Homework 2

Chu Tiến Đạt - 20020289

Ngày 20 tháng 4 năm 2024

1 Ex 1

```
def padding_img(img, filter_size=3):
    pad_size = filter_size // 2
    padded_img = np.pad(img, pad_width=pad_size, mode='
        edge')
    return padded_img
```

Hàm *padding_img* thêm viền vào quanh ảnh để khi áp dụng bộ lọc (kernel) với kích thước cho trước, kích thước của ảnh sau khi lọc vẫn giữ nguyên so với ảnh gốc.

Nhận xét: Padding kiểu replicate có hiệu quả trong việc duy trì các giá trị đặc trưng ở rìa ảnh, không làm thay đổi quá nhiều so với nội dung gốc. Kỹ thuật này giúp tránh hiện tượng biên được lọc mất đi do thiếu dữ liệu xung quanh.

```
def mean_filter(img, filter_size=3):
    padded_img = padding_img(img, filter_size)
    smoothed_img = np.zeros_like(img)
    pad = filter_size // 2
    for i in range(pad, padded_img.shape[0] - pad):
        for j in range(pad, padded_img.shape[1] - pad):
            window = padded_img[i-pad:i+pad+1, j-pad:
                j+pad+1]
            smoothed_img[i-pad, j-pad] = np.mean(
                window)
    return smoothed_img
```

Hàm *mean_filter* áp dụng bộ lọc trung bình (box filter) để làm mịn ảnh. Bộ lọc này lấy trung bình giá trị các pixel trong cửa sổ lân cận và thay thế giá trị pixel hiện tại bằng giá trị trung bình đó.

Nhận xét: Bộ lọc trung bình rất hiệu quả trong việc giảm nhiễu ngẫu nhiên trong ảnh. Tuy nhiên, bộ lọc này cũng làm mờ cạnh và các chi tiết trong ảnh, có thể không phù hợp cho việc bảo toàn kết cấu hoặc cạnh của đối tượng trong ảnh.

```
def median_filter(img, filter_size=3):
    padded_img = padding_img(img, filter_size)
    smoothed_img = np.zeros_like(img)
    pad = filter_size // 2
```

```
    for i in range(pad, padded_img.shape[0] - pad):
        for j in range(pad, padded_img.shape[1] - pad):
                window = padded_img[i-pad:i+pad+1, j-pad
                    :j+pad+1]
                smoothed_img[i-pad, j-pad] = np.median(
                    window)
    return smoothed_img
```

Hàm *median_filter* sử dụng bộ lọc trung vị (median filter) để làm mịn ảnh. Nó thay thế giá trị của mỗi pixel bằng giá trị trung vị của các giá trị pixel trong cửa sổ lân cận.

Nhận xét: Bộ lọc trung vị rất hiệu quả trong việc loại bỏ nhiễu loại muối tiêu (salt-and-pepper) mà không làm mờ cạnh, do đó nó thường được sử dụng trong việc bảo toàn cạnh. So với bộ lọc trung bình, bộ lọc trung vị thường giữ được chi tiết tốt hơn trong ảnh khi giảm nhiễu.

```
def psnr(gt_img, smooth_img):
    mse = np.mean((gt_img - smooth_img) ** 2)
    if mse == 0:
        return float('inf')
    max_pixel = 255.0
    psnr_score = 20 * math.log10(max_pixel / math.sqrt(
        mse))
    return psnr_score
```

Định lượng: $PSNR$ cung cấp một cách định lượng để đánh giá hiệu suất của các hàm lọc, giúp so sánh chúng dựa trên cơ sở toán học.

Chất lượng ảnh: Một $PSNR$ cao cho thấy ảnh sau khi lọc giữ được sự tương đồng cao với ảnh gốc, ít bị ảnh hưởng bởi nhiễu hoặc sự mất mát thông tin.

