
                    P1 = I[i, j - 1, c] if j - 1 >= 0 else 0
                    P2 = I[i - 1, j, c] if i - 1 >= 0 else 0
                    P3 = I[i, j + 1, c] if j + 1 < I.shape[1] else 0
                    neighborhood = [P1, P2, P3]
                    if c == 1:
                        new_I[i, j, c] = sum(neighborhood) / len(neighborhood)
                    else:
                        difference = neighborhood[0]
                        for x in range(len(neighborhood)):
                            difference -= neighborhood[x]
                        new_I[i, j, c] = abs(difference)

                # First column
                if j == 0:
                    P1 = I[i - 1, j, c] if i - 1 >= 0 else 0
                    P2 = I[i, j + 1, c] if j + 1 < I.shape[1] else 0
                    P3 = I[i + 1, j, c] if i + 1 < I.shape[0] else 0
                    neighborhood = [P1, P2, P3]
                    if c == 1:
                        new_I[i, j, c] = sum(neighborhood) / len(neighborhood)
                    else:
                        difference = neighborhood[0]
                        for x in range(len(neighborhood)):
                            difference -= neighborhood[x]
                        new_I[i, j, c] = abs(difference)

                # Last column
                if j == I.shape[1] - 1:
                    P1 = I[i - 1, j, c] if i - 1 >= 0 else 0
                    P2 = I[i, j - 1, c] if j - 1 >= 0 else 0
                    P3 = I[i + 1, j, c] if i + 1 < I.shape[0] else 0
                    neighborhood = [P1, P2, P3]
                    if c == 1:
                        new_I[i, j, c] = sum(neighborhood) / len(neighborhood)
                    else:
                        difference = neighborhood[0]
                        for x in range(len(neighborhood)):
                            difference -= neighborhood[x]
                        new_I[i, j, c] = abs(difference)

    return new_I
```

Exercise 7 - Decimation methods

In [35]:

```
def decimation_avg(I):
    new_rows = (I.shape[0] + 1) // 2
    new_cols = (I.shape[1] + 1) // 2
    new_I = np.zeros((new_rows, new_cols, I.shape[2]), dtype = "uint8")
    for i in range(0, I.shape[0], 2):
        for j in range(0, I.shape[1], 2):
            for c in range(I.shape[2]):
                subpatch = I[i:min(i + 2, I.shape[0]), j:min(j + 2, I.shape[1]), c]
                mean = np.mean(subpatch)
                new_I[i // 2, j // 2, c] = mean.astype("uint8")
                if i == I.shape[0] and j == I.shape[1]:
                    new_I[i // 2, j // 2, c] = I[i, j, c]
    return new_I

def decimation_min(I):
    new_rows = (I.shape[0] + 1) // 2
    new_cols = (I.shape[1] + 1) // 2
    new_I = np.zeros((new_rows, new_cols, I.shape[2]), dtype = "uint8")
    for i in range(0, I.shape[0], 2):
        for j in range(0, I.shape[1], 2):
            for c in range(I.shape[2]):
                subpatch = I[i:min(i + 2, I.shape[0]), j:min(j + 2, I.shape[1]), c]
                minimum = np.min(subpatch)
                new_I[i // 2, j // 2, c] = minimum.astype("uint8")
                if i == I.shape[0] and j == I.shape[1]:
                    new_I[i // 2, j // 2, c] = I[i, j, c]
    return new_I

def decimation_max(I):
    new_rows = (I.shape[0] + 1) // 2
    new_cols = (I.shape[1] + 1) // 2
    new_I = np.zeros((new_rows, new_cols, I.shape[2]), dtype = "uint8")
    for i in range(0, I.shape[0], 2):
        for j in range(0, I.shape[1], 2):
            for c in range(I.shape[2]):
                subpatch = I[i:min(i + 2, I.shape[0]), j:min(j + 2, I.shape[1]), c]
                maximum = np.max(subpatch)
                new_I[i // 2, j // 2, c] = maximum.astype("uint8")
                if i == I.shape[0] and j == I.shape[1]:
                    new_I[i // 2, j // 2, c] = I[i, j, c]
    return new_I
```

Exercise 8 - Zooming

In [72]:

```
def zoom_in(I, k):
    I2 = np.zeros((I.shape[0]*k, I.shape[1]*k, I.shape[2]), dtype = I.dtype)
    return I2

def interp(I, k, mz):
    for i in range(I.shape[0]):
        for j in range(I.shape[1]):
            mz[i*k:(i+1)*k, j*k:(j+1)*k] = I[i,j]
    return mz

k = int(input("Zoom the image by: "))
mz = zoom_in(I, k)
mi = interp(I, k, mz)

#print("Final result \n", mi)
```

```
Zoom the image by: 2
```

Exercise 9 - Bitplanes swapping

In [85]:

```
def swap_bitplanes(I):
    bitplanes = []
    for bit in range(8):
        for channel in range(I.shape[2]):
            bitplanes.append((I[:, :, channel] >> bit) & 1)
    random.shuffle(bitplanes)
    new_I = np.zeros_like(I, dtype = "uint8")
    bp_index = 0
    for bit in range(8):
        for channel in range(I.shape[2]):
            new_I[:, :, channel] |= (bitplanes[bp_index] << bit).astype("uint8")
            bp_index += 1
    return new_I
```

Exercise 10 - Bayer Pattern (replication interpolation)

In [42]:

```
def bayer_pattern(I):
    new_I = np.zeros_like(I, dtype = "uint8")
    for i in range(I.shape[0]):
        for j in range(I.shape[1]):
            if i % 2 == 0: # Even row
                if j % 2 == 0: # Even column --> green channel
                    new_I[i, j, 1] = I[i, j, 1]
                else: # Odd column --> red channel
                    new_I[i, j, 0] = I[i, j, 0]
            else: # Odd row
                if j % 2 == 0: # Even column --> blue channel
                    new_I[i, j, 2] = I[i, j, 2]
                else:
                    new_I[i, j, 1] = I[i, j, 1]
    return new_I

BP = bayer_pattern(I)

def bayer_rep_interp(BP):
    rep_BP = BP.copy()
    for i in range(BP.shape[0]):
        for j in range(BP.shape[1]):
            if i % 2 == 0 and j % 2 != 0: # Red channel
                if i + 1 < BP.shape[0] and j - 1 >= 0:
                    rep_BP[i:i + 2, j - 1:j + 1, 0] = BP[i, j, 0]
            if i % 2 != 0 and j % 2 == 0: # Blue channel
                if i - 1 >= 0 and j + 1 < BP.shape[1]:
                    rep_BP[i - 1:i + 1, j:j + 2, 2] = BP[i, j, 2]
            if i % 2 == 0 and j % 2 == 0: # Green channel
                if j + 1 < BP.shape[1]:
                    rep_BP[i, j + 1, 1] = BP[i, j, 1]
            if i % 2 != 0 and j % 2 != 0: # Green channel
                if j - 1 >= 0:
                    rep_BP[i, j - 1, 1] = BP[i, j, 1]
    return rep_BP

MN = I.shape[0] * I.shape[1]
for x in range(I.shape[0]):
    for y in range(I.shape[1]):
        MSE = (sum(rep_BP[x, y] - I[x, y])**2) // MN
print("MSE: ", MSE)

PSNR1 = -10 * math.log10(MSE // np.max(I))
PSNR2 = 20 * math.log10(np.max(I) // math.sqrt(MSE))
print("PSNR: ", PSNR2)
```

Exercise 10 - Bayer Pattern (bilinear interpolation)

In [45]:

```
import numpy as np
import cv2

def compute_bilinear_coefficients(P0, P1, P2, P3):
    """
    Calcola i coefficienti per l'interpolazione bilineare.

    Parameters:
    P0, P1, P2, P3: Valori dei pixel (vicini) su una griglia 2x2.

    Returns:
    numpy.ndarray: Coefficienti [a0, a1, a2, a3].
    """
    # Matrice dei coefficienti
    M = np.array([
        [1, 0, 0, 0],
        [1, 1, 0, 0],
        [1, 0, 1, 0],
        [1, 1, 1, 1]
    ])

    # Vettore dei valori dei pixel
    P = np.array([P0, P1, P2, P3])

    # Risoluzione del sistema lineare
    A = np.linalg.solve(M, P)

    return A

def bilinear_interpolation(x, y, coeffs):
    """
    Calcola il valore interpolato in (x, y) usando i coefficienti.

    Parameters:
    x, y: Coordinate normalizzate (valori compresi tra 0 e 1).
    coeffs: Coefficienti [a0, a1, a2, a3].

    Returns:
    float: Valore interpolato.
    """
    a0, a1, a2, a3 = coeffs
    return a0 + a1 * x + a2 * y + a3 * x * y

def bayer_bilinear_interpolation(BP):
    """
    Esegue l'interpolazione bilineare su una matrice Bayer per stimare i valori mar
    nei canali rosso, verde e blu.

    Parameters:
    BP (numpy.ndarray): Immagine in formato Bayer (altezza, larghezza, 3).

    Returns:
    numpy.ndarray: Immagine RGB con tutti i canali stimati.
    """
    interpolated_BP = BP.copy()
    height, width, _ = BP.shape

    # Scorriamo la griglia 2x2 per ogni pixel
    for i in range(1, height - 1):
        for j in range(1, width - 1):
            # Griglia 2x2 intorno al pixel
            neighbors = {
                'red': [
                    BP[i, j, 0], BP[i + 1, j, 0], BP[i, j + 1, 0], BP[i + 1, j + 1,
                ],
                'green': [
                    BP[i, j, 1], BP[i + 1, j, 1], BP[i, j + 1, 1], BP[i + 1, j + 1,
                ],
                'blue': [
                    BP[i, j, 2], BP[i + 1, j, 2], BP[i, j + 1, 2], BP[i + 1, j + 1,
                ]
            }

            # Calcola i coefficienti per ogni canale
            red_coeffs = compute_bilinear_coefficients(*neighbors['red'])
            green_coeffs = compute_bilinear_coefficients(*neighbors['green'])
            blue_coeffs = compute_bilinear_coefficients(*neighbors['blue'])

            # Coordinate normalizzate (frazione all'interno della griglia 2x2)
            x, y = 0.5, 0.5  # Esempio: il centro della griglia

            # Interpola i valori
            interpolated_BP[i, j, 0] = bilinear_interpolation(x, y, red_coeffs)
            interpolated_BP[i, j, 1] = bilinear_interpolation(x, y, green_coeffs)
            interpolated_BP[i, j, 2] = bilinear_interpolation(x, y, blue_coeffs)

    return interpolated_BP
```

Bitplanes

Reconstruct an image from a bitplane (simulation 1, exercise 1)

In [51]:

```
t = int(input("Value in {0, 1}: "))
if t != 0 and t != 1:
    raise ValueError("Invalid input!")

