Blind watermarking of images

Wenxuan Li

Content

Abstract

What we want to explain in the report is the **blind watermarking** technology of images. We used the following two methods to achieve this goal:

1. **DFT**: Discrete Fourier Transform
2. **DWT**: Discrete wavelet transform

These two methods have completed embedding a blind watermark to the image, and then we use a variety of attack methods to attack the image that has been embedded to the watermark. Re-extract the watermark after the attack and check the damage of the watermark. Based on this, we have completed the **robustness test** of the two methods, and we have the following conclusions:

1. DFT:
    a) The robustness increases with the increase of the energy coefficient α, while the **imperceptibility** decreases.
    b) When the energy coefficient α is high, the resistance to various attack methods is good. Poor anti-rotation ability, but we can still see the watermark content.
    c) When the energy coefficient is low, the anti-noise and anti-rotation ability is poor, and it has good resistance to other attack methods.
2. DWT
    a) The robustness is worse than that of the DFT algorithm. Poor resistance to blur and rotation, good resistance to other attack methods.
    b) The original image and the watermark are required to be square. If it is not, it will be treated as a square during the transformation process. The output result is gray, so the use limit is relatively large.

Key words: Blind watermark, DFT, DWT, Robustness test, imperceptibility

Introduction

With the development of the digital age, the Internet has become an indispensable part of everyone on this blue planet. Under this trend, we have entered the era of big data, and the scale of data in the network is expanding at an unimaginable rate. Then information security has become the most important issue. The embezzlement of images and documents is endless. The images on the Internet can be used as your own with a little adjustment. Therefore, digital watermarking technology has become very important, and our theme is also image digital watermarking technology.

Table 1 Comparison of ordinary watermark and blind watermark

Ordinary watermarks	Blind watermark
Based on spatial domain	Based on frequency domain
Can be disposed of by cutting and smearing	Resistant to common watermark attacks

Ordinary watermarks only modify and adjust the RGB of the image. Such watermarks not only affect the viewing effect of the image itself, but can also be removed in many ways, and the robustness is not good. The blind watermark can be changed by changing the frequency domain of the image. On the premise that the content of the image is not affected, the existence and integrity of the watermark can be guaranteed for a variety of attack methods. This is why we use blind watermarking.

In the follow-up content, we will introduce two methods: Discrete Fourier Transform (DFT) and Discrete Wavelet Transform (DWT), to implement the blind watermarking algorithm and perform robustness checks respectively.

DFT: Discrete Fourier Transform

1. Principle and nature

The Fourier transform is a linear combination of a trigonometric function (sine or cosine) or its integral satisfying certain conditions. In different research fields, Fourier transform has many different variant forms, such as continuous Fourier transform and discrete Fourier transform.

Because this application mainly uses two-dimensional Fourier transform and its properties, we explain in detail the two-dimensional mode of the Fourier transform.

$$F(u,v) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-i2\pi(\frac{ux}{M}+\frac{vy}{N})} \quad (1)$$

$$f(x,y) = \frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} F(u,v)e^{i2\pi(\frac{ux}{M}+\frac{vy}{N})} \quad (2)$$

$$\{x,u,u,v \in N | 0 \le x,u < M, 0 \le y,u < N, M\ and\ N\ \in N^*\}$$

In these two formulas:

$$f(x,y): The\ spatial\ domain$$

$$F(u,v): The\ frequency\ domain$$

The following are the main properties of the two-dimensional Fourier transform:

a) Separability

For a two-dimensional image $f(x,y)$ with M rows and N columns, a one-dimensional discrete Fourier transform of length N is performed according to the row queue variable Y, and then a Fourier transform of length M is performed on the variable X to obtain the Fourier transform of the image according to the column direction The result of the inner transformation is shown in the formula:

$$F(u,v) = \frac{1}{MN}\sum_{x=0}^{M-1}\left[\sum_{y=0}^{N-1} f(x,y)\exp\left(-j2\pi\frac{vy}{N}\right)\right]\exp\left(-j2\pi\frac{ux}{M}\right) \quad (3)$$

Decompose the above equation into the following two parts: $F(x,v)$, and $F(x,v)$ get from $F(x,v)$.

$$F(x,v) = \frac{1}{N}\sum_{y=0}^{N-1} f(x,y)\exp\left(-j2\pi\frac{vy}{N}\right)\ v = 0,1,2 \dots N-1 \tag{4}$$

$$F(u,v) = \frac{1}{M}\sum_{x=0}^{N-1} F(x,v)\exp\left(-j2\pi\frac{ux}{M}\right)\ u,v = 0,1,2 \dots M-1 \tag{5}$$

In the same way, to do the inverse Fourier transform, first do the one-dimensional inverse Fourier transform of the column direction, and then do the one-dimensional inverse Fourier transform of the rows, as shown in the following formula:

$$f(x,y) = \sum_{u=0}^{M-1}\left[\sum_{v=0}^{N-1} F(u,v)\exp\left(j2\pi\frac{vy}{N}\right)\right]\exp\left(j2\pi\frac{ux}{M}\right) \tag{6}$$

$$x = 0,1,2 \dots M-1; y = 0,1,2 \dots N-1$$

b) Periodicity

From the basic properties of the Fourier transform, we know that the frequency spectrum of a discrete signal is periodic. Discrete Fourier Transform DFT and its internal transformation are periodic at n times of Fourier Transform.

For the one-dimensional Fourier transform, we have:

$$F(u) = F(u \pm kn), k = 0,1,2 \dots \tag{7}$$

For the two-dimensional Fourier transform, we have:

$$F(u,v) = F(u \pm in, v \pm jn), i = 0,1,2 \dots, j = 0,1,2 \dots \tag{8}$$

c) Conjugate symmetry

Although $F(u,v)$ repeats for an infinite number of u and v values, you can get $f(x,y)$ from $F(u,v)$ only according to the N values in any period. $F(u,v)$ can be completely determined in the frequency domain with only a transformation in one cycle. The same conclusion holds for $f(x,y)$ in the airspace.

When the real parts of two complex numbers are equal and the imaginary parts are opposite to each other, the two complex numbers are called mutually conjugate complex numbers. If $f(x,y)$ is a real function, its Fourier transform has conjugate symmetry.

$$F(u,v) = F(-u,-v) \tag{9}$$

$$|F(u,v)| = |F(-u,-v)| \tag{10}$$

Among them, $F(u,v)$ is the complex conjugate of $F(u,v)$

d) Gyration

If $f(x,y)$ is rotated by an angle, then the Fourier transform of the rotated image of $f(x,y)$ is also rotated by the same angle. Rewrite the plane Cartesian coordinates to polar coordinates:

$$\begin{cases} x = r\,cos\theta \\ y = r\,sin\theta \end{cases} and \begin{cases} u = \omega\,cos\varphi \\ v = \omega\,sin\varphi \end{cases} \tag{11}$$

By the formula (11), we can get:

$$f(x,y) \rightarrow f(r,\theta) \Leftrightarrow F(\omega,\varphi) \tag{12}$$

If $f(x,y)$ rotates ω, then $f(u,v)$ rotates by the same angle. Fourier transform:

$$f(r,\theta+\theta_0) \Leftrightarrow F(\omega,\varphi\theta_0) \tag{13}$$

2. Flow chart and explanation

We use MATLAB to implement the DFT algorithm and draw a flow chart to clarify the idea. The image below is our flow chart:

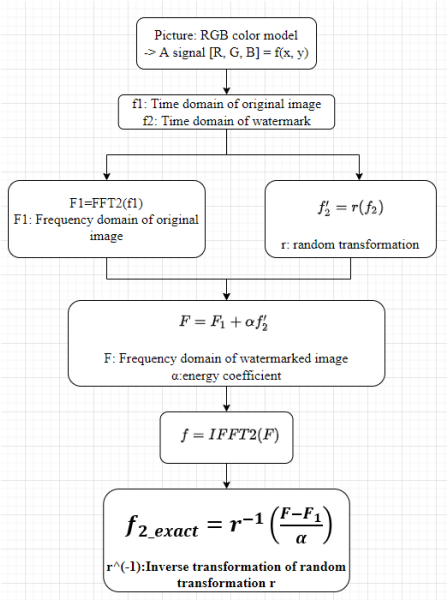

Figure 1 DFT algorithm implementation flowchart

a) For an image, we assume that the RGB color model is used, so it can be simply regarded as an MxNx3 matrix. Then, we can regard this image as a signal and get the spatial domain representation of this signal. Then, the discrete Fourier transform is applied to obtain its frequency domain representation $F(u, v)$.

$$[R, G, B] = f(x, y) \tag{14}$$

b) For the watermark and the original image, it is easy to obtain the spatial domain representation f1 and f2

$$f_1: Spatial\ domain\ of\ original\ image$$

$$f_2: Spatial\ domain\ of\ the\ watermark$$

c) Perform Fourier transform on the original image to get the frequency domain

$$F_1 = FFT2(f_1) \tag{15}$$

$$F_1: The\ frequency\ domain\ of\ original\ image$$

d) In order to make the information of the watermark evenly distributed in the frequency domain, random transformation r(f) is introduced to transform the spatial-domain watermark

$$f_2' = r(f_2) \tag{16}$$

$$f_2' : The\ frequency\ domain\ of\ watermark$$

$$r() : The\ random\ transformation$$

e) Introduce the energy coefficient α to merge the frequency domain of the watermark and the original image

$$F = F_1 + \alpha(f_2') \tag{17}$$

$$\boldsymbol{F} : The\ frequency\ domain\ of\ watermarked\ image$$

$$\alpha : energy\ coefficient$$

f) Through the inverse Fourier transform, the watermarked Image can be obtained

$$f = IFFT2(F) \tag{18}$$

$$F : The\ frequency\ domain\ of\ watermarker\ image$$

$$f : The\ watermarked\ image$$

g) Use the frequency domain of the watermarked Image to subtract the frequency domain of the original Image, and perform the inverse transform of the random transformation to get the watermark spatial domain

$$f_{2_{exact}} = r^{-1}\left(\frac{F - F_1}{\alpha}\right) \tag{19}$$

$$f_{2_{exact}} : The\ spatial\ domain\ of\ exacted\ watermark$$

$$r^{-1} : The\ inverse\ transformation\ of\ random\ transformation$$

3. Algorithm implementation

The following is our step-by-step process of using DFT to embed and extract blind watermarks to images.

a) First check the unprocessed original image and watermark

Figure 2 Unprocessed original image

Figure 3 Unprocessed watermark

b) The data type of read image is uint8, which domain is integers from 0 to 255. But the data type usually used for calculation in MATLAB is double, which domain is from 0 to 1. So, we need to adjust the data type and value.

