MATH3320 Project Report
Topic: Image Compression

ZHANG Xinfang 1155141566
ZHANG Yifan 1155141570

Monday 8th November, 2021

1. Introduction

In recent years, with the rapid developement in technology, multimedia product of digital information grows increasingly fast, which requires a large memory space and sufficient bandwidth in the storage and transmission process. Therefore, data compression becomes extremely vital for reducing the data redundancy to save more hardware space and transmission bandwidth.

Image compression is the process of removing redundant and less irrelevant information, and efficiently encoding or reverting what remains without affecting or degrading its quality. The objective of image compression is to store or transmit data in an efficient form and to reduce the storage quantity as much as possible.

One useful technique in image compression is to decompose an image into linear combination of elementary images with specific properties. By truncating some less important components in the image decomposition, we can compress an image to reduce the image size and achieve transform coding.

In this paper, we will discuss some useful image decomposition methods, demonstrate the applications of these decomsotion methods for image compression and compare their compression performance.

2. Image Compression Models

2.1 Singular Value Decomposition (SVD)

2.1.1 Definition

Every $m \times n$ image g has a singular value decomposition.
For any $g \in M_{m\times n}$, the singular value decomposition (SVD) of g is a matrix factorization give by

$$g = U\Sigma V^T$$

with $U \in M_{m\times m}$ and $V \in M_{n\times n}$ both unitary, and $\Sigma \in \mathbb{R}^{m\times n}$ is a diagonal matrix with diagonal elements $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$ with r $\leq$ min (m,n). The diagonal elements are called singular values. Then we have

$$g = U\Sigma V^T = \sum_{i=1}^{r} \sigma_i \vec{u_i}\vec{v_i}^T \quad ,$$

where $\vec{u_i}\vec{v_i}^T$ is called the eigenimages of g under SVD.

The most interesting part of SVD methods is that the data arrange in such a way that the most important data is stored on the top, which corresponds to the top eigenvalues and eigenimages.

2.1.2 Application

SVD can be applied for image compression, by removing terms (eigenimages) associated to small singular values. We will show examples of image compression using SVD, so-called 'rank-k approximation'. The whole implementation of this report is done using Python. For simplicity, we convert all images into grayscale images first. The packages and grayscalized process are as follows:

```python
import math
import cmath
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import skimage.io as skio
from skimage.color import rgb2gray
```

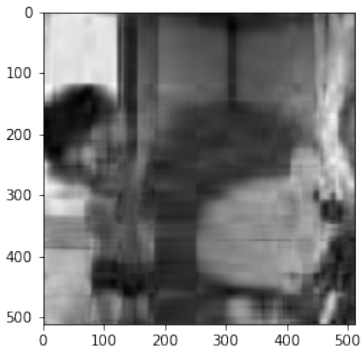
(a) Import packages

```python
# Read the image and convert colorful images to grayscale
lenna = rgb2gray(skio.imread('Lenna.png'))
dora = rgb2gray(skio.imread('dora.png'))
platform = rgb2gray(skio.imread('platform.png'))
hibiscus = skio.imread('hibiscus.png')
einstein = skio.imread('einstein2.png')
balloon = skio.imread('balloon.png')
```

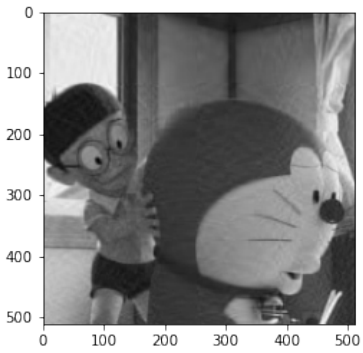
(b) Read and convert images

Figure 1: Packages and pre-processing

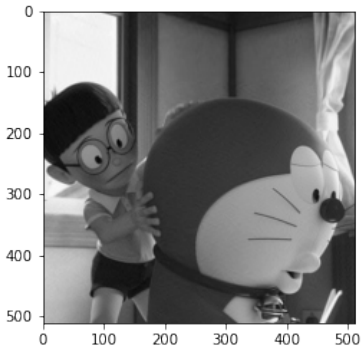
Here is the results for different rank-k approximations. (# of singular values = 512)



