Họ tên : Nguyễn Quang Anh

Mã sinh viên : B21DCDT036

Chương 2 : Lấy mẫu , Biến đổi Fourier, và Tích chập (Sampling, Fourier Transform, and Convolution)

PHẦN 1 : LẤY MẪU VÀ LƯỢNG TỬ HÓA (SAMPLING)

Phần 1 – a : Lấy mẫu

Sampling (lấy mẫu) là quá trình chọn hoặc loại bỏ các pixel của ảnh, nghĩa là nó là một thao tác không gian (spatial operation). Chúng ta có thể sử dụng kỹ thuật lấy mẫu để tăng hoặc giảm kích thước của một hình ảnh, lần lượt gọi là **up-sampling** (tăng mẫu) và **down-sampling** (giảm mẫu).

**Sampling được chia làm hai loại chính:**

* **Down-Sampling (Giảm mẫu)**:
  + Mục tiêu: Giảm số lượng pixel trong ảnh.
  + Ứng dụng: Nén ảnh, giảm kích thước dữ liệu.
  + Ví dụ: Một ảnh 1000x1000 pixel có thể được giảm xuống còn 500x500 pixel.
* **Up-Sampling (Tăng mẫu)**:
  + Mục tiêu: Tăng số lượng pixel trong ảnh bằng cách nội suy thêm thông tin.
  + Ứng dụng: Phóng to ảnh mà không làm mờ hoặc mất chi tiết.
  + Ví dụ: Một ảnh 500x500 pixel được phóng to thành 1000x1000 pixel.

**Các kỹ thuật Sampling phổ biến**

Sampling được thực hiện bằng các phương pháp cụ thể, bao gồm:

**Down-Sampling**

1. **Nearest Neighbor (Láng giềng gần nhất)**:
   * Giữ lại giá trị pixel gần nhất từ ảnh gốc.
   * Đơn giản và nhanh, nhưng có thể tạo ra ảnh bị răng cưa.
2. **Average Pooling (Lấy trung bình)**:
   * Lấy giá trị trung bình của một nhóm pixel trong vùng lân cận.
   * Tạo ra ảnh mượt mà hơn, nhưng có thể làm mất chi tiết.

##### ****Up-Sampling****

1. **Nearest Neighbor Interpolation (Nội suy láng giềng gần nhất)**:
   * Sao chép giá trị của pixel gần nhất.
   * Dễ triển khai nhưng có thể làm ảnh bị răng cưa.
2. **Bilinear Interpolation (Nội suy song tuyến tính)**:
   * Tính giá trị mới dựa trên nội suy tuyến tính theo cả hai chiều (x, y).
   * Kết quả ảnh mượt hơn so với Nearest Neighbor.
3. **Bicubic Interpolation (Nội suy ba khối)**:
   * Tính giá trị mới bằng cách nội suy theo khối pixel 4x4.
   * Tạo ra ảnh mượt mà hơn Bilinear.

THUẬT TOÁN 1 : THUẬT TOÁN NỘI SUY BẬC BA HAI CHIỀU ( Bi-cubic interpolation )

a, Giới thiệu qua về thuật toán nội suy bậc ba hai chiều :

- **Nội suy bậc ba (cubic interpolation)**: Nội suy giữa các điểm dữ liệu bằng cách sử dụng đa thức bậc ba.

**- Bi-cubic interpolation**: Là mở rộng của nội suy bậc ba từ 1 chiều sang 2 chiều, áp dụng nội suy bậc ba cả trên trục X và Y.

- **Nguyên lý hoạt động**:

* **Lưới 4x4 pixel**: Thuật toán xem xét giá trị của 16 điểm lân cận (4x4) quanh điểm cần nội suy.
* **Công thức nội suy bậc ba**:
  + Tính nội suy bậc ba dọc theo một chiều (ví dụ trục X).
  + Sau đó, tiếp tục nội suy bậc ba trên chiều còn lại (trục Y).

- **Ưu điểm**:

* Tạo ra hình ảnh phóng to mượt mà hơn, ít bị răng cưa so với nội suy song tuyến tính.
* Hiệu quả khi phóng to ảnh với độ chi tiết cao.

b, Các bước thực hiện thuật toán Bi-cubic interpolation

B1: **Chọn điểm cần nội suy**:

* Xác định tọa độ của điểm cần nội suy trên ảnh phóng to (ảnh đầu ra).
* Tính tọa độ tương ứng trên ảnh gốc (ảnh đầu vào) bằng cách chia tỷ lệ.

B2: **Xác định vùng lân cận (4x4 pixel)**:

* Lấy giá trị của 16 điểm ảnh (pixel) gần nhất từ ảnh gốc. Đây là vùng lân cận 4x4 bao quanh điểm nội suy.

B3: **Tính nội suy theo một chiều (trục X)**:

* Với mỗi hàng pixel trong vùng lân cận, áp dụng nội suy bậc ba để tính giá trị nội suy trên trục X.
* Công thức nội suy bậc ba trên trục X: f(x)=ax3+bx2+cx+df(x) = ax^3 + bx^2 + cx + df(x)=ax3+bx2+cx+d Trong đó, a,b,c,da, b, c, da,b,c,d được xác định dựa trên giá trị của 4 pixel liên tiếp.

B4: **Tính nội suy trên chiều còn lại (trục Y)**:

* Sau khi nội suy trên trục X, lấy giá trị từ các điểm đã nội suy trên trục X.
* Áp dụng nội suy bậc ba trên trục Y với giá trị đã tính được.

B5: **Gán giá trị vào pixel nội suy**:

* Giá trị tính được sẽ được gán vào tọa độ tương ứng trên ảnh đầu ra.

B6: **Lặp lại quá trình**:

* Thực hiện lại các bước trên cho từng pixel trong ảnh đầu ra.

c, Giải thích đoạn mã nguồn :

from PIL import Image  
import matplotlib.pyplot as plt  
  
# Mở ảnh gốc  
im = Image.open(r"C:\Users\Dell\Desktop\Sandipan\_Dey\_2018\_Sample\_Images\images\clock.jpg")  
  
# Tăng kích thước ảnh lên 10 lần sử dụng nội suy bậc ba  
im\_resized = im.resize((im.width \* 10, im.height \* 10), Image.BICUBIC)  
  
# Hiển thị ảnh gốc và ảnh phóng to  
plt.figure(figsize=(10, 5))  
  
# Ảnh gốc  
plt.subplot(1, 2, 1)  
plt.title("Original Image")  
plt.imshow(im)  
plt.axis("off")  
  
# Ảnh phóng to (Bi-cubic)  
plt.subplot(1, 2, 2)  
plt.title("Resized Image (Bi-cubic)")  
plt.imshow(im\_resized)  
plt.axis("off")  
  
plt.show()

 PIL (Python Imaging Library): Dùng để xử lý ảnh (mở, chỉnh sửa, thay đổi kích thước, v.v.).

 matplotlib.pyplot: Thư viện vẽ biểu đồ và hiển thị ảnh. Chúng ta dùng nó để hiển thị ảnh gốc và ảnh phóng to.

 Image.open(path): Dùng để mở một file ảnh từ đường dẫn cụ thể.

* **Đường dẫn**: Đảm bảo ảnh clock.jpg nằm ở vị trí bạn đã chỉ định. Nếu không đúng, mã sẽ báo lỗi FileNotFoundError.
* Biến im chứa đối tượng ảnh gốc.

 im.width và im.height: Trả về chiều rộng và chiều cao của ảnh gốc (số pixel).

 **Hàm resize()**:

* **Tham số đầu tiên**: (im.width \* 10, im.height \* 10)
  + Tăng kích thước ảnh lên **10 lần** theo cả chiều rộng và chiều cao.
* **Tham số thứ hai**: Image.BICUBIC
  + Sử dụng thuật toán **bi-cubic interpolation** để nội suy, đảm bảo ảnh được phóng to một cách mượt mà.

 Biến im\_resized chứa ảnh phóng to

**Hàm plt.figure()**: Tạo một cửa sổ vẽ có kích thước 10x5 inch để hiển thị các ảnh

 **Hàm plt.subplot(1, 2, 1)**:

* Chia cửa sổ vẽ thành **1 hàng, 2 cột**.
* Ảnh gốc sẽ hiển thị ở vị trí **ô thứ nhất**.

 **Hàm plt.title("Original Image")**: Đặt tiêu đề cho ảnh gốc là *"Original Image"*.

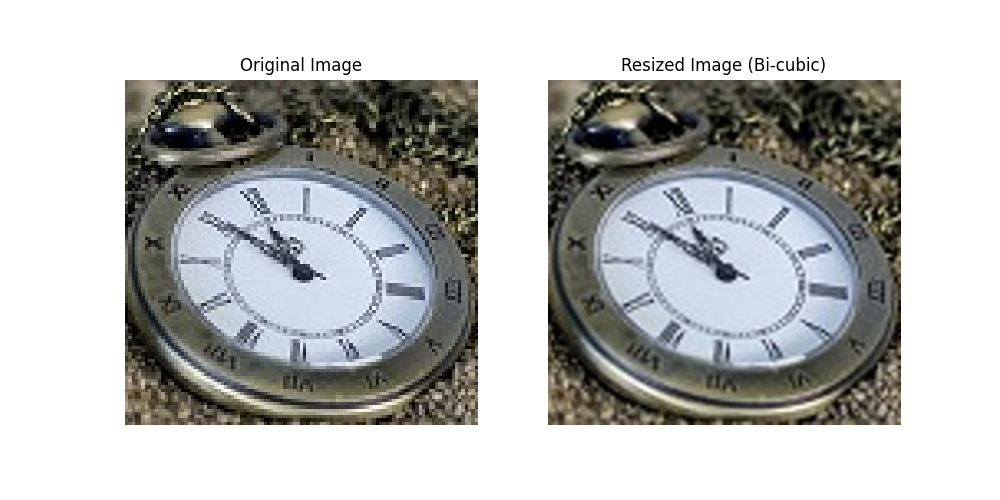
 **Hàm plt.imshow(im)**: Hiển thị ảnh gốc.