Figure 4 Processed original image

Figure 5 Processed watermark

Through the comparison of figures 3 and 4 with figures 1 and 2, we found that there is almost no difference between the images before and after processing, which is invisible to the naked eye.

c) Because the image signal is a two-dimensional discrete signal, we use a two-dimensional discrete Fourier transform. In addition, we use Fast Fourier Transform instead of Discrete Fourier Transform. According to formula 15, we use the $fft2$ function in MATLAB to complete the two-dimensional fast Fourier transform.

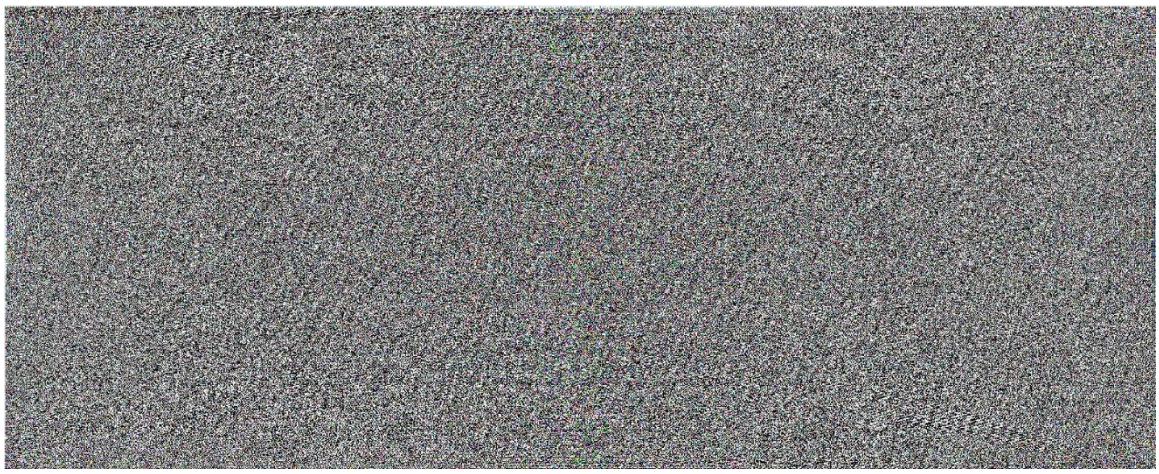

Figure 6 Frequency domain of original image

d) In addition to random transformation, we also need watermark symmetry to ensure that the watermarked frequency domain image will not lose important information of the watermark during the inverse Fourier transform process, and maximize the concealment of the blind watermark.

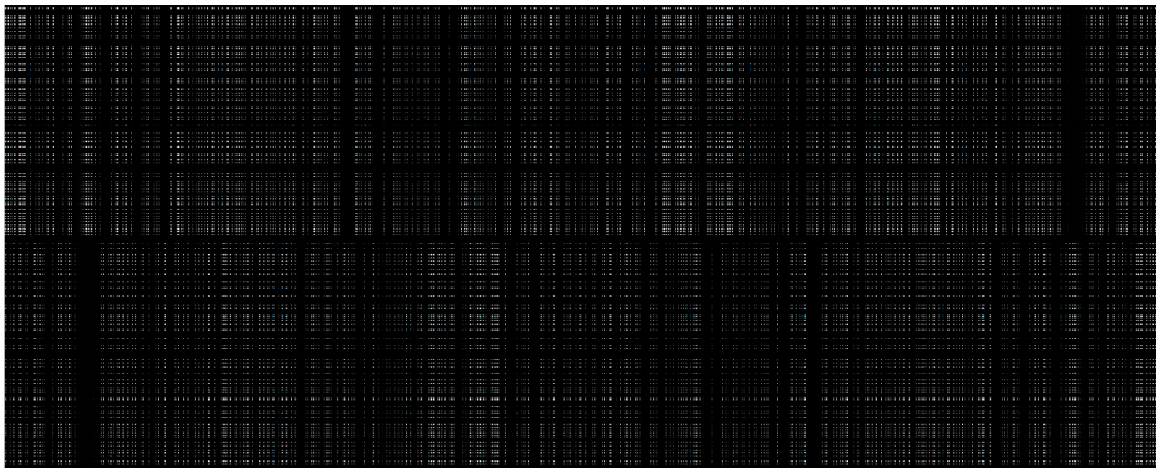

Figure 7 Frequency domain of encoded watermark

e) We set the energy coefficient to 1 and perform watermark embedding according to formula 17.

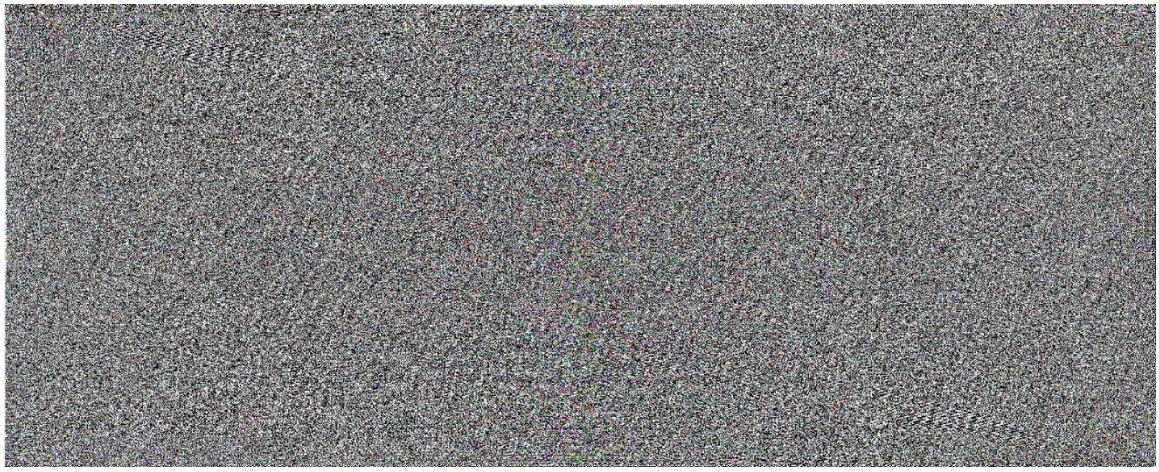

Figure 8 Frequency domain of watermarked image

f) According to formula 18, we perform inverse Fourier transform on the frequency domain image of the watermarked image to obtain the spatial domain.

Figure 9 Watermarked image

By comparing Figure 8 with Figures 1 and 3, we find that there is no obvious change in the watermarked image.

g) According to formula 19, the embedded watermark is obtained by inverse random transformation

Figure 10 Extracted watermark

So far, we have completed the realization of the blind watermarking algorithm using DFT. Next, we try to use the DWT method to implement the blind watermarking algorithm.

DWT: Discrete Wavelet Transform

1. Principle and nature

DWT stands for Discrete Wavelet Transform, which is a new spectrum analysis tool used for the scale and translation of discrete basic wavelets. It can not only test the frequency domain characteristics of the local frequency domain process, but also the spatial domain characteristics of the local frequency domain process, so even those non-stationary processes can be transformed and processed well. For images, it can convert the image into a series of wavelet coefficients, and effectively compress and store these coefficients. In addition, the rough edge of wavelet eliminates the square effect common in DCT compression, so as to better restore and represent the image.

Discrete wavelet transform:

$$W_f(a,b) =< f, \psi_{a,b}(t) > \tag{20}$$

Reconstruction formula:

$$f(t) = \sum_{a,b\in Z} < f, \psi a, b > \psi_{a,b}(t) \tag{21}$$

In formulas 20 and 21:

$$a: Scale\ parameters$$

$$b: Translation\ parameters$$

2. Flow chart and explanation

The implementation of this method is through Python, and we divide the whole method into two parts: embedding and extraction. We will show the flowchart and explanation separately.