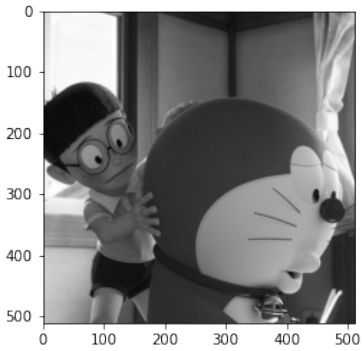
(a) k = 10



(b) k = 40



(c) k = 100



(d) Original

Figure 2: Doraemon and its compressed images using SVD

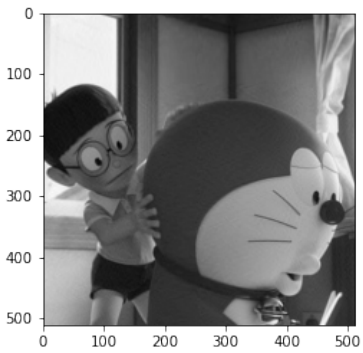
Since the weightings of eigenimages depend on corresponding singular values, the larger the singular value is, the more important the eigenimage is. Hence, we consider ignoring singular values which are less than 0.5% of the largest singular value automatically so that the image can be compressed without losing too much information and influcing the image quality. Our code and result are as follows:

Code:

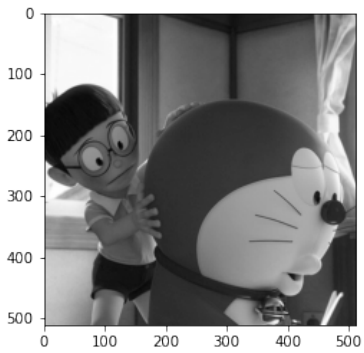
```
# Use the SVD function to compute SVD of the image
u, s, vh = np.linalg.svd(lenna, full_matrices=True)
r, c = lenna.shape
# Generate a rank-k approximation image
img = np.zeros((r, c))
uh = np.array(u).T
k = 0

for i in range(0, r):
    if s[i]/s[0] >= 0.005:
        img = np.add(img, np.matmul(np.array([uh[i]]).T, np.array([vh[i]])) * s[i])
        k = i+1
    else:
        break

print(f'k = {k}')
plt.imshow(img, cmap="gray")
plt.savefig('lenna_auto_{k}.png')
```

Figure 3: SVD Algorithm

Results:



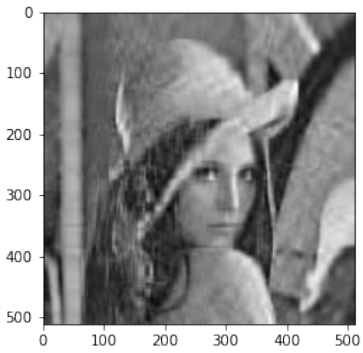
(a) Algorithm with k = 86



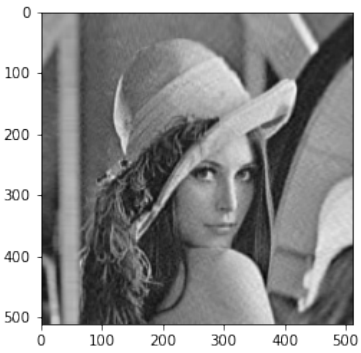
(b) Original

Figure 4: SVD algorithm results

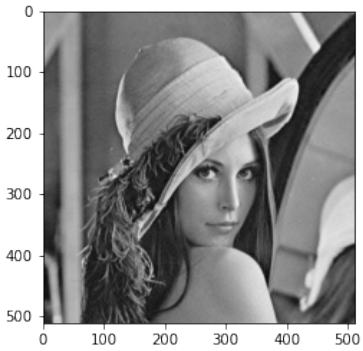
Now we use the same algorithm to test another image called 'Lenna'. The comparison between SVD compressed images and the original image are as follows: (# of singular values = 512)



