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 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 techniques 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 analyze their advantages, disadvantages and applicability.

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 numpy as np
import matplotlib.pyplot as plt
import skimage.io as skio
from skimage.color import rgb2gray
```

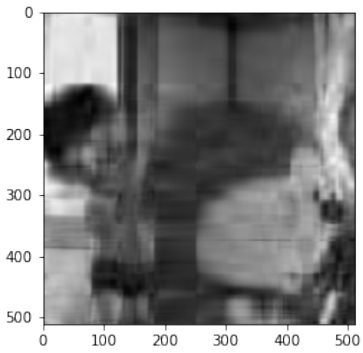
(a) Packages

```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')
```

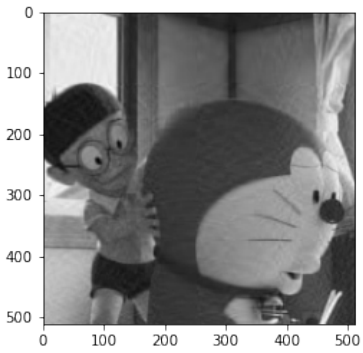
(b) rgb2gray

Figure 1: Packages and pre-processing

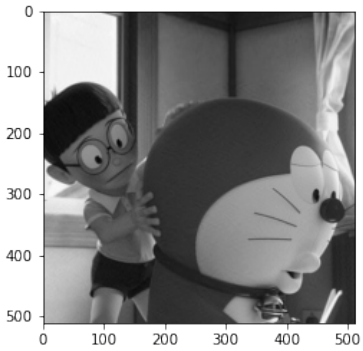
Here is the results for different rank-k. (# of singular values = 512)



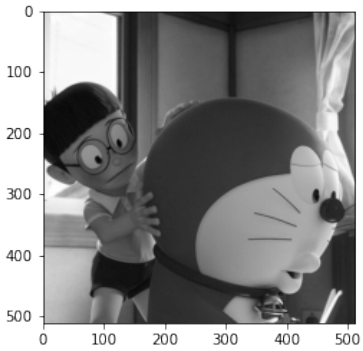
(a) k = 10



(b) k = 40



(c) k = 100



(d) Original

Figure 2: Doraemon and its compressed images using SVD

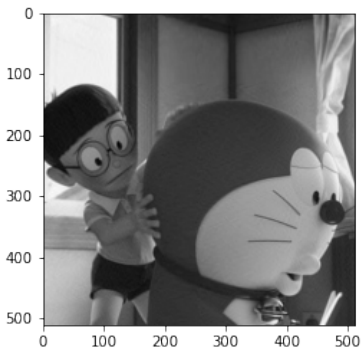
Since the weightings of eigenimages depend on corresponding singular values, the larger the singular value is, the more important the eigen-image is. Hence, we consider ignoring singular values which are less than 0.5% of the largest singular value automatically so that the image can be compressed without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

Code:

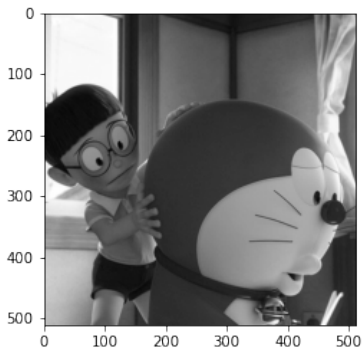
```
# Use the SVD function to compute SVD of the image
u, s, vh = np.linalg.svd(lenna, full_matrices=True)
r, c = lenna.shape
# Generate a rank-k approximation image
img = np.zeros((r, c))
uh = np.array(u).T
k = 0

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:



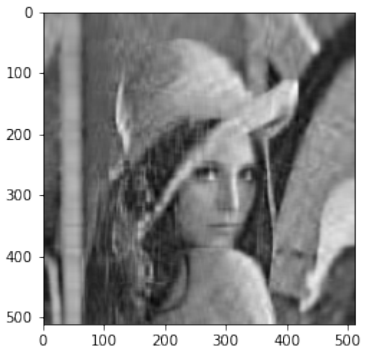
(a) Algorithm with k = 86



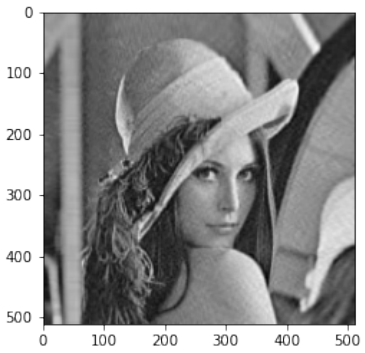
(b) Original

Figure 4: SVD algorithm results

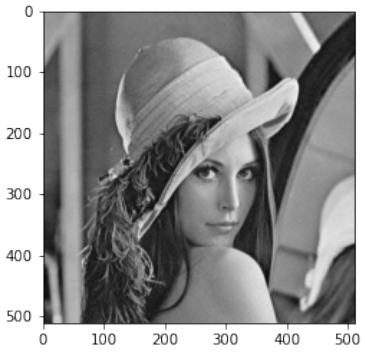
Now we use the same algorithm to test another image called 'Lenna'. The comparison between SVD compressed images and the original image are as follows: (# of singular values = 512)