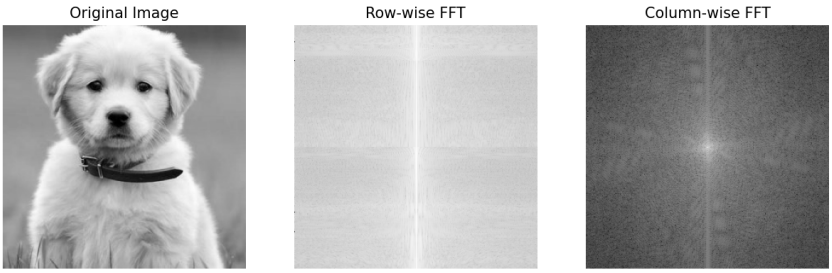
2 Ex 2

```
def DFT_slow(data):
    N = len(data)
    dft = np.zeros(N, dtype=complex)
    for s in range(N):
            for n in range(N):
                dft[s] += data[n] * np.exp(-1j * 2 * np.pi
                    * s * n / N)
    return dft

def DFT_2D(gray_img):
    row_fft = np.apply_along_axis(np.fft.fft, axis=1, arr
        =gray_img)
    row_col_fft = np.apply_along_axis(np.fft.fft, axis=0,
        arr=row_fft)
    return row_fft, row_col_fft
```

DFT_slow (DFT 1D):

Hàm *DFT_slow* thực hiện biến đổi Fourier rời rạc cho tín hiệu một chiều (1D). Đây là cách tiếp cận cơ bản nhất và rõ ràng nhất để hiểu về *DFT*.



Hình 1: Result images

Ưu điểm:

- Không phụ thuộc: Không cần dựa vào thư viện bên ngoài, hữu ích trong các môi trường hạn chế hoặc khi học cách cài đặt các thuật toán cơ bản.

Nhược điểm:

- Hiệu suất: Rất chậm so với các thuật toán FFT (Fast Fourier Transform) tối ưu như thuật toán Cooley-Tukey.
- Không thực tế: Không phù hợp cho việc xử lý dữ liệu thực tế hoặc trong các ứng dụng thực sự cần hiệu suất tính toán cao.

DFT_2D (DFT 2 chiều):
Hàm DFT_2D mở rộng cài đặt DFT_slow để xử lý tín hiệu hai chiều (2D), thường là ảnh.
Ưu điểm:

- Hiểu biết: Cung cấp cách tiếp cận cơ bản để hiểu cách DFT có thể được áp dụng cho dữ liệu hai chiều như ảnh.
- Tương thích: Có thể sử dụng để kiểm tra đầu ra chính xác của các hàm DFT nhanh hơn hoặc các thư viện bên ngoài.

Nhược điểm:

- Hiệu suất: Tương tự như DFT_slow, hiệu suất tính toán rất thấp, không thể sử dụng cho các ảnh có kích thước lớn hoặc xử lý thời gian thực.
- Mất mát thông tin: Khi làm việc với ảnh lớn, việc cắt giảm thông tin (do bộ nhớ hạn chế) có thể dẫn đến kết quả không chính xác.

```
def filter_frequency(orig_img, mask):
    f_orig_img = fft2(orig_img)
    f_shifted = fftshift(f_orig_img)
    f_filtered = f_shifted * mask
    f_ishifted = ifftshift(f_filtered)
    img_filtered = np.abs(ifft2(f_ishifted))
    return f_filtered, img_filtered
```

Bước 1: Biến đổi Fourier $f\_orig\_img = fft2(orig\_img)$

Đánh giá: $fft2$ thực hiện biến đổi Fourier nhanh hai chiều trên ảnh đầu vào. Đây là bước đầu tiên để chuyển đổi ảnh từ miền không gian sang miền tần số, nơi các thao tác lọc được thực hiện hiệu quả hơn.

Bước 2: Dịch chuyển tần số về trung tâm $f\_shifted = fftshift(f\_orig\_img)$

Đánh giá: $fftshift$ dịch các thành phần tần số thấp về trung tâm của mảng, làm cho việc phân tích và thao tác trên các thành phần này trở nên trực quan hơn. Điều này hữu ích khi áp dụng các mặt nạ lọc tần số.