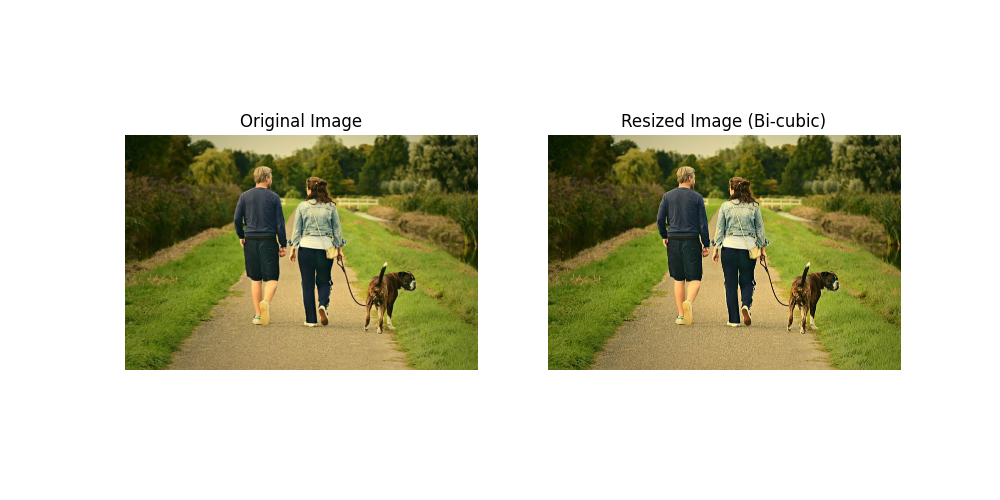
 **Hàm plt.axis("off")**: Tắt hiển thị trục tọa độ x, y để ảnh trông rõ ràng hơn.

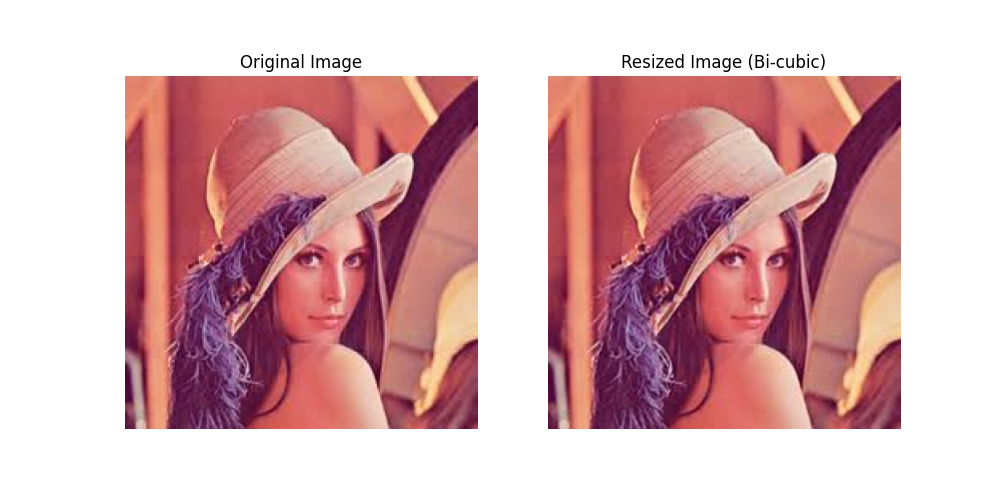
**Hàm plt.show()**: Hiển thị cửa sổ vẽ với hai ảnh:

* **Ảnh gốc** bên trái.
* **Ảnh phóng to (bi-cubic)** bên phải.

d, Hiển thị kết quả

Ví dụ 1 : 

Ví dụ 2 : 

Ví dụ 3 : 

**Nhận xét kết quả đầu ra**

Khi chạy mã này, chúng ta sẽ thấy ảnh :

* **Bên trái**: Ảnh gốc với kích thước ban đầu.
* **Bên phải**: Ảnh đã phóng to bằng bi-cubic interpolation.
  + Ảnh sẽ mượt hơn, ít răng cưa và chi tiết được cải thiện so với các phương pháp nội suy khác như nearest-neighbor hay bi-linear.

Phần 1 – b : Lượng tử hóa ( Quantization )

**Lượng tử hóa (Quantization)** liên quan đến **mức độ cường độ của một ảnh** và có thể được định nghĩa bởi số lượng **bit được sử dụng trên mỗi pixel**. Các ảnh số thường được lượng tử hóa thành **256 mức xám** (8-bit)

THUẬT TOÁN 2 : LƯỢNG TỬ HÓA VỚI THƯ VIỆN PIL ( Quantizing with PIL )

a, Giới thiệu :

 Hàm convert() của PIL với chế độ P và tham số colors chỉ định số lượng màu tối đa có thể có được dùng để thực hiện lượng tử hóa màu sắc :

**Hàm convert('P')**:

* Chuyển ảnh RGB sang **chế độ bảng màu (palette mode)**, phù hợp để giảm số lượng màu.
* **Tham số colors**: Số lượng màu tối đa sau lượng tử hóa.

 Sử dụng hàm signaltonoise() từ thư viện SciPy để tính **Tỷ lệ Tín hiệu trên Nhiễu (SNR)**:

* **SNR** được định nghĩa là trung bình giá trị pixel chia cho độ lệch chuẩn của mảng ảnh.
* Tính tỷ lệ SNR: SNR= mean(I) \ std(I)
* Mean(I) : Giá trị trung bình của các pixel.
* std(I) : Độ lệch chuẩn của các pixel.
* Ý nghĩa: Giá trị SNR càng cao, chất lượng ảnh càng tốt.

b, Các bước thực hiện thuật toán Lượng tử hóa bằng PIL :

**B1 : Chuẩn bị ảnh đầu vào**

* Mở ảnh đầu vào sử dụng thư viện PIL.
* Ảnh đầu vào phải ở định dạng RGB hoặc có thể chuyển đổi được sang chế độ P (bảng màu).

**B2. Xác định số lượng màu cần lượng tử hóa**

* Tạo danh sách số lượng màu C theo cấp số nhân của 2 (giảm dần): C={256,128,64,32,16,8,4,2}
* Mỗi giá trị trong danh sách đại diện cho số lượng màu tối đa được giữ lại sau lượng tử hóa.

**B3. Thực hiện lượng tử hóa**

* Với mỗi số lượng màu c trong danh sách C:
  1. Chuyển đổi ảnh sang chế độ bảng màu (P mode) bằng hàm:

im1 = im.convert('P', palette=Image.ADAPTIVE, colors=c)

* + - **palette=Image.ADAPTIVE**: Chọn bảng màu tối ưu dựa trên ảnh.
    - **colors=c**: Giới hạn số lượng màu tối đa trong ảnh.
  1. Kết quả là ảnh được lượng tử hóa với ccc màu.

**B4. Tính toán Tỷ lệ Tín hiệu trên Nhiễu (SNR)**

* Chuyển đổi ảnh lượng tử hóa sang mảng numpy:

array = np.array(im1)

* Sử dụng hàm: SNR=mean(array) \ std(array)
  + **mean(array)**: Trung bình giá trị các pixel.
  + **std(array)**: Độ lệch chuẩn của các pixel.

**B5. Lưu trữ và hiển thị**

* Lưu kết quả SNR tương ứng với từng số lượng màu.
* Hiển thị các ảnh đã lượng tử hóa theo thứ tự, kèm theo thông tin:
  + Số lượng màu.
  + Giá trị SNR.

