به نام خدا

عنوان

تکلیف دوم درس پردازش تصویر

استاد

دکتر منصوری

دانشجو

محمدعلی مجتهدسلیمانی

٤٠٣٣٩٠٤٥٠٤

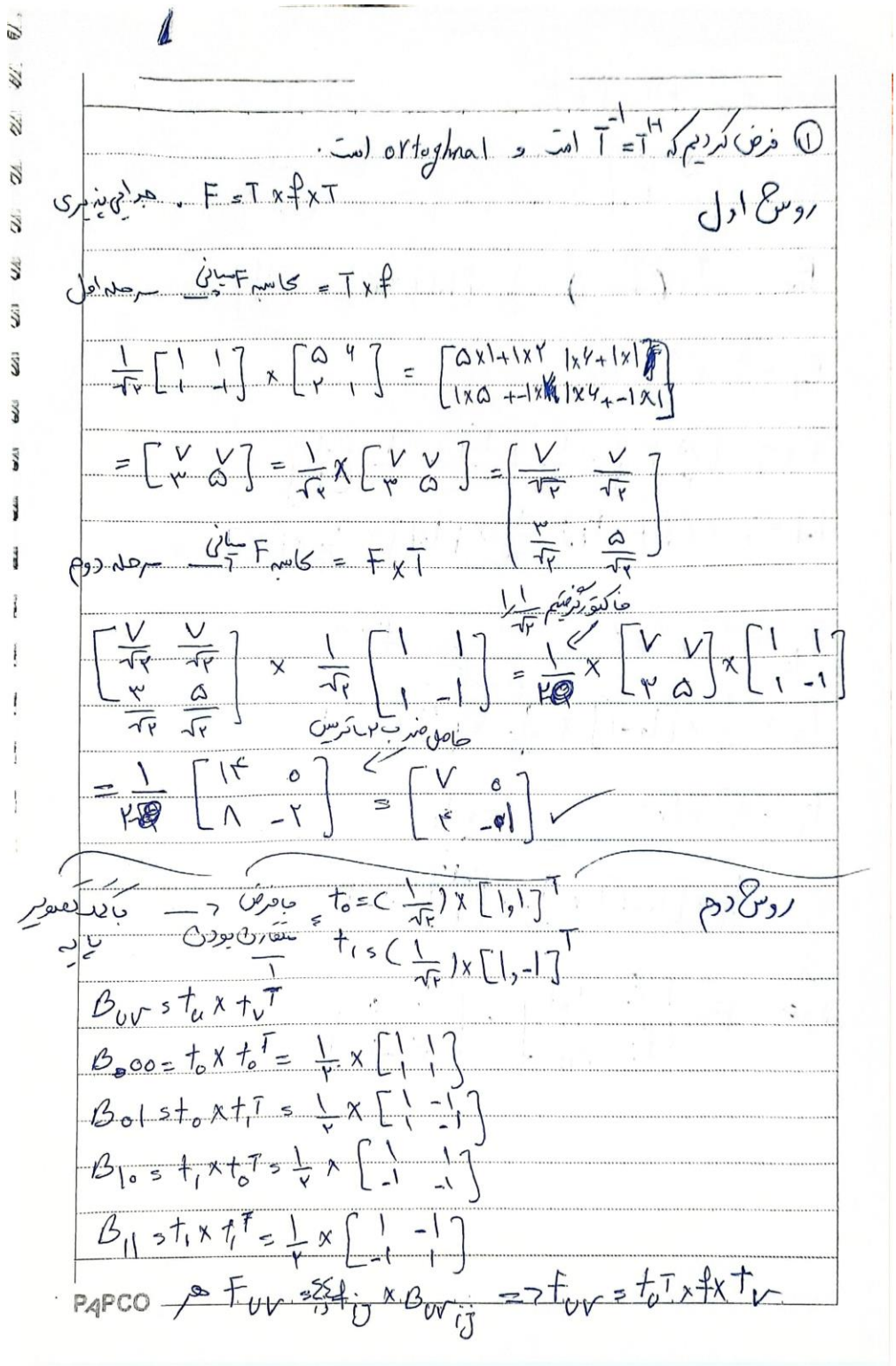
فهرست پاسخ

سوال اول

این سوال در ۳ بخش حل شده است.

بخش الف

فایل این بخش تحت نام `01_PARTA_1&2` در پوشه ۰۱ وجود دارد. در ادامه نیز خواهد آمد:


① فرض کردیم که $T^{-1}=T^H$ است و orthogonal است.
روش اول
$F=T\times f\times T$ ← جداپذیری
مرحله اول ← محاسبه F میانی $= T\times f$
$\frac{1}{\sqrt{2}}\begin{bmatrix}1&1\\1&-1\end{bmatrix}\times\begin{bmatrix}5&4\\2&1\end{bmatrix}=\begin{bmatrix}5\times1+1\times2 & 1\times4+1\times1\\ 1\times5+-1\times2 & 1\times4+-1\times1\end{bmatrix}$
$=\begin{bmatrix}7&5\\3&3\end{bmatrix}=\frac{1}{\sqrt{2}}\times\begin{bmatrix}7&5\\3&3\end{bmatrix}=\begin{bmatrix}\frac{7}{\sqrt{2}}&\frac{5}{\sqrt{2}}\\ \frac{3}{\sqrt{2}}&\frac{3}{\sqrt{2}}\end{bmatrix}$
مرحله دوم ← محاسبه F نهایی $= F\times T^{-1}$
ماتریس ترانهاده $\frac{1}{\sqrt{2}}$
$\begin{bmatrix}\frac{7}{\sqrt{2}}&\frac{5}{\sqrt{2}}\\ \frac{3}{\sqrt{2}}&\frac{3}{\sqrt{2}}\end{bmatrix}\times\frac{1}{\sqrt{2}}\begin{bmatrix}1&1\\1&-1\end{bmatrix}=\frac{1}{2}\times\begin{bmatrix}7&5\\3&3\end{bmatrix}\times\begin{bmatrix}1&1\\1&-1\end{bmatrix}$
حاصل ضرب ۲ ماتریس
$=\frac{1}{2}\begin{bmatrix}12&2\\6&0\end{bmatrix}=\begin{bmatrix}6&1\\3&0\end{bmatrix}$ ✓
روش دوم
فرض ← $t_0=(\frac{1}{\sqrt{2}})\times[1,1]^T$ ← باید متعامد باشد
$t_1=(\frac{1}{\sqrt{2}})\times[1,-1]^T$
$B_{uv}=t_u\times t_v^T$
$B_{00}=t_0\times t_0^T=\frac{1}{2}\times\begin{bmatrix}1&1\\1&1\end{bmatrix}$
$B_{01}=t_0\times t_1^T=\frac{1}{2}\times\begin{bmatrix}1&-1\\1&-1\end{bmatrix}$
$B_{10}=t_1\times t_0^T=\frac{1}{2}\times\begin{bmatrix}1&1\\-1&-1\end{bmatrix}$
$B_{11}=t_1\times t_1^T=\frac{1}{2}\times\begin{bmatrix}1&-1\\-1&1\end{bmatrix}$
$F_{uv}=\sum_{ij} f_{ij}\times B_{uv_{ij}}$ ⇒ $f_{uv}=t_0^T\times f\times t_v$

$F_{00} = t_0^T \times f \times t_0$

$f \times t_0 = \begin{bmatrix} 5 & 4 \\ 2 & 1 \end{bmatrix} \times \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 11 \\ 3 \end{bmatrix}$

$F_{00} = t_0^T \times (f \times t_0) = \frac{1}{\sqrt{2}} \times [1,1] \times \left(\frac{1}{\sqrt{2}}\right) \times \begin{bmatrix} 11 \\ 3 \end{bmatrix} = 7$

$F_{01} = t_0^T \times f \times t_1$

$f \times t_1 = \begin{bmatrix} 5 & 4 \\ 2 & 1 \end{bmatrix} \times \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} -1 \\ 1 \end{bmatrix}$

$F_{01} = t_0^T \times (f \times t_1) \quad \frac{1}{\sqrt{2}} \times [1,1] \times \frac{1}{\sqrt{2}} \times \begin{bmatrix} -1 \\ 1 \end{bmatrix} = 0$

$F_{10} = t_1^T \times f \times t_0$ ← $f \times t_0$

$F_{10} = \frac{1}{\sqrt{2}} \times [1,-1] \times \frac{1}{\sqrt{2}} \times \begin{bmatrix} 11 \\ 3 \end{bmatrix} = -1$

$F_{11} = t_1^T \times f \times t_1$ ← $F \times t_1$

$F_{11} = \frac{1}{\sqrt{2}} \times [1,-1] \times \frac{1}{\sqrt{2}} \times \begin{bmatrix} -1 \\ 1 \end{bmatrix} = -1$

بنابراین $F = \begin{bmatrix} F_{00} & F_{01} \\ F_{10} & F_{11} \end{bmatrix} = \begin{bmatrix} 7 & 0 \\ -1 & -1 \end{bmatrix}$

P4PCO

از هر ۲ روش خواسته شده پیاده کردیم. بخش هایی از محاسبات نوشته نشده اند.

بخش ب

با مشاهده نتیجه میتوان گفت که انرژی تا حدود زیادی در مولفه F(0,0)=7 است متمرکز شده است و ضریب بسیار بزرگتری نسبت به سایر موارد است. برای سایر جزئیات نیز F(0,1)=0 بیانگر این است که جزئیات افقی خاصی نداریم. F(1,0)=4 یعنی اینکه یک سری جزئیات عمودی برخلاف جزئیات افقی داریم. F(1,1)=-1 یعنی جزئیات افقی نیز قابل مشاهده است. به طور کلی این به ما میگوید که سمت بالا چپ بیشتر انرژی در آن قرار دارد. و بقیه جزئیات فرکانس بالا اندازه کمی دارند. پس به طور کلی میتوان گفت یک تصویر هموار شبیه به DC وجود دارد.

بخش ج

مولفه F(0,0) در تصویر تبدیل یافته بیانگر یک میانگین scale داده شده است. انگار تخمینی از ضرایب یا مولفه‌های low frequency از تصویر هستند. و یک نسخه scale داده شده از جمع یا میانگین تمام مقادیر پیکسل ها هست. زیرا اگر محاسبه کنیم میبینیم جمع f00+ f01 + f10 + f11 برابر با جمع مقادیر پیکسل های تصویر ۲ در ۲ است. که ضرب این مقدار در ½ به ما یک میانگین scale داده شده میدهد. خود F(0,0) را به این شکل بدست آوردیم:

```
F(0,0) = T(row 1) * f * T(col 1)
```

بدست آوردیم. همچنین با فرض اینکه میدانیم T متقارن است.

به صورت کلی:

```
F(0,0) = (1/2) * [1 * (f00+f01) + 1 * (f10+f11)]
F(0,0) = (1/2) * (f00 + f01 + f10 + f11)
```

که برای تصویر ما:

```
f = [[5, 6], [2, 1]]

f00 = 5, f01 = 6, f10 = 2, f11 = 1.
F(0,0) = (1/2) * (5 + 6 + 2 + 1)
F(0,0) = (1/2) * (14)
F(0,0) = 7
```

که دقیقا برابر با مقداری است که برای F(0,0) در بخش الف بدست آوردیم.

سوال دوم

این سوال را در ۲ بخش حل میکنیم. تبدیل فوریه برای اولین بار توسط فوریه معرفی شد و ادعا میکرد هر سیگنالی یا هر تابعی میتوان به صورت جمع یک سری توابع سینوس و کسینوس نوشته شود حتی اگر متناوب نباشند. البته این ادعای فوریه فقط برای سیگنال هایی بود که شرایط دریکله (Dirichlet) را داشته باشند.

بخش الف

تبدیل فوریه و سری فوریه را از لحاظ گسسته/پیوسته و متناوب/نامتناوب بودن ورودی و نتیجه مقایسه میکنیم.

سری فوریه

این سری با این هدف استفاده میشود که یک سیگنال متناوب و زمان-پیوسته را به عنوان جمع نامحدود یک سری توابع سینوس و کسینوس نمایش دهد. که هر کدام از این توابع یا مولفه ها یک اندازه و فاز دارند.

ورودی:

ورودی این سری باید پیوسته باشد همچنین ورودی حتما باید متناوب باشد.

خروجی:

خروجی سری فوریه گسسته خواهد بود یعنی ضرایب آن گسسته هستند. `spectrum` آن خطی خواهد بود. همچنین هیچ انرژی بین فرکانس‌های هارمونیک وجود ندارد.

تبدیل فوریه

این تبدیل با این هدف استفاده میشود که یک سیگنال زمان-پیوسته غیر متناوب را آنالیز کند و به توابع سینوس و کسینوس تجزیه کند و انتگرال یک سری سینوس و کسینوس خواهد بود.

ورودی:

ورودی این سیگنال از جنس پیوسته و غیر متناوب خواهد بود. سیگنال ورودی حتما باید اصول دریکه را رعایت بکند.

خروجی:

خروجی نیز پیوسته خواهد بود همچنین **spectrum** آن نیز پیوسته خواهند بود. در واقع محتوای سیگنال را نه فقط در یک نقطه گسسته بلکه در یک محدوده بازگو میکند.

در واقع تبدیل فوریه به نوعی همان سری فوریه است که انگار پارامتر T به سمت بینهایت رفته است.

تبدیل فوریه	سری فوریه	ویژگی
پیوسته	پیوسته	نوع ورودی
غیر متناوب	متناوب	متناوب بودن/نبودن
پیوسته	گسسته	خروجی در دامنه فرکانس

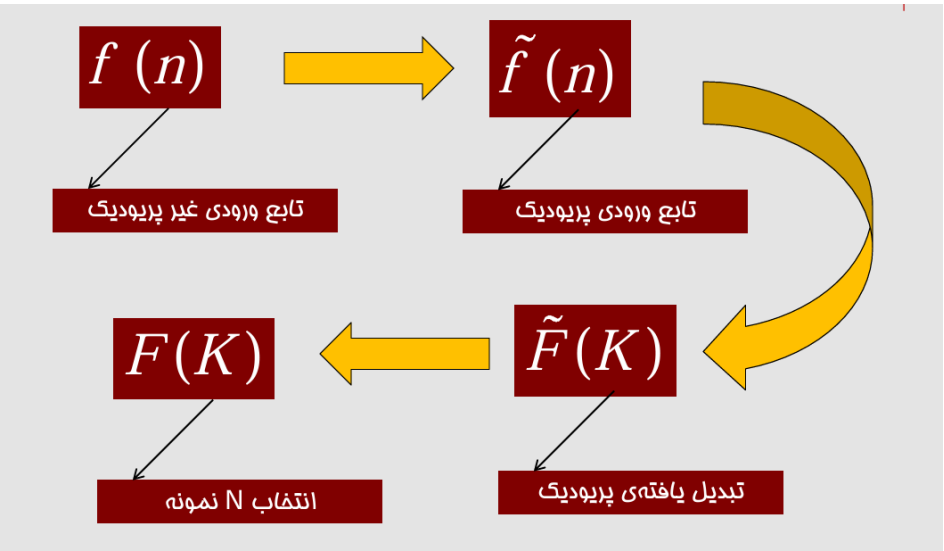
بخش ب

ما در این بخش تبدیل فوریه گسسته را که بر روی سیگنال های زمان گسسته و محدود صورت میگیرد را به متناظر آن بر روی سیگنال های زمان پیوسته یعنی سری فوری و تبدیل فوریه متصل کنیم.

۲ روش برای این کار وجود دارد.

استفاده از سری فوریه: در این روش ما سیگنال زمان گسسته محدود خود را به عنوان یک دوره تناوب از یک سیگنال گسسته متناوب نامحدود فرض میکنیم. یعنی میگیم این یک سیگنالی که وجود دارد در واقع متناوب و نامحدود است. برای این کار ابتدا باید یک سیگنال متناوب بسازیم و بعد سری فوریه گسسته را روی آن اعمال کنیم و در نهایت ضرایب سری فوریه گسسته را به تبدیل فوریه گسسته مرتبط کنیم. چرا این کار را باید بکنیم؟ چونکه ضرایب تبدیل فوریه گسسته (DFT) اساسا نسخه‌ای مقیاس داده شده از ضرایب سری فوریه گسسته (DTFS) هستند. DFT به اندازه N نمونه از یک تناوب را میگیرد.

استفاده از تبدیل فوریه پیوسته: در این روش ابتدا از تبدیل فوریه پیوسته مربوط به یک سیگنال زمان پیوسته یا تبدیل فوریه زمان گسسته مربوط به یک دنباله گسسته نمونه گیری میکنیم. بعد نمونه ها DTFT را که تبدیل فوریه زمان گسسته ما هست را به DFT که تبدیل فوریه گسسته است مرتبط میکنیم.



مثال

یک سیگنال تصویر تک‌بعدی ساده را در نظر بگیریم: یک ردیف از ۸ پیکسل.

x[n] = [۰, ۰, ۲۵۵, ۲۵۵, ۲۵۵, ۲۵۵, ۰, ۰] برای n = 0, ..., 7. (N=8)

این نشان دهنده یک نوار روشن در وسط یک پس زمینه تاریک است.

محاسبه ضرایب DFT :

X[0] (DC component/ k=0)

X[0] = Σ x[n] * e^0 = 0+0+255+255+255+255+0+0 = 4 * 255 = 1020.

این نشان دهنده مجموع شدت پیکسل‌ها (متناسب با شدت متوسط) است.

`X[k]` برای `k > 0`: این ضرایب نحوه تغییر سیگنال را ثبت می‌کنند. انتقال‌های سریع از ۰ به ۲۵۵ و برگشت به ۰ به مؤلفه‌های فرکانس بالاتر کمک می‌کنند. برای مثال، `X[4]` (مولفه فرکانس نایکوئیست برای `N=8`) به دنبال الگوهایی مانند [+، −، +، −، +، −، +، −] خواهد بود. سیگنال ما دقیقاً در این بالاترین فرکانس، مولفه قوی ندارد، اما به دلیل شکل `"car"` خود، انرژی آن در مقادیر مختلف `k` پخش خواهد شد.

۱. `DFT` از دیدگاه سری فوریه:

ما تصور می‌کنیم ردیف ۸ پیکسلی `x[n]` ما یک دوره از یک الگوی تکرار بی‌نهایت است:

[۰,۰,۲۵۵,۲۵۵,۲۵۵,۲۵۵,۰,۰],[۰,۰,۲۵۵,۲۵۵,۲۵۵,۲۵۵,۰,۰],...

این سیگنال گسسته دوره‌ای به عنوان نمونه‌هایی از یک سیگنال دوره‌ای پیوسته `x_p(t)` (مانند یک موج مربعی دوره‌ای) در نظر گرفته می‌شود.

ضرایب `CTFS c_k` این `x_p(t)` قدرت فرکانس اساسی `w_0` و هارمونیک‌های آن `kw_0` را توصیف می‌کنند. ضرایب `DFT` ما `X[k]` برابر با `N * c_k` هستند. بنابراین، `X[0] = 8 * c_0. c_0` مقدار میانگین `x_p(t)` در یک دوره تناوب است که

برای سیگنال ما (۴۵۵*۴)/۸ = ۱۲۷.۵ است. بنابراین X[0] = 8 * 127.5 = 1020، که با محاسبه مستقیم ما مطابقت دارد.

مقادیر دیگر X[k] (برای k=1..7) نسخه‌های مقیاس‌بندی شده ضرایب c_k هستند که نشان‌دهنده قدرت فرکانس‌های هارمونیک گسسته πk/N۲ می‌باشند.

۲. تبدیل فوریه فوریه (DFT) از دیدگاه تبدیل فوریه:

ما تصور می‌کنیم که ردیف ۸ پیکسلی x[n] ما با نمونه‌برداری از یک سیگنال پیوسته و غیرمتناوب x_a(t) به دست آمده است (مثلاً سیگنالی که صفر است، سپس برای مدتی به مقدار بالایی افزایش می‌یابد، سپس به صفر کاهش می‌یابد و صفر می‌ماند).

تبدیل فوریه دنباله ۸ نمونه‌ای متناهی x[n] ما به صورت X_{N,DTFT}(Ω) = Σ_{n=0}^{7} x[n] * e^(-j * Ω * n خواهد بود. این X_{N,DTFT}(Ω) یک تابع پیوسته از Ω است و با π۲ تناوبی است.