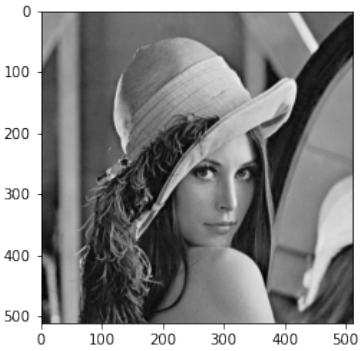
(a) k = 20



(b) k = 50



(c) Algorithm with k = 107



(d) Original

Figure 5: Lenna and its compressed images using SVD

2.1.3 Error Analysis

For any k with $0 \leq k \leq r$, we define g_k a rank-k approximation of g by

$$g_k = \sum_{j=1}^{k} \sigma_j \vec{u_j} \vec{v_j}^T,$$

This low rank matrix approximation can be applied to image compression.

Here we apply Frobenius norm to compute the error of approximation:

Let $f = \sum_{j=1}^{r} \sigma_j \vec{u_j} \vec{v_j}^T$ be the SVD of an $M \times N$ image f. For any k with $k < r$, and $f_k = \sum_{j=1}^{k} \sigma_j \vec{u_j} \vec{v_j}^T$, we have

$$\|f - f_k\|_F^2 = \sum_{j=k+1}^{r} \sigma_j^2$$

Proof: Let $f = \sum_{j=1}^{r} \sigma_j \vec{u_j} \vec{v_j}^T$.

Approximate f by f_k with $k < r$ where $f_k = \sum_{j=1}^{k} \sigma_j \vec{u_j} \vec{v_j}^T$

Define the error of the approximation by $D \equiv f - f_k = \sum_{j=k+1}^{r} \sigma_j \vec{u_j} \vec{v_j}^T \in M_{m \times n}$ Then the m-th row, n-th column entry of D is given by

$$D_{mn} = \sum_{j=k+1}^{r} \sigma_j u_{im} v_{in}$$

where $\vec{u_i} = \begin{pmatrix} u_{i1} \\ \vdots \\ u_{iM} \end{pmatrix}$, $\vec{v_i} = \begin{pmatrix} v_{i1} \\ \vdots \\ v_{iN} \end{pmatrix}$. Then,

$$\begin{aligned}
\|D\|_F^2 &= \sum_m \sum_n D_{mn}^2 \\
&= \sum_m \sum_n \left(\sum_{i=k+1}^{r} \sigma_i^2 u_{im}^2 v_{in}^2 + 2 \sum_{i=k+1}^{r} \sum_{\substack{j=k+1 \\ j \neq i}}^{r} \sigma_i \sigma_j u_{im} v_{in} u_{jm} v_{jn} \right) \\
&= \sum_{i=k+1}^{r} \sigma_i^2 \cancelto{1}{\sum_m u_{im}^2} \cancelto{1}{\sum_n v_{in}^2} + 2 \sum_{\substack{j=k+1 \\ j \neq i}}^{r} \sigma_i \sigma_j \cancelto{0}{\sum_m u_{im} u_{jm}} \cancelto{0}{\sum_n v_{in} v_{jn}} \\
&= \sum_{i=k+1}^{r} \sigma_i^2
\end{aligned}$$

Therefore, we prove that
Sum of square error of the approximation = Sum of omitted eigenvalues.

2.2 Haar Transform and Walsh Transform

2.2.1 Haar Function and Haar Transform

Haar function:

$$H_m(t) = H_{2^p+n}(t) = \begin{cases} 2^{\frac{p}{2}} & \text{if } \frac{n}{2^p} \leq t \leq \frac{n+0.5}{2^p} \\ -2^{\frac{p}{2}} & \text{if } \frac{n+0.5}{2^p} \leq t \leq \frac{n+1}{2^p} \\ 0 & \text{elsewhere} \end{cases}$$

where $p = 1, 2, \ldots$; $n = 0, 1, 2, \ldots, 2^{p-1}$

Based on the Haar function, Haar Transform of $N \times N$ image is defined as follow.
Let $H(k,i) \equiv H_k\left(\frac{i}{N}\right)$ where $k, i = 0, 1, 2, \ldots, N$-1
We obtain the Haar Transfrom matrix:

$$\tilde{H} \equiv \frac{1}{\sqrt{N}} H, \quad H \equiv (H(k,i))_{0 \leqslant k,i \leqslant N-1}$$