**B6. So sánh chất lượng ảnh**

* Dựa vào số lượng màu và SNR:
  + Ảnh có số lượng màu lớn hơn giữ lại nhiều chi tiết hơn, chất lượng cao hơn.
  + Khi số lượng màu giảm, chất lượng giảm do lỗi lượng tử hóa.

c, Giải thích đoạn mã nguồn :

from PIL import Image  
import matplotlib.pylab as pylab  
import numpy as np  
  
# Mở ảnh đầu vào  
im = Image.open(r"C:\Users\Dell\Desktop\Sandipan\_Dey\_2018\_Sample\_Images\images\messi.jpg") # Đường dẫn ảnh  
  
# Tạo danh sách số lượng màu (giảm dần từ 256 đến 2)  
num\_colors\_list = [1 << n for n in range(8, 0, -1)] # [256, 128, 64, 32, 16, 8, 4, 2]  
snr\_list = [] # Danh sách lưu SNR  
i = 1  
  
# Thiết lập kích thước hiển thị  
pylab.figure(figsize=(20, 30))  
  
# Duyệt qua các số lượng màu  
for num\_colors in num\_colors\_list:  
 # Lượng tử hóa ảnh  
 im1 = im.convert('P', palette=Image.ADAPTIVE, colors=num\_colors)  
  
 # Hiển thị ảnh đã lượng tử hóa  
 pylab.subplot(4, 2, i) # Hiển thị trên lưới 4x2  
 pylab.imshow(im1) # Hiển thị ảnh  
 pylab.axis('off') # Tắt trục tọa độ  
  
 # Tính SNR (thay thế signaltonoise bằng công thức)  
 im\_array = np.array(im1) # Chuyển ảnh thành mảng numpy  
 snr = im\_array.mean() / im\_array.std() # Tính SNR  
 snr\_list.append(snr)  
  
 # Đặt tiêu đề  
 pylab.title(  
 f"Image with # colors = {num\_colors}\nSNR = {np.round(snr, 3)}",  
 size=20  
 )  
 i += 1  
  
# Điều chỉnh khoảng cách giữa các ô  
pylab.subplots\_adjust(wspace=0.2, hspace=0)  
  
# Hiển thị toàn bộ ảnh  
pylab.show()

**. Mở ảnh đầu vào:**

im = Image.open(r"C:\Users\Dell\Desktop\Sandipan\_Dey\_2018\_Sample\_Images\images\messi.jpg")

* Image.open là một hàm của thư viện PIL dùng để mở và tải ảnh từ đường dẫn chỉ định. Ở đây, ảnh có tên là messi.jpg được mở từ thư mục đã chỉ định.

**. Tạo danh sách số lượng màu:**

num\_colors\_list = [1 << n for n in range(8, 0, -1)] # [256, 128, 64, 32, 16, 8, 4, 2]

* Danh sách này tạo ra các số lượng màu mà ảnh sẽ được lượng tử hóa (giảm dần từ 256 màu xuống 2 màu). 1 << n là phép dịch bit, tương đương với phép nhân với số mũ 2, tạo ra các giá trị từ 256, 128, 64, ... cho đến 2.

**. Khởi tạo các biến cần thiết:**

snr\_list = [] # Danh sách lưu SNR  
i = 1

* snr\_list là danh sách lưu trữ giá trị SNR (Signal-to-Noise Ratio) của mỗi ảnh sau khi lượng tử hóa.
* i là biến đếm để hiển thị ảnh trên lưới 4x2.

**. Thiết lập kích thước hiển thị:**

pylab.figure(figsize=(20, 30))

* Dùng pylab.figure để thiết lập kích thước của cửa sổ hiển thị. figsize có nghĩa là chiều rộng là 20 inch và chiều cao là 30 inch.

**. Duyệt qua các số lượng màu và lượng tử hóa ảnh:**

for num\_colors in num\_colors\_list:  
 # Lượng tử hóa ảnh  
 im1 = im.convert('P', palette=Image.ADAPTIVE, colors=num\_colors)

* Vòng lặp này sẽ duyệt qua tất cả các giá trị trong num\_colors\_list và sử dụng im.convert('P', palette=Image.ADAPTIVE, colors=num\_colors) để chuyển ảnh về định dạng ảnh với số màu tương ứng. 'P' là chế độ chỉ sử dụng palette màu (chế độ bảng màu). Image.ADAPTIVE sẽ tự động chọn bảng màu tốt nhất cho ảnh. colors=num\_colors là số lượng màu muốn giảm xuống.

**. Hiển thị ảnh đã lượng tử hóa:**

pylab.subplot(4, 2, i) # Hiển thị trên lưới 4x2  
pylab.imshow(im1) # Hiển thị ảnh  
pylab.axis('off') # Tắt trục tọa độ

* pylab.subplot(4, 2, i) xác định vị trí của ảnh trên lưới 4x2 (4 hàng và 2 cột). i là chỉ số của ảnh trên lưới.
* pylab.imshow(im1) để hiển thị ảnh.
* pylab.axis('off') tắt các trục tọa độ.

**. Tính toán SNR:**

im\_array = np.array(im1) # Chuyển ảnh thành mảng numpy  
snr = im\_array.mean() / im\_array.std() # Tính SNR  
snr\_list.append(snr)

* np.array(im1) chuyển ảnh thành một mảng numpy để có thể tính toán dễ dàng.
* SNR được tính bằng công thức mean / std, trong đó mean() là giá trị trung bình của các pixel và std() là độ lệch chuẩn của các pixel. Công thức này đo lường mức độ nhiễu của ảnh, giá trị SNR cao nghĩa là ảnh ít nhiễu.

**. Đặt tiêu đề cho từng ảnh:**

pylab.title(  
 f"Image with # colors = {num\_colors}\nSNR = {np.round(snr, 3)}",  
 size=20  
)

* Tiêu đề của mỗi ảnh sẽ bao gồm số lượng màu được sử dụng và giá trị SNR. SNR được làm tròn đến 3 chữ số thập phân.

**.Điều chỉnh khoảng cách giữa các ô:**

pylab.subplots\_adjust(wspace=0.2, hspace=0)

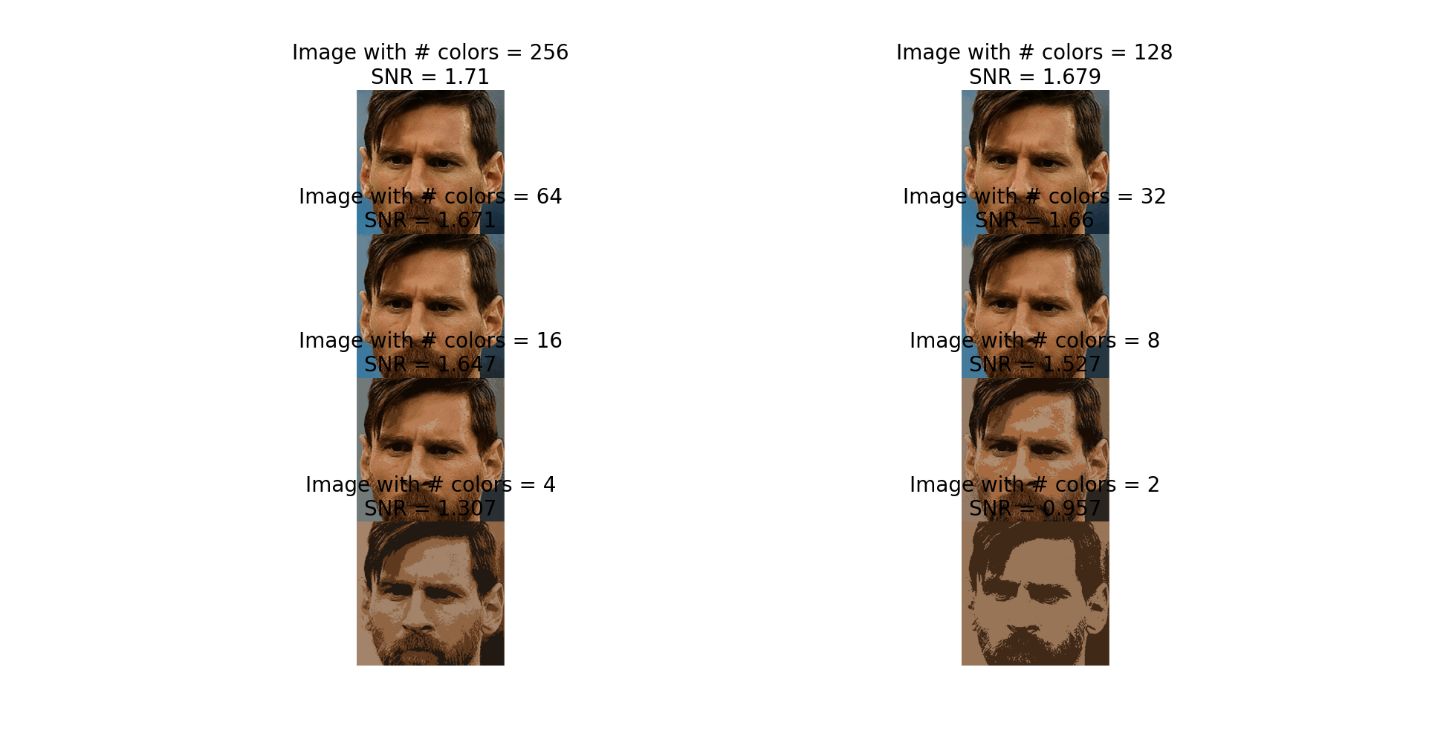
* wspace=0.2 điều chỉnh khoảng cách giữa các cột của các ảnh trong lưới.
* hspace=0 điều chỉnh khoảng cách giữa các hàng của các ảnh trong lưới.