ضرایب DFT X[k] به سادگی N=8 نمونه از این X_{N,DTFT}(Ω) در فرکانس‌های Ω_k = k * 2π/8 = kπ/4 هستند.

X[0] = X_{N,DTFT}(۰)

$X[1] = X_{N,DTFT}(\pi/4)$

$X[2] = X_{N,DTFT}(\pi/2)$

...

$X[7] = X_{N,DTFT}(7\pi/4)$

شکل $X_{N,DTFT}(\Omega)$ به CTFT $X_a(w)$ سیگنال پیوسته زیرین $x_a(t)$ مربوط می‌شود، به طور خاص یک تابع سینک مانند برای یک پالس مستطیلی، که با اثرات نمونه‌برداری و پنجره‌بندی کانولوشن شده است. DFT $X[k]$ نقاط خاصی را روی این شکل طیفی به ما می‌دهد.

مثال زده شده در حالت یک بعدی بوده و برای حالت ۲ بعدی نیز همین شکلی صدق میکند.

سوال سوم

این سوال در ۳ بخش انجام شده است که کد های آن در فایل `03_PARTA.ipynb` و `03_PARTC.ipynb` قرار دارد به همراه تصاویر خروجی بدست آمده و توضیح مربوط به کد هر بخش در همان بخش قرار دارد.

بخش الف

در این قسمت تصاویر پایه ۸ در ۸ DFT را به صورت ۲ قسمت مجزا موهومی و حقیقی نمایش میدهیم و توضیح مربوط به کد آن در قسمت گزارش کار قرار دارد.

گزارش کار

```python
import numpy as np
import matplotlib.pyplot as plt
```

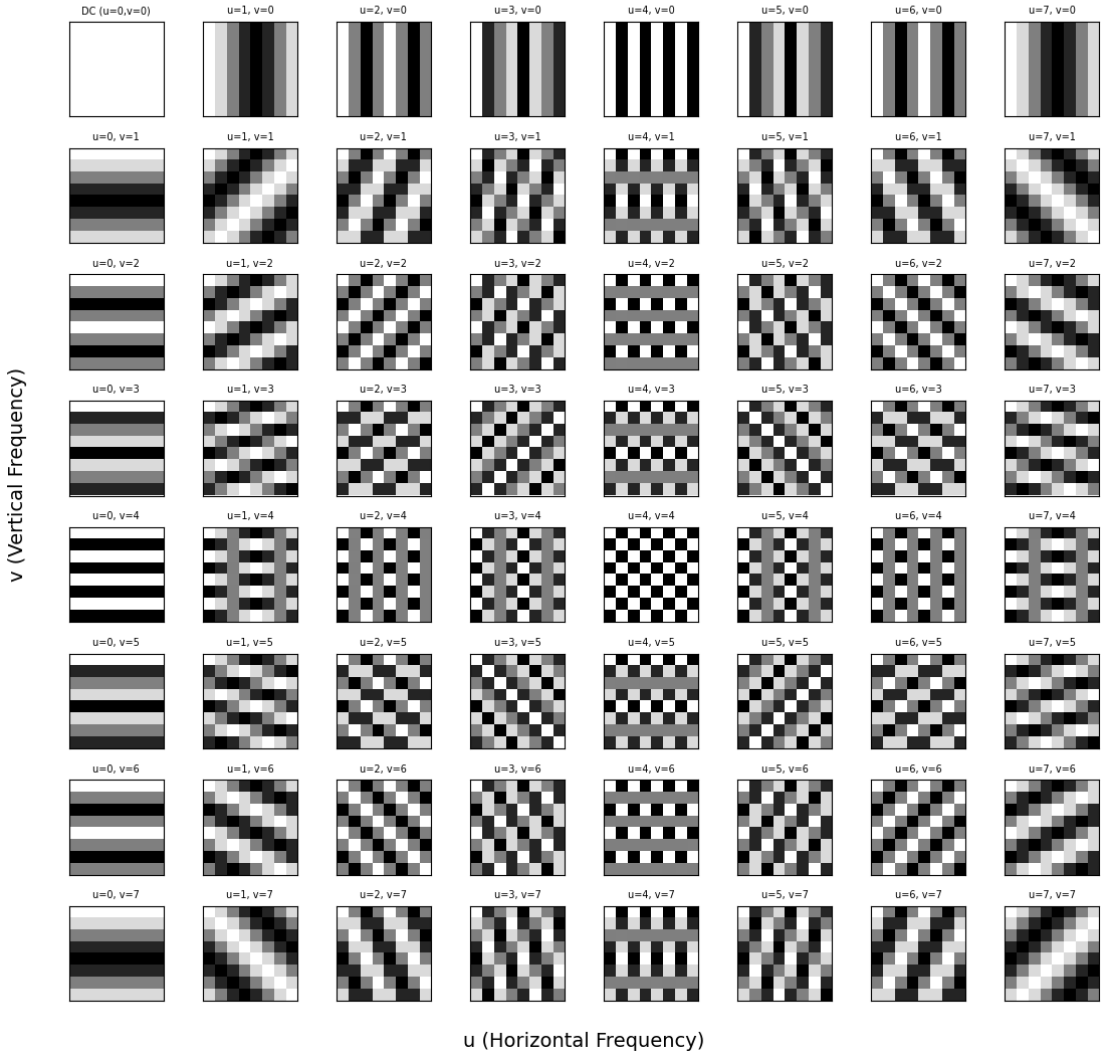
کتابخانه مورد نیاز خود را وارد کردیم.

```python
def generate_dft_basis_function(N, u_freq, v_freq):
    x_coords = np.arange(N)
    y_coords = np.arange(N)

    xx, yy = np.meshgrid(x_coords, y_coords)

    term = 2 * np.pi * ((u_freq * xx / N) + (v_freq * yy / N))

    real_part = np.cos(term)
    imag_part = np.sin(term)

    return real_part, imag_part
```

تابع مورد نظر که برای تولید تصاویر پایه DFT به کار میرود را تعریف کردیم خروجی به صورت دو بخش موهومی و حقیقی خواده بود. با `term` فاز را محاسبه میکنیم. و به دو قسمت بعدی میدهیم.

```python
N_size = 8

fig_real, axes_real = plt.subplots(N_size, N_size, figsize=(12, 12))
english_title_real = "8x8 Discrete Fourier Transform Basis Functions (Real Part)"
fig_real.suptitle(english_title_real, fontsize=16)

for v_idx in range(N_size):
    for u_idx in range(N_size):
        real_part, _ = generate_dft_basis_function(N_size, u_idx, v_idx)

        ax = axes_real[v_idx, u_idx]
        ax.imshow(real_part, cmap='gray', vmin=-1, vmax=1)
        ax.set_xticks([])
        ax.set_yticks([])

        if v_idx == 0 and u_idx == 0:
            ax.set_title(f'DC (u={u_idx},v={v_idx})', fontsize=7)
        else:
            ax.set_title(f'u={u_idx}, v={v_idx}', fontsize=7)


fig_real.text(0.5, 0.03, 'u (Horizontal Frequency)', ha='center', va='center', fontsize=14)
fig_real.text(0.03, 0.5, 'v (Vertical Frequency)', ha='center', va='center', rotation='vertical', fontsize=14)


plt.tight_layout(rect=[0.05, 0.05, 0.95, 0.93])
plt.show()
```
Python

قسمت حقیقی خواسته شده که فقط با کسینوس ها کار داریم برای نمایش بخش حقیقی و بعد خروجی را **plot** میکنیم. را برای تصویر پایه ۸ در ۸ نشان میدهیم:

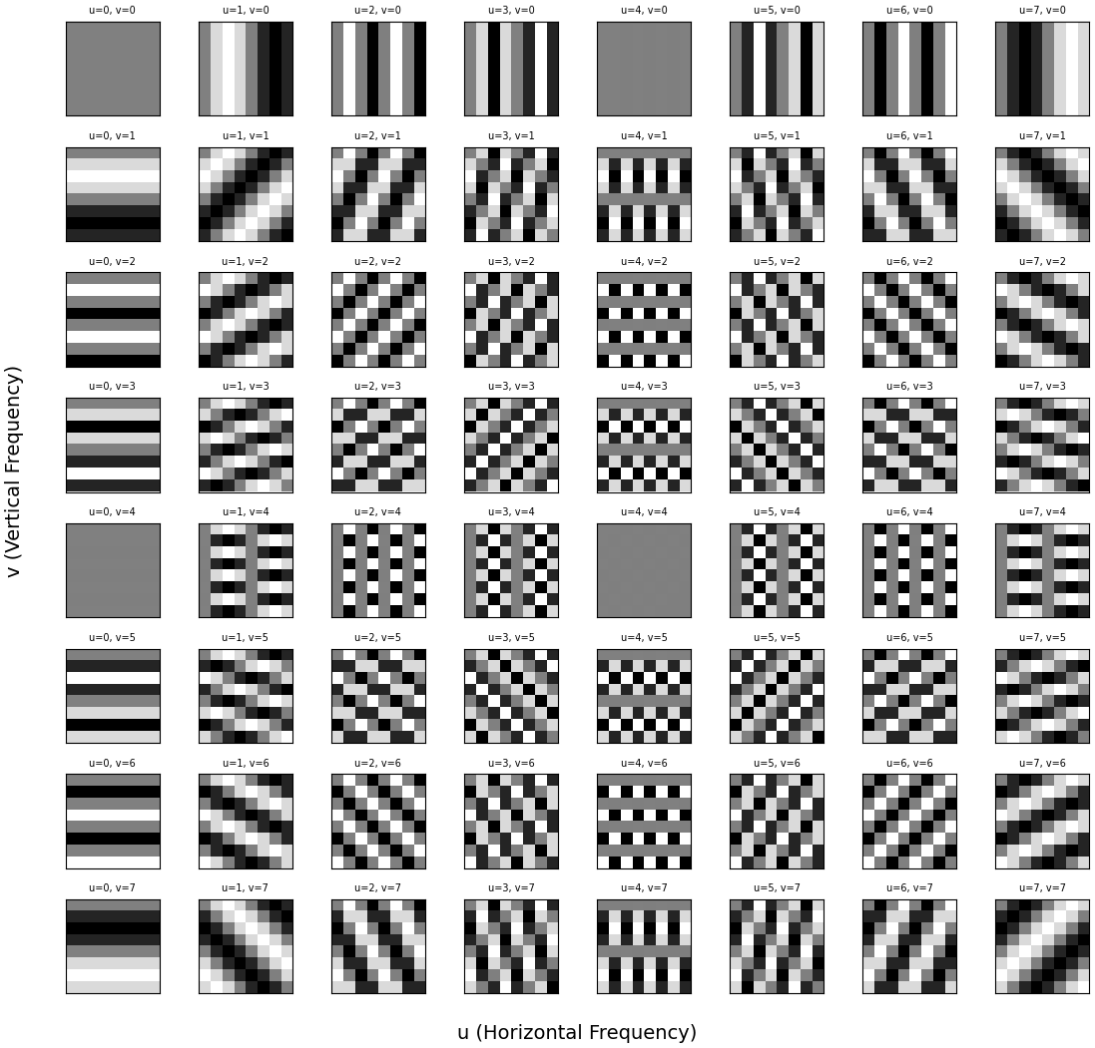


تصویر پایه قسمت حقیقی DFT با سایز ۸ در ۸.

```python
fig_imag, axes_imag = plt.subplots(N_size, N_size, figsize=(12, 12))
english_title_imag = "8x8 Discrete Fourier Transform Basis Functions (Imaginary Part)"
fig_imag.suptitle(english_title_imag, fontsize=16)

for v_idx in range(N_size):
    for u_idx in range(N_size):

        _, imag_part = generate_dft_basis_function(N_size, u_idx, v_idx)

        ax = axes_imag[v_idx, u_idx]
        ax.imshow(imag_part, cmap='gray', vmin=-1, vmax=1)
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_title(f'u={u_idx}, v={v_idx}', fontsize=7)


fig_imag.text(0.5, 0.03, 'u (Horizontal Frequency)', ha='center', va='center', fontsize=14)
fig_imag.text(0.03, 0.5, 'v (Vertical Frequency)', ha='center', va='center', rotation='vertical', fontsize=14)


plt.tight_layout(rect=[0.05, 0.05, 0.95, 0.93])
plt.show()
```

Python

در این قسمت نیز قسمت موهومی تبدیل **DFT** را نمایش میدهیم که برای اینکار با الگوهای موج سینوس کار داریم و بعد خروجی را **plot** کردیم:



تصویر پایه قسمت موهومی تبدیل DFT با سایز ۸در۸.

بخش ب

یک تبدیل فوریه گسسته ۲ بعدی یک تصویر را به جمع یک سری اعداد مختلط توابع سینوس و کسینوس تجزیه میکند. هر تصویر پایه در واقع یک موج ۲ بعدی از سینوس یا کسینوس هست که اندازه و فاز دارد که با ضرایب DFT بیانگر همین اندازه و فاز متناظر با هر تصویر پایه خواهند بود. هر چه قدر اندازه بزرگتر باشد برای F(u,v) یعنی تصاویر پایه متناظر با (u,v) وزن بیشتری دارند یا انگار درصد مشارکتی بیشتری دارند. F(0,0) یک تصویر ثابت هموار هست با میانگین روشنایی ورودی. در مواقعی که U> 0 است یعنی خط عمودی داریم یا به اصطلاح سینوس ها در جهت محور x حرکت میکنند برای مواقعی V>0 هست یعنی خط افقی داریم و به اصطلاح سینوس در جهت y تغییر میکنند. اگر هر دو بزرگتر صفر باشند خطوط افقی یا شطرنجی داریم. هر چه قدر این مقادیر بزرگتر شود شدت تغییرات را بیانگر هستند که تغییرات دارند سریع رخ میدهند. پس به طور کلی اگر تصویر ورودی به یک الگو خاصی از تصویر پایه شبیه باشد ضرایب DFT متناظر با آن نیز بزرگتر هستند اندازه بزرگتری دارند و وزن بیشتری میگیرند.

اگر لبه افقی وجود داشته باشد یعنی یک تغییر سریع در جهت عمودی رخ داده است و اگر لبه عمودی وجود داشته باشد یعنی یک تغییر سریع در جهت افقی رخ داده است. اگر تغییرات هم در جهت افقی و هم عمودی سریع و ناگهانی و به سرعت باشند یعنی الگو شطرنجی داریم، یعنی مولفه های مورب آن قوی تر هستند و هم u هم v جز high frequency هستند.

کدام تصویر پایه وزن بیشتری میگیرد؟ اون تصویر پایه ای که بهتر بتواند ساختار در تصویر ورودی را نشان بدهد. به طور خاص، ضرایبی مانند F(M/2, N/2) ("گوشه‌های" طیف DFT بدون تغییر)، F(M/2, 0) و F(0, N/2) قوی خواهند بود. مؤلفه F(M/2, N/2) نشان

دهنده یک تصویر پایه است که خود یک صفحه شطرنجی با بالاترین فرکانس است. F(M/2, 0) نشان دهنده یک تصویر پایه از خطوط عمودی در بالاترین فرکانس افقی است.

F(0, N/2) نشان دهنده یک تصویر پایه از خطوط افقی در بالاترین فرکانس عمودی است.یک صفحه شطرنجی کامل، مجموعه ای از این تصاویر پایه فرکانس بالا است. تصویر اغلب از نوع F(M/2, N/2) است.

بخش ج

گزارش کار مربوط به این بخش را در بخش گزارش کار تهیه کردیم و خروجی های تهیه شده تحت عنوان o3_PARTC_... .PNG قرار دارند. کد مربوط به این بخش در فایل 03_PARTC.ipynb قرار دارد. در این قسمت نیاز داریم که الگوهای گفته شده را پیاده سازی کنیم بعد بر روی آنها DFT اعمال کنیم و بعد تصویر اصلی، magnitude spectrum، phase spectrum و بخش موهومی و حقیقی DFT را نشان بدهیم.

گزارش کار

```
import numpy as np
import matplotlib.pyplot as plt
```

3.9s Python

کتابخانه مورد نیاز را وارد کردیم.

```python
def display_dft_components(image, title_prefix=""):
    dft = np.fft.fft2(image)

    dft_shifted = np.fft.fftshift(dft)


    magnitude_spectrum = np.log(1 + np.abs(dft_shifted))

    phase_spectrum = np.angle(dft_shifted)

    real_part = np.real(dft_shifted)

    imaginary_part = np.imag(dft_shifted)

    plt.figure(figsize=(15, 10))

    plt.subplot(2, 3, 1)
    plt.imshow(image, cmap='gray')
    plt.title(f'{title_prefix} - Original Image')
    plt.axis('off')

    plt.subplot(2, 3, 2)
    plt.imshow(magnitude_spectrum, cmap='gray')
    plt.title(f'{title_prefix} - Magnitude Spectrum')
    plt.axis('off')

    plt.subplot(2, 3, 3)
    plt.imshow(phase_spectrum, cmap='gray')
    plt.title(f'{title_prefix} - Phase Spectrum')
```

یک تابع کمکی برای نمایش تصاویر و مولفه های **DFT** ساختیم. که تصویر اصلی به همراه بخش های موهومی و حقیقی و فاز و اندازه را نشان میدهیم به کمک این تابع. با کمک **NP.FFT.FFT2** تبدیل **DFT** ۲ بعدی را محاسبه میکنیم و بعد با کمک **NP.FFT.FFTSHIFT** میایم و مولفه ها را به سمت مرکز برای نمایش شیفت میدهیم. برای اینکه بتوانیم **Magnitude spectrum** را نشان بدهیم آن را با یک **log**، **scale** میکنیم تا بهتر نمایش داده شود. فاز هم با کمک **np.angle** نمایش میدهیم. برای قسمت های موهومی و حقیقی نیز از کتابخانه **numpy** کمک گرفتیم. در نهایت خروجی را نشان میدهیم.