The Haar Transfrom of $f \in M_{n \times n}$ is defined as:

$$g = \tilde{H} f \tilde{H}^T$$

2.2.2 Walsh Function and Walsh Transform

Walsh function is defined recursively as follows:

$$W_{2j+q}(t) = (-1)^{\lfloor \frac{j}{2} \rfloor + q} \{W_j(2t) + (-1)^{j+q} W_j(2t-1)\}$$

where $q = 0$ or 1; $j = 0, 1, 2, \ldots$ and $W_0(t) = \begin{cases} 1 & \text{if } 0 \leq t < 1 \\ 0 & \text{elsewhere} \end{cases}$

Based on Walsh function, Walsh Transfrom of $N \times N$ image is defined as follow.
Let $W(k,i) \equiv W_k\left(\frac{i}{N}\right)$ where $k, i = 0, 1, 2, \ldots, N$-1
We obtain the Walsh Transfrom matrix:

$$\tilde{W} \equiv \frac{1}{\sqrt{N}} W, \quad W \equiv (W(k,i))_{0 \leqslant k,i \leqslant N-1}$$

The Walsh Transfrom of $f \in M_{n \times n}$ is defined as:

$$g = \tilde{W} f \tilde{W}^T$$

2.2.3 Application

According to the definition in last subsection, the transformation codes and transformed images are implemented as follows:

(a) Haar Transform

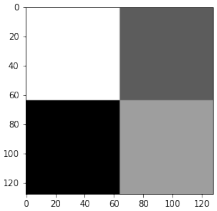
Code:

```
h, w = balloon.shape
# Construct the Haar matrix
H = np.zeros((h, w))
H[0, :] = 1
H[1, 0:int(w/2)] = 1
H[1, int(w/2):(w+1)] = -1
for i in range(2, h):
    for j in range(0, w):
        p = math.floor(math.log(i, 2))
        n = i - pow(2, p)
        t = j/w
        if n/pow(2, p) <= t and t < (n+0.5)/pow(2, p):
            H[i][j] = pow(math.sqrt(2), p)
        elif (n+0.5)/pow(2, p) <= t and t < (n+1)/pow(2, p):
            H[i][j] = -pow(math.sqrt(2), p)
H = np.true_divide(H, math.sqrt(h))
Ht = H.transpose()
# Compute the Haar coefficients matrix
G = np.matmul(np.matmul(H, balloon), Ht)
# Generate the approximation of the image using only the coefficients in the upper left corner
img = np.zeros((h, w))
k = 10
for i in range(0,k):
    for j in range(0, k):
        img = np.add(img, np.matmul(np.array([H[i]]).T, np.array([H[j]])) * G[i][j])
plt.imshow(img, cmap="gray")
plt.savefig(f'balloon_HaarTo_{k}.png')
```

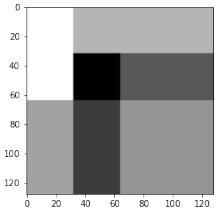
Figure 6: Haar Transform Codes

Results:

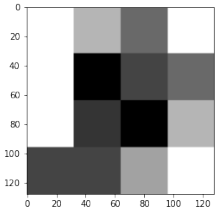
In forllowing figures, we keep upper left corner Haar coefficients in Haar transform matrix up to 2×2, 3×3, 4×4, ... respectively. (Size of this 'balloon' image $= 128 \times 128$)