ptOut = "C:\\Users\\emanu\\Desktop\\Images directory"

im = cv2.imread(r"C:\Users\emanu\Immagini\15343NSM_yoshi_main.jpg")
im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
#plt.imshow(im, cmap = "gray")
#plt.show()

def bitplane(im, t, ptOut):
    # Bitplane creation
    bitplanes = []
    for bit in range(8):
        bitplanes.append((im >> bit) & 1)
    BP_im = np.array(bitplanes)
    # Set to zero
    if t == 0:
        BP_im[1::2] = 0
    if t == 1:
        BP_im[::2] = 0
    # Image reconstruction
    reconstructed_im = np.zeros_like(BP_im[0], dtype = "uint8")
    for bit in range(8):
        reconstructed_im += (BP_im[bit] << bit)
    # Saving results
    t_file = input("File: ")
    filename = f"bitplane_{t_file}.png"
    os.makedirs(ptOut, exist_ok = True)
    output_path = os.path.join(ptOut, filename)
    cv2.imwrite(output_path, reconstructed_im)
    # Display
    plt.imshow(reconstructed_im, cmap = "gray")
    plt.show()
```

```
Value in {0, 1}: 0
```

Show the bitplanes in a single image (simulation 4, exercise 1, t == even number)

In [161…

```
def random(im, t):
    # Bitplanes
    if t % 2 == 0:
        im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
        x = rd.randint(0, im.shape[0])
        y = rd.randint(0, im.shape[1])
        h = rd.randint(0, im.shape[0] - x)
        w = rd.randint(0, im.shape[1] - y)
        patch_im = im[x:x + h, y:y + w]
        bitplanes = []
        for bit in range(8):
            bitplanes.append((patch_im >> bit) & 1)
        bp_img = np.concatenate(bitplanes, axis = 1) * 255
        # Saving results
        filename = f"bitplane_conct.png"
        os.makedirs(ptOut, exist_ok = True)
        output_path = os.path.join(ptOut, filename)
        cv2.imwrite(output_path, bp_img)
        # Display
        plt.imshow(bp_img, cmap = "gray")
        plt.show()
    # Half channels
    else:
        red_half = np.zeros_like(im, dtype = "uint8")
        blue_half = np.zeros_like(im, dtype = "uint8")
        red_half[:, :im.shape[1] // 2, 0] = im[:, :im.shape[1] // 2, 0]
        blue_half[:, im.shape[1] // 2:, 2] = im[:, im.shape[1] // 2:, 2]
        RB = red_half + blue_half
        # Saving results
        filename = f"half_channels.png"
        os.makedirs(ptOut, exist_ok = True)
        output_path = os.path.join(ptOut, filename)
        cv2.imwrite(output_path, RB)
        # Display
        plt.imshow(RB)
        plt.show()
```

Image mosaic

In [173…

```
# MAKING A MOSAIC WITH AN IMAGE

rd.seed(42)
img = cv2.imread(r"C:\Users\emanu\Immagini\15343NSM_yoshi_main.jpg")
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
n = 50
img1 = img.copy()
img = img[0:img.shape[0]//n*n, 0:img.shape[1]//n*n, :]
A = [(x, y) for x in range(0, img.shape[0], n) for y in range(0, img.shape[1], n)]
rd.shuffle(A)
for a in range(0, img.shape[0], n):
    for b in range(0, img.shape[1], n):
        C = A.pop()
        x, y = C
        img1[a:a + n,b:b + n] = img[x:x + n,y:y + n]

cv2.imshow("Shuffled Image", cv2.cvtColor(img1, cv2.COLOR_RGB2BGR))
cv2.waitKey(0)
cv2.destroyAllWindows()
```

In [172…

```
# Coming back
# COMING BAAAAACK!!!!!!

B = [(x, y) for x in range(0, img.shape[0], n) for y in range(0, img.shape[1], n)]
D = B.copy()
rd.shuffle(B)
B = B[::-1]
img2 = img1.copy()
for a in range(0, img.shape[0], n):
    for b in range(0, img.shape[1], n):
        C = B.index((a,b))
        x = D[C][0]
        y = D[C][1]
        img2[a:a + n, b:b + n] = img1[x:x + n, y:y + n]
cv2.imshow("Original Image", cv2.cvtColor(img2, cv2.COLOR_RGB2BGR))
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Histogram

Histogram computation (absolute frequencies, relative frequencies, equalization)

In [2]:

```
def histogram(img):

    hist = np.zeros(256, dtype = "int")

    # Absolute frequencies
    for x in range(img.shape[0]):
        for y in range(img.shape[1]):
            for z in range(img.shape[2]):
                hist[(img[x, y, z])] += 1
    #plt.bar(range(256), pr, color='black', width=1.0)
    #plt.xlim(0, 255)  # Limita l'asse x al range dei valori dei pixel
    #plt.show()

    # Relative frequencies
    pr = np.zeros(256, dtype = "float")
    MN = img.shape[0] * img.shape[1]
    for i in range(256):
        pr[i] = hist[i] / MN
    #plt.bar(range(256), pr, color='black', width=1.0)
    #plt.xlim(0, 255)  # Limita l'asse x al range dei valori dei pixel
    #plt.show()

    # Equalization algorithm
    eq = np.zeros(256, dtype = "float")
    for i in range(256):
        summation = 0.0
        for k in range(i):
            summation += pr[i]
        eq[i] = (255 // MN) * summation
    plt.bar(range(256), eq, color='black', width=1.0)
    plt.xlim(0, 255)  # Limita l'asse x al range dei valori dei pixel
    plt.show()
```

Histogram in random blocks (simulation 2, exercise 3)