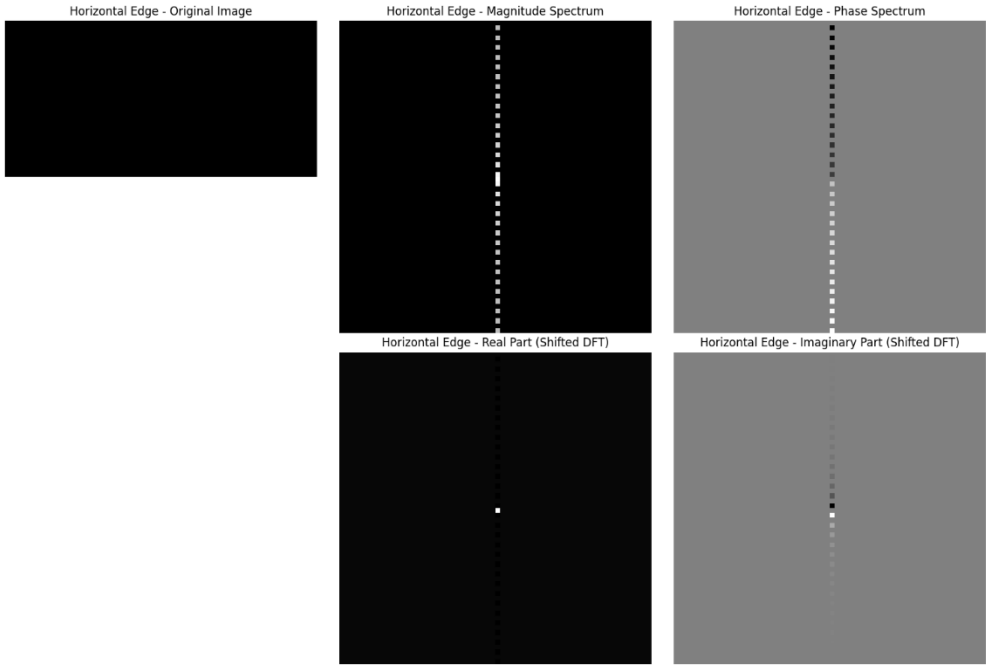
تعریف لبه افقی و عمودی و شطرنجی را نیز در قسمت قبل مشخص کردیم. فقط لازم است بدانیم که در لبه افقی انرژی در محور عمودی متمرکز میشود و در لبه عمودی انرژی در محور افقی متمرکز میشود. برای الگوی شطرنجی نیز انرژی در مولفه های **high frequency** متمرکز

میشوند که گوشه های **DFT** هستند که متناظر با تصویر پایه **fine-grained checkerboards** خواهد بود.

```
horizontal_edge = np.zeros((N, N))
horizontal_edge[N//2:, :] = 255

display_dft_components(horizontal_edge, "Horizontal Edge")
```

✓ 1.1s Python

بر اساس توضیحات داده شده لبه افقی را ساختیم و نمایش میدهیم:

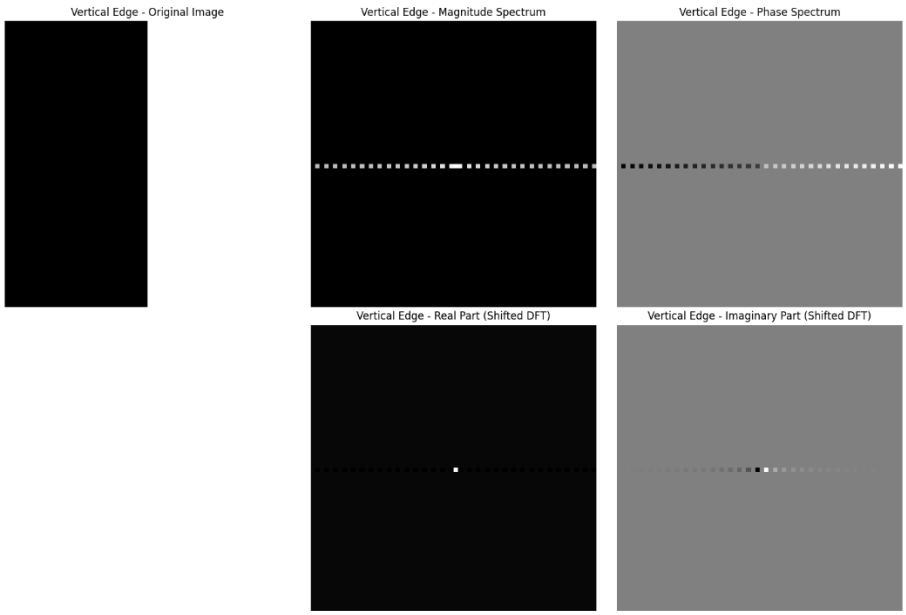


تصویر حاصل از لبه افقی هست که فاز و بخش موهومی و بخش حقیقی و **magnitude** آن را نمایش دادیم و تبدیل **DFT** اعمال کردیم.

```
vertical_edge = np.zeros((N, N))
vertical_edge[:, N//2:] = 255

display_dft_components(vertical_edge, "Vertical Edge")
```

✓ 1.1s Python

در ادامه لبه عمودی را ساختیم و خروجی را نمایش میدهیم:

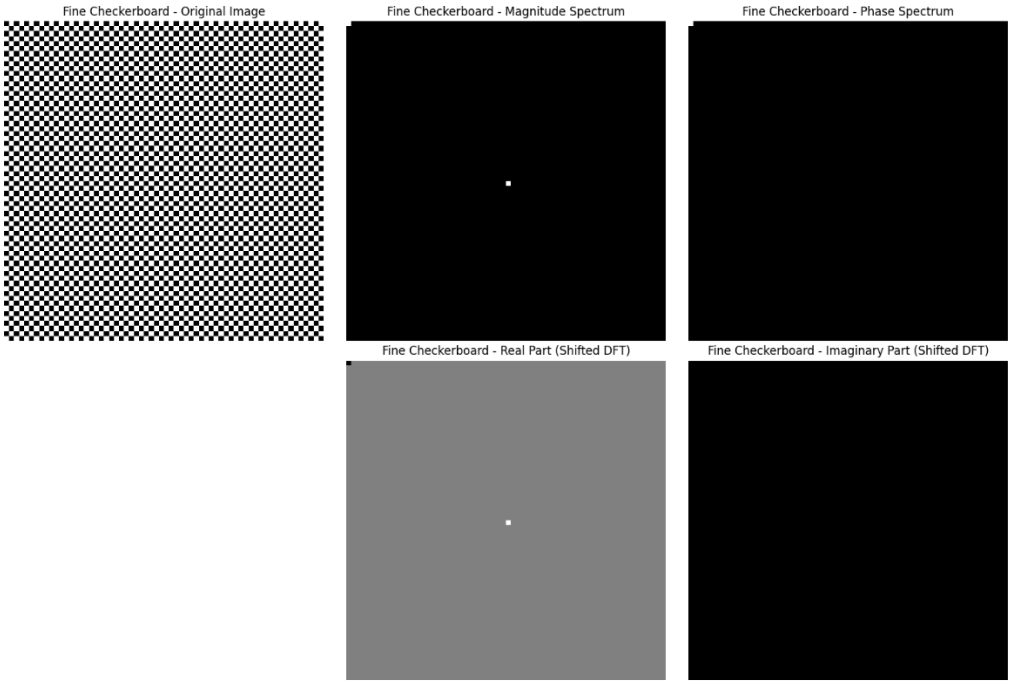


تصویر حاصل از ساخت لبه عمودی است. همراه با موارد خواسته شده.

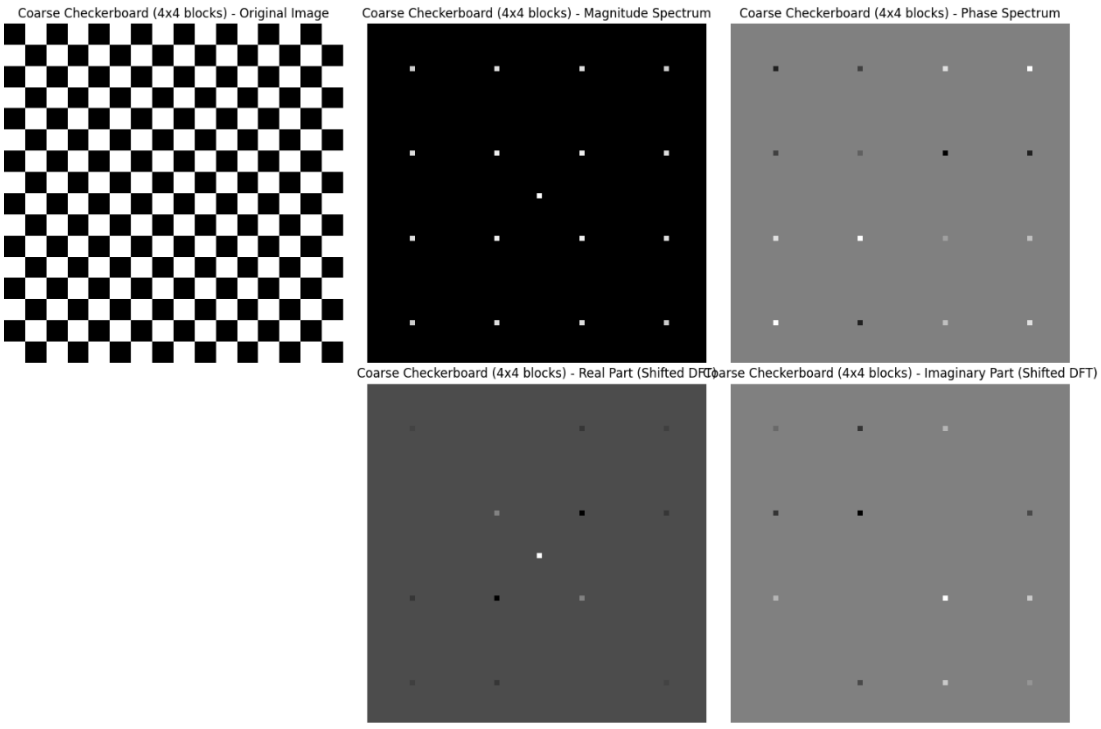
```python
checkerboard = np.zeros((N, N))
for i in range(N):
    for j in range(N):
        if (i + j) % 2 == 0:
            checkerboard[i, j] = 0
        else:
            checkerboard[i, j] = 255


fine_checkerboard = np.zeros((N,N))
x, y = np.meshgrid(np.arange(N), np.arange(N))
fine_checkerboard = ((x % 2) ^ (y % 2)) * 255

block_size = 4
coarse_checkerboard = np.zeros((N, N))
x_block, y_block = np.meshgrid(np.arange(N // block_size), np.arange(N // block_size))
pattern_block = ((x_block % 2) ^ (y_block % 2))
coarse_checkerboard = np.kron(pattern_block, np.ones((block_size, block_size))) * 255



display_dft_components(fine_checkerboard, "Fine Checkerboard")


display_dft_components(coarse_checkerboard, "Coarse Checkerboard (4x4 blocks)")
```

✓ 2.2s Python

در ادامه الگوی شطرنجی را ساختیم و نشان دادیم بر اساس موارد گفته شده. الگو را هم به صورت **fine** و هم به صورت **coarse** نمایش دادیم:



الگوی شطرنجی **fine** به همراه موارد خواسته شده.



الگوی **coarse** همراه با موارد خواسته شده.

تحلیل

در لبه افقی به صورت عمده تصویر پایه های متناظر با خطوط افقی فعال میشوند که انرژی در محور عمودی جریان دارد. تصویر اصلی، به وضوح یک لبه افقی را نشان می‌دهد. Magnitude spectrum یک خط عمودی روشن در مرکز میبینیم جایی که u=0 است و فرکانس ها غالبا عمودی هستند. روشن ترین نقطه در مرکز قرار دارد. ساختار قابل توجهی را نشان می‌دهد که نشان‌دهنده عدم تقارن الگوی لبه است. اجزای سینوسی (که لبه‌ها را تشکیل می‌دهند) اغلب در نمایش فوریه خود دارای بخش‌های موهومی هستند، مگر اینکه توابع کاملاً زوج باشند (مانند کسینوس).

در لبه عمودی به صورت عمده تصویر پایه های متناظر با خطوط عمودی فعال میشوند که انرژی در محور افقی جریان دارد. تصویر اصلی، به وضوح یک لبه عمودی را نشان می‌دهد. Magnitude spectrum همانطور که پیش‌بینی می‌شد، یک خط افقی روشن در مرکز می‌بینیم (v=0، پس از جابجایی). این بدان معناست که فرکانس‌های غالب کاملاً افقی هستند (u ≠ 0). مؤلفه DC روشن‌ترین نقطه است. انرژی در امتداد محور افقی نشان می‌دهد که تصاویر پایه شبیه خطوط عمودی به شدت وزن‌دار هستند.

در الگوی شطرنجی هم انرژی در گوشه ها جریان دارد و الگوی افقی وجود دارد و high frequency است.

سوال چهارم

در این سوال ما در ۲ بخش ۲ ماسک خواسته شده را طراحی کردیم و در دامنه فرکانس بر روی تصویر ورودی که لنا بود تحت عنوان **lena512** اعمال کردیم. هر بار تا ۳ مرحله شعاع را افزایش دادیم و نتایج را بررسی کردیم و گزارش دادیم. در نهایت در بخش سوم تحلیل مربوط به این قسمت را ارائه کردیم. ما در این سوال از معیار **PSNR** استفاده کردیم که به طور عمده برای این به کار میرود که کیفیت تصویر بازسازی شده را بر اساس تصویر اصلی بررسی کند، این معیار نسبت سیگنال به نویز را میسنجد یعنی چه مقدار نویز در حین پردازش این تصویر افزوده شده اند نسبت به مقدار عکس اولیه. این معیار از روش **MSE** برای تعریف تابع هزینه خود استفاده میکند به این شکل که برای هر پیکسل تفاوت بین مقدار اصلی آن با مقدار بازسازی شده آن را بدست میاورد و بعد از آن به توان ۲ میرساند و در نهایت همه این مقادیر را با هم جمع میکند. بعد از محاسبه **MSE**، خود **PSNR** را محاسبه میکنیم با فرمول زیر:

$$PSNR = 10 * LOG10 ((MAX_I)^2 / MSE)$$

که MAX_I در واقع بیشترین مقدار ممکن برای یک پیکسل هست (مثلا در ۸ بیت میشود ۲۵۵). در نهایت خروجی به صورت **decibels** ظاهر میشود.

هر چه قدر میزان **PSNR** بالاتر باشد به این معنی است که تصویر بازسازی شده کیفیت بالاتری دارد، یعنی به تصویر اصلی بسیار نزدیک است و نویز کمی دارد. اگر مقدار **MSE** صفر شود یعنی اینکه تصویر بازسازی شده دقیقا برابر با تصویر اصلی است، مقدار **PSNR** بینهایت میشود.

در هر بخش توضیح گزارش کار مربوط به آن بخش را آوردیم.

بخش الف

در این بخش یک ماسک ایده آل ایجاد کردیم و ۳ بار شعاع آن را افزایش دادیم و نتایج مختلف بدست آوردیم و در پایین هر مرحله مقدار **PSNR** آن را گزارش کردیم.

گزارش کار

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import math
```
✓ 0.0s

ابتدا کتابخانه‌های مورد نیاز را **import** کردیم. از **matplotlib** برای نشان دادن تصاویر خروجی استفاده کردیم.

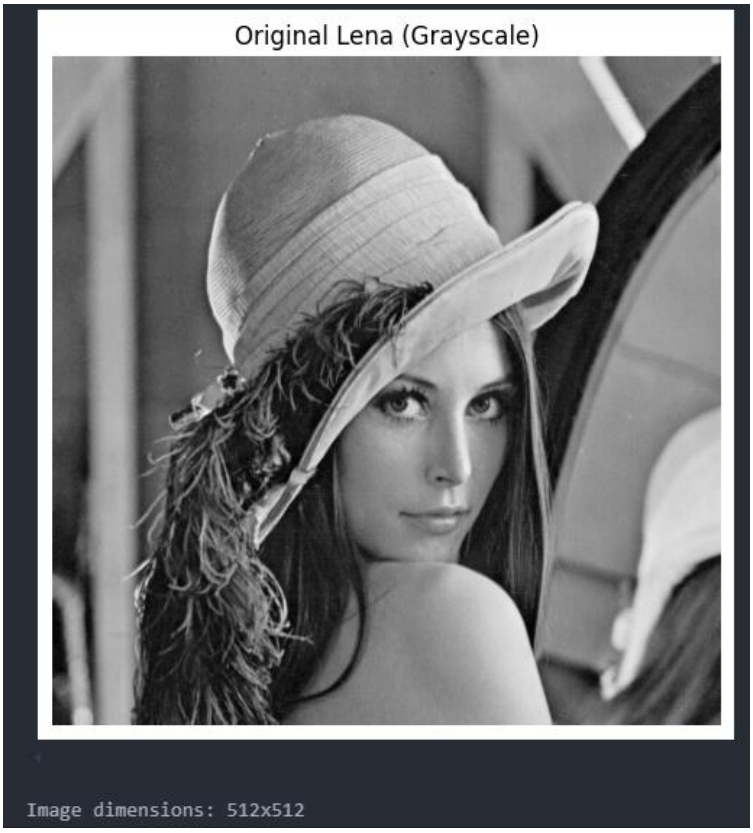
```
def display_image(image, title="Image", cmap=None):
    plt.figure(figsize=(6, 6))
    plt.imshow(image, cmap=cmap)
    plt.title(title)
    plt.axis('off')
    plt.show()
```
✓ 0.0s

یک تابع تعریف کردیم برای اینکه بتوانیم تصاویر را نشان بدهیم و در ادامه آن را فراخوانی کنیم.

```python
lena_path = 'lena512.bmp'
original_image = cv2.imread(lena_path, cv2.IMREAD_GRAYSCALE)

if original_image is None:
    print(f"Error: Could not load image from {lena_path}")
else:
    print("Original Lena Image:")
    display_image(original_image, title="Original Lena (Grayscale)", cmap='gray')
    img_height, img_width = original_image.shape
    print(f"Image dimensions: {img_width}x{img_height}")
```

0.3s Python

تصویر ورودی که **lena512.bmp** بود با کمک تابع **imread** از کتابخانه **open cv** خواندیم. نتایج خروجی در ادامه قرار دارد:

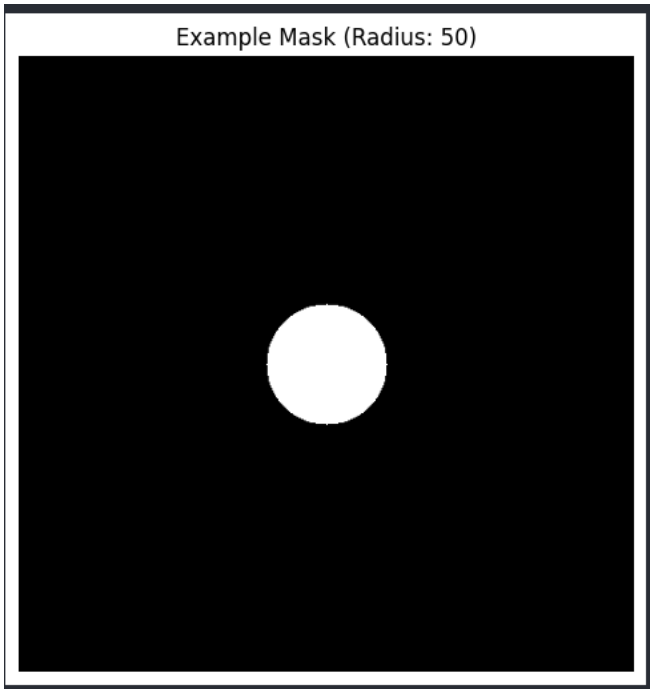


نتایج خروجی خواندن تصویر داده شده.

```python
def create_circular_mask(h, w, center_x, center_y, radius):
    Y, X = np.ogrid[:h, :w]
    dist_from_center = np.sqrt((X - center_x)**2 + (Y - center_y)**2)
    mask = dist_from_center <= radius
    return mask.astype(float)

if original_image is not None:
    rows, cols = original_image.shape
    center_x, center_y = cols // 2, rows // 2
    example_radius = 50
    example_mask = create_circular_mask(rows, cols, center_x, center_y, example_radius)
    display_image(example_mask, title=f"Example Mask (Radius: {example_radius})", cmap='gray')
```

✓ 0.1s Python

در این قسمت یک ماسک **low-filter** یا به اصطلاح پایین گذر تعریف کردیم که فقط مولفه‌های **low frequency** را عبور میدهد و از عبور مولفه‌های **high frequency** جلوگیری میکند. در نهایت در بخش دوم ماسک تهیه شده را نمایش دادیم. دقت شود در نهایت مقدار **Boolean** را به **float** تبدیل میکنیم یعنی ۰.۰ یا ۱.۰ چون که یک ماسک باینری است. در قسمت محاسبه فاصله از فاصله اقلیدوسی استفاده کردیم و با کمک **mask** مشخص کردیم همه نقاطی که فاصله آنها کمتر از شعاع است **True** و بقیه **False** باشند. با کمک مقادیر **True** یک ناحیه دایره‌ای ایجاد کردیم. نتایج خروجی به شکل زیر خواهد بود:



ماسک ساخته شده در قسمت قبل را نمایش دادیم که شعاع آن ۵۰ است.

```python
def calculate_psnr(img1, img2):
    img1 = img1.astype(np.float64)
    img2 = img2.astype(np.float64)

    mse = np.mean((img1 - img2) ** 2)
    if mse == 0:
        return float('inf')

    max_pixel_value = 255.0
    psnr = 20 * math.log10(max_pixel_value / math.sqrt(mse))
    return psnr
```

0.0s Python

در این قسمت همانطور که توضیح دادیم معیار **PSNR** آن را پیاده سازی کردیم. هر دو تصویر را در فرمت **float64** تنظیم کردیم تا از **overflow** در حین محاسبات جلوگیری شود مخصوصا وقتی قرار هست میزان تفاوت را به توان ۲ برسانیم در محاسبه **MSE**. بعد از محاسبه **MSE**، حالت خاصی که در ابتدا توضیح دادیم را که در آن **PSNR** بینهایت میشد را مدیریت کردیم. بعد از این مرحله **PSNR** را محاسبه کردیم فقط فرقی که در این قسمت دارد این است که فرمول محاسبه **PSNR** را کمی ساده کردیم یعنی توان ۲ بر روی MAX_I بیرون آوردیم و در ۱۰ پشت آن ضرب کردیم و به شکلی که در کد آمده است تبدیل شده است. در نهایت خروجی را برگرداندیم.