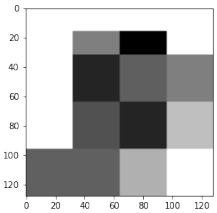
(a) Haar to 2



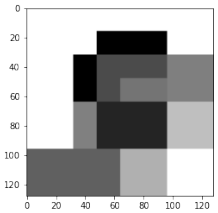
(b) Haar to 3



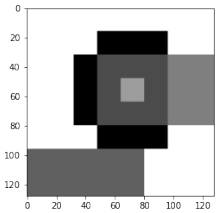
(c) Haar to 4



(d) Haar to 5

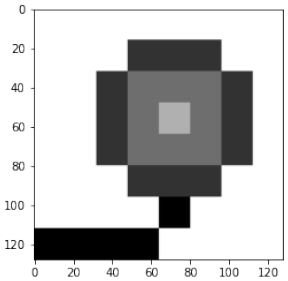


(e) Haar to 6

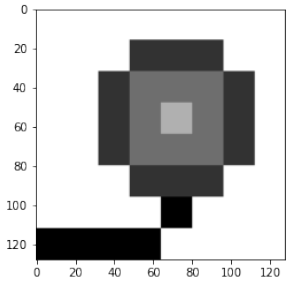


(f) Haar to 7

Figure 7: Balloon and its compressed images using Haar Transform



(a) Haar to 10



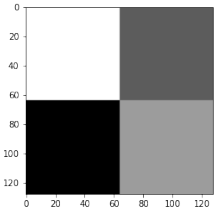
(b) Original

Figure 8: Haar Compression VS Original Image

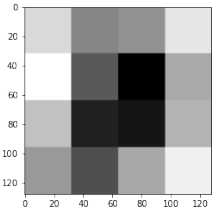
It is obvious that 'Haar to 10' has already achieved a good performance compared to the original image size.

(b) Walsh Transform

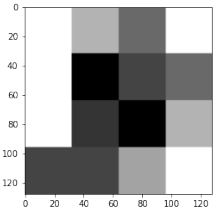
The implementation logic for Walsh Transform is similar to that for Haar Transform. Therefore, we only put the sample images of different size of upper left corner of Walsh coefficients here.



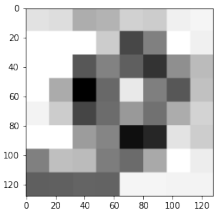
(a) Walsh to 2



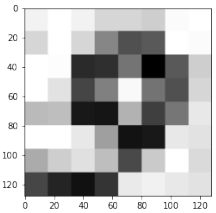
(b) Walsh to 3



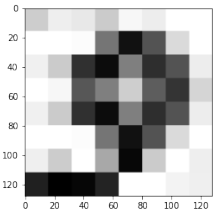
(c) Walsh to 4



(d) Walsh to 5



(e) Walsh to 6



(f) Walsh to 7

Figure 9: Balloon and its compressed images using Walsh Transform

2.3 Discrete Fourier Transform (DFT)

2.3.1 Definition

The 2D **discrete Fourier Transform (DFT)** of a $M \times N$ image $g = (g(k,l)_{k,l})$ where k = 0,1,..., M-1 and l = 0,1,..., N-1, is defined as

$$\hat{g}(m,n) = \frac{1}{MN} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} g(k,l) e^{-2\pi j(\frac{km}{M} + \frac{ln}{N})}$$

And the inverse discrete Fourier Transform is given by

$$g(p,q) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \hat{g}(m,n) e^{2\pi j(\frac{pm}{M} + \frac{qn}{N})}$$

Define matrix $U = (U_{x\alpha})_{0 \leqslant x,\alpha \leqslant N-1} \in M_{N \times N}(\mathbb{C})$ by $U_{x\alpha} = \frac{1}{N} e^{-2\pi j \frac{x\alpha}{N}}$, where $0 \leqslant x, \alpha \leqslant N-1$
Then U is symmetric and

$$\hat{g} = UgU$$

Besides, we can prove that rows of U are mutually orthogonal but not orthonormal.
Then we conclude that $UU^* = \frac{1}{N} I$

2.3.2 Application

(a) Fast Fourier Transform Algorithm (FFT)

Since DFT is seperable, for a 2D image, to compute its DFT is to compute two 1D DFT. Hence, it is sufficient to consider how to compute 1D DFT fast.