Bước 3: Áp dụng mặt nạ lọc $f\_filtered = f\_shifted \times mask$

Đánh giá: Phép nhân mảng này áp dụng mặt nạ trực tiếp lên biểu diễn tần số của ảnh. Mặt nạ quyết định các thành phần tần số nào được giữ lại và thành phần nào bị loại bỏ. Điều này có thể giúp loại bỏ nhiễu hoặc làm nổi bật các đặc điểm nhất định của ảnh.

Bước 4: Đảo ngược dịch chuyển tần số $f\_ishifted = ifftshift(f\_filtered)$

Đánh giá: Đảo ngược lại hoạt động của $fftshift$ để chuẩn bị cho việc biến đổi ngược lại về miền không gian. Đây là bước cần thiết để đảm bảo rằng ảnh biến đổi ngược có cấu trúc không gian đúng.

Bước 5: Biến đổi Fourier ngược $img\_filtered = np.abs(ifft2(f\_ishifted))$

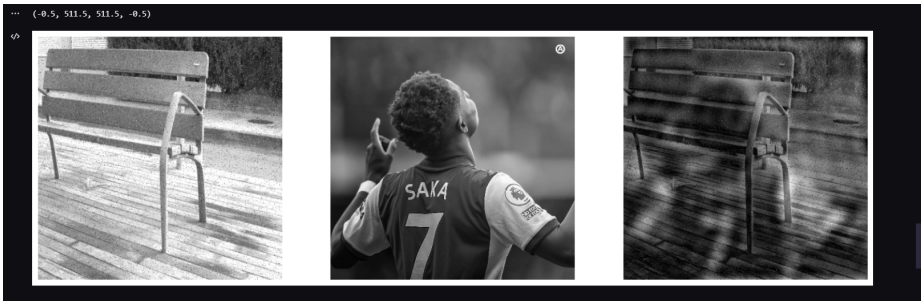
Đánh giá: $ifft2$ thực hiện Biến đổi Fourier ngược để chuyển đổi dữ liệu từ miền tần số trở lại miền không gian. Kết quả cuối cùng là một ảnh trong đó đã được áp dụng lọc tần số. Sử dụng *np.abs* là cần thiết để tính toán giá trị tuyệt đối của các số phức, tạo ra ảnh cuối cùng có giá trị thực.

```
def create_hybrid_img(img1, img2, r):
    img1 = cv2.resize(img1, img2.shape[::-1])
    f_img1 = fft2(img1)
    f_img2 = fft2(img2)
    f_img1_shifted = fftshift(f_img1)
    f_img2_shifted = fftshift(f_img2)
    y, x = np.indices((img1.shape[0], img1.shape[1]))
    center_x, center_y = int(img1.shape[1] / 2), int(img1
        .shape[0] / 2)
    mask = np.sqrt((x - center_x)**2 + (y - center_y)**2)
        <= r
    f_hybrid_shifted = f_img1_shifted * mask +
       f_img2_shifted * (~mask)
    f_hybrid_ishifted = ifftshift(f_hybrid_shifted)
    hybrid_img = np.abs(ifft2(f_hybrid_ishifted))
    return hybrid_img
```

Hàm *create_hybrid_img* là một công cụ trong xử lý ảnh để tạo ra ảnh hybrid bằng cách kết hợp các thành phần tần số thấp của một ảnh với các thành phần tần số cao của ảnh khác. Cách tiếp cận này thường được sử dụng để tạo ra các hiệu ứng thị giác độc đáo hoặc cải thiện chất lượng ảnh trong một số ứng dụng nhất định.

Ưu điểm:

- Hiệu quả: Hàm hiệu quả trong việc tạo ra ảnh hybrid mà giữ được tính chất mượt mà của một ảnh và chi tiết của ảnh kia.
- Linh hoạt: Có thể điều chỉnh dễ dàng thông qua tham số bán kính r để kiểm soát mức độ của các thành phần tần số được trộn lẫn.



Hình 2: Result images