In [203…

```
def blocks(im):
    padded_h = ((im.shape[0] + 7) // 8) * 8
    padded_w = ((im.shape[1] + 7) // 8) * 8
    padded_im = np.zeros((padded_h, padded_w, im.shape[2]), dtype = "uint8")
    padded_im[:im.shape[0], :im.shape[1], :] = im
    # Blocks creation
    blocks = []
    for i in range(0, padded_h, 8):
        for j in range(0, padded_w, 8):
            block = padded_im[i:i + 8, j:j + 8, :]
            blocks.append(block)
    # Histogram per block
    for idx, block in enumerate(blocks):
        for z in range(im.shape[2]):
            hist = np.zeros(256, dtype = "int")
            for x in range(block.shape[0]):
                for y in range(block.shape[1]):
                    hist[(block[x, y, z])] += 1
            plt.bar(range(256), hist, color='black', width=1.0)
            plt.xlim(0, 255)
            plt.title(f"Histogram for block {idx}, channel {z}")
            plt.show()
```

Punctual operators

Negative

In [56]:

```
def negative(img):
    neg_img = img.copy()
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            for c in range(img.shape[2]):
                neg_img[i, j, c] = 255 - img[i, j, c]
    #plt.imshow(neg_img)
    #plt.show()
```

Binarization

In [ ]:

```
def binarization(img): # Punctual operator
    bw_img = np.zeros_like(img)
    T = int(input("Threshold in [0, 255]: "))
    if T < 0 or T > 255:
        raise ValueError("Insert valid threshold.")
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            for c in range(img.shape[2]):
                if img[i, j, c] <= T:
                    bw_img[i, j, c] = 0
                else:
                    bw_img[i, j, c] = 255
    plt.imshow(bw_img, cmap = "gray")
    plt.show()
    return bw_img
```

Image darkening

In [56]:

```
def img_darkening(img):
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    dark = img.copy()
    for x in range(img.shape[0]):
        for y in range(img.shape[1]):
            if img[x, y] > 100:
                dark[x, y] = img[x, y] - 100
            else:
                dark[x, y] = 100 - img[x, y]
    dark = np.clip(dark, 0, 255).astype("uint8")
    plt.imshow(dark, cmap = "gray")
    plt.show()
```

Image clarification

In [58]:

```
def img_clarification(img):
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    light = img.copy()
    for x in range(img.shape[0]):
        for y in range(img.shape[1]):
            light[x, y] = img[x, y] + 100
    light = np.clip(light, 0, 255).astype("uint8")
    plt.imshow(light, cmap = "gray")
    plt.show()
```

Contrast decrease

In [75]:

```
def contrast_decrease(img):
    # Calcola il contrasto utilizzando una formula scalabile
    # Convertiamo i valori dei pixel a float per evitare problemi di overflow
    img = img.astype(float)

    # Centro del contrasto attorno a 127 e lo aumenta
    factor = 0.5  # Fattore di contrasto, maggiore di 1 aumenta il contrasto
    mean_value = 127  # Punto centrale per il contrasto
    contr = mean_value + factor * (img - mean_value)

    # Clippiamo i valori per restare nel range valido per le immagini
    contr = np.clip(contr, 0, 255).astype("uint8")

    # Visualizzazione del risultato
    plt.imshow(contr)
    plt.axis('off')  # Nascondi gli assi
    plt.show()
```

Contrast increase

In [77]:

```
def contrast_increase(img):
    # Calcola il contrasto utilizzando una formula scalabile
    # Convertiamo i valori dei pixel a float per evitare problemi di overflow
    img = img.astype(float)

    # Centro del contrasto attorno a 127 e lo aumenta
    factor = 1.5  # Fattore di contrasto, maggiore di 1 aumenta il contrasto
    mean_value = 127  # Punto centrale per il contrasto
    contr = mean_value + factor * (img - mean_value)

    # Clippiamo i valori per restare nel range valido per le immagini
    contr = np.clip(contr, 0, 255).astype("uint8")

    # Visualizzazione del risultato
    plt.imshow(contr)
    plt.axis('off')  # Nascondi gli assi
    plt.show()
```

Local operators: convolution

In [2]:

```
kernel = np.array([[-1, 0, -1], [0, 4, 0], [-1, 0, -1]]) # laplacian kernel (see th
```

Types of convolution (standard, zero padding, toroidal padding, replication padding)