Let $\omega_N = e^{-\frac{2\pi}{N}j}$ and assume $N = 2^n = 2M$.
Then the 1D DFT is: $\hat{f}(u) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)\omega_N^{ux} = \frac{1}{2M}\sum_{x=0}^{2M-1} f(x)\omega_{2M}^{ux}$.
Seperate it into odd and even parts, we have

$$\begin{aligned}
\hat{f}(u) &= \frac{1}{2}\left\{\frac{1}{M}\sum_{y=0}^{M-1} f(2y)\omega_{2M}^{u(2y)} + \frac{1}{M}\sum_{y=0}^{M-1} f(2y+1)\omega_{2M}^{u(2y+1)}\right\} \\
&= \frac{1}{2}\left\{\frac{1}{M}\sum_{y=0}^{M-1} f(2y)\omega_M^{uy} + \frac{1}{M}\sum_{y=0}^{M-1} f(2y+1)\omega_M^{uy}\omega_{2M}^{u}\right\} \\
&= \begin{cases} \frac{1}{2}\left\{\hat{f}_{\mathbf{even}}(u) + \hat{f}_{\mathbf{odd}}(u)\omega_{2M}^u\right\} & \text{for } u < M \\ \frac{1}{2}\left\{\hat{f}_{\mathbf{even}}(u) - \hat{f}_{\mathbf{odd}}(u)\omega_{2M}^u\right\} & \text{for } u \geq M \end{cases}
\end{aligned}$$

$$\hat{f}_{\mathbf{even}}(u) = \frac{1}{M}\sum_{y=0}^{M-1} f(2y)\omega_M^{uy} \quad \text{DFT of even part of } f$$

$$\hat{f}_{\mathbf{odd}}(u) = \frac{1}{M}\sum_{y=0}^{M-1} f(2y+1)\omega_M^{uy} \quad \text{DFT of odd part of } f$$

Therefore, **Fast Fourier Transform (FFT) Algorithm** can be stated as:

Let $f \in \mathbb{R}^N$ where $N = 2^n = 2M$.
Step 1: Split f into

$$\begin{aligned}
f_{\mathbf{even}} &= [f(0), f(2), \ldots, f(2M-2)]^T \\
f_{\mathbf{odd}} &= [f(1), f(3), \ldots, f(2M-1)]^T.
\end{aligned}$$

Step 2: Compute $\hat{f}_{\mathbf{even}} = F_M f_{\mathbf{even}}$ and $\hat{f}_{\mathbf{odd}} = F_M f_{\mathbf{odd}}$, where $F_M = (\omega_M^{ux})_{0\leq u,x\leq M-1}$ is an $M \times M$ matrix.
Step 3: Compute $\hat{f}$ using the following formula:

For $u = 0, 1, 2, \ldots, M-1$.

$$\begin{aligned}
\hat{f}(u) &= \frac{1}{2}[\hat{f}_{\mathbf{even}}(u) + \hat{f}_{\mathbf{odd}}(u)\omega_{2M}^u] \\
\hat{f}(u+M) &= \frac{1}{2}[\hat{f}_{\mathbf{even}}(u) - \hat{f}_{\mathbf{odd}}(u)\omega_{2M}^u].
\end{aligned}$$

Let C_m be the **computational cost** of $F_m\mathbf{x}$. Then $C_1 = 1$.
Obviously, $C_N = 2C_M + 3M$ (2 matrix multiplication, M multiplication, addition and subtraction)
Hence, we can conclude that the computational cost C_N is bounded by $KN\log_2 N$. We denote it by $\mathcal{O}(N\log_2 N)$.