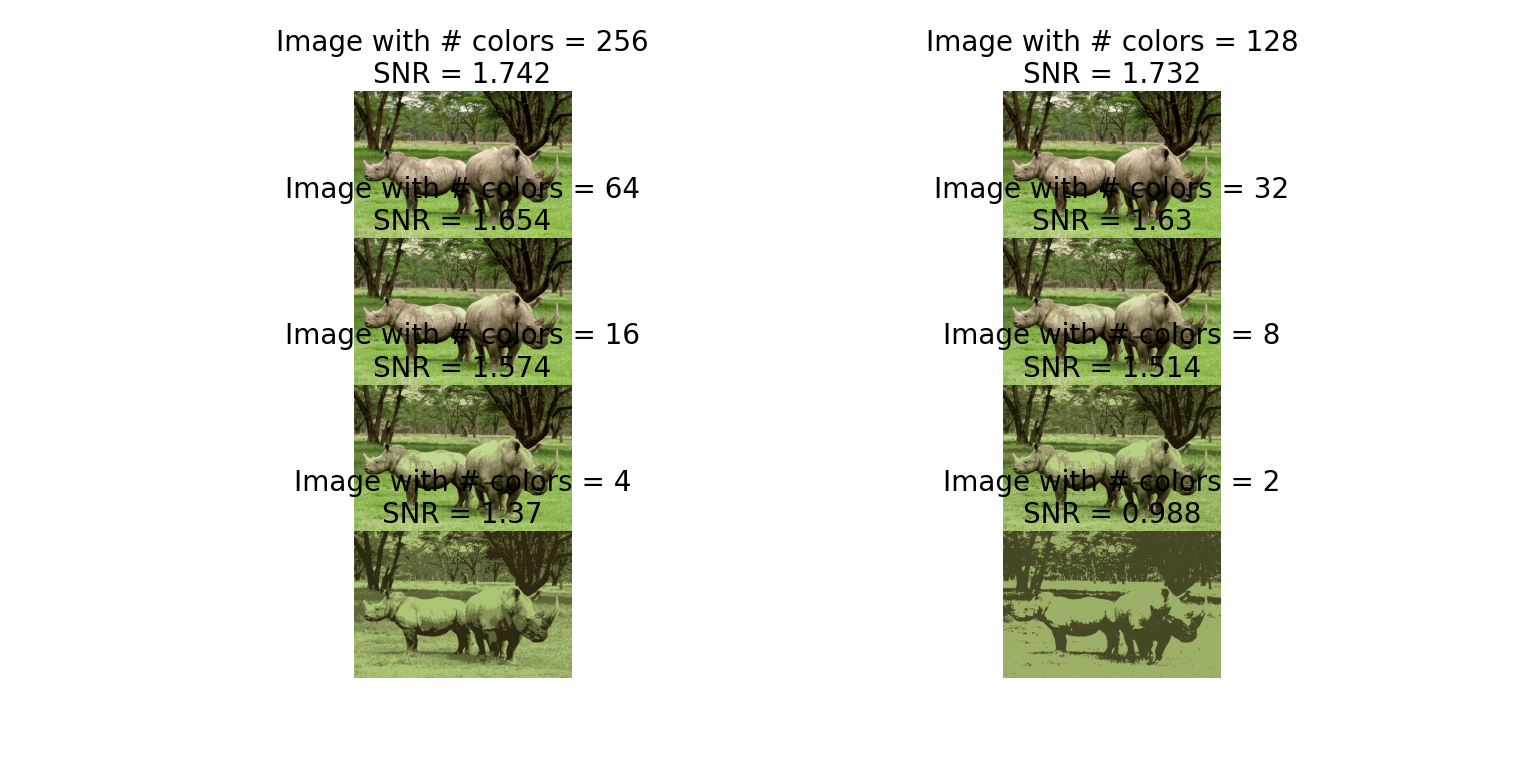
**. Hiển thị tất cả ảnh:**

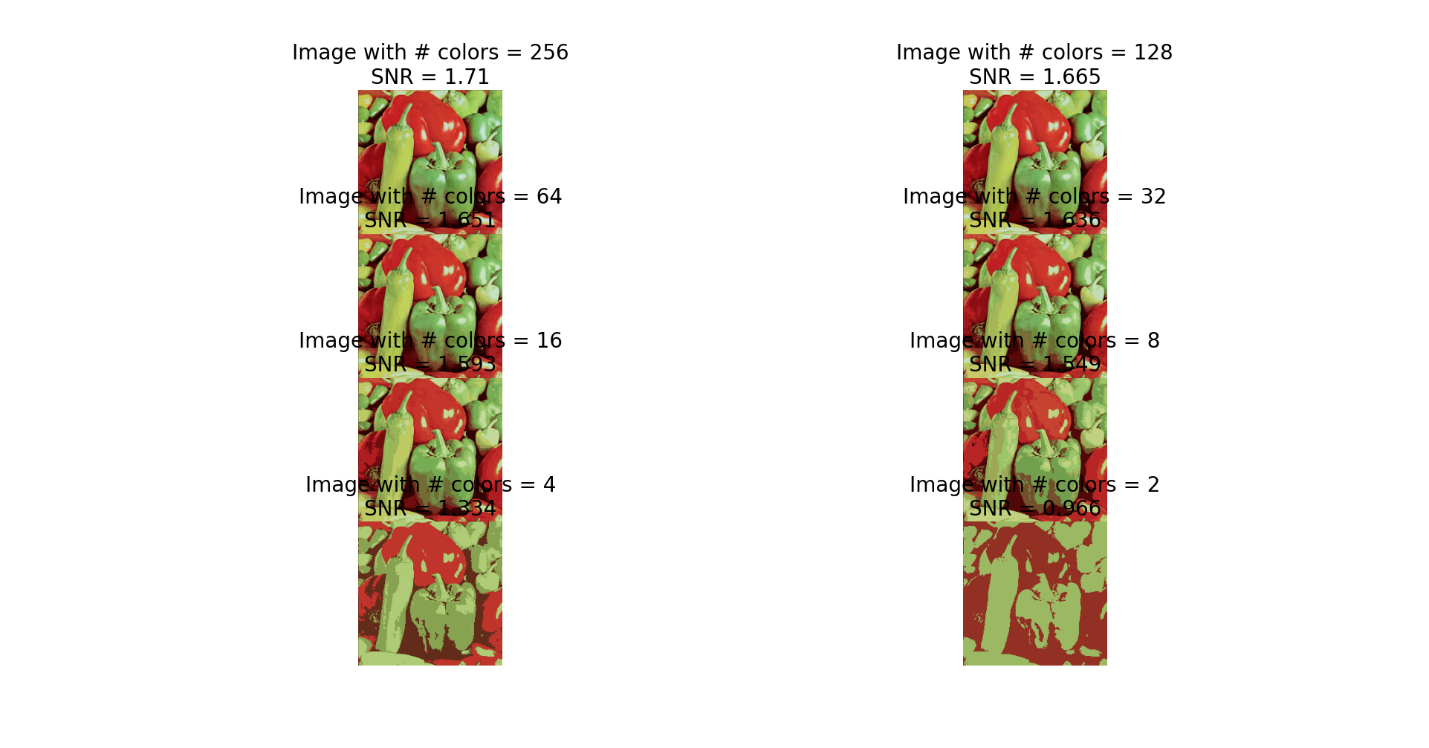
pylab.show()

Dùng pylab.show() để hiển thị tất cả ảnh đã được lượng tử hóa với các tiêu đề tương ứng.

d, Kết quả thực hiện

Ví dụ 1 : 

Ví dụ 2 : 

Ví dụ 3 : 

Nhận xét ảnh : Ta sẽ có một cửa sổ hiển thị 8 ảnh, mỗi ảnh có số lượng màu khác nhau và kèm theo giá trị SNR của ảnh đó. Ảnh có số màu càng ít sẽ có SNR thấp hơn, phản ánh sự suy giảm chất lượng hình ảnh.

PHẦN 2 : BIẾN ĐỔI FOURIER RỜI RẠC (Discrete Fourier Transform)

THUẬT TOÁN 3 : THỰC HIỆN TÍNH TOÁN BIÊN ĐỘ VÀ PHA CỦA MỘT DFT

Thuật toán Biến đổi Fourier Nhanh (FFT) là một thuật toán phân chia và chinh phục để tính toán DFT một cách nhanh chóng hơn (với độ phức tạp thời gian O(N.log2 N)) so với phương pháp tính toán chậm hơn nhiều O(N²) đối với ảnh có kích thước n x n. Trong Python, cả hai thư viện numpy và scipy đều cung cấp các hàm để tính toán DFT/IDFT 2D sử dụng thuật toán FFT

1.Các bước thực hiện thuật toán :

B1: **Đọc ảnh**:

* Đầu tiên, ảnh được đọc vào từ hệ thống sử dụng thư viện skimage.io.imread().
* Nếu ảnh có 4 kênh màu (RGBA), chỉ giữ lại 3 kênh (RGB) để xử lý.

B2 : **Chuyển ảnh sang ảnh xám (Grayscale)**:

* Chuyển ảnh màu RGB sang ảnh xám để giảm độ phức tạp và chỉ giữ lại thông tin độ sáng (intensity) của ảnh. Việc này giúp đơn giản hóa quá trình phân tích trong không gian tần số.

B3: **Áp dụng phép biến đổi Fourier (FFT)**:

* Áp dụng phép biến đổi Fourier nhanh (FFT) 2 chiều (fft2()) cho ảnh xám. Phép biến đổi Fourier chuyển đổi ảnh từ không gian không gian (spatial domain) sang không gian tần số (frequency domain), giúp phân tích các tần số có trong ảnh.
* Kết quả của phép biến đổi là một ma trận các số phức, trong đó mỗi giá trị đại diện cho một tần số và có 2 thành phần: biên độ (magnitude) và pha (phase).