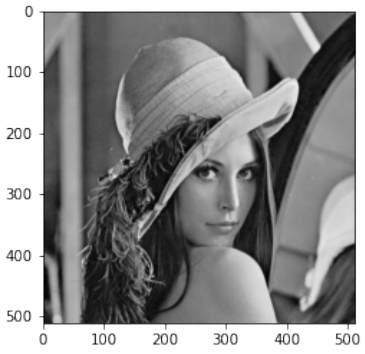
(a) k = 20



(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

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

where g_k is called a rank-k approximation of g. 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

$$D_{mn} = \sum_{j=k+1}^{r} \sigma_j^2 u_{im} v_{in}$$

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 $W(k,i) \equiv H_k\left(\frac{i}{N}\right)$ 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

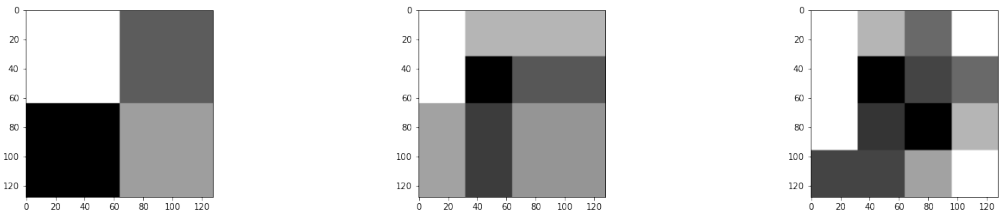
Code:

```
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')
```

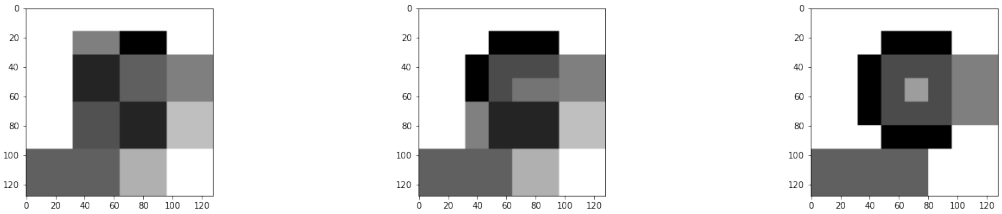
Figure 6: Haar Transform Codes

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 $= 128 \times 128$)

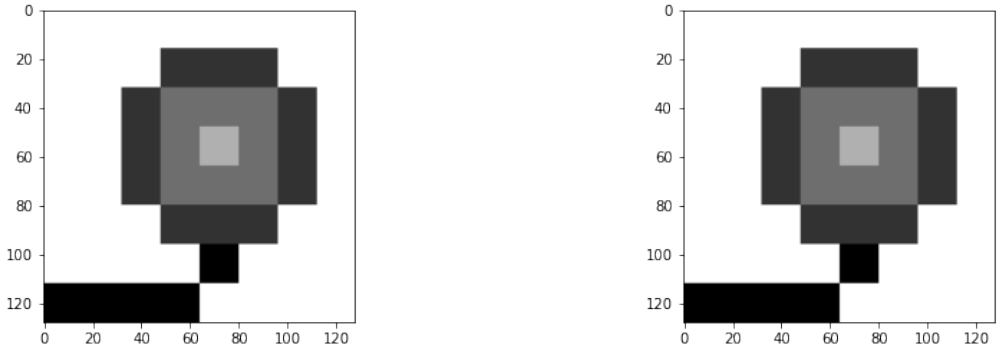


(a) Haar to 2 (b) Haar to 3 (c) Haar to 4



(d) Haar to 5 (e) Haar to 6 (f) Haar to 7

Figure 7: Balloon and its compressed images using Haar Transform



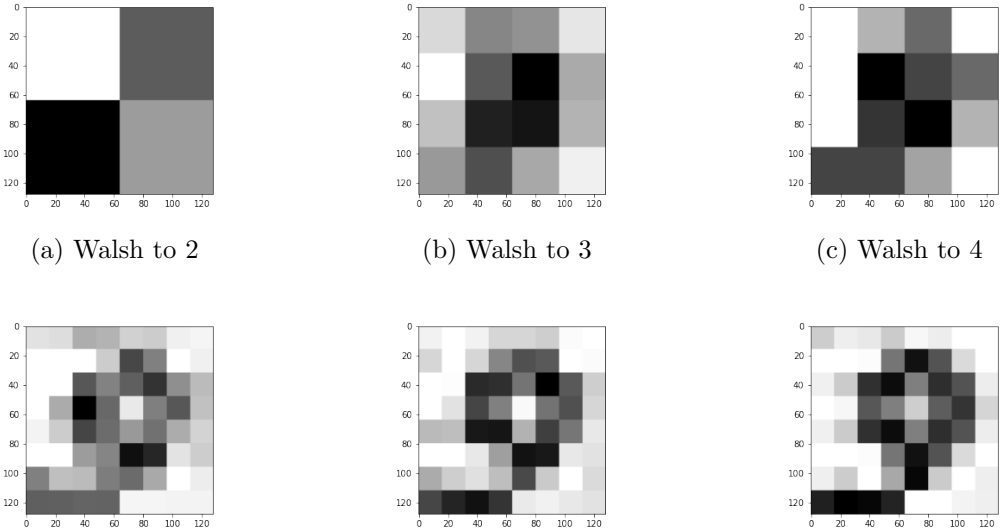
(a) Haar to 10 (b) Original

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)

2.3.4 Even Discrete Cosine Transform (EDCT)

3. Conclusion