```python
if original_image is not None:
    f_transform = np.fft.fft2(original_image)
    f_transform_shifted = np.fft.fftshift(f_transform)

    magnitude_spectrum_original = 20 * np.log(np.abs(f_transform_shifted) + 1e-9)
    display_image(magnitude_spectrum_original, title="Magnitude Spectrum of Original Image", cmap='gray')

    radii = [20, 60, 120]

    results = []

    print("\n--- Processing with different mask radii ---")
    for i, radius in enumerate(radii):···
```

2.0s Python

از آنجایی که این بخش طولانی بود فقط قسمت اول کد را عکس گرفتیم و بقیه کد در فایل موجود است اما کد را به صورت کامل توضیح دادیم و توضیح کامل کد در ادامه قرار دارد.

در این قسمت ماسک طراحی شده را در دامنه فرکانس اعمال کردیم و تصویر را بازسازی کردیم. همچنین چون از تبدیل فوریه استفاده کردیم و محتویات فرکانسی را نیز نشان دادیم، هر **Magnitude Spectrum** این کار را انجام دادیم است که فقط قسمت **magnitude** هر عدد مختلط را از خروجی تبدیل فوریه گرفته است و نشان داده است که همین، مقدار هر پیکسل را میسازد.

در ابتدا با کمک **f_transform** یک تبدیل دو بعدی فوریه زدیم و از دامنه زمان–مکان به دامنه فرکانسی رفتیم. سپس از تبدیل فوریه شیفت یافته استفاده کردیم.

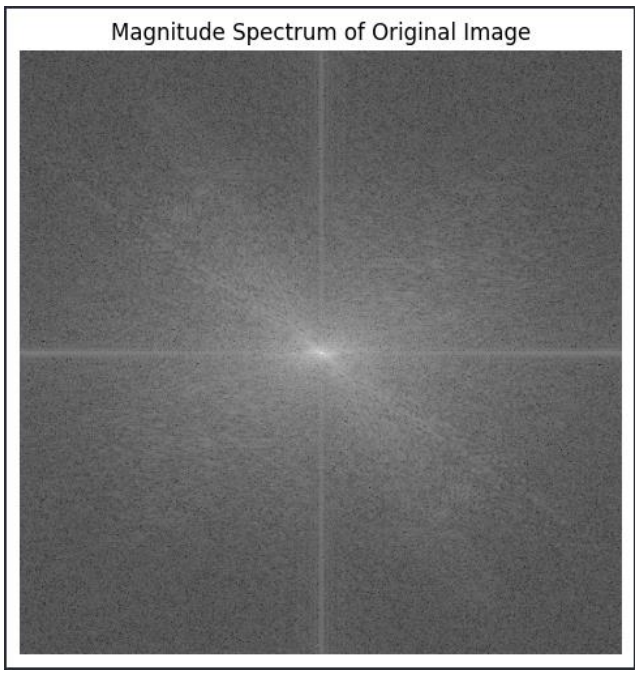
بعد با کمک **radii** شعاع های مختلف را برای فیلتر پایین گذر خورد تعریف کردیم. هر چه قدر شعاع کوچک تر باشد فقط مولفه‌های فرکانس پایین را نگه میدارد، شعاع‌های بالاتر جزئیات و مولفه‌های فرکانس بالاتری را نگه میدارند.

در ادامه با کمک **f_transform_shifted_mask** فیلتر پایین گذر خود را اعمال کردیم به این صورت که به صورت **element wise** تبدیل **FFT** را با ماسک ضرب کردیم. تا فقط مولفه‌های پایین گذر را نگه داریم.

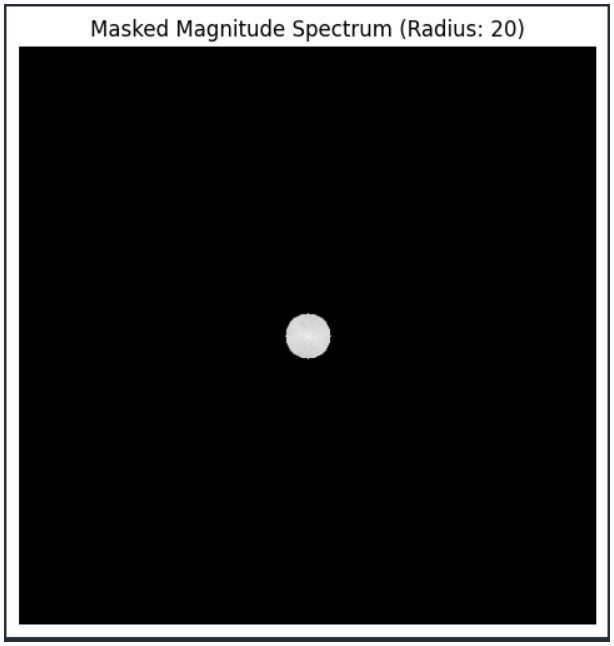
در نهایت با کمک **f_ishift** به نسخه معمولی فوریه برگرشتیم و تبدیل معکس **FFT** را زدیم تا عکس را بازسازی کنیم و از دامنه فرکانسی به دامنه زمان–مکان برویم. قدر مطلق نیز گرفتیم بخاطر وجود عدد مختلط هنگام استفاده از تبدیل فوریه.

در نهایت با کمک **psnr_value** میزان **PSNR** را محاسبه کردیم و گزارش کردیم.

در نهایت برای هر ۳ شعاع خواسته شده تصویر را نشان دادیم و گزارش **PSNR** را تحویل دادیم که در ادامه خواهد آمد:

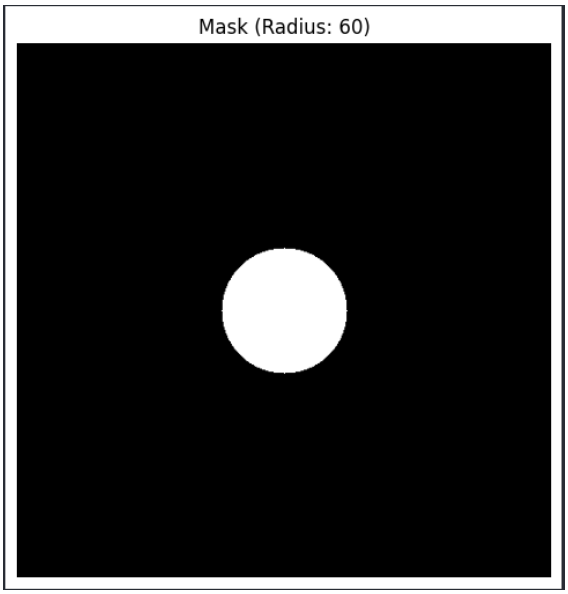


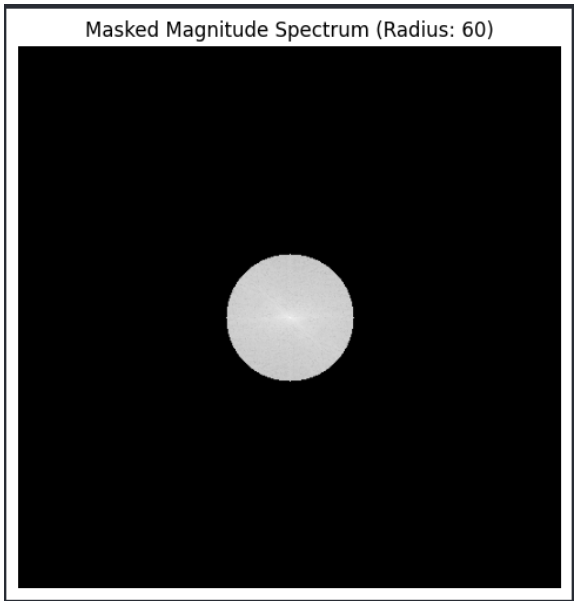
ابتدا تصویر **Magnitude Spectrum** را که توضیح دادیم نمایش دادیم.



در ابتدا ماسک را با شعاع ۲۰ اعمال کردیم و نتیجه تصویر بازسازی شده آن در ادامه خواهد آمد که در زیر آن مقدار **PSNR** را نیز گزارش کردیم:

تصویر بازسازی شده با ماسک با شعاعی ۲۰ به همراه گزارش PSNR که مقدار 22.82 dB را دارد.

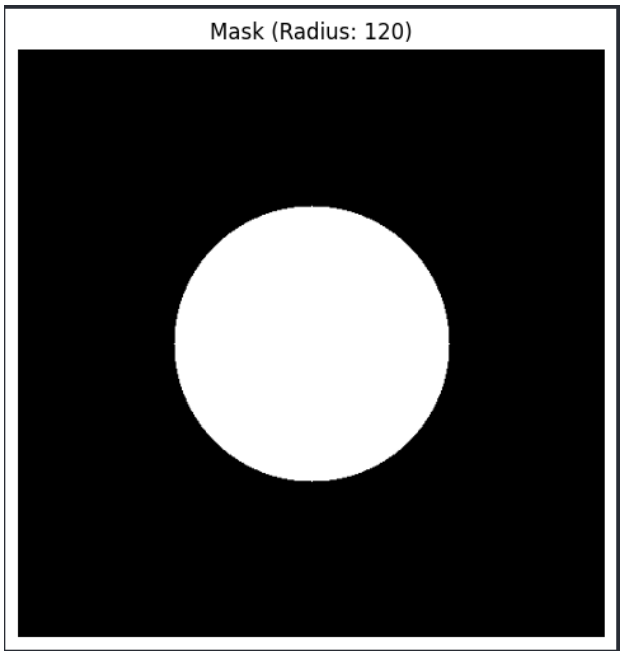


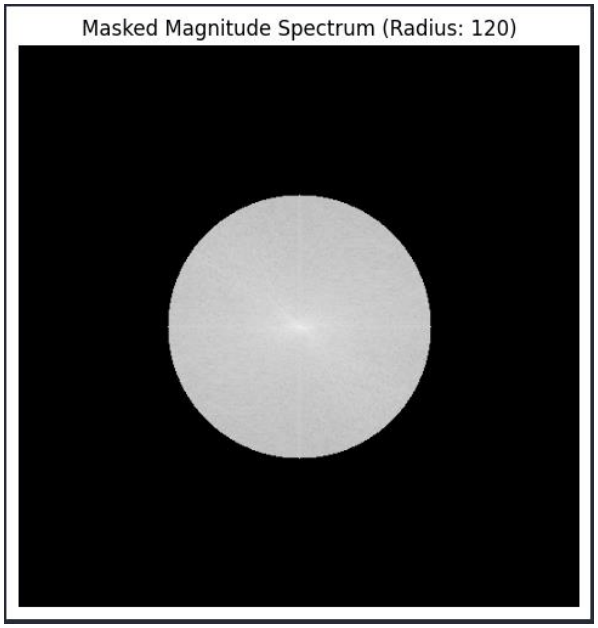


در مرحله بعد از ماسکی با شعاع ٦٠ کمک گرفتیم که نتایج آن در ادامه قرار دارد:

عکس بدست آمده با ماسکی با شعاع ۶۰ درجه و گزارش PSNR آن برابر با 28.38 dB است

همانطور که مشخص است از لحاظ بصری تصویر بهتری گرفتیم با افزایش شعاع.





ماسک با شعاع ۱۲۰ درجه اعمال کردیم و نتایج آن در ادامه خواهد آمد:

همانطور که مشخص است با اعمال ماسک با شعاع ۱۲۰ درجه مولفه های دیگری نیز عبور کرده اند و تصویر خیلی با جزئیات تر شده است و از لحاظ بصری بهتر است همچنین PSNR آن نیز افزایش قابل توجهی پیدا کرده است و به مقدار 33.73 Db رسیده است.

تحلیل

همانطور که توضیح دادیم و در تصاویر خروجی مشخص است، شعاع ماسک با میزان تار شدگی رابطه مستقیمی دارد، هر چه قدر شعاع کوچک تر باشد، سهم کمتری از مولفه‌های فرکانس پایین که در نزدیکی مرکز spectrum هستند اجازه عبور میکنند. این امر باعث میشود که ما به طور قابل توجهی مولفه‌های فرکانس بالا و جزئیات را از دست بدهیم مانند texture و لبه و جزئیات و همین باعث میشود که تصویر بازسازی شده کمتر sharp بنظر برسد و بسیار تار بشود. هنگامی که از شعاع متوسط استفاده میکنیم مثلا ٦٠، اوضاع کمی بهتر میشود شعاع افزایش پیدا میکند و مولفه های بیشتری اجازه عبور پیدا میکنند و تصویر کمتر تار میشود و جزئیات کمی قابل مشاهده میشوند. وقتی از شعاع بزرگی استفاده میکنیم مثلا ۱۲۰، بخش قابل توجهی از spectrum عبور پیدا ممیکند و تصویر بازسازی شده به تصویر اصلی نزدیک میشود. ما sharpness بهتری خواهیم داشت و جزئیات بیشتری اضافه میشوند.

معیار PSNR نیز را همین را به ما خواهد گفت در شعاع های پایین میبینیم که میزان PSNR نیز کم است که بیانگر کیفیت پایین تصویر است و هر چه میزان شعاع بزرگتر میشود میزان PSNR نیز افزایش پیدا میکند.

بخش ب

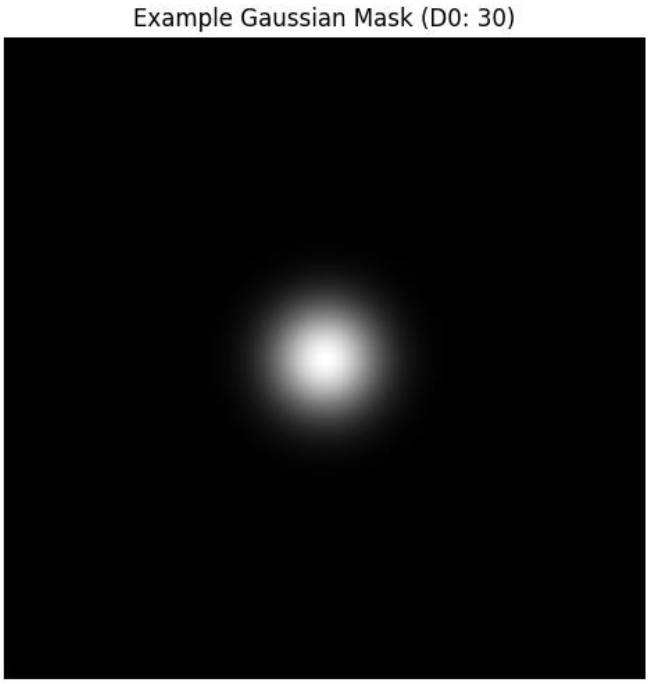
در این بخش ما یک فیلتر پایین گذر گوسی پیاده‌سازی و سپس روی تصویر اعمال میکنیم. توجه شود که بخش‌های بارگذاری تصویو، قسمت تبدیل فوریه و محاسبه **PSNR** مانند بخش الف بوده و از توضیح این موارد در این بخش صرف نظر میکنیم. در این قسمت میدانیم شعاع همان سیگما خواهد بود که آن را D0 نشان میدهیم. هدف فیلتر گوسی با فیلتر ایده آل یکی است هر ۲ فیلتر پایین گذر هستند ولی به جای اینکه مانند فیلتر ایده آل باینری عمل کند یعنی مولفه‌هایی زیر **cutoff** هستند عبور کنند (ضربدر ۱ شوند) و مولفه‌هایی که بالای **cutoff** هستند عبور نکنند (در ۰ ضرب شوند)، رویکرد آرام هموارتر و نرم تری را در پیش میگیرد و به آرامی عمل میکند. به عبارت دیگر پایین ترین فرکانس ها را با قدرت عبور میدهد (که دقیقا در مرکز **spectrum** قرار دارند یا در نوک تابع گوسی هستند) و هر چه قدر که به مقادیر فرکانس بالاتر حرکت میکنیم، به آرامی در مقادیر کمتری از ۱ ضرب میکند و پیوسته ضریب را کاهش میدهد هر چه قدر که به مقادیر فرکانس بالاتر نزدیک تر میشویم. همین جا هست که فرق بین فیلتر **gaussian** و ایده آل مشخص میشود زیرا فیلتر ایده آل بخاطر این تغییرات سریع دچار اثر **“ringing artifacts”** میشود.

گزارش کار

```python
def create_gaussian_mask(h, w, center_x, center_y, D0):
    u = np.arange(w) - center_x
    v = np.arange(h) - center_y

    U, V = np.meshgrid(u, v)

    D_sq = U**2 + V**2

    mask = np.exp(-D_sq / (2 * (D0**2)))
    return mask

if 'original_image_gaussian_nb' in locals() and original_image_gaussian_nb is not None:
    rows, cols = original_image_gaussian_nb.shape
    center_x_display, center_y_display = cols // 2, rows // 2
    example_D0_display = 30
    example_gaussian_mask_display = create_gaussian_mask(rows, cols, center_x_display, center_y_display, example_D0_display)
    display_image(example_gaussian_mask_display, title=f"Example Gaussian Mask (D0: {example_D0_display})", cmap='gray')
    print("create_gaussian_mask function defined and example shown.")
```

0.1s Python

در این قسمت میخواهیم یک فیلتر پایین گذر ۲ بعدی گوسی بسازیم که در دامنه فرکانس مورد استفاده قرار بگیرد. ابتدا یک سری **coordinate** تعیین میکنیم. سپس مربع فاصله را محاسبه میکنیم با کمک **D_sq** که فاصله هر نقطه در دامنه فرکانسی را تا مرکز محاسبه میکند. با کمک متغیر **mask**، تابع **guassian** را میسازیم که مقادیر نزدیک به ۱ بیانگر نزدیکی به مرکز خواهند بود و مقادیر از ۱ به آرامی کم میشود و به سمت صفر حرکت میکند. فرمول اعمال شده در کد مشخص است.در نهایت برای درک بهتر خروجی تهیه شده را نمایش میدهیم:

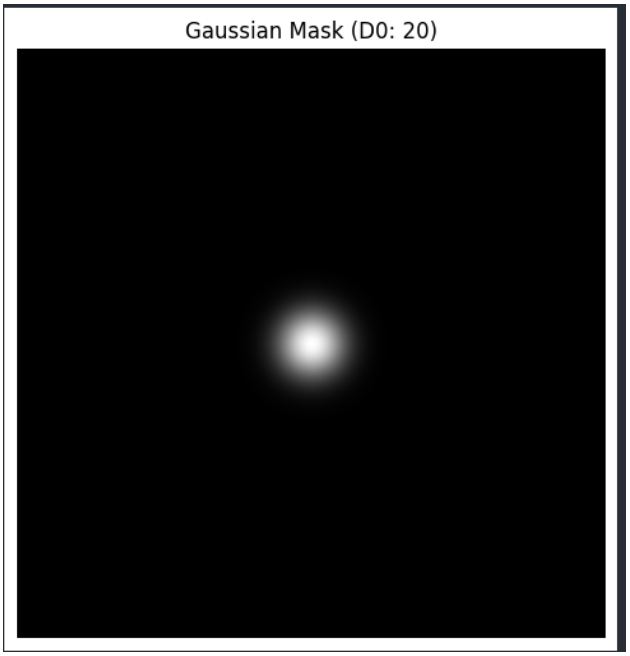


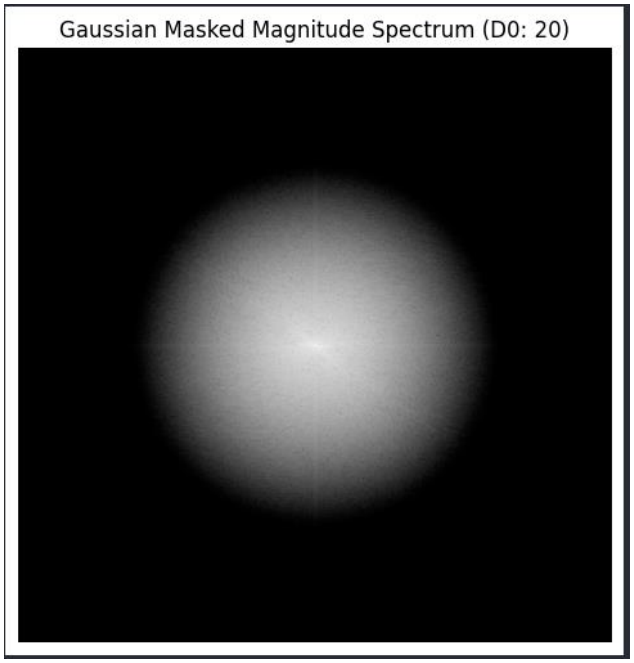
تصویر ماسک گوسی طراحی شده است.