B4 : **Tính toán ảnh hồi phục từ DFT**:

* Để lấy lại ảnh từ không gian tần số, thực hiện phép nghịch đảo của FFT (ifft2()). Kết quả này sẽ là ảnh hồi phục từ các tần số, gần giống như ảnh gốc nếu không có thông tin tần số nào bị mất.

B5 : **Hiển thị các kết quả**:

* **Hiển thị ảnh gốc**: Đây là ảnh xám ban đầu.
* **Hiển thị phổ FFT (Magnitude Spectrum)**: Tính độ lớn của các số phức trong phổ FFT và hiển thị dưới dạng ảnh, với tần số thấp nhất ở giữa ảnh.
* **Hiển thị pha FFT (Phase Spectrum)**: Hiển thị pha của các số phức trong phổ FFT. Thông tin này chứa các đặc điểm hình học và cấu trúc của ảnh.
* **Hiển thị ảnh hồi phục**: Sau khi thực hiện phép nghịch đảo FFT, ảnh được tái tạo lại từ không gian tần số.

2.Giải thích đoạn mã nguồn :

image = io.imread(r"C:\Users\Dell\Desktop\Sandipan\_Dey\_2018\_Sample\_Images\images\house.png")

 **io.imread()** từ thư viện skimage được sử dụng để đọc ảnh từ đường dẫn cho trước.

if image.shape[2] == 4:  
 image = image[:, :, :3]

 Nếu ảnh có định dạng RGBA (có 4 kênh màu), bạn chỉ lấy 3 kênh đầu tiên (RGB) bằng cách cắt ảnh, điều này tránh việc xử lý không cần thiết với kênh alpha (kênh trong suốt).

im1 = rgb2gray(image)

**rgb2gray()** là một hàm trong thư viện skimage.color, chuyển ảnh RGB (màu) thành ảnh xám. Việc này giúp loại bỏ thông tin màu sắc, chỉ giữ lại độ sáng của ảnh, làm cho quá trình xử lý tần số dễ dàng hơn

freq1 = fp.fft2(im1)

 **fp.fft2()** từ thư viện numpy.fft thực hiện phép biến đổi Fourier nhanh (FFT) 2 chiều trên ảnh xám. Điều này chuyển ảnh từ không gian không gian (spatial domain) sang không gian tần số (frequency domain).

 Sau khi áp dụng FFT, ảnh sẽ được biểu diễn dưới dạng các tần số, với mỗi giá trị là một số phức chứa thông tin về biên độ và pha của tần số đó.

im1\_ = fp.ifft2(freq1).real

 **fp.ifft2()** thực hiện phép nghịch đảo của DFT, chuyển ảnh từ không gian tần số trở lại không gian không gian.

 .real lấy phần thực của ảnh hồi phục (phần ảo có thể không cần thiết trong trường hợp ảnh thực tế).

pylab.subplot(2, 2, 1)  
pylab.imshow(im1, cmap='gray')  
pylab.title('Original Image', size=20)

 **pylab.subplot()** chia màn hình hiển thị thành một lưới 2x2 để chứa các đồ thị.

 **imshow()** được sử dụng để hiển thị ảnh dưới dạng mảng 2 chiều.

 Ở đây, ảnh gốc (chuyển sang xám) được hiển thị

pylab.subplot(2, 2, 2)  
pylab.imshow(20 \* np.log10(0.01 + np.abs(fp.fftshift(freq1))), cmap='gray')  
pylab.title('FFT Spectrum Magnitude', size=20)

 **np.abs()** tính độ lớn của mỗi số phức trong phổ DFT.

 **fp.fftshift()** dịch chuyển phổ tần số để gốc tần số (tần số thấp) ở giữa ảnh (thường tần số thấp nằm ở góc trên bên trái sau khi áp dụng FFT).

 **np.log10()** áp dụng logarit để tăng cường các tần số cao, giúp dễ dàng nhìn thấy các chi tiết.

 Đây là phổ FFT của ảnh, cho biết mức độ xuất hiện của các tần số trong ảnh.

pylab.subplot(2, 2, 3)  
pylab.imshow(np.angle(fp.fftshift(freq1)), cmap='gray')  
pylab.title('FFT Phase', size=20)

 **np.angle()** tính toán pha của các số phức trong phổ DFT, cho biết vị trí không gian của các tần số.

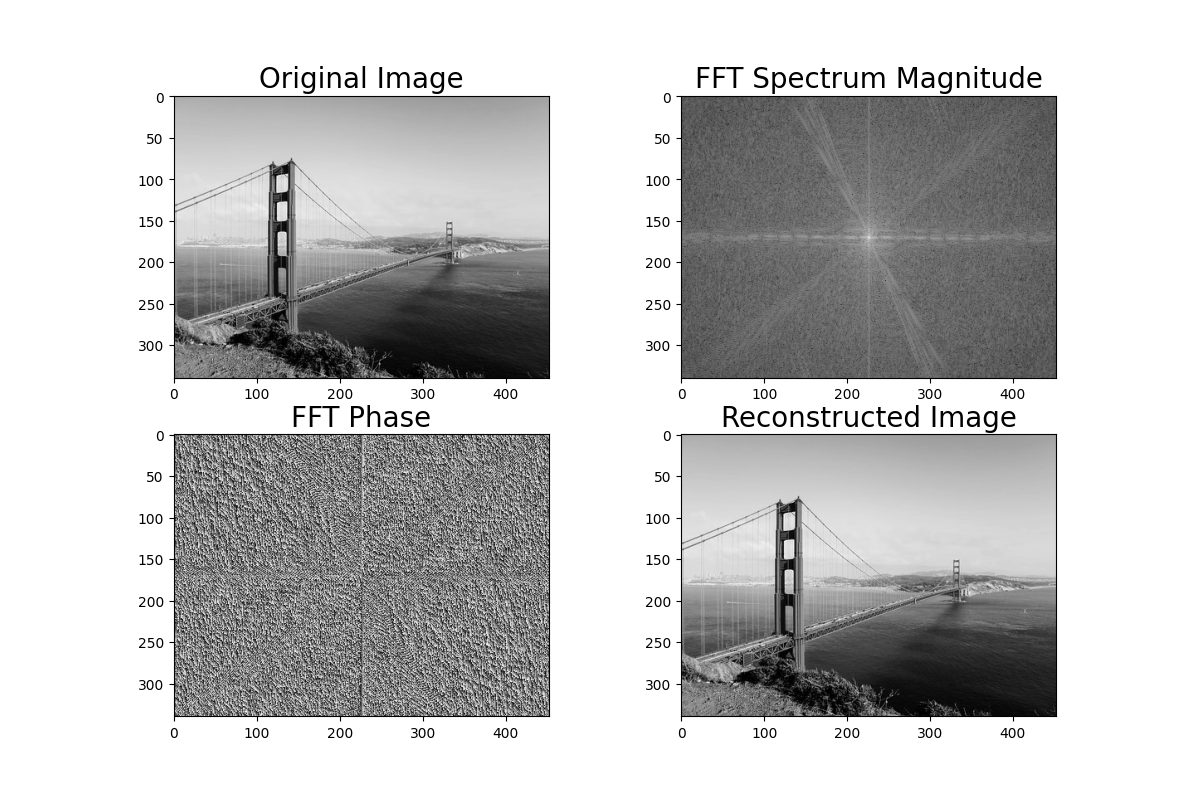
 Pha chứa thông tin về cấu trúc và hình học của ảnh.

pylab.subplot(2, 2, 4)  
pylab.imshow(np.clip(im1\_, 0, 255), cmap='gray')  
pylab.title('Reconstructed Image', size=20)