In [104…

```
def std_convolution(img, kernel): # no padding
    conv_img = np.zeros_like(img, dtype = "uint8")
    for i in range(1, img.shape[0] - 2):
        for j in range(1, img.shape[1] - 2):
            subpatch = img[i - 1:i + 2, j - 1:j + 2, :]
            product = subpatch * kernel
            result = np.sum(product)
            conv_img[i, j, :] = result
    plt.imshow(conv_img)
    plt.show()

def zero_convolution(img, kernel): # zero padding
    conv_img = np.zeros_like(img, dtype = "uint8")
    pad_zero = np.zeros((img.shape[0] + 2, img.shape[1] + 2, img.shape[2]), dtype =
    pad_zero[1:-1, 1:-1, :] = img
    for i in range(1, pad_zero.shape[0] - 1):
        for j in range(1, pad_zero.shape[1] - 1):
            subpatch = pad_zero[i - 1:i + 2, j - 1:j + 2, :]
            product = subpatch * kernel
            result = np.sum(product)
            conv_img[i - 1, j - 1, :] = result
    plt.imshow(conv_img)
    plt.show()

def toroid_convolution(img, kernel): # toroidal padding
    conv_img = np.zeros_like(img, dtype = "uint8")
    pad_trd = np.zeros((img.shape[0] + 2, img.shape[1] + 2, img.shape[2]), dtype =
    pad_trd[1:-1, 1:-1, :] = img
    # Toroidal rows
    pad_trd[0, 1:-1, :] = img[-1, :, :] # first row (no corners)
    pad_trd[-1, 1:-1, :] = img[0, :, :] # last row (no corners)
    # Toroidal columns
    pad_trd[1:-1, 0, :] = img[:, -1, :] # first column (no corners)
    pad_trd[1:-1, -1, :] = img[:, 0, :] # last column (no corners)
    # Corners
    pad_trd[0, 0, :] = img[-1, -1, :] # top-left corner (copied from bottom-right)
    pad_trd[0, -1, :] = img[-1, 0, :] # top-right corner(copied from bottom-left)
    pad_trd[-1, 0, :] = img[0, -1, :] # bottom-left corner (copied from top-right)
    pad_trd[-1, -1, :] = img[0, 0, :] # bottom-right corner (copied from top-left)
    # Convolution
    for i in range(1, pad_trd.shape[0] - 1):
        for j in range(1, pad_trd.shape[1] - 1):
            subpatch = pad_trd[i - 1:i + 2, j - 1:j + 2, :]
            product = subpatch * kernel
            result = np.sum(product)
            conv_img[i - 1, j - 1, :] = result
    plt.imshow(conv_img)
    plt.show()

def rep_convolution(img, kernel):
    conv_img = np.zeros_like(img, dtype = "uint8")
    pad_rep = np.zeros((img.shape[0] + 2, img.shape[1] + 2, img.shape[2]), dtype =
    pad_rep[1:-1, 1:-1, :] = img
    # Replicated rows
    pad_rep[0, 1:-1, :] = img[0, :, :] # first row (no corners)
    pad_rep[-1, 1:-1, :] = img[-1, :, :] # last row (no corners)
    # Replicated columns
    pad_rep[1:-1, 0, :] = img[:, 0, :] # first column (no corners)
    pad_rep[1:-1, -1, :] = img[:, -1, :] # last column (no corners)
    # Corners
    pad_rep[0, 0, :] = img[0, 0, :] # top-left corner
    pad_rep[0, -1, :] = img[0, -1, :] # top-right corner
    pad_rep[-1, 0, :] = img[-1, 0, :] # bottom-left corner
    pad_rep[-1, -1, :] = img[-1, -1, :] # bottom-right corner
    # Convolution
    for i in range(1, pad_rep.shape[0] - 1):
        for j in range(1, pad_rep.shape[1] - 1):
            subpatch = pad_rep[i - 1:i + 2, j - 1:j + 2, :]
            product = subpatch * kernel
            result = np.sum(product)
            conv_img[i - 1, j - 1, :] = result
    plt.imshow(conv_img)
    plt.show()
```

Mathematical morphology

In [175…

```
def opening(img):
    op_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    kernel = np.ones((3, 3), dtype = "uint8")
    op_img = cv2.erode(op_img, kernel, iterations = 10)
    op_img = cv2.dilate(op_img, kernel, iterations = 10)
    op_img = np.clip(op_img, 0, 255).astype(np.uint8)
    plt.imshow(op_img, cmap = "gray")
    return op_img

def closing(img):
    cl_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    kernel = np.ones((3, 3), dtype = "uint8")
    cl_img = cv2.dilate(cl_img, kernel, iterations = 10)
    cl_img = cv2.erode(cl_img, kernel, iterations = 10)
    cl_img = np.clip(cl_img, 0, 255).astype(np.uint8)
    plt.imshow(cl_img, cmap = "gray")
    return cl_img

def edge_extraction(img):
    kernel = np.ones((3, 3), dtype = "uint8")
    D = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    D = cv2.dilate(img, kernel, iterations = 10)
    E = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    E = cv2.erode(img, kernel, iterations = 10)
    edge = D - E
    return edge

def contour_extraction(img):
    kernel = np.ones((3, 3), dtype = "uint8")
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    E = cv2.erode(img, kernel, iterations = 10)
    contour = img - E
    return contour

def outer_borders(img):
    kernel = np.ones((3, 3), dtype = "uint8")
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    D = cv2.dilate(img, kernel, iterations = 10)
    outer = img - D
    return outer

def laplacian(img):
    kernel = np.ones((3, 3), dtype = "uint8")
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    D = cv2.dilate(img, kernel, iterations = 10)
    E = cv2.erode(img, kernel, iterations = 10)
    lap = D + E - 2 * img
    return lap
```

Skeletonization

In [65]:

```
img = cv2.imread(r"C:\Users\emanu\Immagini\edd732e3-1671-4eff-9f59-a620cd9011ec.jpg
size = np.size(img)
shape = np.zeros(img.shape, dtype = "uint8")
skel = np.zeros(img.shape, dtype="uint8")

ret, img = cv2.threshold(img, 127, 255, 0)
element = cv2.getStructuringElement(cv2.MORPH_CROSS, (3, 3))
```

```
print("Element ", element)
done = False

while (not done):
    eroded = cv2.erode(img, element)
    temp = cv2.dilate(eroded, element)
    temp = cv2.subtract(img, temp)
    skel = cv2.bitwise_or(skel, temp)
    img = eroded.copy()

    zeros = size - cv2.countNonZero(img)
    if zeros == size:
        done = True

#plt.imshow(skel, cmap = "gray")
```

```
Element  [[0 1 0]
 [1 1 1]
 [0 1 0]]
```

Fourier transform

In [64]:

```
img = img[:,:,0]
# Apply 2D FFT and shift the frequencies
f_transform = np.fft.fft2(img)
f_shift = np.fft.fftshift(f_transform)
# Obtain the magnitude spectrum and plot it
magnitude_spectrum = 20 * np.log(np.abs(f_shift))
#plt.imshow(magnitude_spectrum, cmap='gray')
#plt.title('Magnitude Spectrum')
#plt.show()
```