```python
if 'original_image_gaussian_nb' in locals() and original_image_gaussian_nb is not None:

    f_transform_gaussian_nb = np.fft.fft2(original_image_gaussian_nb)
    f_transform_shifted_gaussian_nb = np.fft.fftshift(f_transform_gaussian_nb)

    magnitude_spectrum_original_gaussian_nb = 20 * np.log(np.abs(f_transform_shifted_gaussian_nb) + 1e-9)
    display_image(magnitude_spectrum_original_gaussian_nb, title="Magnitude Spectrum of Original Image", cmap='gray')

    D0_values_gaussian_nb = [20, 60, 120]

    gaussian_results_notebook2 = []

    print("\n--- Processing with different Gaussian mask D0 values ---")
    for i, D0_val in enumerate(D0_values_gaussian_nb):···
```

✓ 2.8s Python

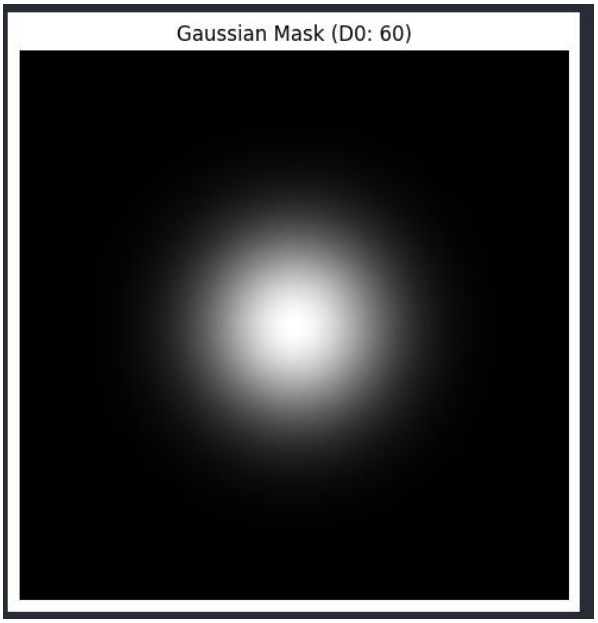
در این قسمت ماسک گوسی را روی تصویر خواسته شده اعمال میکنیم و سایر مراحل مانند قبل است. همچنین همینطور که در بخش الف نیز توضیح دادیم **Magnitude Spectrum** را نیز نمایش میدهیم. در این بخش فقط مقادیر **D0** که همان شعاع خواهد بود تغییر خواهد کرد و برای مقادیر مختلف خروجی را بررسی میکنیم و برای هر کدام **PSNR** را گزارش میدهیم:

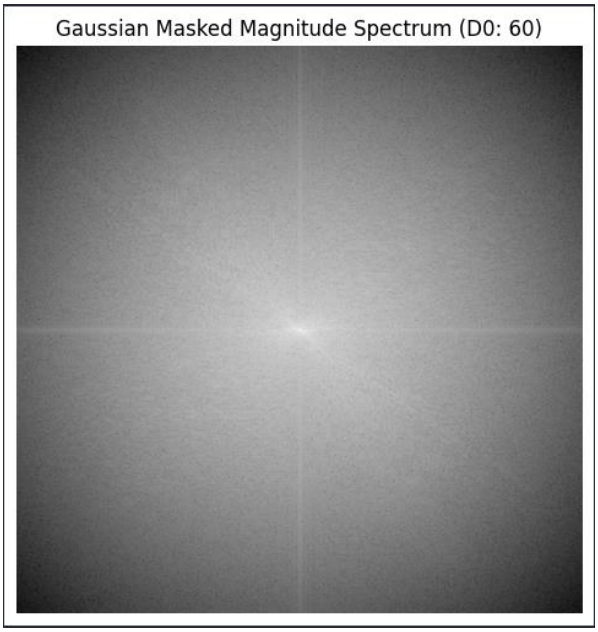




ابتدا از شعاع ۲۰ استفاده میکنیم که خروجی آن به شکل زیر خواهد بود:

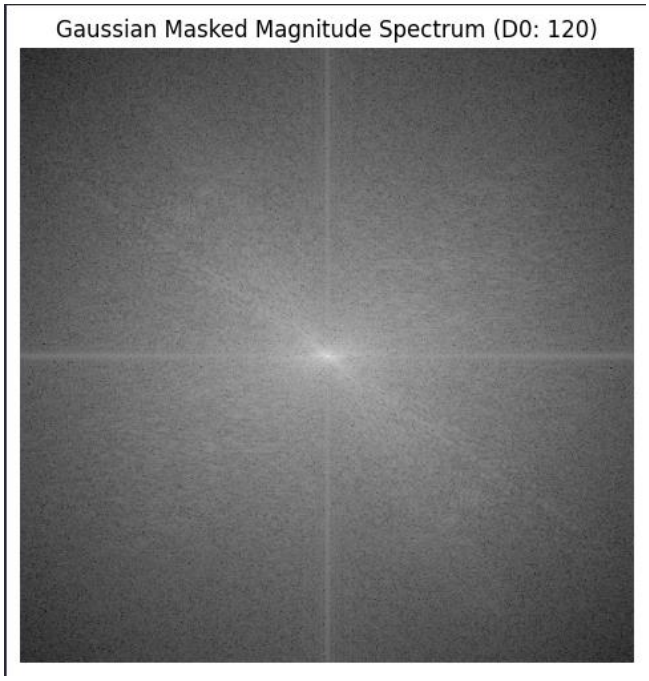
تصویر بازسازی شده با شعاع ۲۰ قابل مشاهده است صرفا ساختار تصویر حفظ شده است و تصویر به شدت تار است و لبه ها مشخص نیستند و جزئیات به سختی قابل مشاهده و تشخیص است. همچنین PSNR گزارش شده برای آن 24.41 است که عدد پایینی است.





سپس همین کار را با شعاع ۶۰ انجام میدهیم و تصویر را بازسازی میکنیم. نتیجه:

تصویر بازسازی به خوبی قابل مشاهده است. ساختار حفظ شده است، تاری کمتر شده است، لبه ها و جزئیات قابل مشاهده تر شده اند و همچنین PSNR بالاتری دریافت کرده‌ایم.



این بار با شعاع ۱۲۰ اعمال کردیم و تصویر را بازسازی کردیم. نتیجه:

تصویر بازسازی شده کیفیت بالایی دارد. لبه ها و جزئیات در آن قابل به راحتی قابل مشاهده است. تاری تصویر کم شده است. گزارش **PSNR** نیز عدد بالایی را گزارش کرده است.

تحلیل

قابل مشاهده است که هر چه قدر مقدار **D0** که شعاع ما است کوچک تر باشد، ماسک گوسی ما محدود تر و باریک تر میشود در دامنه فرکانس. یعنی مقادیر که دورتر از **DC** یا مرکز هستند را با شدت رد میکند در حالی که مولفه های فرکانس پایین را عبور میدهد. بخش قابل توجهی از مولفه های فرکانس بالا عبور نمیکنند. تصویر بازسازی شده کاملا تار شده است مانند فیلتر ایده آل با همین مقدار. با افزایش مقدار **D0** تصویر بهتر و با کیفیت تر میشود و تاری تصویر کم میشود و جزئیات و لبه ها مشخص تر میشوند و از نرم و هموار شدن لبه ها کاسته میشود زیرا که مولفه های فرکانس بالا نیز عبور میکنند آنهایی که از مرکز گوسی فاصله دارند. معیار **PSNR** نیز همین را

میگوید برای شعاع های کوچک تر، عدد پایین تری گزارش میکند که یعنی کیفیت آن پایین تر هست و هر چه قدر شعاع بالا میرود عدد گزارش شده نیز بالا میرود و رابطه بین مستقیمی بین شعاع و افزایش عدد **PSNR** وجود دارد.

بخش ج

نتایج الف و ب را از لحاظ تصویر بازسازی شده، ماسک هر کدام در دامنه فرکانس و مقدار **PSNR** بررسی کرد.

از لحاظ تصویر بازسازی شده: وقتی فیلتر ایده آل را اعمال میکنیم به سبب برخورد ناگهانی و سریع با مقادیر همانطور که گفتیم دچار **ringing artifacts** میشویم در تصویر بازسازی شده. در شعاع های کوچک تر میزان تار شدگی بسیار زیاد تر است نسبت به فیلتر گوسی. در مقابل در فیلتر گوسی چون به آرامی با مقادیر برخورد میکند دچار **ringing artifacts** کمتری میشوند یا نمیشوند. همچنین میزان تار شدگی کمتر است نسبت به فیلتر ایده آل در مقادیر مشابه.

از لحاظ ماسک در دامنه فرکانس: ماسک فیلتر ایده ال بسیار تیز هست در دامنه فرکانس، که داخل آن ۱ و بیرون آن صفر است. اما در فیلتر گوسی این مقدار به آرامی تغییر میکند و تیز نیست.

از لحاظ مقدار PSNR: از لحاظ سطوح تار شدگی و مقدار **PSNR**، در شعاع های یکسان، مقادیر اندکی متفاوت است. فیلتر گوسی در شعاع یکسان عدد بالاتری از لحاظ **PSNR** دریافت میکنند در حالی که فیلتر ایده آل دچار **ringing artifact** میشود و **MSE** تاثیر خود را میگذارد و مقادیر در فیلتر ایده آل پایین تر میاد.

به طور کلی از آنجایی که وقتی تبدیل فوریه از تابعی که فیلتر ایده آل دارد میگیریم حضور **SINC** باعث میشود که این اثر **ringing artifact** ظاهر شود زیرا **SINC** یک سری **ripple** در اطراف حلقه اصلی دارد و از آنجایی که فیلتر کردن در دامنه فرکانس با کانولوشن در دامنه زمان–مکان برابر است، **convolve** کردن تصویر در دامنه زمان–مکان به سبب همان **ripple**ها دچار **ringing artifact** در این دامنه میشود مخصوصا در نزدیک جاهایی که لبه داریم. اگر تغییر به صورت آرام صورت بگیرد مانند فیلتر گوسی این مشکل حل خواهد شد.

سوال پنجم

این سوال در ٤ بخش مختلف انجام شده است و گزارش کار کد هر بخش در همان بخش قرار دارد. بخش‌های پیاده‌سازی شده به طور کلی در یک فایل تحت عنوان **05.ipynb** قرار دارند. تبدیل **DCT** نوعی خاص از تبدیل فوریه گسسته هست که در آن فقط سیگنال‌های زوج وجود دارند و یک سیگنال را به صورت جمع یک سری توابع کسینوسی نمایش میدهد. این تبدیل به خاطر اینکه عمده انرژی را در یک گوشه (معمولا در قسمت بالا سمت چپ) متمرکز میکند بسیار در فشرده سازی مفید است.

بخش الف

در این بخش ما ماسک خواسته شده را پیاده‌سازی کردیم و تصویر **mandril.tiff** در ورودی خواندیم و تبدیل **DCT** را روی آن اعمال کردیم. عدد **PSNR** و زمان اجرا در انتها تهیه شده است و خروجی بررسی میشود. توضیح کد در قسمت گزارش کار قرار دارد.

گزارش کار

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import dct, idct
from skimage.metrics import peak_signal_noise_ratio
import time
import os
```

کتابخانه‌های مورد نیاز خود را در این بخش وارد کردیم از **time** برای سنجیدن و گزارش زمان اجرا استفاده کردیم. در این کد برای گزارش **PSNR** به جای پیاده‌سازی از کتابخانه **skimage** برای این کار کمک گرفتیم. از **scipy** استفاده کردیم برای وقتی که میخواهیم تبدیل **DCT** را به صورت کاربرد ۲ بعدی اعمال کنیم. از **matplotlib** نیز برای نمایش خروجی استفاده کردیم.

```python
def display_image(image, title="Image", cmap=None):
    plt.figure(figsize=(6, 6))
    if cmap:
        plt.imshow(image, cmap=cmap)
    else:
        plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB) if len(image.shape) == 3 and image.shape[2] == 3 else image, cmap=
    plt.title(title)
    plt.axis('off')
    plt.show()
```

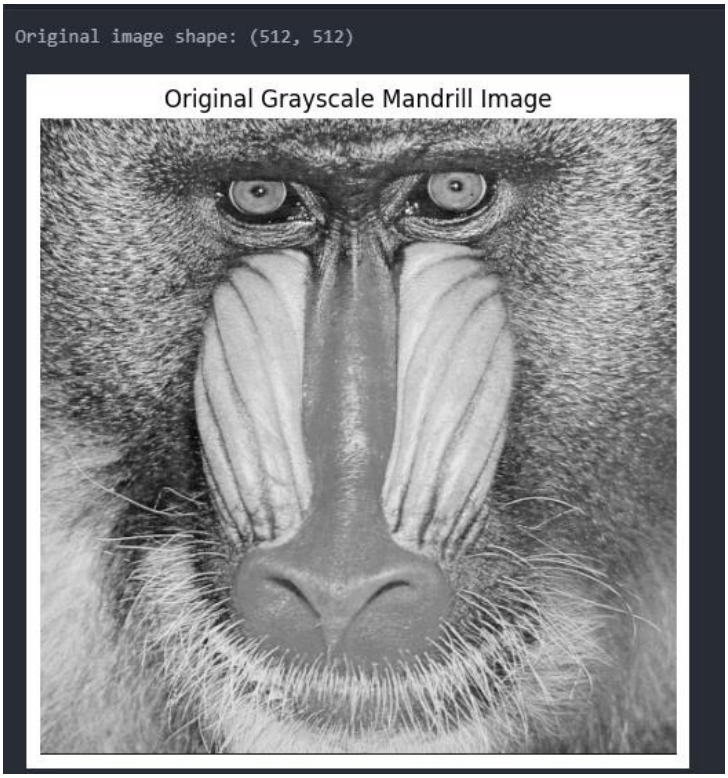
مانند سوال قبل تابعی برای نمایش تصاویر تعریف کردیم.

```python
def dct2(block):
    return dct(dct(block.T, norm='ortho', axis=0).T, norm='ortho', axis=0)

def idct2(block):
    return idct(idct(block.T, norm='ortho', axis=0).T, norm='ortho', axis=0)
```

۲ تابع برای اعمال ۲ بعدی تبدیل **DCT** و **IDCT** تعریف کردیم با کمک کتابخانه **scipy.fftpack**. این ۲ تابع ابتدا بر روی سطرها و بعد روی ستون‌ها اعمال میشوند. قسمت **"ortho"** برای حفظ انرژی و معکوس پذیری آن استفاده شده است.

```python
image_path = "mandril.tiff"

if not os.path.exists(image_path):
    print(f"Error in reading image")
    original_image_color = None
else:
    original_image_color = cv2.imread(image_path)

if original_image_color is None:
    print(f"Failed to load image")
else:
    original_image_gray = cv2.cvtColor(original_image_color, cv2.COLOR_BGR2GRAY)
    original_image_float = original_image_gray.astype(np.float32)

    print(f"Original image shape: {original_image_gray.shape}")
    display_image(original_image_gray, title="Original Grayscale Mandrill Image")
```

عکس داده شده در صورت سوال را در که در **directory** یکسان قرار دارند با کمک کتابخانه **open cv** خواندیم و بعد از آن به یک عکس **grayscale** یا خاکستری تبدیل کردیم زیرا تبدیل **DCT** معمولا بر روی این تصاویر که یک کانال هستند اعمال میشود و بهتر است تصاویر را از رنگی به خاکستری تبدیل کنیم. بعد از آن با کمک **original_image_float** تصویر را به **float32** برای اعمال **DCT** تبدیل کردیم. در نهایت خروجی را نمایش دادیم:

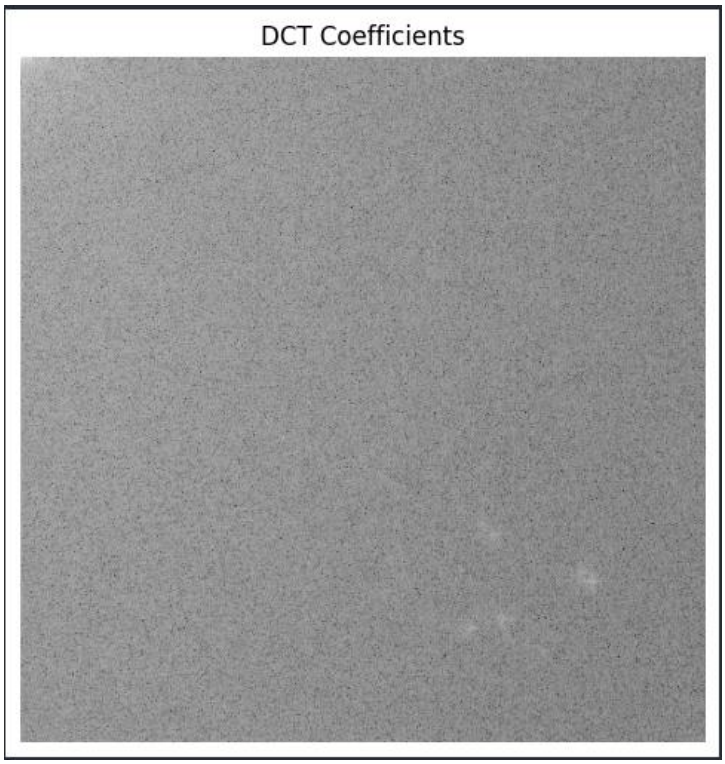


تصویری که حاصل از انجام عملیات های گفته شده تهیه شده است.

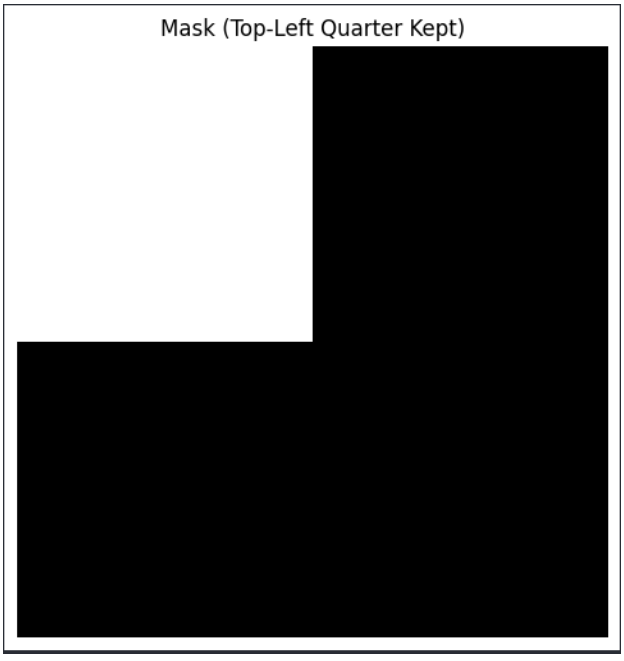
```python
start_time = time.time()
dct_coeffs = dct2(original_image_float)

dct_coeffs_log_scaled = np.log(np.abs(dct_coeffs) + 1e-9)
display_image(dct_coeffs_log_scaled, title="DCT Coefficients")

rows, cols = dct_coeffs.shape
mask = np.zeros_like(dct_coeffs, dtype=np.float32)

mask_rows_quarter = rows // 2
mask_cols_quarter = cols // 2

mask[0:mask_rows_quarter, 0:mask_cols_quarter] = 1.0

display_image(mask, title="Mask (Top-Left Quarter Kept)")

masked_dct_coeffs = dct_coeffs * mask

masked_dct_coeffs_log_scaled = np.log(np.abs(masked_dct_coeffs) + 1e-9)
display_image(masked_dct_coeffs_log_scaled, title="Masked DCT Coefficients (Log Scaled)")
```

Python

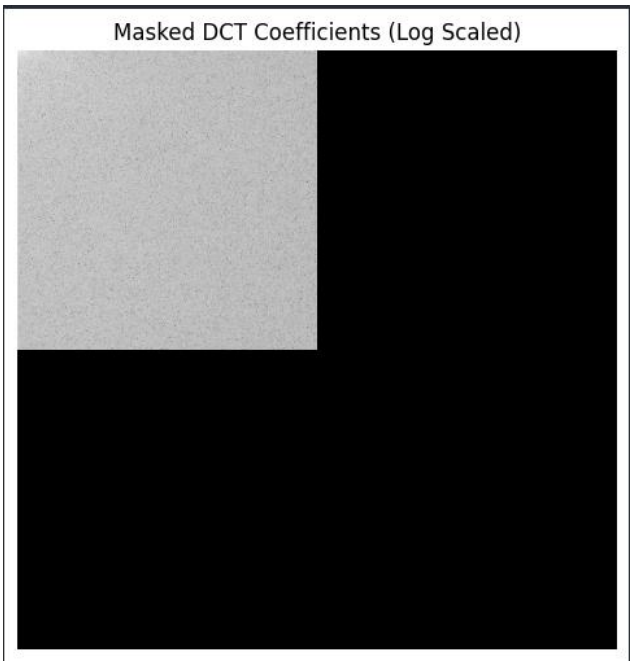
در این قسمت **DCT** را اعمال کردیم، ماسک را ساختیم و آن را نیز اعمال کردیم از **time** هم برای سنجیدن مقدار صرف شده زمان برای گزارش استفاده کردیم. با کمک تابعی که در بالا تعریف کردیم تبدیل ۲ بعدی **DCT** را اعمال کردیم. صرفا برای نمایش خروجی که خروجی بهتری داشته باشیم با یک **log** تابع را **scale** کردیم تا از لحاظ بصری بهتر باشد. اینکار صرفا برای خروجی بهتر از لحاظ بصری انجام شده و تاثیری در روند اعمال تبدیل ما نداشته است. حاصل اینکار در ادامه تحت عنوان **DCT Coefficients** نمایش داده شده است. در ادامه با کمک **mask** ، ماسک خواسته شده را ساختیم. ماسک خواسته شده و در گوشه بالا سمت چپ قسمت سفید دارد و مقدار ۱ میگیرد و در بقیه بخش ها مقدار ۰ میگیرد یا همان سیاه است. در واقع ماسک میگوید که فقط قسمت **low frequency** ها را نگه دار. نمایش خروجی گفته شده در ادامه خواهد آمد:



تصویر **DCT coefficients** که برای نمایش بهتر با یک **log**، **scale** شده است.



