



Practica 5

Esteganografía

Códigos y Criptografía
Curso 2023/24
Ingeniería Informática UCO

Alumno:	Cáceres Gaitán, Pablo
Email:	i12cacgp@uco.es

Índice

1. Introducción	2
2. Ejemplo de ejecución	3
2.1. LSB Simple	3
2.1.1. Codificación	3
2.1.2. Decodificación	4
2.2. LSB Complejo	4
2.2.1. Codificación	4
2.2.2. Decodificación	4
2.3. Desordenación de imágenes	5
2.3.1. Desordenar/Ordenar	5
2.3.2. Desordenar con varios k	6
3. Análisis del código fuente	7
3.1. Dependencias	7
3.2. LSB Simple	7
3.3. LSB Complejo	8
3.4. Desordenación	8
3.4.1. Funciones auxiliares	8
3.4.2. Ordenación y desordenación	8
3.4.3. Generación de imágenes	9
4. Conclusión	9

1. Introducción

En esta práctica se implementan varias técnicas de esteganografía en imágenes. Así como técnicas para ocultar el contenido de una imagen.

Además, este código incluye también una pequeña aplicación de terminal que permite probar las funciones implementadas. En este caso, no se utiliza una interfaz interactiva, sino que se interactúa mediante flags pasadas como argumentos.

Dado que la base teórica ha sido estudiada en clase, no será incluida en este documento.

2. Ejemplo de ejecución

Para poder probar todas las funciones, se ha implementado una interfaz por línea de comandos, que permite llamar a las distintas funcionalidades mediante *flags*.

Para las pruebas, trabajaremos sobre la siguiente imagen (Fig. 1):



Figura 1: Imagen base.

En caso de necesitar ver las posibles *flags*, podemos llamar al menú de ayuda:

```
python3 lsb.py -h
```

2.1. LSB Simple

2.1.1. Codificación

Para codificar un mensaje en una imagen, debemos especificar el mensaje con *-m*, y utilizar la opción *-e* (*encode*):

```
python3 lsb.py -m "Hola esto es una prueba" -e ./imagen.jpg
```

Lo que nos debería dar la imagen en blanco y negro y con el mensaje codificado (Fig. 2):



Figura 2: Imagen codificada.

2.1.2. Decodificación

Una vez tenemos una imagen codificada, podemos decodificarla con la opción *-d* (*decode*). Podemos especificar con *-l* la cantidad de bytes a mostrar, aunque por defecto es 150.

```
python3 lsb.py -d -l 200 ./imagen_codificada.png
```

```
Mensaje oculto: Hola esto es una pruebaÃ
```

2.2. LSB Complejo

2.2.1. Codificación

De forma similar, podemos utilizar configuración compleja con la *flag -Ce*. En este caso, podemos especificar también el nivel de complejidad con *-c*:

```
python3 lsb.py -Ce -m "Hola esto es una prueba" ./imagen.jpg
```

De igual forma, obtenemos la imagen con el mensaje codificado:



Figura 3: Imagen codificada compleja.

2.2.2. Decodificación

Para decodificar, debemos especificar la opción de decodificación compleja *-Cd*, además de especificar una complejidad con *-c*:

```
python3 lsb.py -Cd imagen_codificada_compleja.png -l 200
```

```
Mensaje oculto: Hola esto es una pruebaÃ_
```

2.3. Desordenación de imágenes

2.3.1. Desordenar/Ordenar

Finalmente, se ha implementado un método para desordenar imágenes. En este caso, tenemos la opción de desordenar una imagen con `-d`. Tendremos que pasar el valor de k :

```
python3 desordenar.py -d imagen.jpg

Imagen recortada guardada en imagen_cuadrada.png
Puedes usar los valores de k menores que: 2880
Ingrese un valor k para desordenar la imagen (1 <= k): 2
Imagen desordenada con k=2 guardada en imagen_desordenada_k.png
```

La imagen se recorta para ser cuadrada y se desordena (Fig. 4):

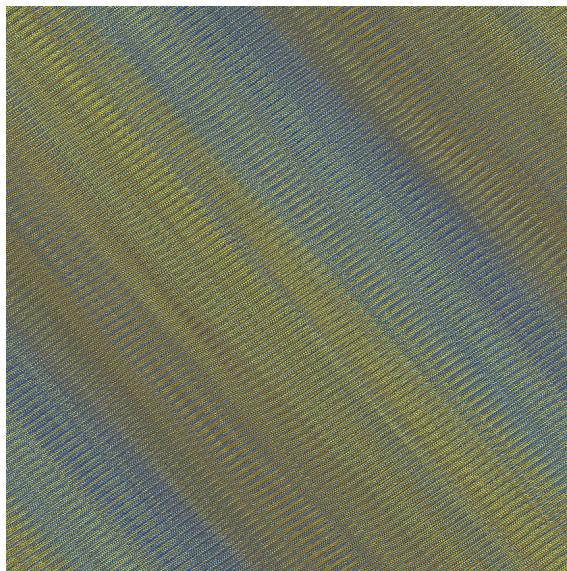


Figura 4: Imagen desordenada con $k=2$

Para ordenar, lanzamos el programa con los siguientes parámetros:

```
python3 desordenar.py -o imagen_desordenada_k.png

Imagen recortada guardada en imagen_cuadrada.png
Ingrese el mismo valor k utilizado para desordenar la imagen: 2
Puedes usar los valores de k menores que: 1440
Imagen ordenada con k=2 guardada en imagen_ordenada.png
```

La imagen se reordenará y se guardará en `imagen_ordenada.png`.