Code:

```
# FFT functions
# 1d FFT functions
def omega(p, q):
    '''The omega term in DFT formula'''
    return cmath.exp((2.0 * cmath.pi * 1j * q) / p)
def pad(lst):
    '''padding the list to next nearest power of 2 as FFT implemented is radix 2'''
    k = 0
    while 2**k < len(lst):
        k += 1
    return np.concatenate((lst, ([0] * (2 ** k - len(lst)))))
def fft(x):
    '''FFT of 1d signals'''
    n = len(x)
    if n == 1:
        return x
    Feven, Fodd = fft(x[0::2]), fft(x[1::2])
    combined = [0] * n
    for m in range(int(n/2)):
      combined[m] = Feven[m] + omega(n, -m) * Fodd[m]
      combined[m + int(n/2)] = Feven[m] - omega(n, -m) * Fodd[m]
    return combined
def ifft(X):
    '''Inverse FFT of 1d signals'''
    x = fft([x.conjugate() for x in X])
    return [x.conjugate()/len(X) for x in x]
# 2d FFT functions
def pad2(x):
    m, n = np.shape(x)
    M, N = 2 ** int(math.ceil(math.log(m, 2))), 2 ** int(math.ceil(math.log(n, 2)))
    F = np.zeros((M,N), dtype = x.dtype)
    F[0:m, 0:n] = x
    return F, m, n
def fft2(f):
    '''FFT of 2d images
    usage X, m, n = fft2(x), where m and n are dimensions of original signal'''
    f, m, n = pad2(f)
    return np.transpose(fft(np.transpose(fft(f)))), m, n
def ifft2(F, m, n):
    '''Inverse FFT of 2d images'''
    f, M, N = fft2(np.conj(F))
    f = np.matrix(np.real(np.conj(f)))/(M*N)
    return f[0:m, 0:n]
```

Figure 10: FFT Algorithm

(b) Application of FFT

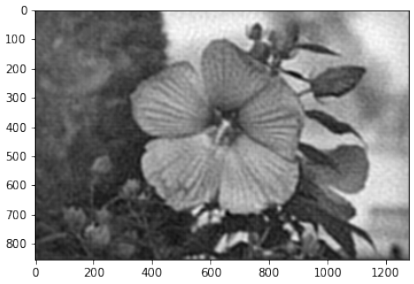
To compress an image, we can calculate corresponding discrete Fourier matrix and then calculate norm of each entry. Ignoring the relatively small coefficients and do inverse DFT finally.

In the following code, we set k as a parameter to dominate the percentage of coefficients to be kept.

```
A, m, n = fft2(platform)
# Compute the norm of each entry in DFT matrix A
norm_A = np.real(np.sqrt(np.multiply(np.conjugate(A), A)))
# Remove the least 100k percent Fourier coefficients
k = 0.005  # k is a value we can shoose
max = np.amax(norm_A)
r, c = norm_A.shape
cnt = int(k * r * c)
# Descending order of all norms of entries in Fourier coefficients matrix
stacked_norm_A = pd.DataFrame(np.reshape(norm_A, r * c).transpose(), columns=['values'])
sorted_norm_A = stacked_norm_A.sort_values(by=['values'], ascending=False).reset_index(drop=True)
# Pick out the bounded value nmax which is the boundary value of top 100k largest value
# of all norms of entries in A
nmax = sorted_norm_A['values'][cnt]
# Assign 0 to values less than nmax
A = np.where(norm_A < nmax, 0, A)
# Calculate iDFT
img = ifft2(A, m, n)
plt.imshow(img, cmap="gray")
plt.savefig(f'platform_fft_{k}.png')
```

Figure 11: FFT compression code

Results:



(a) FFT up to 0.5% coefficients

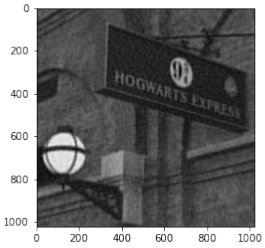
(b) FFT up to 5% coefficients

(c) FFT up to 10% coefficients

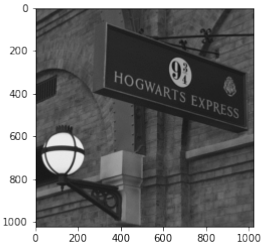
(d) Original

Figure 12: FFT compression results for Hibiscus

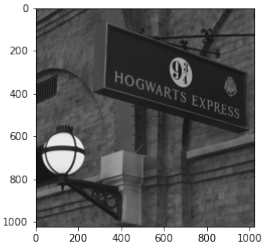
From the 'Hibiscus' example, it is clear that the compression performance is already good enough when top 10% coefficients are kept. The following 'platform' image is also compressed by the same logic.