ماسک طراحی شده که در صورت سوال آمده بود.

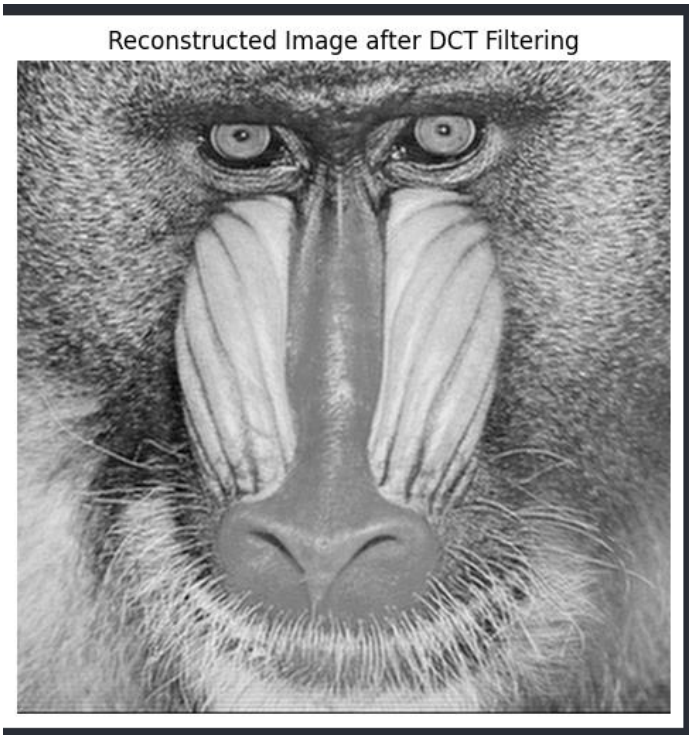


اعمال ماسک طراحی شده بر روی تبدیل **DCT** که برای نمایش بهتر با یک **log**، **scale** شده است.

```python
reconstructed_image_float = idct2(masked_dct_coeffs)
reconstructed_image = np.clip(reconstructed_image_float, 0, 255)
reconstructed_image = reconstructed_image.astype(np.uint8)
end_time = time.time()
execution_time = end_time - start_time
display_image(reconstructed_image, title="Reconstructed Image after DCT Filtering")
```

در این قسمت با کمک تابع کمکی که طراحی کرده بودیم معکوس تبدیل را انجام میدهیم تا به دامنه **spatial** برگردیم و بتوانیم عکس را بازسازی کنیم. تبدیل معکوس نیز به صورت ۲ بعدی خواهد بود. در مرحله بعد مقادیر را **clip** کردیم و به **unit8** تبدیل کردیم. مقدار **clip** کردن برای مواقعی هست که مقادیر از بازه **[0, 255]** خارج شوند. بعد از آن تایمر را قطع میکنیم و زمان را در نظر میگیریم. بعد با کمک تابع کمکی که در بالاتر تعریف کردیم تصویر خروجی را نمایش میدهیم:

خروجی



تصویر بازسازی شده بعد از اعمال **DCT**.

```python
if original_image_color is not None:

    psnr_value = peak_signal_noise_ratio(original_image_gray, reconstructed_image, data_range=255)

    print(f"\n--- Results ---")
    print(f"Original Image Dimensions: {original_image_gray.shape}")
    print(f"Reconstructed Image Dimensions: {reconstructed_image.shape}")
    print(f"Mask kept top-left {mask_rows_quarter}x{mask_cols_quarter} coefficients.")
    print(f"PSNR: {psnr_value:.2f} dB")
    print(f"Total Execution Time (DCT, Masking, IDCT): {execution_time:.4f} seconds")
```

Python

در این قسمت موارد خواسته شده را گزارش میکنیم مانند زمان اجرا و مقدار **PSNR**. که خروجی آن به صورت زیر خواهد بود:

```
...
    --- Results ---
    Original Image Dimensions: (512, 512)
    Reconstructed Image Dimensions: (512, 512)
    Mask kept top-left 256x256 coefficients.
    PSNR: 24.53 dB
    Total Execution Time (DCT, Masking, IDCT): 3.7148 seconds
```

تصویر بازسازی شده نمره **24.53** از معیار **PSNR** گرفت و زمان اجرا ۳.۷۱ ثانیه را ثبت کرد.

قابل اشاره است که از آنجا که **DCT** قسمت عمده انرژی را در گوشه بالا سمت چپ فشرده میکند و ماسک نیز همان قسمت را اجازه عبور میدهد در نتیجه تصویر بازسازی شده هر چند که اطلاعاتی نظیر بعضی جزئیات را از دست داده است، اما توانسته ظاهر خوبی داشته باشد.

بخش ب

در این قسمت همانطور که در صورت سوال خواسته شده بود تصویر را به بلوک‌هایی با اندازه‌های گفته شده تقسیم کردیم و تبدیل **DCT** را روی هر بلوک اعمال کردیم. گزارش کار مربوط به کد در ادامه همین بخش آمده است. نتایج **PSNR** و زمان اجرا را در ادامه گزارش آوردیم و تحلیل رابطه آن با سایز بلوک در ادامه همین بخش وجود دارد.

گزارش کار

در همان فایل تحت عنوان **PART B** بخش ب شروع میشود.

```python
def process_image_blockwise(image_to_process_float, original_img_gray_for_psnr, block_size_N):
    original_rows, original_cols = image_to_process_float.shape

    pad_rows_bottom = (block_size_N - original_rows % block_size_N) % block_size_N
    pad_cols_right = (block_size_N - original_cols % block_size_N) % block_size_N

    padded_image = np.pad(image_to_process_float, ...

    padded_rows, padded_cols = padded_image.shape
    reconstructed_padded_image = np.zeros_like(padded_image, dtype=np.float32)

    mask_block = np.zeros((block_size_N, block_size_N), dtype=np.float32)
    keep_dim = block_size_N // 2
    mask_block[0:keep_dim, 0:keep_dim] = 1.0


    start_time_proc = time.time()

    for r_idx in range(0, padded_rows, block_size_N):...

    end_time_proc = time.time()
    execution_time_proc = end_time_proc - start_time_proc

    reconstructed_image_cropped_float = reconstructed_padded_image[0:original_rows, 0:original_cols]

    reconstructed_image_uint8 = np.clip(reconstructed_image_cropped_float, 0, 255).astype(np.uint8)

    psnr_value = peak_signal_noise_ratio(original_img_gray_for_psnr, reconstructed_image_uint8, data_range=255)
```

تابعی که تعریف کردیم تحت عنوان **PROCESS_IMAGE_BLOCKWISE** میاد و تصویر را به بلوک هایی با سایز خواسته شده تقسیم میکند بعد از آن تبدیل **DCT** را میزنیم و بعد ماسک را اعمال میزنیم و بعد از آن تبدیل معکوس میگیریم.

ابتدا به تصویر **padding** اعمال کردیم تا ابعاد آن به طور کامل قابل تقسیم برای سایزهای خواسته شده باشد. در قسمت **padded_image** از **reflect** استفاده کردیم تا اثر **edge artifacts** کمینه شود.

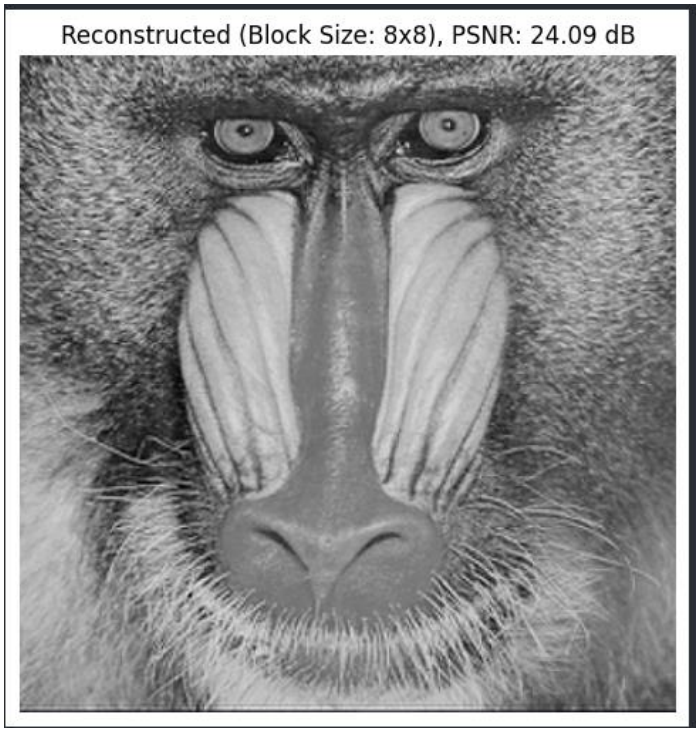
در مرحله بعد با کمک **mask_block** یک ماسک با سایز **N*N** تعریف کردیم. چون هر کدام **N/2** میشوند در نهایت ¼ ضریب‌ها باقی میماند برای هدف خواسته شده در صورت سوال. بعد از آن تایمر را شروع میکنیم، با یک **for** بر روی تصویر حرکت میکنیم و بلوک‌ها را اعمال میکنیم. با **current_block** یک بلوک را استخراج میکنیم و بعد روی آن با کمک **block_dct_coeffs** تبدیل **DCT** را روی آن بلوک اعمال میکنیم. بعد از آن با کمک

MASKED_BLOCK_COEFFS ماسک را اعمال میکنیم. در نهایت تبدیل معکوس را کمک تابع کمکی که بالاتر تعریف کرده بودیم میزنیم و بلوک بازسازی شده را داریم، در نهایت بلوک را به همان **position** قبلی برمیگردانیم و تایمر را قطع میکنیم. با کمک **reconstructed_image_cropped_float** تصویر را برای برگردانیم به ابعاد اولیه **crop** میکنیم. مقادیری که بین **[0, 255]** نیستند را **clip** میکنیم و بعد به **unit8** تبدیل میکنیم. در مرحله بعد **PSNR** را محاسبه میکنیم و نتیجه را برمیگردانیم.

```
if 'original_image_gray' not in globals() or original_image_gray is None:
    print("ERROR in image")
else:

    original_image_float_for_processing = original_image_gray.astype(np.float32)

    block_sizes_to_test = [8, 16, 64]
    block_processing_results = {}


    for N in block_sizes_to_test:
        print(f"--- Processing with block size: {N}x{N} ---")

        reconstructed_img, exec_time, psnr = process_image_blockwise(
            original_image_float_for_processing,
            original_image_gray,
            N
        )

        block_processing_results[N] = {
            'psnr': psnr,
            'time': exec_time,
            'image': reconstructed_img
        }
```

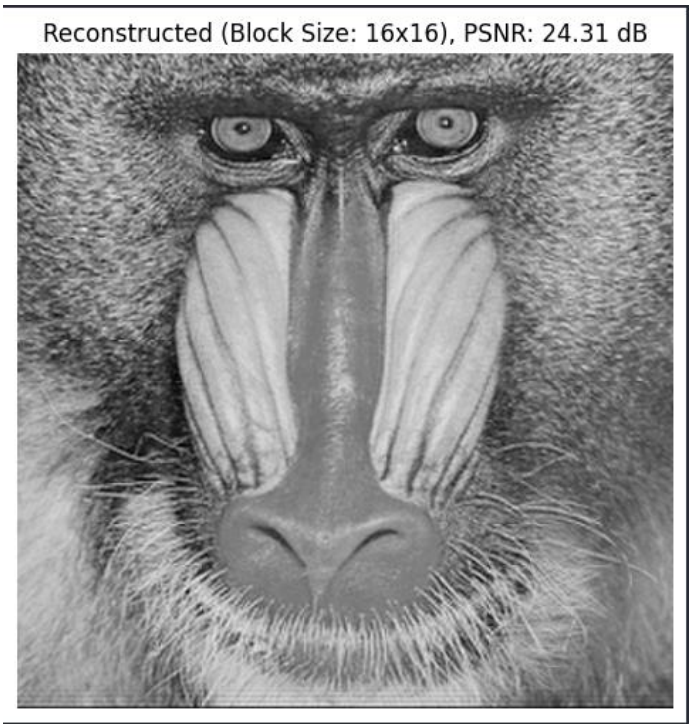
در قسمت بعدی سایزهای خواسته شده را مینویسیم و تابع را فراخوانی میکنیم و نتایج و موارد قابل گزارش مانند **PSNR** و زمان و عکس بازسازی شده را برای هر سایز بلوک ذخیره میکنیم و نمایش میدهیم که در ادامه خواهد آمد:

خروجی



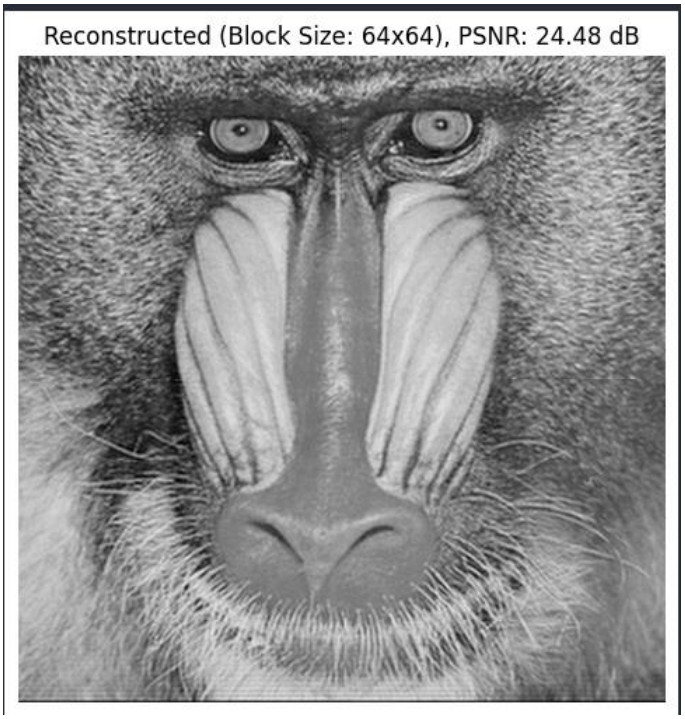

```
Block Size: 8x8
PSNR: 24.09 dB
Execution Time: 0.2414 seconds
```

تصویر بازسازی شده با سایز بلوک ۸ در ۸ که **PSNR** آن برابر با **24.09** و زمان اجرای آن **0.24** ثانیه شده است.




```
Block Size: 16x16
PSNR: 24.31 dB
Execution Time: 0.0695 seconds
```

تصویر بازسازی شده با سایز بلوک‌های ۱۶ در ۱۷ که **PSNR** آن برابر با **24.31** شده است و **0.06** ثانیه طول کشیده است.




```
Block Size: 64x64
PSNR: 24.48 dB
Execution Time: 0.0080 seconds
------------------------------
```

نتیجه تصویر بازسازی شده با بلوک با سایز ٦٤در٦٤ که معیار **PSNR** آن برابر با **24.48** است.

و زمان اجرای آن **0.008** ثانیه شده است.

```
--- Analysis: Block Size vs. PSNR and Execution Time ---
============================================================
| Block Size | PSNR (dB) | Execution Time (s) |
|------------|-----------|--------------------|
| 8          | 24.09     | 0.2414             |
| 16         | 24.31     | 0.0695             |
| 64         | 24.48     | 0.0080             |
```

تحلیل

قابل مشاهده است که هر چه قدر سایز بلوک‌ها بالاتر میرود، مقدار PSNR نیز متناسب با آن افزایش پیدا میکند. این بخاطر این است که DCT بهتر میتواند انرژی را در ناحیه اشاره شده جمع آوری و فشرده کند. از آنجایی که ¼ از ضرایب را نگه میداریم، بلاک‌های بزرگتر مقدار اطلاعات low frequency بیشتری نگه داری میکند متناظر با ساختار بزرگتر.

هر چه قدر سایز بلاک‌ها کوچک تر میشود مقدار PSNR نیز کمتر میشود و ما شاهد اثری تحت عنوان "blocking artifact" خواهیم بود زیرا به اندازه کافی ضرایبی را نگه نداشتیم تا بتوانیم با کمک آن جزئیات را به طور آرام و هموار در مرزهای بلاک ها نشان بدهیم.

نسبت به تصویری که در بخش الف تهیه شده است و مقدار PSNR گزارش شده بخش الف که بلاک سازی نکرده بودیم عدد بهتری گرفته بودیم وقتی به یک میزان از ضرایب را نگه داشتیم نسبت به حالتی که به طور محلی بلاک بندی کرده بودیم. میتوان نتیجه گرفت که DCTهای سراسری بهتر انرژی را فشرده میکنند نسبت به بلاک بندی.

در رابطه با زمان اجرا نیز میتوان گفت وقتی تبدیل DCT با بلوک بندی داشتیم پیچیدگی زمانی ما برابر با O(N^2 LOG N) بوده است و زمانی که بلاک بندی نداشتیم زمان اجرای بیشتری صرف شده بود درحالتی که داشتیم به طور سراسری DCT میزدیم.

بخش ج

وقتی در بخش الف بر روی کل تصویر فیلتر اعمال کردیم فقط به یک عملیات بزرگ DCT و معکوس آن نیاز داشتیم برای کل تصویر. ولی وقتی در بخش ب به صورت محلی بلاک بندی کردیم بر اساس سایز هر بلاک ما Nدر N تبدیل DCT و معکوس آن را اعمال کردیم. که از لحاظ محاسباتی بخش ب سریعتر است (بر اساس پیچیدگی زمانی که در بخش الف از مرتبه M^2 LOG M است و در بخش ب از مرتبه N^2 LOG N و N LOG N کمی کوچک تر از M است زیرا N < M). هر چه قدر سایز بلاک کوچک تر باشد زمان اجرا نیز بیشتر میشود زیرا تعداد بلاک ها افزایش پیدا میکنند در صورتی که برای بلاک سایز های متوسط و بزرگ مثل ١٦ در ١٦ و ٦٤ در ٦٤ این برقرار نیست و این موارد سریعتر هستند و زمان اجرای کمتری دارند.