Color spaces

In [14]:

```
def RGB2YCbCr(img):
    R = img[:, :, 0]
    G = img[:, :, 1]
    B = img[:, :, 2]
    imgYCC = cv2.cvtColor(img, cv2.COLOR_BGR2YCR_CB)
    Y = imgYCC[:, :, 0]
    CR = imgYCC[:, :, 1]
    CB = imgYCC[:, :, 2]
    channels = [R, G, B, Y, CR, CB]
    rd_channels = rd.sample(channels, 3)
    output = cv2.merge(rd_channels)
    plt.imshow(output)
```

Video processing

Capture a video from your webcam

In [3]:

```
def camframe_saving(ptOut):
    cam = cv2.VideoCapture(0)
    if not cam.isOpened():
        print("Errore: Impossibile aprire la fotocamera")
        return
    frame_height = int(cam.get(cv2.CAP_PROP_FRAME_HEIGHT))
    frame_width = int(cam.get(cv2.CAP_PROP_FRAME_WIDTH))
    fps = int(cam.get(cv2.CAP_PROP_FPS))
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter("output.mp4", fourcc, fps, (frame_width, frame_height))
    frame_count = 0

    while True:
        ret, frame = cam.read()
        if not ret:
            break

        out.write(frame)
        filename = f"frame_{frame_count}.png"
        output_path = os.path.join(ptOut, filename)
        cv2.imwrite(output_path, frame)
        frame_count += 1

        # Mostra il frame in una finestra
        cv2.imshow("Video Capture", frame)

        # Controlla se è stato premuto 'q'
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cam.release()
    out.release()
    cv2.destroyAllWindows()
```

In [8]:

```
video_path = r"C:\Users\emanu\Downloads\Toad Scream (1).mp4"
ptOut = r"C:\Users\emanu\Desktop\Video exercise\Camera frames"
```

Digital video patch (simulation 5, exercise 1)

In [19]:

```
def video_patch(video_path, ptOut):
    video = cv2.VideoCapture(video_path)
    frame_height = int(video.get(cv2.CAP_PROP_FRAME_HEIGHT))
    frame_width = int(video.get(cv2.CAP_PROP_FRAME_WIDTH))
    fps = int(video.get(cv2.CAP_PROP_FPS))
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter("output.mp4", fourcc, fps, (frame_width, frame_height))

    x = int(input("x coordinate: "))
    if x < 0 or x > frame_width:
        raise ValueError("Invalid input.")
    y = int(input("y coordinate: "))
    if y < 0 or y > frame_height:
        raise ValueError("Invalid input.")
    w = int(input("width: "))
    if x + w < 0 or x + w > frame_width:
        raise ValueError("Invalid input.")
    h = int(input("height: "))
    if y + h < 0 or y + h > frame_height:
        raise ValueError("Invalid input.")

    frame_count = 0

    while True:
        ret, frame = video.read()
        if not ret:
            break
        P = frame[y:y + h, x:x + w, :]
        out.write(P)
        filename = f"patch_{frame_count}.png"
        output_path = os.path.join(ptOut, filename)
        cv2.imwrite(output_path, P)
        frame_count += 1

        cv2.imshow("Video Capture", P)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    video.release()
    out.release()
    cv2.destroyAllWindows()
```

In [20]:

```
def rd_video_patch(video_path, ptOut):
    video = cv2.VideoCapture(video_path)
    frame_height = int(video.get(cv2.CAP_PROP_FRAME_HEIGHT))
    frame_width = int(video.get(cv2.CAP_PROP_FRAME_WIDTH))
    fps = int(video.get(cv2.CAP_PROP_FPS))
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter("output.mp4", fourcc, fps, (frame_width, frame_height))

    x = int(input("x coordinate: "))
    if x < 0 or x > frame_width:
        raise ValueError("Invalid input.")
    y = int(input("y coordinate: "))
    if y < 0 or y > frame_height:
        raise ValueError("Invalid input.")
    w = int(input("width: "))
    if x + w < 0 or x + w > frame_width:
        raise ValueError("Invalid input.")
    h = int(input("height: "))
    if y + h < 0 or y + h > frame_height:
        raise ValueError("Invalid input.")

    frame_count = 0

    while True:
        ret, frame = video.read()
        if not ret:
            break
        x_2 = rd.randint(0, frame_width - w)
        y_2 = rd.randint(0, frame_height - h)
        P2 = frame[y_2:y_2 + h, x_2:x_2 + w, :]
        frame[y:y + h, x:x + w, :] = P2
        out.write(frame)
        filename = f"patch_{frame_count}.png"
        output_path = os.path.join(ptOut, filename)
        cv2.imwrite(output_path, frame)
        frame_count += 1

        cv2.imshow("Video Capture", frame)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    video.release()
    out.release()
    cv2.destroyAllWindows()
```

In [26]:

```
def video_sobel(video_path, ptOut):
    video = cv2.VideoCapture(video_path)
    frame_height = int(video.get(cv2.CAP_PROP_FRAME_HEIGHT))
    frame_width = int(video.get(cv2.CAP_PROP_FRAME_WIDTH))
    fps = int(video.get(cv2.CAP_PROP_FPS))
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter("output.mp4", fourcc, fps, (frame_width, frame_height))
    sobel_x = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
    sobel_y = np.array([[-1, 0, 1], [-2, 0, 2], [- 1, 0, 1]])
    frame_count = 0

    while True:
        ret, frame = video.read()
        if not ret:
            break
        new_frame = frame.copy()
        for i in range(1, frame.shape[0] - 2):
            for j in range(1, frame.shape[1] - 2):
                for c in range(frame.shape[2]):
                    subpatch = frame[i - 1:i + 2, j - 1:j + 2, c]
                    if frame_count % 2 != 0: # odd frame
                        product = subpatch * sobel_x
                        result = np.sum(product)
                        new_frame[i, j, c] = result
                    else: # even frame
                        product = subpatch * sobel_y
                        result = np.sum(product)
                        new_frame[i, j, c] = result
        out.write(new_frame)
        filename = f"patch_{frame_count}.png"
        output_path = os.path.join(ptOut, filename)
        cv2.imwrite(output_path, new_frame)
        frame_count += 1

    video.release()
    out.release()
    cv2.destroyAllWindows()
```