A. Watermark embedding

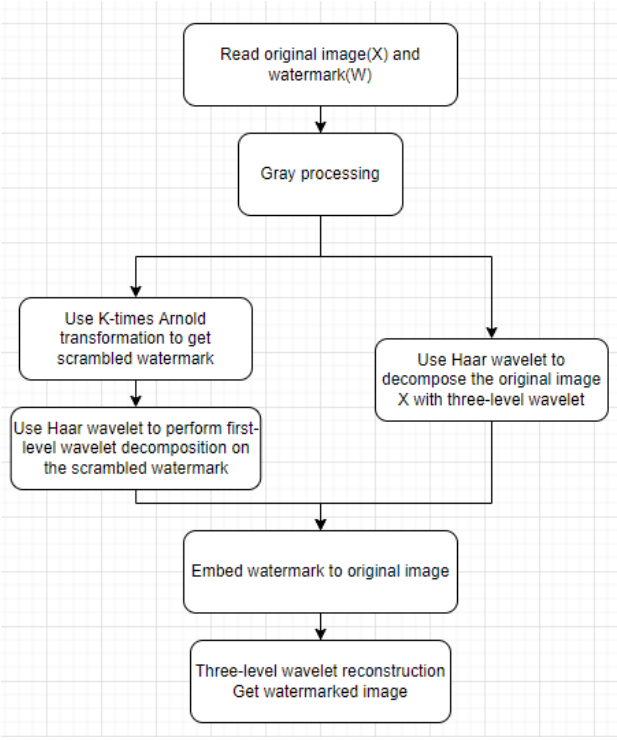

Figure 11 DWT algorithm embedding implementation flowchart

a) Read the original image X and the watermark image W separately.

b) Arnold performs $K-times$ transformations on the watermark image W, and the scrambled watermark is W. The coefficient and the number of transformations is stored as the key.

c) The scrambled watermark W_a is decomposed by $Haar\ wavelet$ to obtain an approximate sub-image ($ca1$) and three detailed sub-images ($ch1, cv1\ and\ cd1$) at first-order resolution.

d) The original image X is decomposed by $Haar\ wavelet$, and several detailed sub-images and an approximate sub-image of different resolution levels are obtained. The specific decomposition subgraphs are $CA3, CH3, CV3, CD3, CH2, CV2, CD2, CH\ 1, CV1, CD1$.

e) Embed each sub-image after W_a first-order wavelet decomposition into the corresponding sub-image after third-order wavelet decomposition of the original image, and select different coefficients for different groups:

$$\begin{cases} CA3' = CA3 + a_1 * ca1 \\ CH3' = CH3 + a_2 * ch1 \\ CV3' = CV3 + a_3 * cv1 \\ CD3' = CD3 + a_4 * cd1 \end{cases} \tag{22}$$

For the system of equations in Equation 22:

$$a_1, a_2, a_3, a_4: The\ corresponding\ weighting\ coefficients$$

$$CA3', CH3', CV3', CD3': The\ corresponding\ subgraphs\ of\ the\ watermarked\ image$$

f) The watermark embedded image is obtained by three-level wavelet reconstruction of wavelet coefficients.

B. Watermark extraction

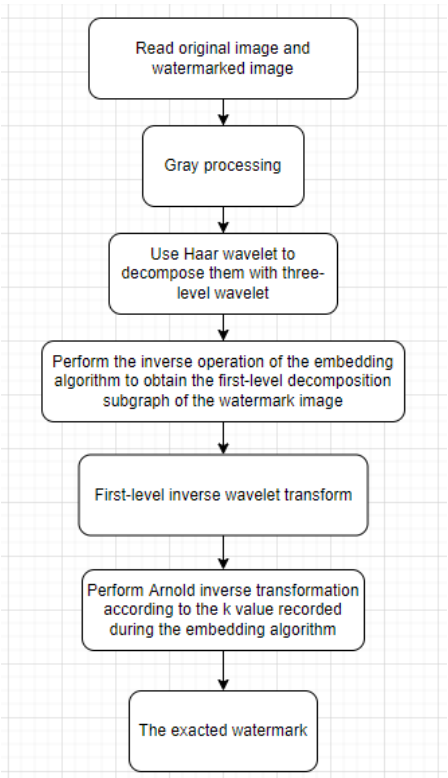

Figure 12 DWT algorithm extraction implementation flowchart

a) Read the original image X and the watermarked image separately.

b) The original image X and the embedded image are respectively subjected to $Haar\ wavelet$ decomposition to obtain multiple detailed sub-images and an approximate sub-image with different resolution levels.

c) Use the subgraphs of watermarked image and original image to extract the watermark image's first level decomposition subgraph: $ac1, ch1, cv1\ and\ cd1$. Carry on the inverse wavelet transform of the extracted wavelet coefficients to get the watermark scrambled image.

$$\begin{cases} ca1 = \frac{CA3' - CA3}{a_1} \\ ch1 = \frac{CH3' - CH3}{a_2} \\ cv1 = \frac{CV3' - CV3}{a_3} \\ cd1 = \frac{CD3' - CD3}{a_4} \end{cases} \tag{23}$$

d) According to the embedded key, the watermark W is extracted through Arnold inverse transformation.

3. Algorithm implementation

The following is a step-by-step process for embedding and extracting blind watermarks in images using DWT. We are also divided into two parts: embedding and extraction

A. Watermark embedding

a) Read the original image and watermark

Figure 13 Unprocessed original image

Figure 14 Unprocessed watermark

b) Gray processing: using COLOR_RGB2GRAY

Figure 15 Processed original image

Figure 16 Processed watermark

c) K-times Arnold transformation on the watermark (We set key=10): Using the $Arnold()$ function

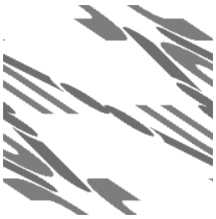

Figure 17 Watermark after K-times Arnold transformation

d) Three-level wavelet transform of original image and one-level wavelet transform of watermark image

```
# original image three-order wavelet transform
c = pywt.wavedec2(Img1,'db2',level=3)
[cl,(cH3,cV3,cD3),(cH2,cV2,cD2),(cH1,cV1,cD1)] = c
# watermark image first order wavelet transform
waterTmg1 = cv2.resize(waterTmg1,(101,101))
d = pywt.wavedec2(waterTmg1,'db2',level=1)
[ca1,(ch1,cv1,cd1)] = d
```

Figure 18 Wavelet transform code

e) Embed watermark to the original image and use three-level wavelet reconstruction to get the watermarked image. Based on formula 22

Figure 19 Watermarked image

B. Watermark extraction

a) Read original image and watermarked image

Figure 20 Unprocessed original image

Figure 21 Unprocessed watermarked image

b) Gray processing: using COLOR_RGB2GRAY

Figure 22 Processed original image

Figure 23 Processed watermarked image

c) Use $Haar\ wavelet$ to decompose the original image and watermarked image with three-level wavelet

```
# watermarked image three-order wavelet transform
c = pywt.wavedec2(Img,'db2',level=3)
[cl,(cH3,cV3,cD3),(cH2,cV2,cD2),(cH1,cV1,cD1)] = c

# Original image three-order wavelet transform
d = pywt.wavedec2(Img1,'db2',level=3)
[dl,(dH3,dV3,dD3),(dH2,dV2,dD2),(dH1,dV1,dD1)] = d
```

Figure 24 Three-level wavelet transform code

d) According to the proportion coefficient of extraction and image reconstruction. Based on the formula 23

Figure 25 Watermark before Arnold inverse transform

e) Perform K times Arnold inverse transformation according to the k value recorded in the embedding algorithm to obtain the watermark

Figure 26 Extracted watermark

Robustness test

The purpose of using different algorithms to realize the blind watermarking technology of images is to ensure the stability of the watermark under the premise that the viewing of images is not greatly affected. Then the robustness test of the watermark can help us to evaluate. Because the two methods use different compilers, the image attack methods used will be different. The attack methods used are shown in the following table:

Table 2 Image attack method used

Attack method
Add noise
Crop
Rotate
Brighten
Blur
Random line

In Table 2, the method with a white background is the attack method used in the robustness test of the two algorithms. The gray background is only used in the robustness test of the DWT algorithm. Next, we will conduct robustness tests on DFT and DWT in turn.

A. DFT algorithm

Because the energy coefficient α in the algorithm is related to the effect of watermark embedding, we set the $\alpha = 100$ and $\alpha = 1000$ respectively for comparison test.

a) Robustness and imperceptibility

Figure 27 Watermark image with $\alpha = 100$

Figure 28 Watermarked image with $\alpha = 1000$

Through Figures 27 and 28, we can see that when $\alpha = 1000$, the image is affected a bit after embedding the watermark. Although we can still clearly see the information in the image, the imperceptibility of the watermark is poor, and it is obvious that the image has been processed. The watermark imperceptibility of the watermarked image with an $\alpha = 100$ is better.

Figure 29 Extracted watermark with $\alpha = 100$

Figure 30 Extracted watermark with $\alpha = 1000$

By comparing Figure 29 and Figure 30, when the $\alpha = 1000$, the extracted watermark has a higher definition.

According to formula 17, we can know that the energy coefficient determines the embedding strength of the watermark in the frequency domain during the watermark embedding process. When α is larger, the frequency domain of the original image is affected by the larger the frequency domain of the watermark. Therefore, the watermarked image will become blurred, and the extracted watermark will become clearer. However, we are currently unable to generalize the relationship between robustness and imperceptibility, and we still need follow-up tests.

b) Noise attack

When some form of noise is added to the image, the frequency domain of the image will be affected, and the extracted watermark will also be affected to a certain extent. What we are using is salt and pepper noise, and the noise density is not changed, which defaults to 0.05.

```
%% Robustness test: adding noise
    watermarked_img_noise=imnoise(watermarked_img,'salt & pepper');
    figure,imshow(watermarked_img_noise);title('The watermarked img adding noise');
```