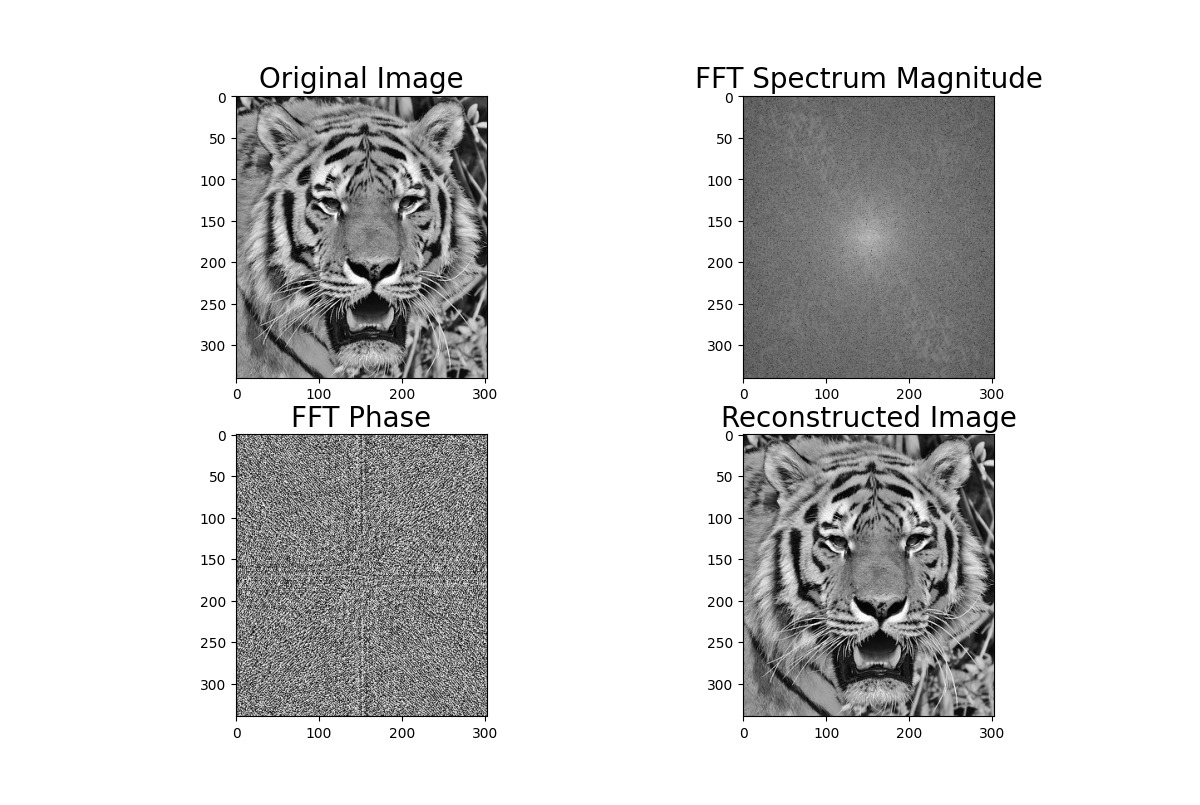
Ảnh hồi phục từ DFT được hiển thị. **np.clip()** giới hạn giá trị pixel trong phạm vi [0, 255] để tránh các giá trị ngoài phạm vi.

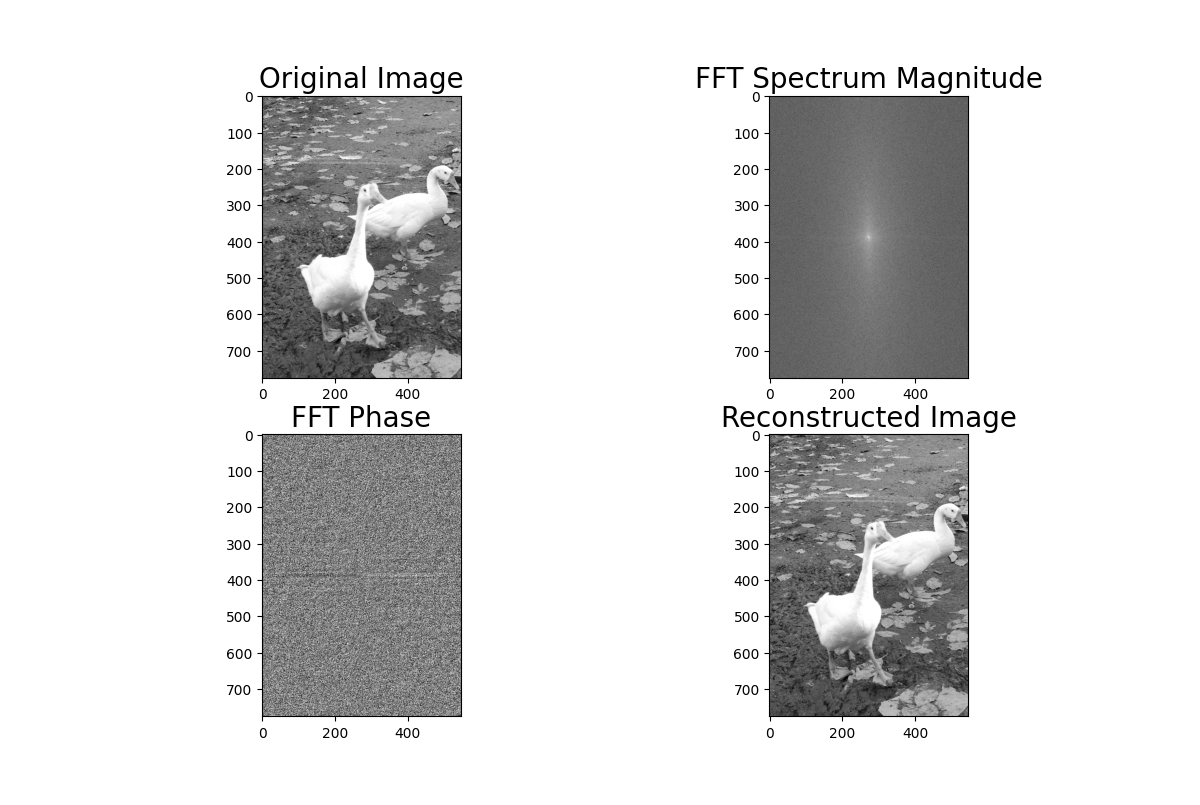
2, Kết quả thực hiện :

Ví dụ 1 :



Ví dụ 2 :



Ví dụ 3 : 

Nhận xét các ví dụ :

 **Ảnh gốc**:

* Đây là ảnh ban đầu, đã được chuyển sang ảnh xám. Ảnh có thể có các chi tiết rõ nét hoặc các vùng đồng nhất.

 **Phổ FFT (Magnitude)**:

* Phổ tần số thường có một "vùng sáng" tại trung tâm, đại diện cho tần số thấp nhất, và giảm dần khi ra xa trung tâm (tần số cao hơn). Điều này cho thấy ảnh chủ yếu có các thành phần tần số thấp (các chi tiết mịn mà mắt thường có thể nhận thấy được).
* Các chi tiết sắc nét hoặc các chuyển động nhanh sẽ được biểu thị ở các tần số cao, điều này có thể khó nhìn thấy rõ ràng trong ảnh phổ.

 **Pha FFT**:

* Pha chứa thông tin về hình học và cấu trúc của ảnh, nhưng không cung cấp thông tin về cường độ ánh sáng của ảnh. Ảnh này có thể hiển thị các mô hình phức tạp hoặc các đường biên của ảnh.

 **Ảnh hồi phục**:

* Ảnh hồi phục từ DFT có thể rất giống với ảnh gốc nếu không có sự thay đổi nào trong quá trình chuyển đổi tần số. Tuy nhiên, nếu có thay đổi hoặc mất mát tần số (ví dụ: nếu tần số cao bị loại bỏ), ảnh hồi phục sẽ bị mờ hoặc mất chi tiết.
* Việc sử dụng .real để lấy phần thực của ảnh hồi phục có thể dẫn đến sự mất mát các chi tiết nhỏ trong quá trình phục hồi.

PHẦN 3 : TÍCH CHẬP ( Convolution )

Giới thiệu về tích chập : Tích chập ( Convolution ) là một phép toán hoạt động trên hai ảnh, một là ảnh đầu vào và ảnh còn lại là mặt nạ (còn gọi là kernel) dùng làm bộ lọc trên ảnh đầu vào, tạo ra một ảnh đầu ra.  
Lọc convolution được sử dụng để thay đổi các đặc tính tần số không gian của một ảnh. Nó hoạt động bằng cách xác định giá trị của một pixel trung tâm bằng cách cộng các giá trị có trọng số của tất cả các pixel xung quanh nó lại để tính toán giá trị mới của pixel trong ảnh đầu ra. Các giá trị pixel trong ảnh đầu ra được tính toán bằng cách di chuyển cửa sổ kernel qua ảnh đầu vào.

THUẬT TOÁN 4 : ÁP DỤNG PHÉP CHẬP CHO ẢNH MÀU (RGB)

1, Giải thích thuật toán : Thuật toán trên sử dụng phép chập để xử lý ảnh RGB. Chúng ta sẽ áp dụng phép chập cho từng kênh màu riêng biệt (đỏ, xanh lá cây, và xanh dương) của ảnh RGB. Cụ thể, ta áp dụng hai phép chập sau:

* + **Embossing (Hiệu ứng nổi)**: Sử dụng kernel emboss để tạo hiệu ứng nổi trên ảnh.
  + **Edge Detection (Phát hiện cạnh)**: Sử dụng kernel Scharr (một loại kernel dùng để phát hiện các cạnh) để xác định các cạnh của ảnh.

2, Các bước thực hiện thuật toán :

B1: **Đọc ảnh**: Đọc ảnh màu và chuẩn hóa giá trị pixel về phạm vi từ 0 đến 1 (bằng cách chia giá trị pixel cho 255).

B2: **Xác định kernel**: Hai kernel được sử dụng:

* **Emboss Kernel**: Tạo hiệu ứng làm nổi trên ảnh.
* **Edge Scharr Kernel**: Phát hiện các cạnh trong ảnh.

B3: **Chập ảnh**: Áp dụng phép chập với từng kernel cho từng kênh màu (đỏ, xanh lá, xanh dương).

B4: **Hiển thị ảnh**: Hiển thị ba ảnh kết quả:

* Ảnh gốc.
* Ảnh với hiệu ứng emboss.
* Ảnh sau khi phát hiện các cạnh bằng Scharr.