وقتی به صورت سراسری تبدیل زدیم توانستیم بهتر انرژی را فشرده بکنیم و در قسمت ضرایب low frequency وقتی که ¾ از ضرایب نگه داشتیم، انگار بخش عمده ای را نگه داشتیم. همین امر سبب میشد ما PSNR بالاتری دریافت بکنیم و از لحاظ بصری بهتر باشد. هیچ اثری از blocking artifact مشاهده نشد. اما وقتی به صورت محلی بلاک بندی کردیم و DCT را مستقل روی هر بلاک اعمال کردیم، روابط داخل هر بلاک را در نظر گرفتیم و انرژی درون هر بلاک فشرده میشد و نگه داشتن ¼ ضرایب فقط برای همان بلاک بود. در این حالت دچار اثر blocking artifact میشدیم زیرا هر بلاک به صورت مستقل فیلتر و تبدیل روش اعمال میشد و عدم پیوستگی بین مرزهای بلوک ها شکل میگرفت وقتی تصویر را بازسازی میکردیم، و از لحاظ بصری نیز این ظاهر میشد و باعث میشد کیفیت عکس پایین بیاد و PSNR نیز سقوط کند. DCT نمیتواند با این اثر در مرز بلوک‌ها برخورد کند و از بین ببرد. هر چه قدر سایز بلاک کوچک تر بود عدد PSNR نیز کوچک تر بود زیرا اثر blocking artifact بیشتر و

زیاد تر ظاهر میشد و حتی نگه داشتن ¼ ضرایب در حالت کافی نبود و روی جزئیات محلی تاثیر قابل مشاهده ای میگذاشت. با افزایش سایز بلوک مقدار PSNR نیز افزایش پیدا کرد زیرا بلاک های بزرگتر میشدند و روابط بزرگتری را شامل میشدند و تعداد بلاک ها نیز در هر بخش کاهش پیدا میکرد و اون اثر نیز کمتر میشد. به طور کلی بخش الف کیفیت بالاتری و PSNR بالاتری داشت.

اثر بلاک بندی از لحاظ محاسباتی خوب بود و باعث کارآمدی میشد همانطور که از لحاظ مرتبه زمانی توضیح دادیم. اگر چه دچار blocking artifact شدیم و فشرده سازی انرژی تصویر به طور سراسری خوب شکل نگرفت و باعث کاهش کیفیت عکس شد. بلاک بندی وفق پذیری خوبی دارد و ما در بلوک های مختلف بر اساس محتویات آنها میتوانیم از جداول مختلف استفاده بکنیم مانند کاری که JPEG انجام میدهد.

بخش د

این بار مانند بخش ب عمل میکنیم اما به جای DCT از تبدیل DFT استفاده میکنیم و تحلیل خودمان و گزارش کار را در ادامه میاوریم. برای اینکار همانند بخش الف دو تابع کمکی باید تعریف کنیم.

گزارش کار

این بخش در همان فایل 05 قرار دارد و از قسمت PART D این بخش شروع میشود.

```python
def dft2(block):
    return np.fft.fft2(block)

def idft2(block_dft_coeffs):
    return np.fft.ifft2(block_dft_coeffs)
```

همانطور که گفتیم مانند بخش الف به ۲ تابع کمکی نیاز داریم برای تبدیل **DFT**، در این بخش از **numpy** برای پیاده‌سازی این توابع کمک گرفتیم هم خود تبدیل و هم معکوس آن را ساختیم.

```python
def process_image_blockwise_dft(image_to_process_float, original_img_gray_for_psnr, block_size_N):
    original_rows, original_cols = image_to_process_float.shape

    pad_rows_bottom = (block_size_N - original_rows % block_size_N) % block_size_N
    pad_cols_right = (block_size_N - original_cols % block_size_N) % block_size_N
    padded_image = np.pad(image_to_process_float, ...

    padded_rows, padded_cols = padded_image.shape

    reconstructed_padded_image_complex = np.zeros_like(padded_image, dtype=np.complex64)

    mask_block = np.zeros((block_size_N, block_size_N), dtype=np.float32)
    keep_dim = block_size_N // 2
    mask_block[0:keep_dim, 0:keep_dim] = 1.0

    start_time_proc = time.time()

    for r_idx in range(0, padded_rows, block_size_N):...

    end_time_proc = time.time()
    execution_time_proc = end_time_proc - start_time_proc

    reconstructed_padded_image_real = np.real(reconstructed_padded_image_complex)

    reconstructed_image_cropped_float = reconstructed_padded_image_real[0:original_rows, 0:original_cols]

    reconstructed_image_uint8 = np.clip(reconstructed_image_cropped_float, 0, 255).astype(np.uint8)

    psnr_value = peak_signal_noise_ratio(original_img_gray_for_psnr, reconstructed_image_uint8, data_range=255)
```

مانند سری قبل با کمک تابع طراحی شده عکس را به بلاک هایی تقسیم کردیم و **DFT** روی آن اعمال کردیم بعد ماسک زدیم و بعد تبدیل معکوس گرفتیم. تمام روند مثل بخش ب هست فقط یک سری موارد وجود دارد:

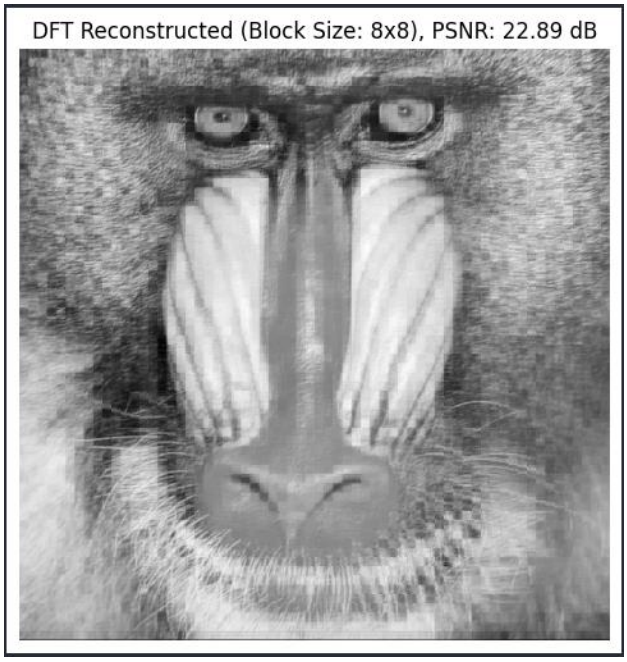
از اونجایی که **DFT** بخش موهومی نیز دارد و با عدد مختلط سروکار داریم ضرایب **DFT** نیز عدد مختلط هستند و قبل از تصویر بازسازی شده که بخش **real** آن را برداریم، مختلط خواهد بود و همین امر را با کمک **reconstructed_padded_image_complex**

مدیریت کردیم. بعد از این ماسک را ساختیم و تقسیم بر N/2 کردیم قسمت های low frequency نزدیک مولفه DC در F[0,0] قرار دارند در خروجی تبدیل و ما این را پیشبینی کردیم. دقت شود که mask_block نیز real است. بقیه مراحل مثل بخش ب است، فقط DFT اعمال شده است. در ادامه با کمک reconstructed_... قسمت read را برداشتیم تا بتوانیم خروجی را نشان بدهیم در تصویر بازسازی شده. بقیه مراحل مثل سابق است.

```
if 'original_image_gray' not in globals() or original_image_gray is None:
    print("ERROR in image")
elif 'block_processing_results' not in globals():
    print("Results from Part B  not found.")
else:
    original_image_float_for_processing = original_image_gray.astype(np.float32)

    block_sizes_to_test = [8, 16, 64]
    dft_block_processing_results = {}


    for N_dft in block_sizes_to_test:
        print(f"--- Processing with DFT, block size: {N_dft}x{N_dft} ---")

        reconstructed_img_dft, exec_time_dft, psnr_dft = process_image_blockwise_dft(
            original_image_float_for_processing,
            original_image_gray,
            N_dft
        )

        dft_block_processing_results[N_dft] = {
            'psnr': psnr_dft,
            'time': exec_time_dft,
            'image': reconstructed_img_dft
        }

        display_image(reconstructed_img_dft, title=f"DFT Reconstructed (Block Size: {N_dft}x{N_dft}), PSNR: {psnr_dft:.2f} d
        print(f"Block Size: {N_dft}x{N_dft}")
        print(f"PSNR (DFT): {psnr_dft:.2f} dB")
        print(f"Execution Time (DFT): {exec_time_dft:.4f} seconds")
```

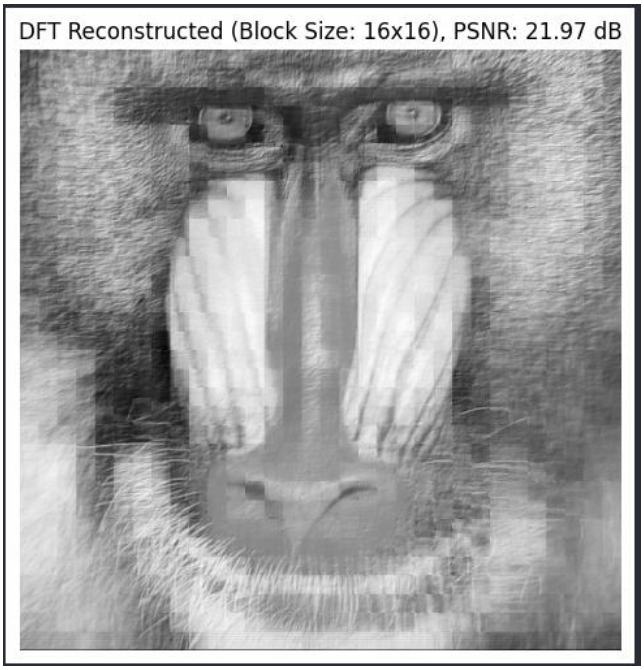
در قسمت بعد تابع را برای بلاک های مختلف فراخوانی کردیم. و نتایج را در ادامه نشان میدهیم:

خروجی



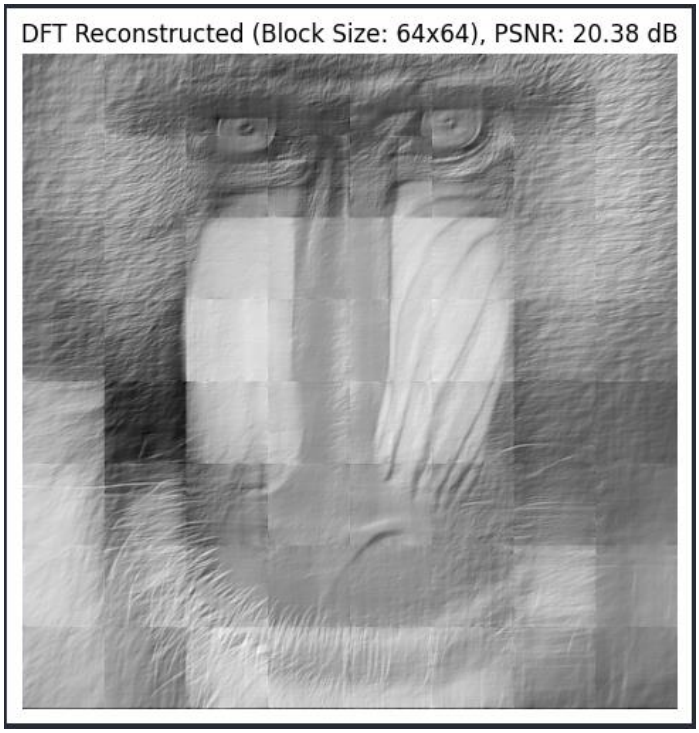

```
Block Size: 8x8
PSNR (DFT): 22.89 dB
Execution Time (DFT): 0.6884 seconds
------------------------------
```

تصویر بازسازی شده با سایز بلوک ۸ در ۸ که زمان اجرا و **PSNR** آن نیز گزارش شده است.



```
Block Size: 16x16
PSNR (DFT): 21.97 dB
Execution Time (DFT): 0.2131 seconds
------------------------------
```

تصویر بازسازی شده با سایز بلوک ۱۶ در ۱۶ که زمان اجرا و **PSNR** آن نیز گزارش شده است.




```
Block Size: 64x64
PSNR (DFT): 20.38 dB
Execution Time (DFT): 0.0300 seconds
------------------------------
```

تصویر بازسازی شده با بلوک ٦٤ در ٦٤ که زمان اجرا و **PSNR** آن نیز گزارش شده است.

```
--- Comparison: DFT vs DCT Block-Based Processing ---
================================================================================
| Block Size | PSNR (DFT) | Time (DFT) (s) | PSNR (DCT) | Time (DCT) (s) |
|------------|------------|----------------|------------|----------------|
| 8          | 22.89      | 0.6884         | 24.09      | 0.2414         |
| 16         | 21.97      | 0.2131         | 24.31      | 0.0695         |
| 64         | 20.38      | 0.0300         | 24.48      | 0.0080         |
```

گزارش تبدیل های DFT و DCT از لحاظ زمان اجرا و مقدار PSNR با سایز بلوک های متناظر و مختلف.

تحلیل

همانطور که قابل مشاهده است مقدار PSNR برای سایز یکسان بلاک ها به طور قابل توجهی در DFT کمتر است بخاطر اینکه DCT به خوبی انرژی را فشرده میکند در صورتی که در DFT این برقرار نیست، و وقتی ماسک و تبدیل را اعمال میکنیم چون DCT مولفه های low frequency را در یک گوشه فشرده کرده است با دور ریختن بقیه قسمت ها اطلاعات چندانی از دست نمیدهیم اما در DFT این شکلی نیست و ما حجم زیادی از اطلاعات را از دست میدهیم و خروجی بی کیفیت میشود.

تصاویر بازسازی در DFT نیز کیفیت پایینی دارند و مشکل blocking artifact اینجا نیز ظاهر میشود به سبب اینکه پردازش روی بلوک های مختلف به صورت مستقل از هم اجرا میشود و عدم پیوستگی در مرز بلاک ها رخ میدهد. همچنین اثر ringing artifact به سبب ذات DFT در برخورد با ضرایب اینجا نیز رخ میدهد و ظاهر میشود زیرا انگار در دامنه فرکانسی مانند فیلتر ایده آل عمل میکند. مخصوصا در نزدیکی لبه‌ها این اثر بیشتر قابل مشاهده

است. به صورت کلی تصاویر DFT ممکن است به طور کلی هموار تر با جزئیات کمتر بنظر برسند که این هم بخاطر همون مسئله فشرده سازی انرژی است. از لحاظ زمان اجرا نیز DFT کمی کند تر هست به سبب وجود اعداد مختلط در آن. چرا این اثر در DFT بیشتر است؟ زیرا توابع پایه DCT کسینوس هستند که زوج هستند و تقارن دارند و در مرز های بلاک ها این اثر کمتر هست اما در DCT به این شکل نیست و توابع پایه از سینوس و کسینوس تشکیل شده است. به همین سبب این آثار در DCT کمتر است. همچنین DFT هر بلاک را به عنوان یک دوره تناوب از یک سیگنال متناوب فرض میکند و اگر لبه سمت چپ به طور هموار به لبه سمت راست متصل نشود باعث یک سری مولفه‌های high frequency مصنوعی میشوند که همین ها هم دچار ringing artifact میشوند.

سوال ششم

این سوال در ۲ بخش حل شده است. تصاویر خوانده شده در این سوال در فایل داده شده HDR وجود دارند. گزارش کار مربوط به کد هر بخش در همان بخش قرار دارد.

بخش الف

در این بخش صرفا عکس های داده شده را خواندیم و برای تولید تصویری دامنه دینامیک بالا، تصاویر را با هم جمع کردیم و میانگین را حساب کردیم و نمایش دادیم. کد این قسمت در فایل **notebook** تحت عنوان **06_PARTA.ipynb** قرار دارد و خروجی آن نیز در ۲ فایل با نام های **06_PART_Average.png** و **06_PARTA_02** قرار دارد البته که در ادامه گزارش نیز آمده است.

گزارش کار

```python
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Python

ابتدا کتابخانه های مورد نیاز خود را **import** کردیم. از **matplotlib** برای نمایش خروجی استفاده کردیم. از **numpy** برای انجام عملیات‌های عددی رو تصاویر استفاده کردیم.

```python
hdr_folder = "HDR"

image_filenames = [
    "StLouisArchMultExpCDR.jpg",
    "StLouisArchMultExpEV+1.51.jpg",
    "StLouisArchMultExpEV+4.09.jpg",
    "StLouisArchMultExpEV-1.82.jpg",
    "StLouisArchMultExpEV-4.72.jpg"
]
```

Python

در این مرحله نام پوشه شامل عکس‌های داده شده و نام تصاویر را مشخص کردیم برای اینکه مشخص باشد دقیقا کدام فایل ها را داریم میخوانیم.

```python
def load_images_from_folder(folder, filenames):
    loaded_images = []
    loaded_paths = []
    print(f"Attempting to read images from folder: {folder}")
    if not os.path.isdir(folder):
        print(f"Error: Folder '{folder}' not found.")
        return [], []

    for filename in filenames:
        path = os.path.join(folder, filename)
        if os.path.exists(path):
            img = cv2.imread(path)
            if img is not None:
                loaded_images.append(img.astype(np.float32))
                loaded_paths.append(path)
                print(f"Loaded {os.path.basename(path)}, shape: {img.shape}, dtype: {loaded_images[-1].dtype}")
            else:
                print(f"Error reading image: {path}")
        else:
            print(f"Image '{path}' not found.")

    if not loaded_images:
        print("Error in reading images.")
    return loaded_images, loaded_paths
```

0.0s Python

یک تابع برای خواندن تصاویر تعریف کردیم. این تابع در پوشه گفته شده حرکت میکند و دانه به دانه تصاویر خواسته شده را میخواند و بارگذاری میکند.

```python
images = []
loaded_image_paths = []

if os.path.isdir(hdr_folder):
    images, loaded_image_paths = load_images_from_folder(hdr_folder, image_filenames)


if not images:
    print("\nError in reading image")
else:
    if len(images) < len(image_filenames):
        print(f"\nNote: Successfully loaded {len(images)} image(s) out of {len(image_filenames)} specified.")
    else:
        print(f"\nSuccessfully loaded all {len(images)} images.")
```

[4] ✓ 2.9s Python

```
Attempting to read images from folder: HDR
Loaded StLouisArchMultExpCDR.jpg, shape: (2112, 2816, 3), dtype: float32
Loaded StLouisArchMultExpEV+1.51.jpg, shape: (2112, 2816, 3), dtype: float32
Loaded StLouisArchMultExpEV+4.09.jpg, shape: (2112, 2816, 3), dtype: float32
Loaded StLouisArchMultExpEV-1.82.jpg, shape: (2112, 2816, 3), dtype: float32
Loaded StLouisArchMultExpEV-4.72.jpg, shape: (2112, 2816, 3), dtype: float32

Successfully loaded all 5 images.
```

در نهایت در این بخش تابع را فراخوانی کردیم و هر ۵ تصویر داده را بارگذاری کردیم.

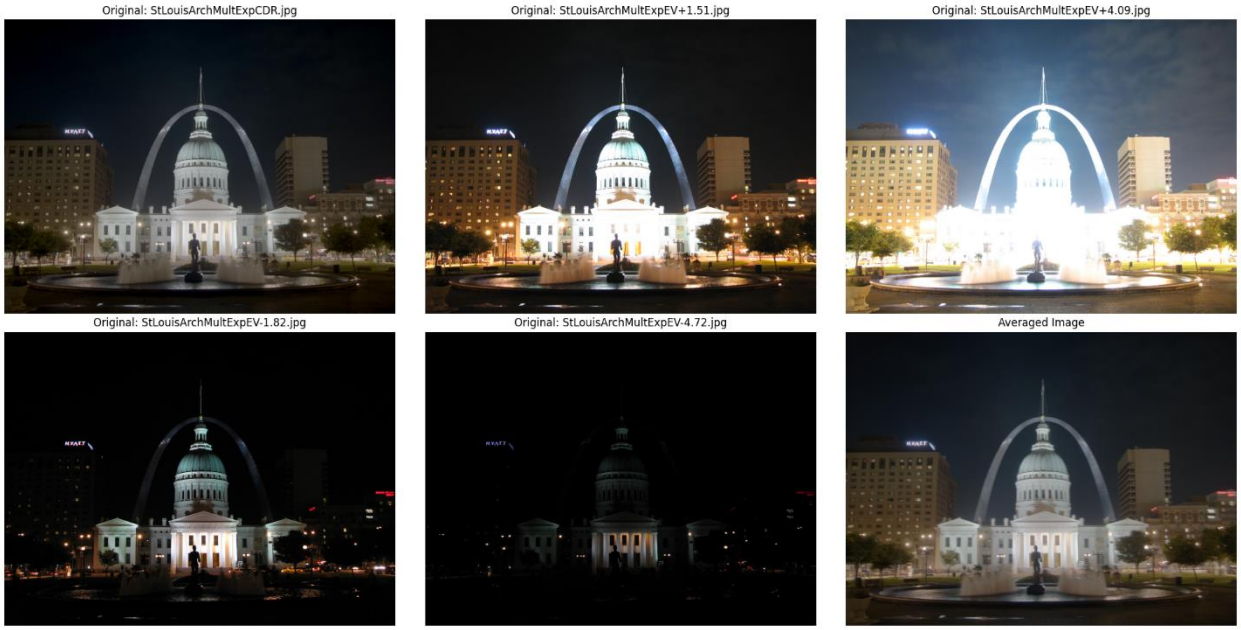
```python
average_image_uint8 = None

if images:
    print("\nCalculating the sum and average of the images...")

    first_shape = images[0].shape
    consistent_shapes = True
    for i, img in enumerate(images[1:], 1):
        if img.shape != first_shape:
            print(f"Error: Image {os.path.basename(loaded_image_paths[i])} has shape {img.shape}, expected {first_shape}.")
            consistent_shapes = False
            break

    if consistent_shapes:
        sum_image = np.zeros_like(images[0], dtype=np.float32)
        for img in images:
            sum_image += img

        average_image_float = sum_image / len(images)

        average_image_uint8 = np.clip(average_image_float, 0, 255).astype(np.uint8)

        print("Summation and averaging complete.")
        print(f"Resulting averaged image dtype: {average_image_uint8.dtype}, shape: {average_image_uint8.shape}")
    else:
        print("Cannot proceed with averaging due to inconsistent image shapes among loaded images.")
```

```
Calculating the sum and average of the images...
Summation and averaging complete.
Resulting averaged image dtype: uint8, shape: (2112, 2816, 3)
```

در این قسمت قرار است از چندین عکسی که بارگذاری کردیم میانگین گیری کنیم. برای اینکه میانگین گیری کنیم همه تصاویر باید یک سایز داشته باشند بنابراین با کمک `first_shape` ابعاد اولین تصویر را نگه میداریم تا به عنوان یک معیار از آن استفاده کنیم. و به کمک `consistent_shape` میخواهیم بررسی کنیم سایر تصاویر هم ابعاد یکسان داشته باشند. یک `for` بر روی باقی مانده تصاویر میزنیم و ابعاد تصویر اولی را با همه آنها مقایسه میکنیم. در نهایت اگر همه ۵ تصویر یک ابعاد داشته باشند سراغ ادامه کد میرویم. با `sum_image` مقدار هر پیکسل جمع میکنیم و نگه میداریم و بعد با کمک `average_image_float` مجموع را تقسیم بر تعداد تصاویر میکنیم تا میانگین بگیریم در نهایت مقادیر که تو بازه `[0, 255]` نیستند را `clip` میکنیم و نمایش میدهیم.