Figure 31 Noise attack code in MATLAB

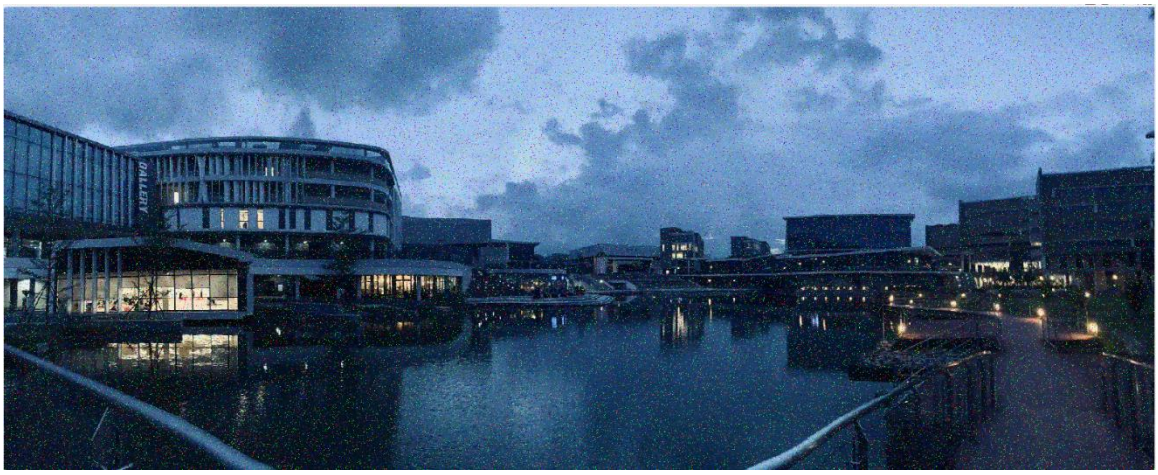

Figure 32 Watermarked image attacked by noise with $\alpha = 100$

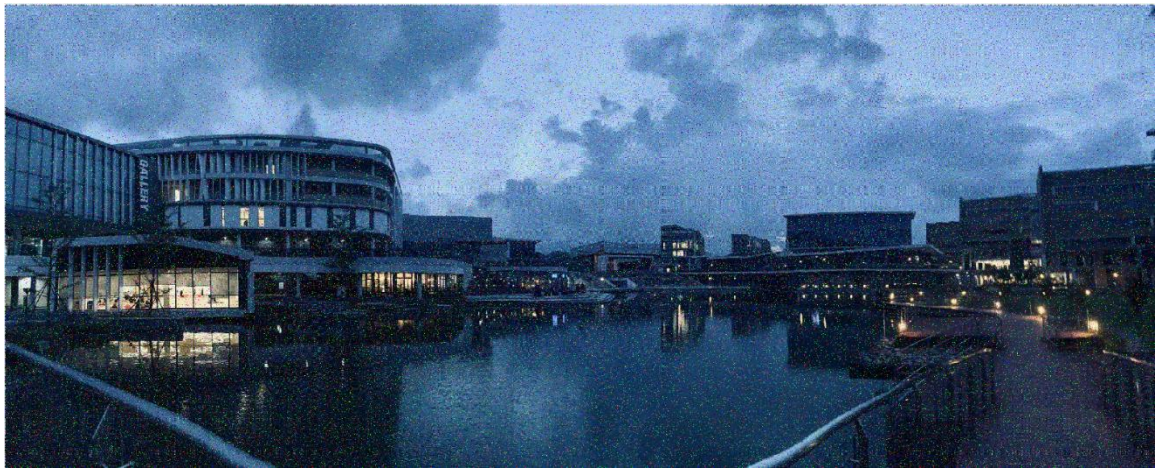

Figure 33 Watermarked image attacked by noise with $\alpha = 1000$

Because the size of the two watermarked images is the same, the attacks received in Figure 32 and Figure 33 are the same when the noise density is both 0.05.

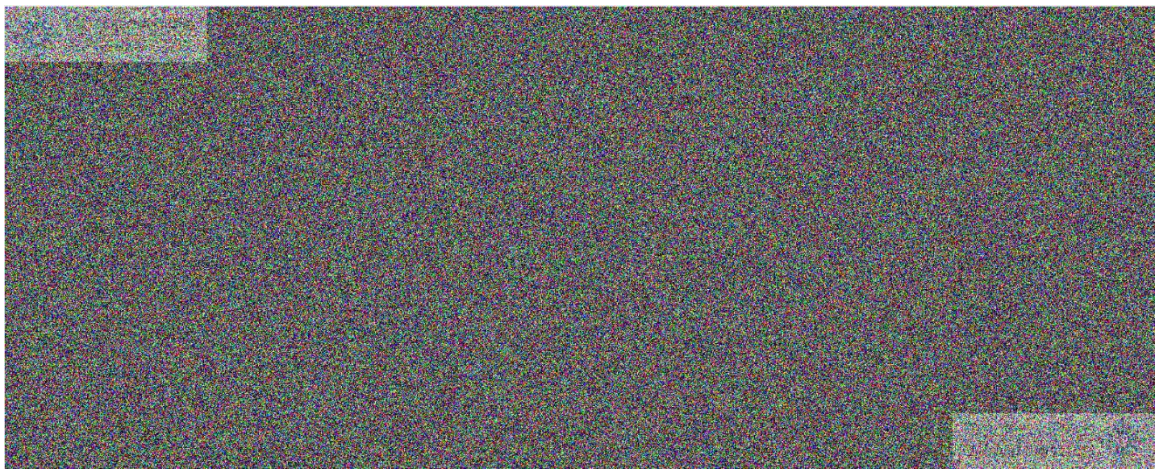

Figure 34 Extracted watermark attacked by noise with $\alpha = 100$

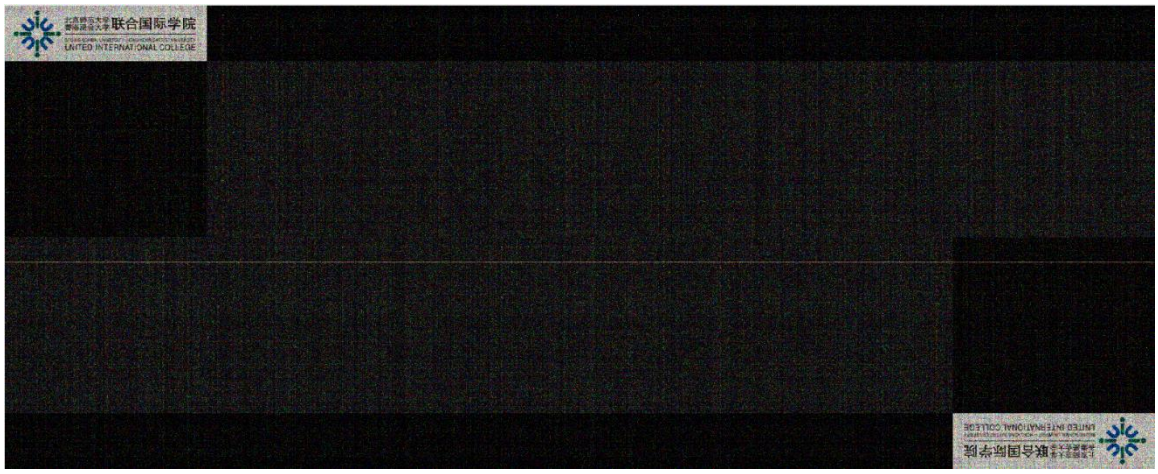

Figure 35 Extracted watermark attacked by noise with $\alpha = 1000$

By comparing Figure 34 and Figure 35, we find that when $\alpha = 1000$, the extracted watermark is clearer. When $\alpha = 100$, after the noise attack, the extracted watermark can hardly read the watermark information. Therefore, we believe that the greater the α, the stronger the noise resistance of the watermark.

c) Cropping attack

```
%% Robustness test: cropping
    watermarked_img_cropping=im2uint8(watermarked_img);
    watermarked_img_cropping_r= watermarked_img_cropping(:,:,1);
    watermarked_img_cropping_r(1:320,1:796.5)=255;
    watermarked_img_cropping_g =watermarked_img_cropping(:,:,2);
    watermarked_img_cropping_g(1:320,1:796.5)=255;
    watermarked_img_cropping_b =watermarked_img_cropping(:,:,3);
    watermarked_img_cropping_b(1:320,1:796.5)=255;
    watermarked_img_cropping(:,:,1) = watermarked_img_cropping_r;
    watermarked_img_cropping(:,:,2) = watermarked_img_cropping_g;
    watermarked_img_cropping(:,:,3) = watermarked_img_cropping_b;
    figure,imshow(watermarked_img_cropping);title('watermarked_img_cropping');
```

Figure 36 Cropping attack code in MATLA

The principle of cropping is to set the RGB in the upper left corner to $(255, 255, 255)$ so that that part becomes white, which looks like it is cut off.

Figure 37 Watermarked image attacked by cropping with $\alpha = 100$

Figure 38 Watermarked image attacked by cropping with $\alpha = 1000$

Comparing Figures 37 and 38 with Figures 27 and 28, after cropping, only the upper left corner of the cropped part changes. There is no change in other parts of the image.