3.Giải thích đoạn mã nguồn :

* import numpy.fft as fp  
  import numpy as np  
  import matplotlib.pylab as pylab  
  from skimage import io  
  from skimage.color import rgb2gray  
  from scipy import signal  
  import imageio # Thêm thư viện imageio  
  import imageio.v2 as imageio  
  # Đọc và chuẩn hóa ảnh  
  im = imageio.imread(r"C:\Users\Dell\Desktop\Sandipan\_Dey\_2018\_Sample\_Images\images\tajmahal.jpg") / 255 # scale each pixel value in [0,1]  
  print(np.max(im))  
  print(im.shape)  
    
  # Định nghĩa các kernel  
  emboss\_kernel = np.array([[-2, -1, 0], [-1, 1, 1], [0, 1, 2]])  
  edge\_schar\_kernel = np.array([[ -3-3j, 0-10j, +3 -3j],  
   [-10+0j, 0+0j, +10+0j],  
   [ -3+3j, 0+10j, +3 +3j]])  
    
  # Khởi tạo ảnh đầu ra  
  im\_embossed = np.ones(im.shape)  
  im\_edges = np.ones(im.shape)  
    
  # Lọc ảnh bằng kernel emboss  
  for i in range(3):  
   im\_embossed[..., i] = np.clip(signal.convolve2d(im[..., i], emboss\_kernel, mode='same', boundary="symm"), 0, 1)  
    
  # Lọc ảnh bằng kernel Scharr để phát hiện cạnh  
  for i in range(3):  
   im\_edges[:, :, i] = np.clip(np.real(signal.convolve2d(im[..., i], edge\_schar\_kernel, mode='same', boundary="symm")), 0, 1)  
    
  # Hiển thị ảnh  
  fig, axes = pylab.subplots(nrows=3, figsize=(20, 30))  
  axes[0].imshow(im)  
  axes[0].set\_title('Original Image', size=20)  
  axes[1].imshow(im\_embossed)  
  axes[1].set\_title('Embossed Image', size=20)  
  axes[2].imshow(im\_edges)  
  axes[2].set\_title('Schar Edge Detection', size=20)  
    
  # Tắt các trục  
  for ax in axes:  
   ax.axis('off')  
    
  # Hiển thị hình ảnh  
  pylab.show()

Các thư viện cần thiết được nhập vào, bao gồm các thư viện xử lý ảnh (imageio, skimage), các thư viện toán học (numpy, scipy), và thư viện vẽ đồ thị (matplotlib).

 Ảnh được đọc bằng imageio.imread(). Sau khi đọc ảnh, giá trị các pixel được chuẩn hóa bằng cách chia cho 255, làm cho tất cả giá trị nằm trong khoảng từ 0 đến 1.

 Hàm np.max(im) trả về giá trị pixel lớn nhất của ảnh, giúp kiểm tra xem ảnh có được chuẩn hóa đúng cách không.

 Hàm im.shape hiển thị kích thước của ảnh (chiều cao, chiều rộng và số kênh màu).

 emboss\_kernel là một kernel 3x3 dùng để tạo hiệu ứng emboss cho ảnh.

 edge\_schar\_kernel là một kernel phức tạp 3x3 dùng để phát hiện các cạnh trong ảnh, với các giá trị phức để tính toán gradient trong không gian phức.

Tạo hai ảnh mới im\_embossed và im\_edges có cùng kích thước với ảnh gốc, ban đầu được gán giá trị 1 (trắng).

 Duyệt qua từng kênh màu của ảnh RGB (có 3 kênh: đỏ, xanh lá, xanh dương).

 Dùng signal.convolve2d() để áp dụng phép chập với kernel emboss cho từng kênh màu. mode='same' nghĩa là ảnh kết quả sẽ có kích thước bằng ảnh gốc.

 boundary="symm" chỉ định cách xử lý biên (biên được làm đối xứng).

 np.clip() dùng để giới hạn giá trị pixel trong phạm vi [0, 1].

 Tương tự như bước trên, nhưng áp dụng kernel Scharr để phát hiện cạnh.

 np.real() được sử dụng để lấy phần thực của kết quả (vì kernel Scharr chứa các số phức).

 Tạo một biểu đồ có 3 ô (subplots) để hiển thị 3 ảnh: ảnh gốc, ảnh emboss và ảnh phát hiện cạnh.

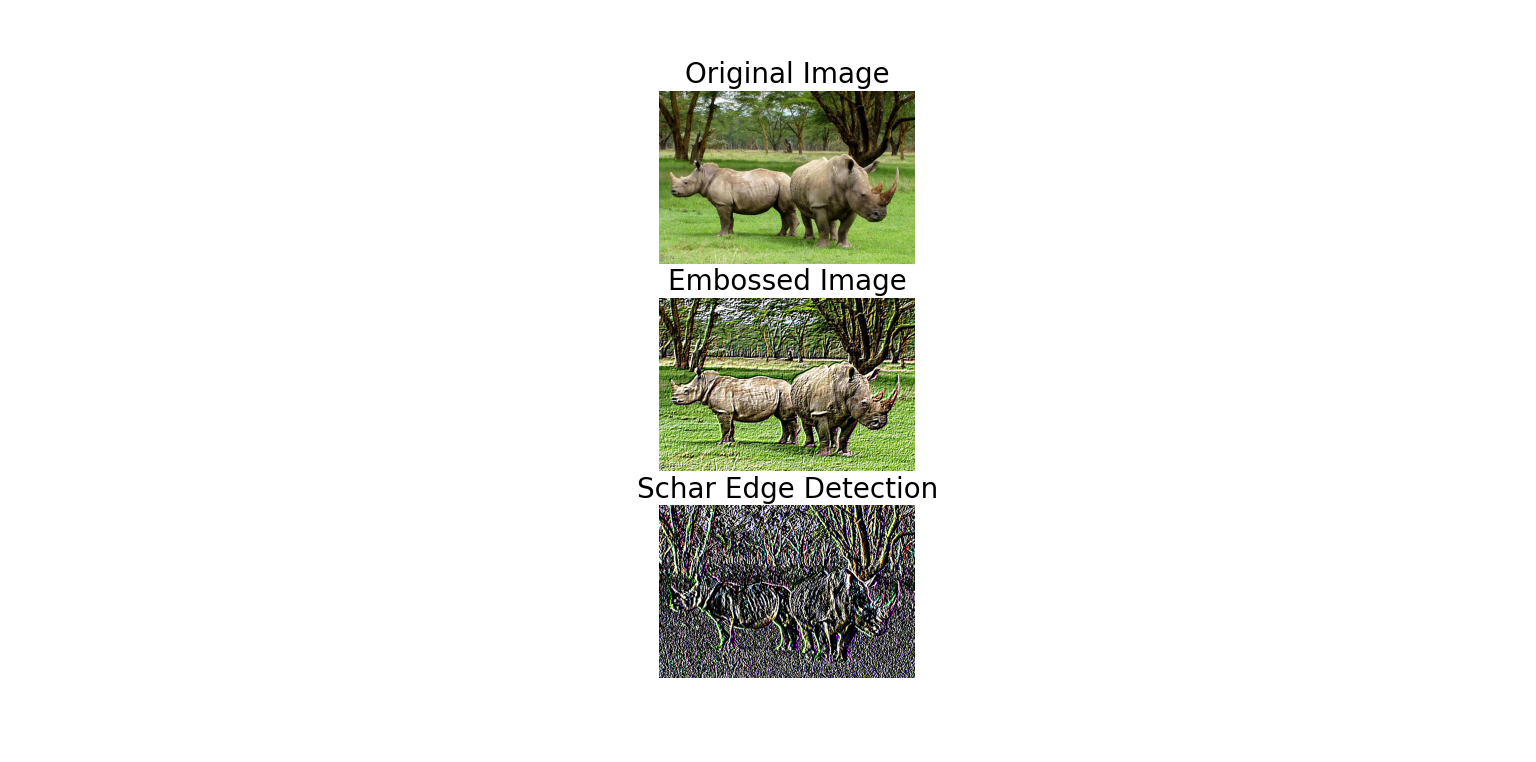
 imshow() được dùng để hiển thị ảnh trong mỗi ô. set\_title() dùng để đặt tiêu đề cho từng ảnh