In [36]:

```
def video_binary(video_path, ptOut):
    video = cv2.VideoCapture(video_path)
    frame_height = int(video.get(cv2.CAP_PROP_FRAME_HEIGHT))
    frame_width = int(video.get(cv2.CAP_PROP_FRAME_WIDTH))
    fps = int(video.get(cv2.CAP_PROP_FPS))
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter("output.mp4", fourcc, fps, (frame_width, frame_height))

    kernel = np.ones((3, 3), dtype = "uint8")
    T = int(input("Threshold for binarization: "))

    while True:
        ret, F = video.read()
        if not ret:
            break
        Fb = cv2.cvtColor(F, cv2.COLOR_BGR2GRAY)
        for i in range(Fb.shape[0]):
            for j in range(Fb.shape[1]):
                if Fb[i, j] <= T:
                    Fb[i, j] = 0
                else:
                    Fb[i, j] = 255
        D = cv2.dilate(Fb, kernel, iterations = 5)
        E = cv2.erode(Fb, kernel, iterations = 5)
        D_E = D - E

    fig = plt.figure()
    ax1 = fig.add_subplot(2, 2, 1)
    ax1.imshow(Fb, cmap = "gray")
    ax2 = fig.add_subplot(2, 2, 2)
    ax2.imshow(D, cmap = "gray")
    ax3 = fig.add_subplot(2, 2, 3)
    ax3.imshow(E, cmap = "gray")
    ax4 = fig.add_subplot(2, 2, 4)
    ax4.imshow(D_E, cmap = "gray")

    video.release()
    out.release()
    cv2.destroyAllWindows()
```

Images overlapping

In [ ]:

```
def images_overlap(img, sfn):
    # Converti l'immagine di input in HSV
    hsv_img = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)

    # Ridimensiona sfn per adattarlo a img
    sfn = cv2.resize(sfn, (hsv_img.shape[1], hsv_img.shape[0]))

    # Crea una copia dell'immagine HSV per il risultato
    result = hsv_img.copy()

    # Correggi l'ordine dei cicli for: altezza (riga) prima, larghezza (colonna) do
    for x in range(hsv_img.shape[0]):  # Altezza (righe)
        for y in range(hsv_img.shape[1]):  # Larghezza (colonne)
            # Applica la condizione sul canale Hue (H)
            if hsv_img[x, y, 0] > 35 and hsv_img[x, y, 0] < 65:
                result[x, y] = sfn[x, y]  # Sovrapponi il pixel di sfn

    # Converti il risultato da HSV a BGR per la visualizzazione
    result = cv2.cvtColor(result, cv2.COLOR_HSV2BGR)

    # Mostra l'immagine risultante usando matplotlib
    plt.imshow(cv2.cvtColor(result, cv2.COLOR_BGR2RGB))
    plt.axis("off")
    plt.show()

    return result
```

Interactive frame over an image

In [ ]:

```
# Shape video
video_path = r"C:\Users\andre\Downloads\Green_fire.mp4"
cap = cv2.VideoCapture(video_path)
ret, frame = cap.read()
w,h,d= frame.shape

# Background
sfn= cv2.imread(r'C:\Users\andre\Python notebooks\Media\sfondo.jpeg')
sfn = cv2.cvtColor(sfn, cv2.COLOR_BGR2HSV)
# Ridimensiona l'immagine come sarà la dimensione del video
sfn = cv2.resize(sfn, (h, w))

# Video reading

n=0
a=1
# Specifica il percorso del file video
video_path = r"C:\Users\andre\Downloads\Green_fire.mp4"
cap = cv2.VideoCapture(video_path)

while True:
    # Legge un fotogramma dal file video
    ret, frame = cap.read()
```

```
# MODIFICHE AL VIDEO
    hsv_img = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    for x in range(hsv_img.shape[0]):
        for y in range(hsv_img.shape[1]):
            if hsv_img[x,y,0] < 65 and hsv_img[x,y,0] > 35:
                hsv_img[x,y]=sfn[x,y]
    frame = cv2.cvtColor(hsv_img, cv2.COLOR_HSV2BGR)

    # Se non ci sono più fotogrammi, esci dal ciclo
    if not ret:
        print("Fine del video.")
        break

    # Percorso della cartella in cui vuoi salvare l'immagine
    output_folder = r'C:\Users\andre\Python notebooks\Media\frames'
    # Nome del file di output
    n+=1
    output_file = os.path.join(output_folder, f'saved_image{n}.jpg')
    # Salva l'immagine
    cv2.imwrite(output_file, frame)
    # Esci dal ciclo premendo 'q'
    if cv2.waitKey(30) & 0xFF == ord('q'):  # 30 ms di ritardo tra i frame
        break
    elif n>=150:
        break
# Rilascia il video e chiudi tutte le finestre
cap.release()
cv2.destroyAllWindows()

#Visualizzare i frame salvati come video

# Percorso della cartella contenente i frame
frames_folder = r'C:\Users\andre\Python notebooks\Media\frames'
fps = 30

# Ottieni la lista dei file nella cartella e ordina i nomi
frame_files = (os.listdir(frames_folder))  # Ordine alfabetico


# Riproduzione dei frame
for frame_file in frame_files:
    # Costruisci il percorso completo del frame
    frame_path = os.path.join(frames_folder, frame_file)

    # Leggi il frame
    frame = cv2.imread(frame_path)

    if frame is None:
        print(f"Impossibile leggere il frame: {frame_file}")
        continue

    # Mostra il frame
    cv2.imshow('Video', frame)

    # Attendi un periodo in base agli FPS
    if cv2.waitKey(int(1000 / fps)) & 0xFF == ord('q'):
        break  # Esci premendo 'q'

# Chiudi la finestra di visualizzazione
cv2.destroyAllWindows()
```