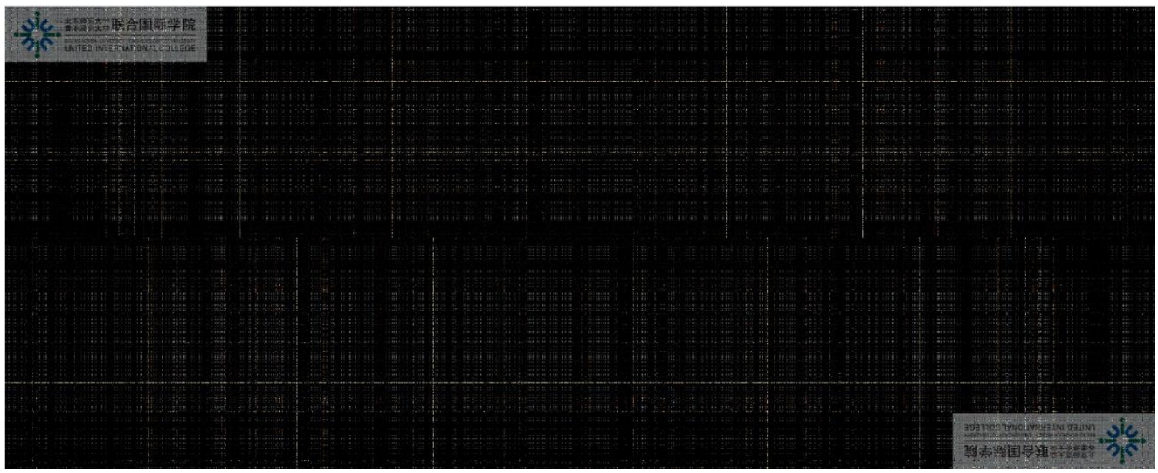

Figure 39 Extracted watermark attacked by cropping with $\alpha = 100$

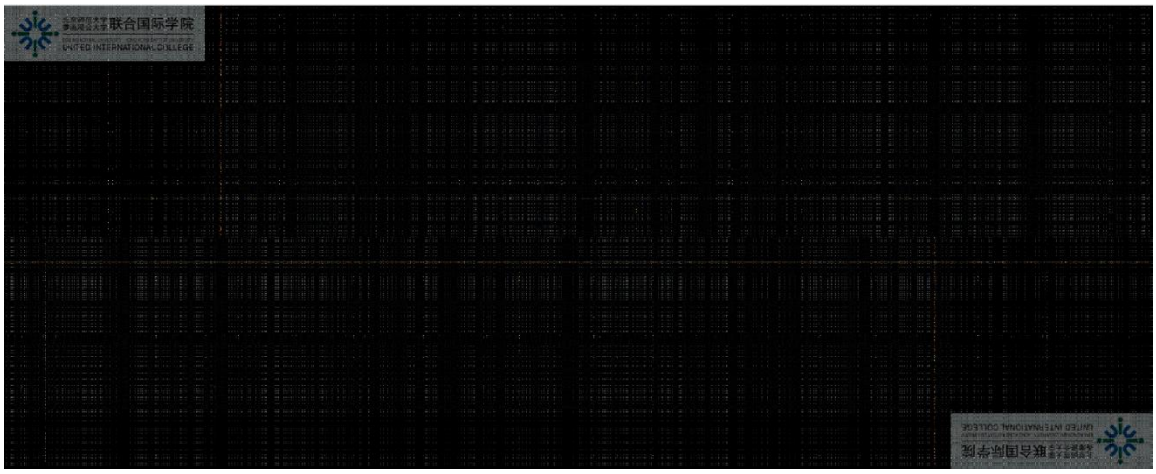

Figure 40 Extracted watermark attacked by cropping with $\alpha = 1000$

After comparison, the clarity of the watermark information in the two extracted watermarks is close. When $\alpha = 1000$, there are fewer noises in other parts of the image.

d) Rotating attack

```
%% Robustness test: rotateing
    watermarked_img_rotateing=imrotate(im2double(watermarked_img),15,'nearest','crop');
    figure,imshow(watermarked_img_rotateing);title('watermarked_img_rotateing');
```

Figure 41 Rotating attack code in MATLA

We use the $imrotate$ function, and set the rotation direction to counterclockwise, and the rotation angle is set to 15 degrees.

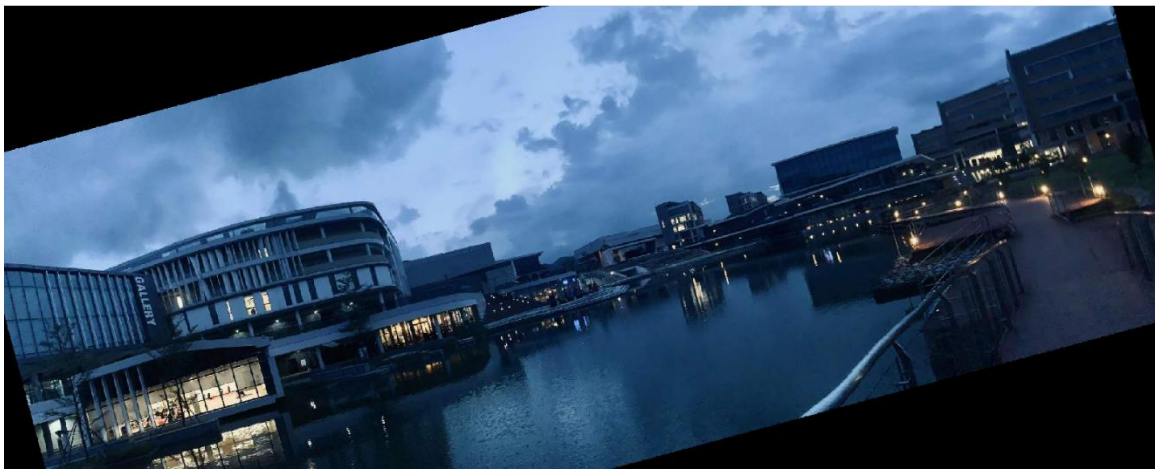

Figure 42 Watermarked image attacked by rotating with $\alpha = 100$

Figure 43 Watermarked image attacked by rotating with $\alpha = 1000$

The two images have been processed in the same way, and the other image information has not changed due to rotating.

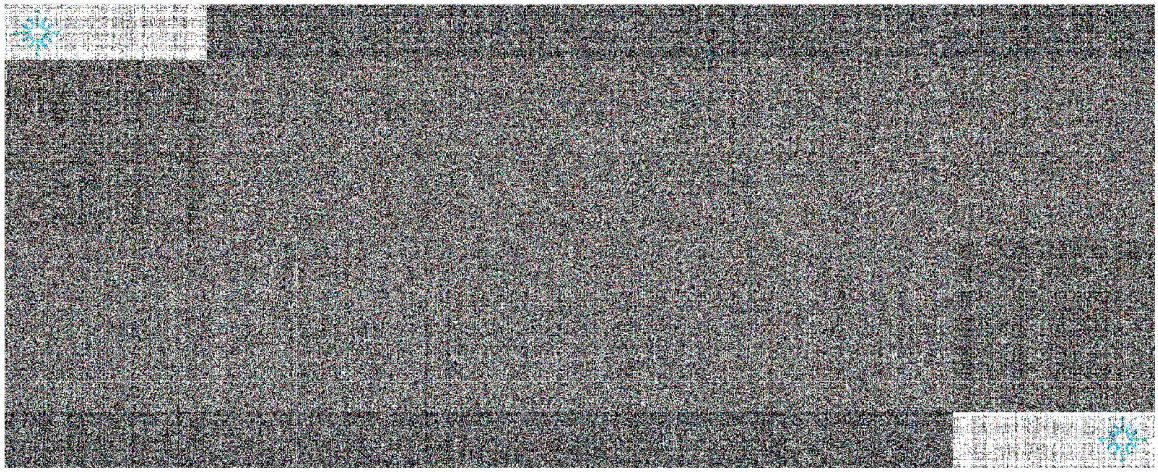

Figure 44 Extracted watermark attacked by rotating with $\alpha = 100$

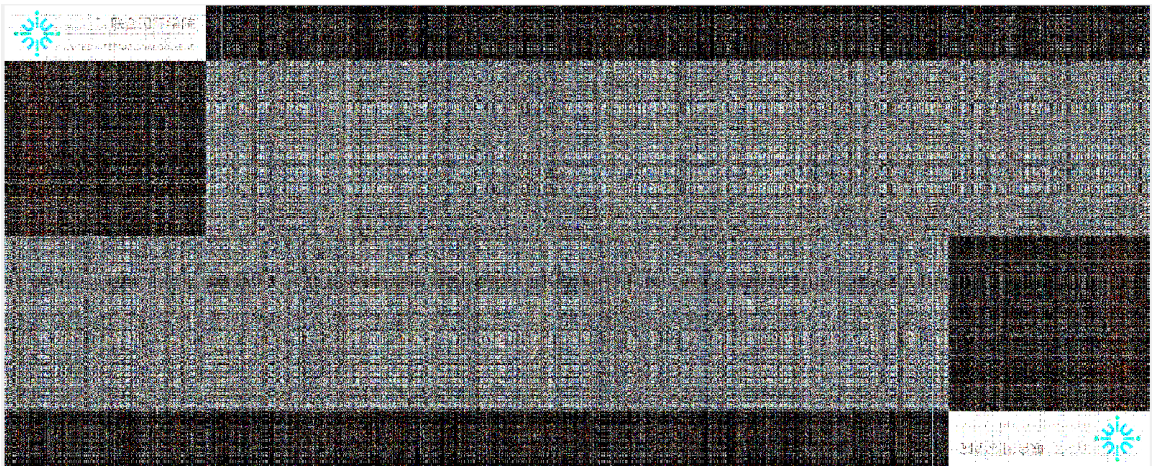

Figure 45 Extracted watermark attacked by rotating with $\alpha = 1000$

From Figure 44 and Figure 45, the watermark information is indeed very serious after being rotated by 15 degrees. When $\alpha = 1000$, the watermark part is relatively clear, and the watermark information can basically be identified.

e) Brightness attack

```
%% Robustness test: brightness increased by 10%
    watermarked_img_brightness10=1.1*im2double(watermarked_img);
    figure,imshow(watermarked_img_brightness10);title('watermarked_img_brightness10');
```

Figure 46 Rotating attack code in MATLA

We increase each value in the spatial domain of the image by 10% to achieve a 10% brightening effect.

Figure 47 Watermarked image attacked by brighten with $\alpha = 100$

Figure 48 Watermarked image attacked by brighten with $\alpha = 1000$

Compared with before processing, the brightness of the screen in Figure 47 and Figure 48 has been improved. When $\alpha = 1000$, the blur in the image still exists.