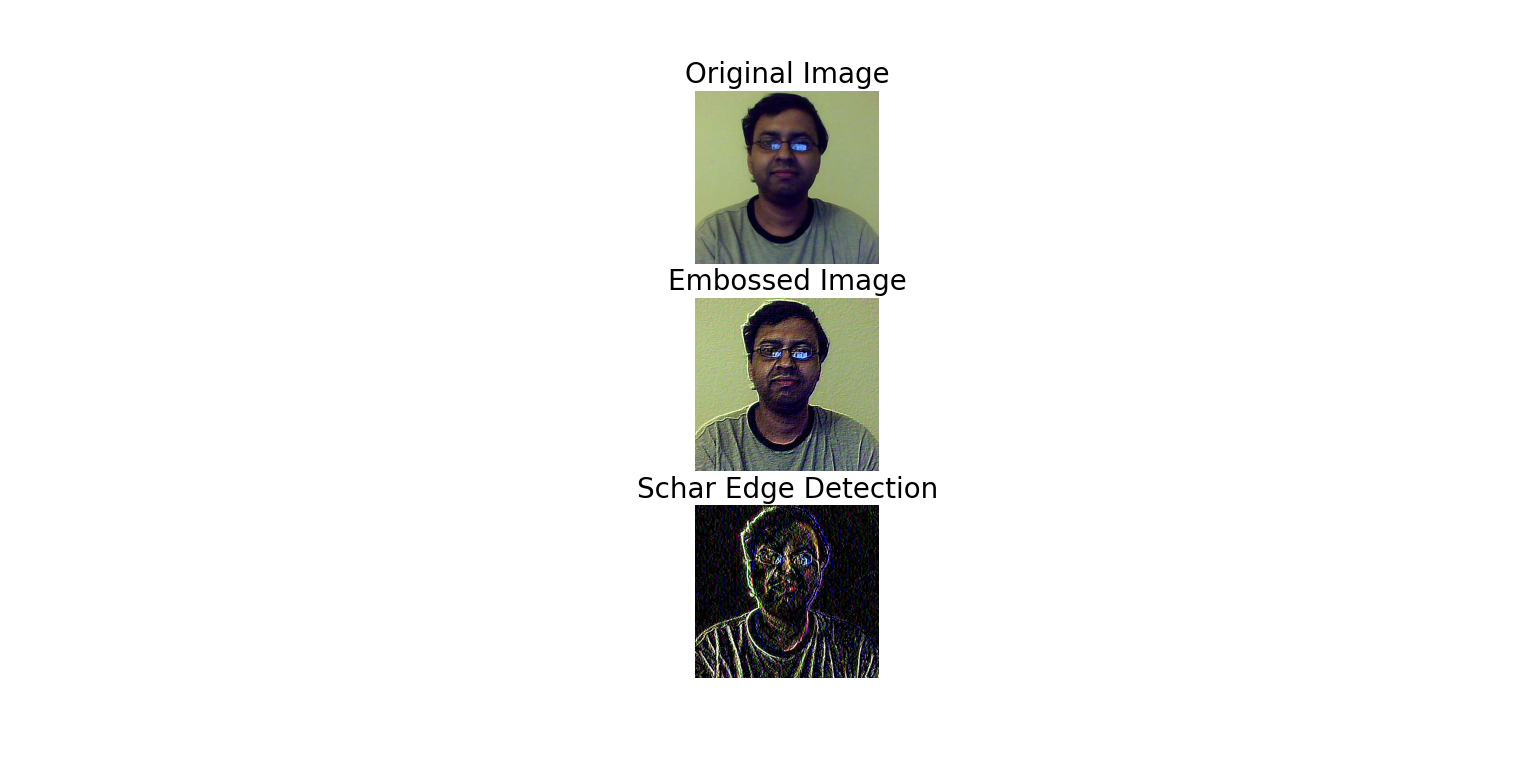
 axis('off') tắt các trục để chỉ hiển thị ảnh mà không có các nhãn trục.

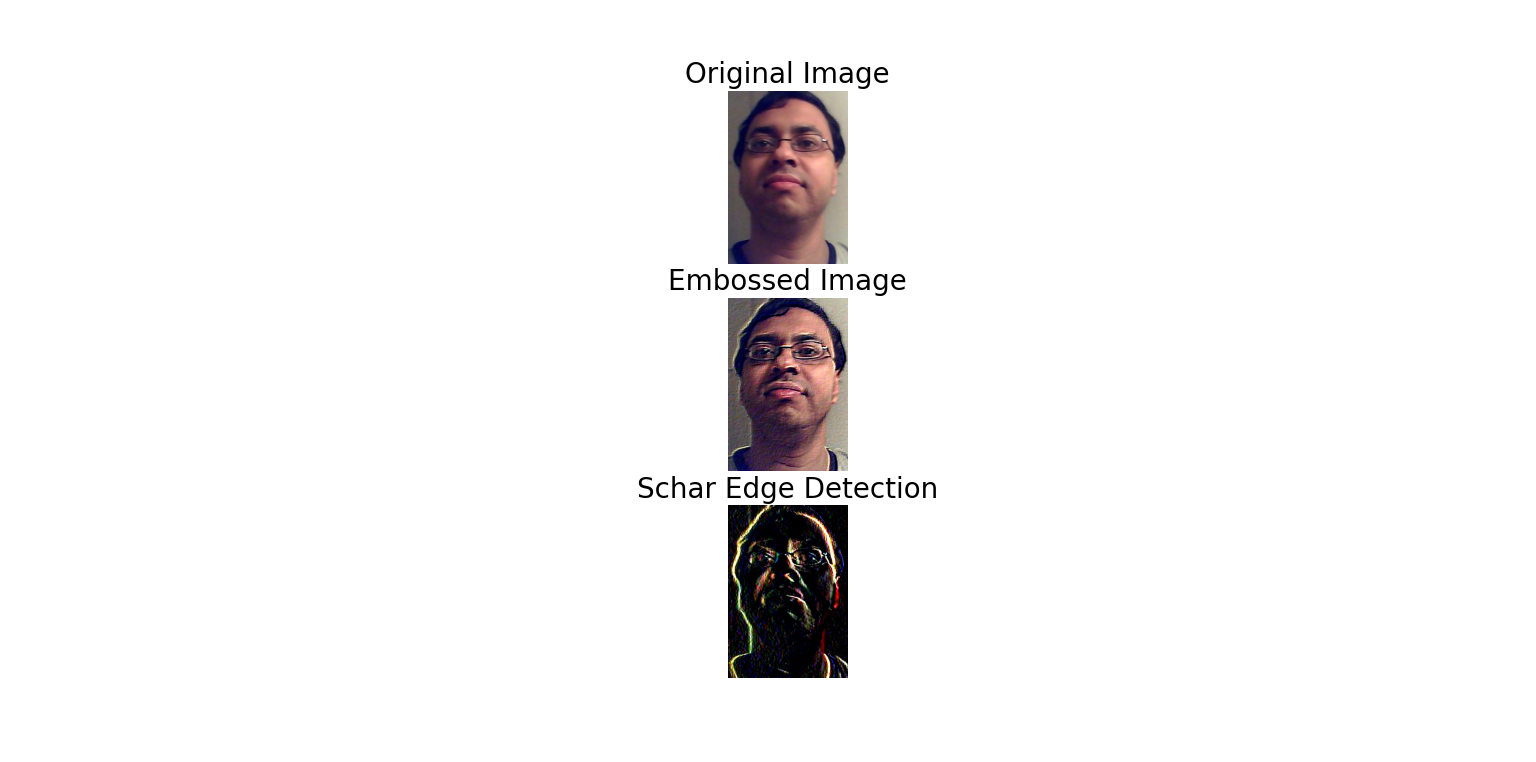
 pylab.show() hiển thị toàn bộ hình ảnh

4.Kết quả thực hiện :

Ví dụ 1 :



Ví dụ 2 : 

Ví dụ 3 : 

Nhận xét ảnh :

**Ảnh gốc (Original Image):**

* Ảnh gốc là một ảnh màu với 3 kênh màu RGB (Đỏ, Xanh lá, Xanh dương).
* Trước khi áp dụng bất kỳ bộ lọc nào, hình ảnh có đầy đủ chi tiết ban đầu, màu sắc tự nhiên và các đặc điểm rõ ràng.

**2. Ảnh được làm nổi bật (Embossed Image):**

Hiệu ứng Emboss: Kernel emboss được sử dụng để tạo hiệu ứng nổi bật cho ảnh. Mục tiêu của bộ lọc này là làm cho các cạnh và các chi tiết trong ảnh trở nên "nổi" lên.

Hiệu quả: Sau khi áp dụng bộ lọc emboss, các vùng sáng và tối trong ảnh sẽ trở nên rõ ràng hơn. Các đối tượng hoặc đường biên trong ảnh có thể xuất hiện với hiệu ứng giống như khắc nổi, nơi mà các chi tiết sẽ có vẻ như được khắc nổi lên bề mặt.

Nhận xét: Ảnh sau khi áp dụng bộ lọc emboss sẽ có các khu vực sáng tối đối lập rõ rệt hơn, giúp làm nổi bật các chi tiết hình khối trong ảnh.

**3. Ảnh phát hiện cạnh (Schar Edge Detection):**

Hiệu quả phát hiện cạnh (Edge Detection): Bộ lọc Scharr được sử dụng để phát hiện các cạnh trong ảnh. Bộ lọc này nhạy cảm với các sự thay đổi đột ngột về độ sáng, đặc biệt là những nơi có sự chuyển tiếp rõ ràng giữa các vùng sáng và tối.

Hiệu quả: Sau khi áp dụng bộ lọc Scharr, ảnh sẽ có nhiều cạnh rõ ràng hơn, đặc biệt là trong các vùng có sự chuyển tiếp mạnh mẽ giữa các đối tượng trong ảnh. Các cạnh này sẽ được thể hiện qua các vùng có độ tương phản mạnh.

Nhận xét: Cạnh của các đối tượng trong ảnh sẽ được làm nổi bật, tạo ra một ảnh với hình dạng của các đối tượng rõ ràng hơn nhưng màu sắc sẽ bị giảm thiểu. Mặc dù các chi tiết vẫn còn, nhưng sẽ chỉ còn lại những đường viền hoặc các vùng có sự thay đổi độ sáng mạnh.