Other exercises (from exam simulations)

Blocks swapping (simulation 4, exercise 3)

In [7]:

```
def swap_blocks(img):
    padded_height = ((img.shape[0] + 4) // 5) * 5
    padded_width = ((img.shape[1] + 4) // 5) * 5
    padded_img = np.zeros((padded_height, padded_width, img.shape[2]), dtype = "uir
    padded_img[:img.shape[0], :img.shape[1], :] = img
    # Blocks creation
    blocks = []
    for i in range(0, padded_height, 5):
        for j in range(0, padded_width, 5):
            block = padded_img[i:i + 5, j:j + 5, :]
            blocks.append(block)
    for b in blocks:
        for i in range(b.shape[0]):
            for j in range(b.shape[1]):
                b[i, i, :] = b[i, 5 - i - 1, :]
                b[i, 5 - i - 1, :] = b[i, i]
                if i != j and j != 5 - i - 1:
                    b[i, j] = 255 - b[i, j]
                padded_img[i:i + 5, j:j + 5, :] = block
    # Results
    plt.imshow(padded_img)
```

Operations on 4 blocks (simulation 3, exercise 3)

In [30]:

```
def random_blocks(img):
    # Padding 4x4
    padded_height = ((img.shape[0] + 3) // 4) * 4
    padded_width = ((img.shape[1] + 3) // 4) * 4
    padded_img = np.zeros((padded_height, padded_width, img.shape[2]), dtype = "uir
    padded_img[:img.shape[0], :img.shape[1], :] = img
    # Blocks creation
    B1 = padded_img[:img.shape[0] // 2, :img.shape[1] // 2, :]
    B2 = padded_img[:img.shape[0] // 2, img.shape[1] // 2:, :]
    B3 = padded_img[img.shape[0] // 2:, :img.shape[1] // 2, :]
    B4 = padded_img[img.shape[0] // 2:, img.shape[1] // 2:, :]
    # B1: mathematical morphology
    kernel = np.ones((3, 3), dtype = "uint8")
    D = cv2.dilate(B1, kernel, iterations = 5)
    E = cv2.erode(B1, kernel, iterations = 5)
    out_B1 = D - E
    # B2: interpolation
    out_B2 = B2[B2.shape[0] // 2:, B2.shape[1] // 2:, :]
    out_B2 = cv2.resize(out_B2, (B2.shape[1], B2.shape[0]), interpolation = cv2.INT
    # B3: negative
    out_B3 = B3.copy()
    for i in range(B3.shape[0]):
        for j in range(B3.shape[1]):
            for c in range(B3.shape[2]):
                out_B3[i, j, c] = 255 - B3[i, j, c]
    # B4: contrast increase
    out_B4 = B4.astype(float)
    factor = 1.5
    mean_value = 127
    out_B4 = mean_value + factor * (B4 - mean_value)
    out_B4 = np.clip(out_B4, 0, 255).astype("uint8")
    # Concatenation to get final output
    B1_B3 = cv2.vconcat([out_B1, out_B3])
    B2_B4 = cv2.vconcat([out_B2, out_B4])
    min_h = min(B1_B3.shape[0], B2_B4.shape[0])
    min_w = min(B1_B3.shape[1], B2_B4.shape[1])
    B1_B3 = cv2.resize(B1_B3, (min_w, min_h))
    B2_B4 = cv2.resize(B2_B4, (min_w, min_h))
    output = cv2.hconcat([B1_B3, B2_B4])
    plt.imshow(output)
```

Images merging (simulation 2, exercise 1)

In [39]:

```
def merge(im1, im2):
    # Initialization
    tp = int(input("Insert 0 or 1: "))
    if tp != 0 and tp != 1:
        raise ValueError("Invalid input.")
    conct = input("Concatenation type (h or v): ")
    if conct != "h" and conct != "v":
        raise ValueError("Invalid input")
    # Resizing
    min_h = min(im1.shape[0], im2.shape[0])
    min_w = min(im1.shape[1], im2.shape[1])
    im1 = cv2.resize(im1, (min_w, min_h))
    im2 = cv2.resize(im2, (min_w, min_h))
    # Fractions of im1
    im1_0H = im1[:, :min_w // 2, :]
    im1_0V = im1[min_h // 2:, :, :]
    im1_1H = im1[:, min_w // 2:, :]
    im1_1V = im1[:min_h // 2, :, :]
    # Fractions of im2
    im2_0H = im2[:, min_w // 2:, :]
    im2_0V = im2[:min_h // 2, :, :]
    im2_1H = im2[:, :min_w // 2, :]
    im2_1V = im2[min_h // 2:, :, :]
    # Merging
    if tp == 0 and conct == "h":
        merged = cv2.hconcat([im1_0H, im2_0H])
    if tp == 0 and conct == "v":
        merged = cv2.vconcat([im2_0V, im1_0V])
    if tp == 1 and conct == "h":
        merged = cv2.hconcat([im2_1H, im1_1H])
    if tp == 1 and conct == "v":
        merged = cv2.vconcat([im1_1V, im2_1V])
    # Display
    plt.imshow(merged)
```