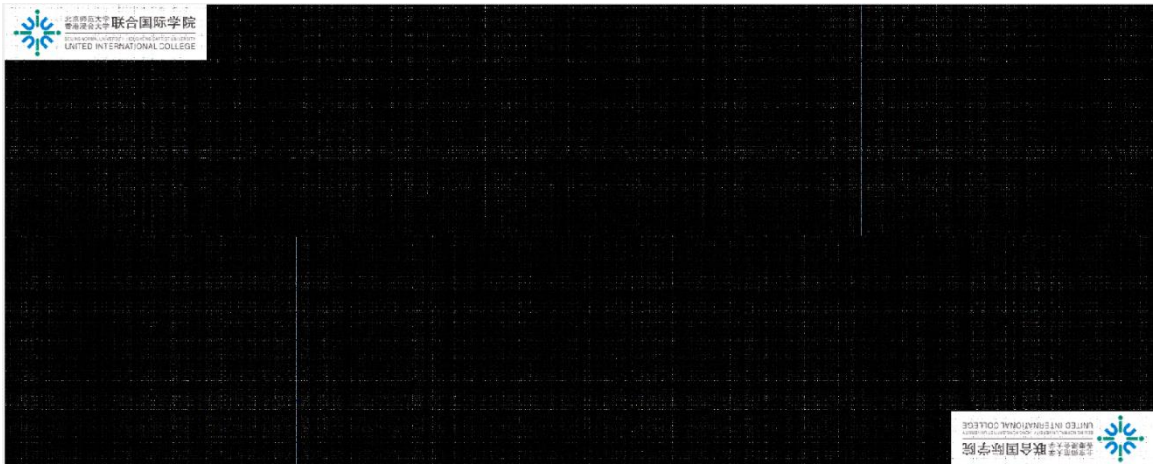

Figure 49 Extracted watermark attacked by brighten with $\alpha = 100$

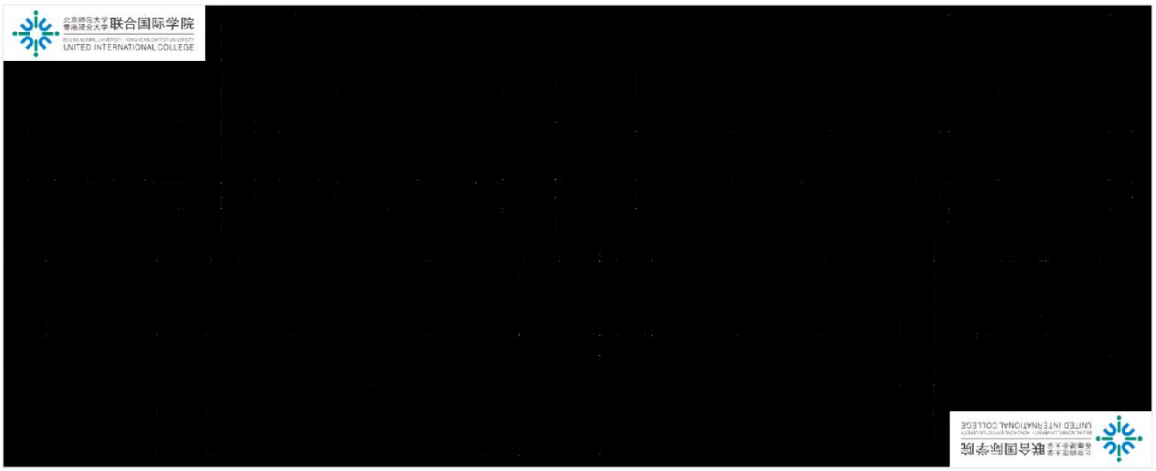

Figure 50 Extracted watermark attacked by brighten with $\alpha = 1000$

According to the comparison between Figure 49 and Figure 50, we find that when $\alpha = 100$, there is noise in the extracted watermark. When $\alpha = 1000$, the extracted watermark is basically the same as the watermark extracted without attack.

Up to now, we have completed the robustness test of the DFT algorithm. After our test, we found the following conclusions:

a) Energy coefficient α is positively correlated with robustness and negatively correlated with imperceptibility

b) Rotation attacks are the most aggressive to the extracted watermark, followed by noise attacks

c) Cropping attack and brightness attack have minimal impact on watermark extraction

B. DWT algorithm

According to our preliminary test, the coefficients in the DWT algorithm have little effect on the watermark result. In the test, we use $a_1 = 0.1, a_2 = 0.2, a_3 = 0.1, a_4 = 0.1$ as an example.

Figure 51 Watermarked image in DWT

Figure 52 Extracted watermark in DWT

a) Noise attack

```
14  if type == "saltnoise":
15      for k in range(1000):
16          i = int(np.random.random() * img.shape[1])
17          j = int(np.random.random() * img.shape[0])
18          if img.ndim == 2:
19              img[j, i] = 255
20          elif img.ndim == 3:
21              img[j, i, 0] = 255
22              img[j, i, 1] = 255
23              img[j, i, 2] = 255
24      return img
```

Figure 53 Noise attack code in Python

When using Python to attack the DWT algorithm in real-time salt and pepper noise, we set to randomly select 1000 pixels in the spatial domain of the image and set their color to white, which RGB is $(255, 255, 255)$.

Figure 54 Watermarked image attacked by noise in DWT

Figure 55 Extracted watermark attacked by noise in DWT

By comparing Fig. 55 with Fig. 52, the extracted watermark image becomes blurred. However, the watermark information can still be seen at present, and the watermark is still identifiable.

b) Brightness attack

By adjusting the RGB value to achieve the brightening effect

Figure 56 Watermarked image attacked by brighten in DWT

Figure 57 Extracted watermark attacked by brighten in DWT

After comparing Figure 57 and Figure 52, we find that the brightness attack has little impact on the DWT algorithm.

c) Rotating attack

We write a function: $rotate\_about\_center$ to complete the image rotation around the center. But this method can only complete 90-degree, 180-degree, 270-degree rotation. Here we choose a 90-degree rotation for testing.

Figure 58 Watermarked image attacked by rotating in DWT

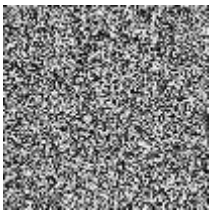

Figure 59 Extracted watermark attacked by brighten in DWT

From the results, the watermark extracted after 90-degree rotation is completely invisible, and no watermark information can be seen.

d) Blur attack

```
if type == "blur":
    kernel = np.ones((3, 3), np.float32) / 9
    return cv2.filter2D(img, -1, kernel)
```

Figure 60 Blur attack code in Python

We blur the image 9 times for a blur attack.

Figure 61 Watermarked image attacked by blur in DWT

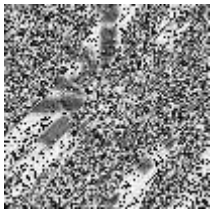

Figure 62 Extracted watermark attacked by blur in DWT

After comparing Figure 62 and Figure 52, the watermark information is severely missing, and the watermark is basically invisible.

e) Cropping attack

The idea and principle of cropping attack in Python are similar to those in MTATLAB. Change the RGB value at a specific position to $(255, 255, 255)$

Figure 63 Watermarked image attacked by cropping in DWT

Figure 64 Extracted watermark attacked by cropping in DWT

By comparing Figure 64 and Figure 52, cropping has little effect on the extracted watermark.

f) Graffiti attack

```
27  if type == "randline":
28      cv2.rectangle(img, (384, 0), (510, 128), (0, 255, 0), 3)
29      cv2.rectangle(img, (0, 0), (300, 128), (255, 0, 0), 3)
30      cv2.line(img, (0, 0), (511, 511), (255, 0, 0), 5)
31      cv2.line(img, (0, 511), (511, 0), (255, 0, 255), 5)
32
33      return img
```

Figure 65 Graffiti attack code in Python

We drew two rectangles and two lines in the watermarked image, both of which have specified colors.

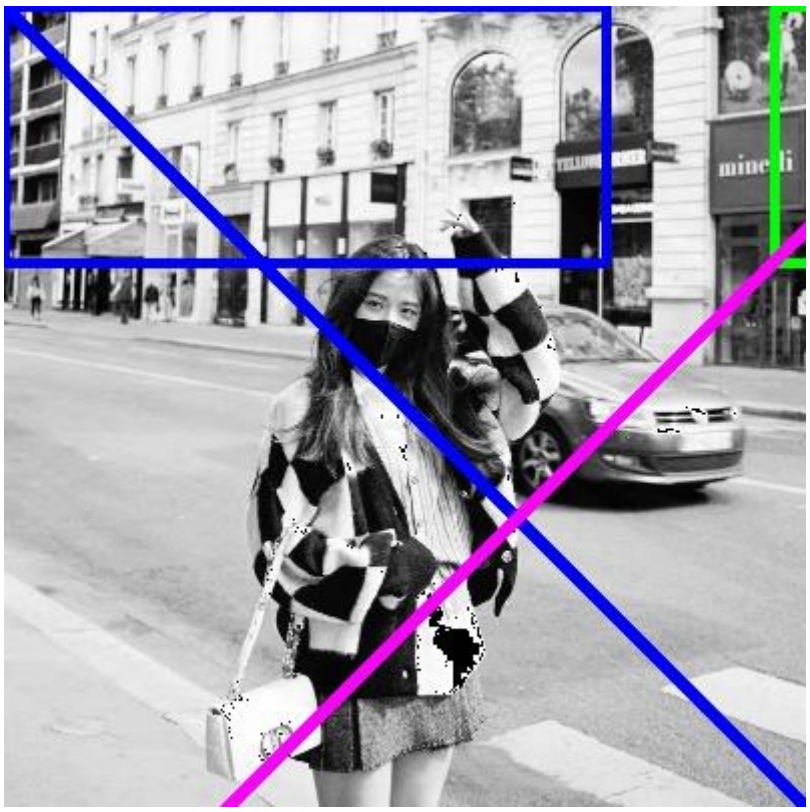

Figure 66 Watermarked image attacked by graffitiing in DWT

Figure 67 Extracted watermark attacked by graffitiing in DWT

We can see these different colored lines on the image. The watermark extracted from the image has been affected a bit. We can still clearly see the watermark information.

So far, we have completed the robustness test of the DWT algorithm. We came to the following conclusions:

a) The resistance to blur attacks and rotating attacks is extremely low

b) The resistance to noise attacks is average. Good resistance to other attack methods

Conclusion

1. DFT is better than DWT

According to our conclusions about the implementation of DFT and DWT algorithms and robustness experiments, we summarize the characteristics of the two algorithms.

Table 2 DFT and DWT comparison

	DFT	**DWT**
Basic principle	Discrete Fourier Transform	Discrete Wavelet Transform
Image requirements	Color Can be any shape	Output is only gray Can only be square
Robustness	Proportional to energy coefficient α larger, robustness better	Very low resistance to rotation and blur

From this, we come to the conclusion that DFT is more advantageous to the blind watermarking algorithm of images.