```python
if images:
    print("\nDisplaying images...")

    num_originals = len(images)
    display_average = average_image_uint8 is not None

    total_plots = num_originals + (1 if display_average else 0)

    if total_plots > 0:
        plt.figure(figsize=(20, 10 if num_originals > 2 else 6 ))

        if total_plots <= 1:
            plot_rows, plot_cols = 1,1
        elif total_plots <= 4:
            plot_rows = 1
            plot_cols = total_plots
        elif total_plots <= 6:
            plot_rows = 2
            plot_cols = 3
        else:
            plot_cols = 4
            plot_rows = (total_plots + plot_cols - 1) // plot_cols

        for i, img_float in enumerate(images):
            plt.subplot(plot_rows, plot_cols, i + 1)
            img_display = np.clip(img_float, 0, 255).astype(np.uint8)
            plt.imshow(cv2.cvtColor(img_display, cv2.COLOR_BGR2RGB))
            plt.title(f"Original: {os.path.basename(loaded_image_paths[i])}")
            plt.axis('off')
```

در این قسمت صرفا فراخوانی میکنیم و ۵ تصویر را به همراه میانگین تصاویر نشان میدهیم، که در ادامه آن را نشان میدهیم:



۵ تصویر خوانده شده و تصویر حاصل از میانگین گیری.

خروجی

تصویر حاصل از میانگین گرفته شده.

```python
if average_image_uint8 is not None:
    output_filename = "StLouisArch_Averaged.png"
    try:
        cv2.imwrite(output_filename, average_image_uint8)
        print(f"\nAveraged image saved as {output_filename}")
    except Exception as e:
        print(f"Error saving image {output_filename}: {e}")
```

Python

```
Averaged image saved as StLouisArch_Averaged.png
```

در این قسمت از خروجی یک فایل **PNG** گرفتیم.

بخش ب

در این قسمت مراحل بارگذاری تصاویر و محاسبه میانگین مانند سابق است و فقط قسمت تبدیل موجک آن فرق میکند و دیگر گزارش این قسمت ها را انجام نمیدهیم. این بخش در فایل **06_PARTB.ipynb** قرار دارد و خروجی آن نیز تحت عنوان **06_PART6_Wavelet.png** تعریف شده است. گزارش کار مربوط به کد نیز در ادامه آمده است. تحلیل مقایسه بین این ۲ نیز در ادامه آمده است.

گزارش کار

```python
coeffs_list_per_image = []
wavelet = 'haar'

if images_float and ('consistent_shapes_b' in globals() and consistent_shapes_b):
    print(f"\n--- Starting Part (B): Wavelet Fusion ---")
    print(f"\n1. Decomposing images using '{wavelet}' wavelet...")

    for i, img_float in enumerate(images_float):
        img_coeffs_channels = []
        for channel_idx in range(img_float.shape[2]):
            channel_data = img_float[:, :, channel_idx]

            coeffs = pywt.dwt2(channel_data, wavelet, mode='periodization')
            img_coeffs_channels.append(coeffs)
        coeffs_list_per_image.append(img_coeffs_channels)
        print(f"Decomposed image {os.path.basename(loaded_image_paths_part_b[i])} ({i+1}/{len(images_float)})")

    if coeffs_list_per_image:
        print("Decomposition complete for all images.")

elif not images_float:
    print("Skipping Wavelet Decomposition: No images loaded.")
```

✓ 3.3s Python

```
--- Starting Part (B): Wavelet

1. Decomposing images using 'haar' wavelet...
Decomposed image StLouisArchMultExpCDR.jpg (1/5)
Decomposed image StLouisArchMultExpEV+1.51.jpg (2/5)
Decomposed image StLouisArchMultExpEV+4.09.jpg (3/5)
Decomposed image StLouisArchMultExpEV-1.82.jpg (4/5)
Decomposed image StLouisArchMultExpEV-4.72.jpg (5/5)
Decomposition complete for all images.
```

با کمک این تابع قرار است تجزیه **Wavelet** را انجام بدهیم. برای اینکار ما از نوع **haar** برای اینکار استفاده میکنیم. خروجی اینکار یک سری ضرایب **wavelet** برای هر عکس هست که بعدا مورد استفاده قرار میگیرند. برای انجام تبدیل ۲ بعدی **wavelet** از کتابخانه

PyWavelets کمک گرفتیم تا تبدیل ۲ بعدی را اجرا کنیم. با کمک **image_float** بر روی هر کانال رنگی حرکت میکنیم و تجزیه **wavelet** را انجام میدهیم هر کانال به صورت جدا از هم و نتایج را در **img_coeffs_channels** نگه میداریم. در نهایت مقادیر را درون **coeffs_list_per_image** نگه میداریم که یک لیستی از تصاویر هستند که هر تصویر یک لیستی از ضرایب در هر کانال هستند.

```
fused_coeffs_all_channels = []

if coeffs_list_per_image:
    print("\n2. Fusing coefficients...")
    num_images_b = len(images_float)
    num_channels_b = images_float[0].shape[2]

    for channel_idx in range(num_channels_b):
        ll_coeffs_channel, lh_coeffs_channel, hl_coeffs_channel, hh_coeffs_channel = [], [], [], []

        for img_idx in range(num_images_b):...


        stacked_ll = np.stack(ll_coeffs_channel, axis=0)
        stacked_lh = np.stack(lh_coeffs_channel, axis=0)
        stacked_hl = np.stack(hl_coeffs_channel, axis=0)
        stacked_hh = np.stack(hh_coeffs_channel, axis=0)


        fused_ll = np.mean(stacked_ll, axis=0)


        fused_lh = np.max(stacked_lh, axis=0)
        fused_hl = np.max(stacked_hl, axis=0)
        fused_hh = np.max(stacked_hh, axis=0)

        fused_coeffs_all_channels.append((fused_ll, (fused_lh, fused_hl, fused_hh)))
        print(f"  Fused coefficients for channel {channel_idx} (B,G,R order).")
```

```
2. Fusing coefficients...
  Fused coefficients for channel 0 (B,G,R order).
  Fused coefficients for channel 1 (B,G,R order).
  Fused coefficients for channel 2 (B,G,R order).
Coefficient fusion complete.
```

در این مرحله میخواهیم ضرایب **wavelet** را با همدیگر ترکیب کنیم در تمامی تصاویر برای هر کانال رنگی تا در نهایت یک مجموعه از ضرایب ترکیب شده برای هر کانال داشته باشیم تا در ادامه بتوانیم تصاویر بازسازی را داشته باشیم. هدف اینکار اینکه اطلاعات مرتبط هر کدام از تصاویر را با همدیگر ترکیب کنیم زیرا **wavelet** هنگامی که یک عکس را تجزیه میکند به اجزایی تجزیه میکند مثل لبه ها و **texture** ها و بر اساس محتوا میکند هر کدام از اینها را **scale**

کند و وقتی ما به این شکل تجمیع شده این اطلاعات را داشته باشیم `scale` هم میتوانیم بکنیم و بر اساس هدف خودمان اون موارد را `scale` بیشتری بدهیم. یک کاربرد دیگر اصل محلی بودن در دامنه فرکانسی و مکان- زمان است که برعکس تبدیل فوریه این تبدیل اصل محلی بودن را میپذیرد و وقتی این ادغام صورت بگیرد ما میدانیم کدام ویژگی از هر عکس را میتوانیم به صورت محلی استفاده کنیم. در ابتدا یک `for` روی هر کانال میزنیم و برای جمع آوری `(LL,LH,HL,HH)` هر کدام یک لیست جدا میسازیم برای همه تصاویر تا بتوانیم ضرایب هر کدام از ٤ تا نگهداری کنیم. با `for` بعدی این ضرایب را برای کانال فعلی برای همه تصاویر جمع آوری میکنیم از روزی ضرایب تجزیه شده و به لیست متناظر با هر کدام از اون ٤ تا اضافه میکنیم. با کمک `fused_ll` برای ضرایب `ll` میانگین گیری را انجام میدهیم با کمک کتابخانه `numpy`. برای ۳ تا باقی مانده همونطور که در صورت سوال خواسته شده بود `max` را بین ضرایب مختلف پیدا میکنیم که این کار با کمک `fused_lh ...` انجام شده است. در نهایت این ضرایب تجمیع شده را در لیست `fused_coeffs_all_channels` ذخیره میکنیم. پس به طور کلی برای `ll` داریم تقریب میزنیم و میانگین گیری میکنیم و ساختار مشترک را نگه میداریم و تفاوت ها را لحاظ نمیکنیم و برای `LH,HL,HH` روی جزئیات و ویژگی ها قرار است تاکید کنیم. در نهایت لیست `fuesed_...` در قسمت بعدی برای تبدیل معکوس مورد استفاده قرار میگیرد تا بتوانیم تصویر را بازسازی کنیم.

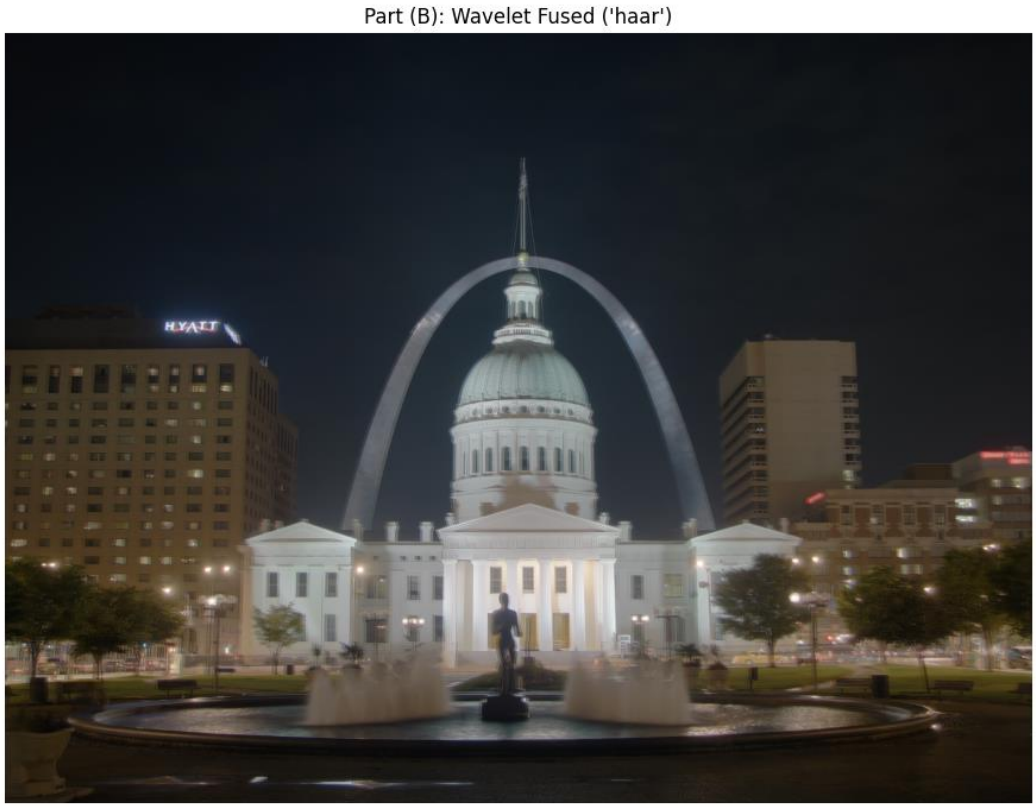
```python
fused_image_wavelet_uint8 = None

if fused_coeffs_all_channels:
    print("\n3. Reconstructing image from fused coefficients...")
    reconstructed_channels = []
    num_channels_b = images_float[0].shape[2]

    for channel_idx in range(num_channels_b):
        coeffs_to_reconstruct = fused_coeffs_all_channels[channel_idx]
        reconstructed_channel = pywt.idwt2(coeffs_to_reconstruct, wavelet, mode='periodization')

        original_channel_shape = images_float[0][:,:,channel_idx].shape
        reconstructed_channel = reconstructed_channel[:original_channel_shape[0], :original_channel_shape[1]]

        reconstructed_channels.append(reconstructed_channel)
        print(f"  Reconstructed channel {channel_idx}.")

    if reconstructed_channels and len(reconstructed_channels) == num_channels_b:
        fused_image_wavelet_float = cv2.merge(reconstructed_channels)
        fused_image_wavelet_uint8 = np.clip(fused_image_wavelet_float, 0, 255).astype(np.uint8)
        print(f"Wavelet fusion and reconstruction complete. Result shape: {fused_image_wavelet_uint8.shape}")
```

✓ 0.5s Python

در نهایت تصویر را با کمک ضرایب **wavelet** که در قسمت قبلی بدست آوردیم و ذخیره کردیم بازسازی میکنیم با کمک تبدیل معکوس **wavelet** تحت عنوان **IDWT**. با کمک متغیر **reconstructed_channels** قرار است بازسازی هر کانال رنگی را انجام بدهیم (قرمز، آبی و سبز). برای اینکه هر کانال را بازسازی کنیم یک **for** میزنیم، ضرایب تجمیع شده را میگیریم با کمک **coeffs_to_reconstruct** و بعد بر روی آن **IDWT** که تبدیل معکوس است را اعمال میکنیم تا بتوانیم تصویر را به طور کلی بسازیم. در نهایت تصویر را **crop** میکنیم تا به ابعاد اصلی برگردد. با **reconstructed_channels** میایم هر کانال بازسازی شده را نگه میداریم. در نهایت بعد از بازسازی همه کانال‌ها، تمام کانال ها را باید با هم ادغام کنیم تا تصویر را بسازیم و این کار را با کمک **fused_image_wavelet_float** انجام میدهیم و در نهایت به **unit8** تبدیل میکنیم.

```python
if fused_image_wavelet_uint8 is not None:
    print("\n4. Displaying results...")

    num_plots = 0
    if average_image_part_a_uint8 is not None:
        num_plots += 1
    if fused_image_wavelet_uint8 is not None:
        num_plots +=1

    if num_plots == 0:
        print("No images available to display.")
    else:
        plt.figure(figsize=(5 * num_plots + 5, 7))
        current_plot = 1

        if average_image_part_a_uint8 is not None:
            plt.subplot(1, num_plots, current_plot)
            plt.imshow(cv2.cvtColor(average_image_part_a_uint8, cv2.COLOR_BGR2RGB))
            plt.title("Part (A): Averaged Image")
            plt.axis('off')
            current_plot += 1
```

در این قسمت فراخوانی میکنیم و خروجی را هم برای میانگین گرفته شده و هم برای تصویر بازسازی شده با تبدیل **wavelet** نشان میدهیم و در ادامه تحلیل میکنیم.



خروجی تصویر بعد از اعمال تبدیل **wavelet**.

مقایسه

مقایسه بین تصویر میانگین گرفته شده از بخش الف با تصویر تبدیل **wavelet** گرفته شده در بخش ب.

تحلیل

در بخش الف وقتی تصویر را میانگین گیری میکنیم باعث میشود نویزها کاهش پیدا کنند البته این امر سبب تاری و محو شدگی جزئیات نیز میشود. در بخش ب اما جزئیات بهتر حفظ میشوند زیرا برای **lh,hl,hh** ما پیکسل هایی با بیشترین مقدار را انتخاب کردیم که جزئیات بیشتری دارند و جزئیات برجسته و لبه ها تیز هستند برای هر تصویر در آن ناحیه. وضوح و کنتراست خوبی دارند و به تصویر نهایی نیز منتقل میشود. این کار ما باعث شد **بهترین اطلاعات high frequency** را از بین تصاویر انتخاب کنیم. به دلیل حفظ جزئیات، کنتراست محلی در تصویر نهایی بهتر از روش الف بوده است. همچنین با میانگین گیری در قسمت **ll** یک تصویر با نویز کم و بدون جزئیات زیاد ایجاد میکند و اطلاعات کلی در تصویر نهایی حفظ میشوند. پس از لحاظ

وضوح و جزئیات روش الف تصویری هموار تر با جزئیات کمتر تولید میکند اما در روش ب تصویر با وضوح بالاتر و جزئیات برجسته تر قرار دارد. کنتراست نیز در روش ب بهتر هست به صورت محلی به دلیل حفظ و انتخاب جزئیات اما در روش الف هم متعادل است. روش الف مستعد artifact هست همانطور که قابل مشاهده هست محو شدگی قابل درکی دارد و روش ب این اثر را بهتر مدیریت میکند البته این کار maxگیری خودش نیز ممکن است آثار جانبی داشته باشد. از لحاظ نویز هم روش اول در کاهش نویز بسیار بهتر است نسبت به روش دوم اگرچه روش دوم هم خوب است اما بخاطر اون maxگیری ممکن است درگیر آثار جانبی شویم. در نگاه اول شاید دو تصویر شبیه بهم بنظر برسند زیرا روشنایی و رنگ های آنها بسیار به هم شبیه است. اما یک سری تفاوت وجود دارد مثلا در قسمت لبه های gateway arch در روش ب کمی برجسته تر و تیز تر هستند نسبت به روش الف. جزئیات ساختمان ها مثل پنجره ها در روش ب بهتر از روش الف هستند. خروجی تصویر روش الف نرم تر است به طور کلی اما روش ب قسمت های برجسته و تیز نیز دارد. نکته ای که مهم است اینکه دلیل شباهت این ۲ بخاطر این هست که ما در روش ب از LL میانگین گیری کردیم و همانطور که میدانیم قسمت عمده انرژی نیز در همان بخش وجود دارد و همین امر سبب شده این ۲ تصویر شبیه هم شوند در رنگ ها و روشنایی و بقیه موارد.