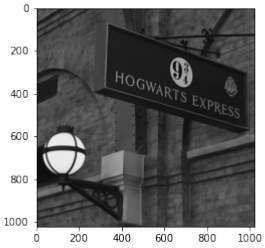
(a) FFT up to 0.5% coefficients



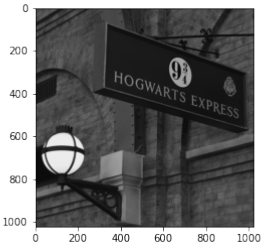
(b) FFT up to 5% coefficients



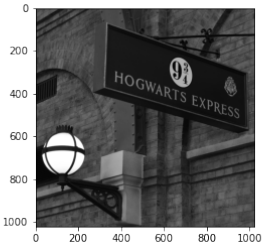
(c) FFT up to 10% coefficients



(d) FFT up to 15% coefficients



(e) FFT up to 20% coefficients



(f) Original

Figure 13: FFT compression results for Platform

(c) Even Discrete Cosine Transform (EDCT)

Even Symmetric Cosine Transform (EDCT) is derived from Discrete Fourier Transform and is applied to **JPEG compression**.

By extending an $N \times N$ image f to a $2M \times N$ image $\tilde{f}$, whose indices are taken from $[-M, M-1]$ and $[-N, N-1]$.
After complicated calculation, we can get that the DFT of $\tilde{f}$:

$$A_1 + A_2 + A_3 + A_4 = \frac{1}{MN} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} f(k,l) cos\Big[\frac{m\pi}{M}\left(k+\frac{1}{2}\right)\Big] cos\Big[\frac{n\pi}{N}\left(l+\frac{1}{2}\right)\Big]$$

$$F(m,n) = \frac{1}{4MN}(A_1 + A_2 + A_3 + A_4) f(k,l) e^{-\pi j \frac{m}{M}(k+\frac{1}{2}) - \pi j \frac{n}{N}(l+\frac{1}{2})}$$

Then **EDCT** of f is defined as:

$$\hat{f}_{ec}(m,n) = \frac{1}{MN} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} f(k,l) cos\Big[\frac{m\pi}{M}\left(k+\frac{1}{2}\right)\Big] cos\Big[\frac{n\pi}{N}\left(l+\frac{1}{2}\right)\Big]$$

with $0 \leq m \leq M-1$, $0 \leq n \leq N-1$.

3. Conclusion

In this part, we would compare the compression results of four different image decomposition techniques: SVD, Haar Transform, Walsh Transform and DFT.

The following table shows the mean square error (MSE) under image compression of an image of a balloon.
In each column with number n, SVD technique refers to MSE under rank-n approximation; Haar transfrom and Walsh transform refers to MSE of image construction by keeping n-size upper left corner coefficients; and DFT technique refers to MSE by reconstruction of $n \times n$ eigenimages.

	1	2	3	4	5	6	7	8	9
SVD	42681	4303	2279	904	220				
Haar	42681	6960	6791	5468	4113	3528	3173	2549	
Walsh	42681	6960	6788	5034	4251	3167	2315		
DFT	42681	5827	3167	2483	2250	1967	1883	1806	1751

Each compression techniques shows different performance at different sizes of component transforms. From results it has been observed that SVD technique gives supervior performance among all with lowest MSE and rapid compression speed. The compression efficiency of Haar Transform, Walsh Transform and DFT is relatively low. They show similar performance when order is lower than 4, however, when order is larger than 4, Walsh Transform gives better performance than the other two techniques.