2. Reasons why DFT has better results

a) Encoding watermark—Random transformation:

To distribute the watermark information as evenly as possible in the frequency domain, so that it is not easy for attacking images to have a greater impact on the watermark.

b) Encoding watermark—Watermark symmetry

Ensure that the watermarked frequency domain image will not lose the important information of the watermark during the inverse Fourier transform process, and hide the blind watermark to the greatest extent

c) Embed redundant information (like noise) through the frequency domain

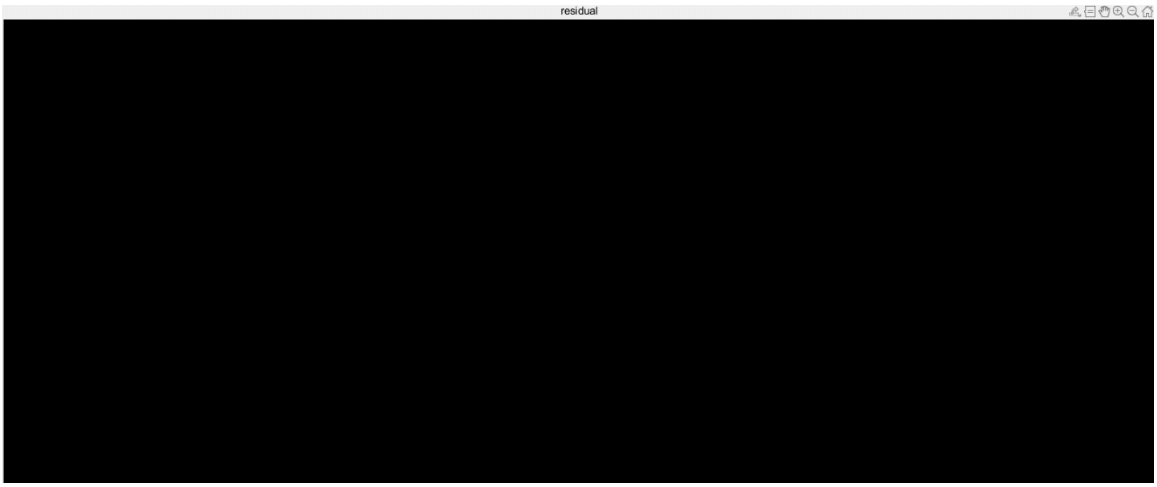

Figure 68 Difference of the frequency domain of watermarked image and original image

Figure 68 shows the residual obtained by subtracting the frequency domain of the original image from the frequency domain of the watermark image in the spatial domain. We can see that it is completely black because it is blank. We embed redundant information (such as noise) through the frequency domain. These voices spread throughout the map. So, they are not easily destroyed in the spatial domain.

Reference

1. 张燕华.(2016).基于分数阶傅里叶变换的数字水印算法研究(硕士学位论文,苏州大学).

   https://kns.cnki.net/KCMS/detail/detail.aspx?dbname=CMFD201701&filename=1016223365.nh

2. 王坤.(2010).基于 DWT 的彩色图像数字水印算法的研究(硕士学位论文,山东师范大学).

   https://kns.cnki.net/KCMS/detail/detail.aspx?dbname=CMFD2011&filename=2010231113.nh

3. 胡安峰.(2009).图像水印算法的鲁棒性研究(硕士学位论文,中南大学).

   https://kns.cnki.net/KCMS/detail/detail.aspx?dbname=CMFD2010&filename=2009237384.nh

4. 秦莉文.(2021).鲁棒数字水印性能优化方法研究(硕士学位论文,北京交通大学).

   https://kns.cnki.net/KCMS/detail/detail.aspx?dbname=CMFDTEMP&filename=1021875601.nh

Appendix

a) $img\_watermark\_DFT.m$: Implementation and robustness test of DFT

```
function img_watermark_DFT(img1_path,img2_path)

%% Eliminate all warnings
    warning('off');

%% Set the value when embedding the watermark
    alpha=1

%% Read the image and show them
    im=im2double(imread(img1_path));
    mark=im2double(imread(img2_path));
    figure, imshow(im),title('The original image');
    figure, imshow(mark),title('The watermark');

%% Encode the watermark
    %Get the original image size
    imsize = size(im);

    % random:The coding method adopts random sequence coding,
    %           through coding, the watermark is distributed to random
    %           distribution to each frequency
    TH=zeros(imsize(1)*0.5,imsize(2),imsize(3));
    TH1 = TH;
    TH1(1:size(mark,1),1:size(mark,2),:) = mark;
    M=randperm(0.5*imsize(1));
    N=randperm(imsize(2));
    save('encode.mat','M','N');
    for i=1:imsize(1)*0.5
        for j=1:imsize(2)
            TH(i,j,:)=TH1(M(i),N(j),:);
        end
    end

    % The watermark is encrypted, and the watermark is symmetrical.
    watmark_encoded = zeros(imsize(1),imsize(2),imsize(3));
    watmark_encoded(1:imsize(1)*0.5,1:imsize(2),:)=TH;
    for i=1:imsize(1)*0.5
        for j=1:imsize(2)
            watmark_encoded(imsize(1)+1-i,imsize(2)+1-j,:)=TH(i,j,:);
        end
    end

    figure,imshow(watmark_encoded),title('encoded watermark');

%% Embed watermark
    % Because the image is a discrete signal, the discrete Fourier transform is used.
    % The two-dimensional fast Fourier transform used in this code, whcih is equivalent to discrete
Fourier transform
```

```
    ori_img_fft=fft2(im); % Get the spectrum of the original image
    watermarked_img_fft=ori_img_fft+alpha*double(watmark_encoded); % Overlay the spectrum of original image and the watermark
    watermarked_img=ifft2(watermarked_img_fft); % Get the watermarked image through the inverse Fourier transform
    residual_ori_embedded = watermarked_img-double(im); % Get the spectrum difference before and after embeding the watermark

    % Draw the key image
    figure,imshow(ori_img_fft);title('spectrum of original image');
    figure,imshow(watermarked_img_fft); title('spectrum of watermarked image');
    figure,imshow(watermarked_img);title('watermarked image');
    figure,imshow(uint8(residual_ori_embedded)); title('residual');

%% Extract watermark
    % Perform Fourier transform on the watermarked image to get the spectrum
    % Obtain the spectrum of the watermark by the difference of the spectrum of image before and after watermarked
    watermarked_img_fft2=fft2(watermarked_img);
    G=(watermarked_img_fft2-ori_img_fft)/alpha;
    watermark_extracted=G;
    for i=1:imsize(1)*0.5
        for j=1:imsize(2)
            watermark_extracted(M(i),N(j),:)=G(i,j,:);
        end
    end
    for i=1:imsize(1)*0.5
        for j=1:imsize(2)
            watermark_extracted(imsize(1)+1-i,imsize(2)+1-j,:)=watermark_extracted(i,j,:);
        end
    end
    figure,imshow(watermark_extracted);title('extracted watermark');

%% Robustness test: adding noise
    watermarked_img_noise=imnoise(watermarked_img,'salt & pepper');
    figure,imshow(watermarked_img_noise);title('The watermarked img adding noise');
    watermarked_img_cropping_fft2=fft2(watermarked_img_noise);
    G=(watermarked_img_cropping_fft2-ori_img_fft)/alpha;
    watermark_extracted_noise=G;
    for i=1:imsize(1)*0.5
        for j=1:imsize(2)
            watermark_extracted_noise(M(i),N(j),:)=G(i,j,:);
        end
    end
    for i=1:imsize(1)*0.5
        for j=1:imsize(2)
            watermark_extracted_noise(imsize(1)+1-i,imsize(2)+1-
j,:)=watermark_extracted_noise(i,j,:);
        end
    end

    figure,imshow(watermark_extracted_noise);title('extracted watermark noise');

%% Robustness test: cropping
```

```
        watermarked_img_cropping=im2uint8(watermarked_img);
        watermarked_img_cropping_r= watermarked_img_cropping(:,:,1);
        watermarked_img_cropping_r(1:320,1:796.5)=255;
        watermarked_img_cropping_g =watermarked_img_cropping(:,:,2);
        watermarked_img_cropping_g(1:320,1:796.5)=255;
        watermarked_img_cropping_b =watermarked_img_cropping(:,:,3);
        watermarked_img_cropping_b(1:320,1:796.5)=255;
        watermarked_img_cropping(:,:,1) = watermarked_img_cropping_r;
        watermarked_img_cropping(:,:,2) = watermarked_img_cropping_g;
        watermarked_img_cropping(:,:,3) = watermarked_img_cropping_b;
        figure,imshow(watermarked_img_cropping);title('watermarked_img_cropping');
        watermarked_img_cropping_fft2=fft2(im2double(watermarked_img_cropping));
        G=(watermarked_img_cropping_fft2-ori_img_fft)/alpha;
        watermark_extracted_cropping=G;
        for i=1:imsize(1)*0.5
            for j=1:imsize(2)
                watermark_extracted_cropping(M(i),N(j),:)=G(i,j,:);
            end
        end
        for i=1:imsize(1)*0.5
            for j=1:imsize(2)
                watermark_extracted_cropping(imsize(1)+1-i,imsize(2)+1-
j,:)=watermark_extracted_cropping(i,j,:);
            end
        end

        figure,imshow(watermark_extracted_cropping);title('extracted watermark cropping');

%% Robustness test: rotateing
        watermarked_img_rotateing=imrotate(im2double(watermarked_img),15,'nearest','crop');
        figure,imshow(watermarked_img_rotateing);title('watermarked_img_rotateing');
        watermarked_img_rotateing_fft2=fft2(watermarked_img_cropping);
        G=(watermarked_img_rotateing_fft2-ori_img_fft)/alpha;
        watermark_extracted_rotateing=G;
        for i=1:imsize(1)*0.5
            for j=1:imsize(2)
                watermark_extracted_rotateing(M(i),N(j),:)=G(i,j,:);
            end
        end
        for i=1:imsize(1)*0.5
            for j=1:imsize(2)
                watermark_extracted_rotateing(imsize(1)+1-i,imsize(2)+1-
j,:)=watermark_extracted_rotateing(i,j,:);
            end
        end

        figure,imshow(watermark_extracted_rotateing);title('extracted watermark rotateing');

%% Robustness test: brightness increased by 10%
        watermarked_img_brightness10=1.1*im2double(watermarked_img);
        figure,imshow(watermarked_img_brightness10);title('watermarked_img_brightness10');
        watermarked_img_brightness_fft2=fft2(watermarked_img_brightness10);
        G=(watermarked_img_brightness_fft2-ori_img_fft)/alpha;
        watermark_extracted_brightness=G;
        for i=1:imsize(1)*0.5
```

```
        for j=1:imsize(2)
            watermark_extracted_brightness(M(i),N(j),:)=G(i,j,:);
        end
    end
    for i=1:imsize(1)*0.5
        for j=1:imsize(2)
            watermark_extracted_brightness(imsize(1)+1-i,imsize(2)+1-
j,:)=watermark_extracted_brightness(i,j,:);
        end
    end

    figure,imshow(watermark_extracted_brightness);title('extracted watermark brightness10');
```