2.3.2. Desordenar con varios k

La última opción, $-p$, nos permite desordenar una imagen varias veces en base a varios valores de k .

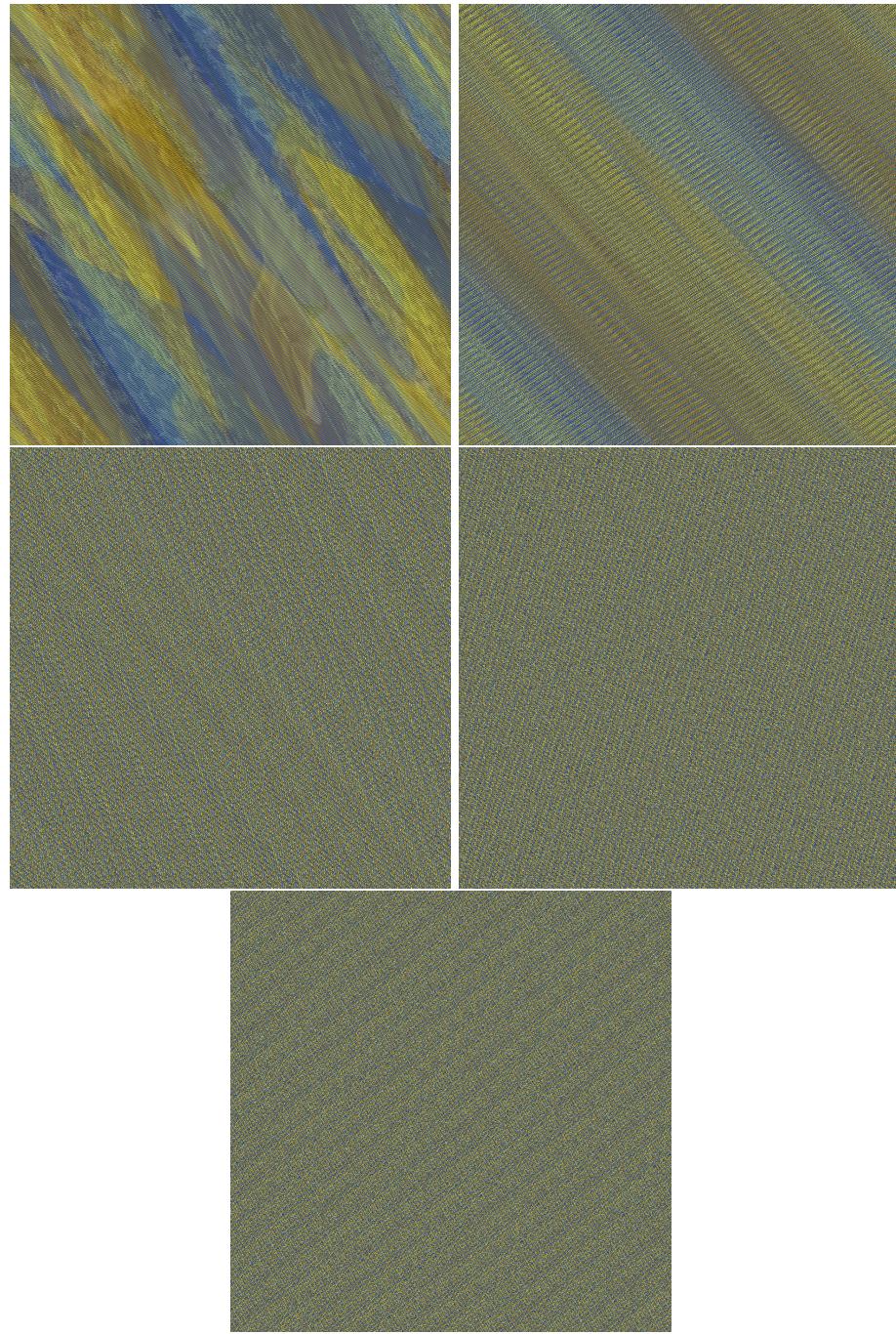


Figura 5: Imagen desordenadas con k de 1 a 5.

3. Análisis del código fuente

3.1. Dependencias

Ambos programas utilizan las siguientes librerías:

```

1 import numpy as np
2 from PIL import Image
3 import os
4 import argparse

```

Se utilizan *numpy* y *pillow* como librerías principales para gestionar matrices e imágenes, respectivamente. Además de *os* para eliminar archivos temporales y *argparse* para gestionar los argumentos.