b) $img\_watermark\_DWT.py$: Implementation and robustness test of DWT

```
# Import the packages we use
import numpy as np
import cv2
import pywt
import random
import math
import cmath


# arnold permutation, k is the number of permutation
def arnold (img, key):
    r = img.shape[0]
    c = img.shape[1]
    p = np.zeros ( (r, c), np.uint8 )

    a = 1
    b = 1
    for k in range ( key ):
        for i in range ( r ):
            for j in range ( c ):
                x = (i + b * j) % r
                y = (a * i + (a * b + 1) * j) % c
                p[x, y] = img[i, j]
    return p


# Inverse Arnold replacement, k is the number of permutation
def deArnold (img, key):
    r = img.shape[0]
    c = img.shape[1]
    p = np.zeros ( (r, c), np.uint8 )

    a = 1
    b = 1
    for k in range ( key ):
        for i in range ( r ):
            for j in range ( c ):
                x = ((a * b + 1) * i - b * j) % r
                y = (-a * i + j) % c
```

```
                    p[x, y] = img[i, j]
    return p


# Watermark embedding
def setwaterMark (waterTmg, Img, key):
    print ( 'Watermark embedding...' )
    Img = cv2.resize ( Img, (400, 400) )
    waterTmg = cv2.resize ( waterTmg, (201, 201) )
    # Grayscale processing of carrier image
    Img1 = cv2.cvtColor ( Img, cv2.COLOR_RGB2GRAY )
    waterTmg1 = cv2.cvtColor ( waterTmg, cv2.COLOR_RGB2GRAY )
    cv2.imshow ( 'original image', Img1 )
    cv2.imshow ( 'watermark image', waterTmg1 )
    cv2.waitKey ( 0 )
    # Arnold permutation is implemented on the watermark image
    waterTmg1 = arnold ( waterTmg1, key )

    # carrier image three-order wavelet transform
    c = pywt.wavedec2 ( Img1, 'db2', level = 3 )
    [cl, (cH3, cV3, cD3), (cH2, cV2, cD2), (cH1, cV1, cD1)] = c
    # watermark image first order wavelet transform
    waterTmg1 = cv2.resize ( waterTmg1, (101, 101) )
    d = pywt.wavedec2 ( waterTmg1, 'db2', level = 1 )
    [ca1, (ch1, cv1, cd1)] = d

    # Custom embedding coefficients
    a1 = 0.1
    a2 = 0.2
    a3 = 0.1
    a4 = 0.1
    # embedding
    cl = cl + ca1 * a1
    cH3 = cH3 + ch1 * a2
    cV3 = cV3 + cv1 * a3
    cD3 = cD3 + cd1 * a4

    # Image reconstruction
    newImg = pywt.waverec2 ( [cl, (cH3, cV3, cD3), (cH2, cV2, cD2), (cH1, cV1, cD1)], 'db2' )
    newImg = np.array ( newImg, np.uint8 )

    print ( 'Watermark embedding complete!' )

    cv2.imshow ( "carrier image", newImg )
    cv2.imwrite ( './after.bmp', newImg )
    cv2.waitKey ( 0 )


def getwaterMark (originalImage, Img, key):
    print ( 'Watermark Extraction...' )
    # Original image gray processing
    originalImage = cv2.resize ( originalImage, (400, 400) )

    Img1 = cv2.cvtColor ( originalImage, cv2.COLOR_RGB2GRAY )
    Img = cv2.cvtColor ( Img, cv2.COLOR_RGB2GRAY )
```

```
        #         cv2.imshow('original image',Img1)
        cv2.waitKey ( 0 )

        # carrier image three-order wavelet transform
        c = pywt.wavedec2 ( Img, 'db2', level = 3 )
        [cl, (cH3, cV3, cD3), (cH2, cV2, cD2), (cH1, cV1, cD1)] = c

        # Original image three-order wavelet transform
        d = pywt.wavedec2 ( Img1, 'db2', level = 3 )
        [dl, (dH3, dV3, dD3), (dH2, dV2, dD2), (dH1, dV1, dD1)] = d

        # The embedding algorithm (reverse)
        # Custom embedding coefficients
        a1 = 0.1
        a2 = 0.2
        a3 = 0.1
        a4 = 0.1

        ca1 = (cl - dl) * 10
        ch1 = (cH3 - dH3) * 5
        cv1 = (cV3 - dV3) * 10
        cd1 = (cD3 - dD3) * 10

        # Reconstruction of watermark image
        waterImg = pywt.waverec2 ( [ca1, (ch1, cv1, cd1)], 'db2' )
        waterImg = np.array ( waterImg, np.uint8 )

        cv2.imshow ( "Reconstruction of watermark image", waterImg )
        cv2.waitKey ( 0 )

        # Inverse Arnold permutation is performed on the extracted watermark image
        waterImg = deArnold ( waterImg, key )
        cv2.imshow ( "Reconstruction of watermark image", waterImg )
        cv2.waitKey ( 0 )
        print ( 'Watermark Extraction Complete !  ' )

        return waterImg

def rotate_about_center(src, angle, scale=1.):
        w = src.shape[1]
        h = src.shape[0]
        rangle = np.deg2rad(angle)    # angle in radians
        nw = (abs(np.sin(rangle) * h) + abs(np.cos(rangle) * w)) * scale
        nh = (abs(np.cos(rangle) * h) + abs(np.sin(rangle) * w)) * scale
        rot_mat = cv2.getRotationMatrix2D((nw * 0.5, nh * 0.5), angle, scale)
        rot_move = np.dot(rot_mat, np.array([(nw - w) * 0.5, (nh - h) * 0.5, 0]))
        rot_mat[0, 2] += rot_move[0]
        rot_mat[1, 2] += rot_move[1]
        return cv2.warpAffine(src, rot_mat, (int(math.ceil(nw)), int(math.ceil(nh))),
flags=cv2.INTER_LANCZOS4)

def attack (fname, type):
        img = cv2.imread(fname)
```

```
    # Get the image before we attack the image
    if type == "ori":
        return img

    # Blur attack
    if type == "blur":
        kernel = np.ones ( (3, 3), np.float32 ) / 9
        return cv2.filter2D ( img, -1, kernel )

    # Rotating attack
    if type == "rotate90":
        return rotate_about_center ( img, 90 )

    # Noise attack
    if type == "saltnoise":
        for k in range ( 1000 ):
            i = int ( np.random.random () * img.shape[1] )
            j = int ( np.random.random () * img.shape[0] )
            if img.ndim == 2:
                img[j, i] = 255
            elif img.ndim == 3:
                img[j, i, 0] = 255
                img[j, i, 1] = 255
                img[j, i, 2] = 255
        return img

    # Graffiti attack
    if type == "randline":
        cv2.rectangle ( img, (384, 0), (510, 128), (0, 255, 0), 3 )
        cv2.rectangle ( img, (0, 0), (300, 128), (255, 0, 0), 3 )
        cv2.line ( img, (0, 0), (511, 511), (255, 0, 0), 5 )
        cv2.line ( img, (0, 511), (511, 0), (255, 0, 255), 5 )

        return img

    # Cropping attack
    if type == "cropping":
        cv2.circle ( img, (256, 256), 63, (255, 255, 255), -1 )
        font = cv2.FONT_HERSHEY_SIMPLEX
        return img
    return img

    # Brightness attack
    if type == "brighter10":
        w, h = img.shape[:2]
        for xi in range ( 0, w ):
            for xj in range ( 0, h ):
                img[xi, xj, 0] = int ( img[xi, xj, 0] * 10 )
                img[xi, xj, 1] = int ( img[xi, xj, 1] * 10 )
                img[xi, xj, 2] = int ( img[xi, xj, 2] * 10 )
        return img

attack_list = {}
attack_list['ori'] = 'Original image'
attack_list['blur'] = 'Blur attack'
```

```
attack_list['rotate90'] = 'Rotating attack'
attack_list['saltnoise'] = 'Noise attack'
attack_list['randline'] = 'Graffiti attack'
attack_list['cropping'] = 'Cropping attack'
attack_list['brighter10'] = 'Brightness attack'

if __name__ == '__main__':

    # read original image and watermark image
    waterImg = cv2.imread ( 'school_badge.jpg' )
    Img = cv2.imread ( 'Jisoo.jpg' )
    # watermark embedding
    setwaterMark ( waterImg, Img, 10 )

    #       # read original image and carrier image
    originalImage = cv2.imread ( 'Jisoo.jpg' )
    Img = cv2.imread ( './after.bmp' )
    # watermark extraction

    getwaterMark ( originalImage, Img, 10 )
    # image attack
    for k, v in attack_list.items ():
        wmd = attack ( 'after.bmp', k )
        cv2.imshow ( 'new', wmd )
        filename = "./output/{}.jpg".format ( k )
        cv2.imwrite ( filename, wmd )

        watermark = getwaterMark ( originalImage, wmd, 10 )
        wm_filename = "./output/watermark/{}.jpg".format ( k )
        cv2.imshow ( "extraction watermark image", watermark )
        cv2.imwrite ( wm_filename, watermark )

        cv2.waitKey ( 0 )
```