3.2. LSB Simple

Para la codificación, disponemos de dos funciones que convierten texto en bits (*text_to_bits*) y bits en texto (*bits_to_text*).

Para la codificación, se utiliza la función *LSB_simple_cypher*, que toma una imagen, un mensaje y un fichero de salida, y oculta el mensaje en los primeros bits de la imagen, guardándola en el fichero de salida. Tanto el principio que consiste en abrir la imagen y convertir el texto en bits, como el final, que es guardar los cambios, la parte importante (donde se realiza la codificación) es:

```

1 # Modificar los bits menos significativos de los pixeles
2 new_pixels = [
3     (pixel & ~1) | int(bits[i]) if i < msg_len else pixel
4     for i, pixel in enumerate(pixels)
5 ]

```

En el caso de la decodificación, se realiza mediante la función *LSB_simple_decipher*, que toma una imagen y una longitud, y extrae dicha cantidad de bits de los menos significativos de la imagen. La parte importante es:

```

1 # Extraer los bits menos significativos
2 bits = ''.join(str(pixel & 1) for pixel in pixels[:msg_length])

```

3.3. LSB Complejo

En el caso de la versión compleja, se añade un salto entre píxeles, ó complejidad. Las funciones son casi idénticas. La parte importante de la función de codificar, *LSB_complex_cypher* es:

```

1 new_pixels = pixels[:]
2 for i, bit in enumerate(bits):
3     idx = i * step
4     new_pixels[idx] = (pixels[idx] & ~1) | int(bit)

```

Y en el caso del descifrado, *LSB_complex_decipher*, es:

```

1 bits = ''.join(str(pixels[i*step] & 1) for i in range(msg_length))

```

3.4. Desordenación

3.4.1. Funciones auxiliares

Para la parte de desordenación de imágenes, tenemos más cantidad de funciones. Sin embargo, las primeras son para operar con matrices en base a un módulo, como *matrix_mod_inverse* ó *mod_matrix_mult*.

Además, disponemos de *crop_to_square* y *delete_image*, que se utilizan para recortar la imagen de entrada y para borrar ficheros temporales.

3.4.2. Ordenación y desordenación

Al igual que antes, la función incluye la lógica necesaria para tratar la imagen y realizar comprobaciones en la matriz. La parte importante en la función de desordenado, *desordenaImagen* es:

```

1 for x in range(h):
2     for y in range(w):
3         new_pos = mod_matrix_mult(matrix, [x, y], n)
4         shuffled[new_pos[0] % h, new_pos[1] % w] = pixels[x, y]

```

Para el caso de *ordenaimagen*, es:

```

1 inverse_matrix = matrix_mod_inverse(matrix, n)
2 desordenaImagen(inverse_matrix, image_path, output_path, n)

```

Como vemos, para ordenar, simplemente desordenamos utilizando la matriz inversa.

Para el caso de las funciones que elevan la matriz de codificación a un valor *k*, *desordenaImagenite* y *ordenaimagenite*, simplemente elevamos la matriz antes de llamar a desordenar la imagen:

```

1 Ak = mod_matrix_power(A, k, n)

```

3.4.3. Generación de imágenes

Para esta parte, necesitamos la función `find_suitable_k`, que encuentra el valor k para el cuál $A^k \equiv I \pmod{n}$. Esto garantiza que tras un número de iteraciones no llegaremos a la imagen de partida.

Para la generación, simplemente vamos elevando la matriz y guardando cada resultado:

```

1 # Iterar sobre valores de k y guardar la evolucion
2 for k in range(1, max_k + 1):
3     # Calcular A^k en Z_n
4     Ak = mod_matrix_power(A, k, n)
5
6     # Crear una nueva matriz de pixeles desordenados
7     shuffled = np.zeros_like(pixels)
8
9     for x in range(h):
10        for y in range(w):
11            new_pos = np.dot(Ak, [x, y]) % n
12            shuffled[new_pos[0] % h, new_pos[1] % w] = pixels[x, y]
13
14     # Guardar la imagen desordenada
15     output_path = os.path.join(output_dir,
16                               f"imagen_desordenada_k{k}.png")
17     shuffled_img = Image.fromarray(shuffled)
18     shuffled_img.save(output_path)
19     print(f"Imagen desordenada k={k} guardada en {output_path}")

```

4. Conclusión

En esta práctica, hemos aprendido varios métodos simples para ocultar información en base a imágenes. Sin embargo, hablamos de codificación y no de encriptación, ya que en caso de ver el mensaje, sería posible interpretarlo por sí solo. Por ejemplo, de darse cuenta de que el mensaje está dividido en cada 4 bits de la imagen, sería posible leerlo.

Además de aprender sobre esteganografía, es curioso observar cómo no cambia la imagen al introducir un mensaje en ella. Ó cómo no podemos interpretar la imagen con tan solo multiplicándola por una matriz.