Informe trabajo final MMA exposición

Pedro Antonio Fondevila Franco

Enero 2020

1 Introducción

Enmarcado en la asignatura de Modelos Matemáticos y Algoritmos vamos a hacer un estudio del paper *Maricela, J. R., Octavio, F. S., Guadalupe, G. N. M. (2015). Sistema para codificar información implementando varias orbitas caóticas. Ingeniería, investigación y tecnología, 16(3), 335-34*

En dicho paper se habla de la codificación de información a través de órbitas caoticas, en principio puede parecer que no cae dentro del ambito de la asignatura pero es simplemente usar uno de los modelos más sencillos de dinámica de poblaciones, el modelo logístico, en nuestro beneficio en el ambito de la criptografía.

2 Modelo logístico

La ecuación del modelo logístico es la siguiente:

$$x_{n+1} = rx_n(1 - x_n)$$

donde $x \in [0, 1]$ que representa la población en porcerntaje (donde 1 sería el número de individuos máximo teórico de la población) y $r \in \mathbb{R}$ que representa la tasa entre nacimientos y defunciones.

Recordemos el diagrama de bifurcaciones asociado a esta ecuación:

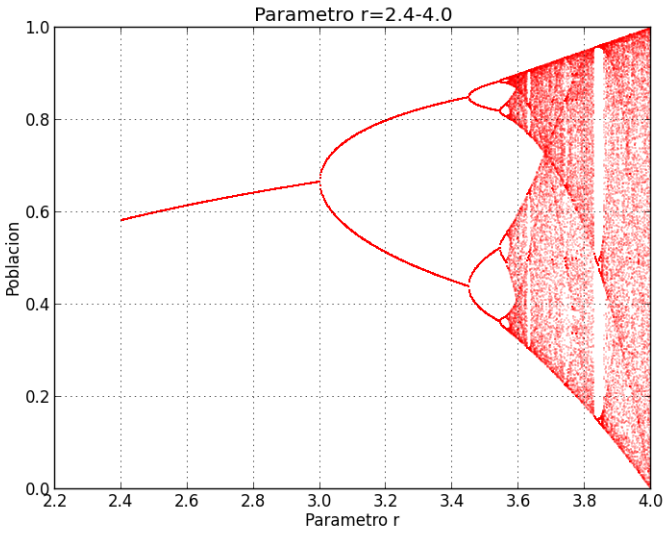

Figure 1: Diagrama de bifuraciones r entre 2.4 y 4.

Como podemos ver a partir de $r = 3.57$ hasta $r = 4$ aparece el caos, donde la propiedad fundamental de este caos es que a pequeñas variaciones de las

condiciones iniciales se tienen resultados diametralmente distintos. Esta es justo la propiedad que vamos a aprovechar para nuestra empresa.

3 Encriptación

Antes de empezar, señalar que el código se puede consultar aquí.

En el paper estudiado describen el proceso exacto para encriptar una foto (aunque se podría generalizar a cualquier tipo de información). Para explicar el proceso vamos a usar la siguiente notación:

- L = longitud del vector con la información original.
- LM = longitud del vector con la información cifrada.
- r_1 y $x1_0$ = parámetro y condición inicial usados para generar la órbita empleada en la técnica de difusión.
- r_2 y $x2_0$ = parámetro y condición inicial para generar la órbita de tamaño LM - L, la cual se emplea para incrementar la longitud de L a LM.
- r_3 y $x3_0$ = parámetro y condición inicial utilizados para generar la órbita que se implementa en la técnica de confusión.
- *ubicación inicial* = posición inicial del vector información cifrada, entre 1 y LM.

Resaltar que es absolutamente imprescindible guardar L, r_1, $x1_0$, r_3, $x3_0$ y *ubicacióninicial* para poder recuperar la información.

A partir de ahora iremos intercalando los pasos descritos en el paper con la implementación hecha en python:

Usaremos la siguiente foto satélite del campus de la Universidad de Cádiz ubicado en Puerto Real.

Figure 2: Diagrama de bifuraciones r entre 2.4 y 4.

- Paso 1. Digitalizar la imagen para guardar cada uno de los subpixeles en el vector Inf_original, el cual es de longitud $L = P_n^R + P_n^G + P_n^B$.

Listing 1: Paso 1

```
from PIL import Image
import random
import numpy as np
from matplotlib import cm

long_extra = 2
def logistic(r,x): return r*x*(1-x)
def r_generator(): return random.uniform(3.57, 4)


original = Image.open('campus.jpg', 'r')
width, height = original.size
# Sacamos los pixeles
im = list(original.getdata())

# Paso 1: Lo ponemos todo en un solo vector
inf_original = [item for sublist in im for item in sublist]
copy = [item for sublist in im for item in sublist]
L = len(inf_original)
LM = L*long_extra
ubicacion_inicial = int(random.uniform(1, LM))
```

- Paso 2. Dividir cada subpixel que contiene el vector Inf_original entre 255, para obtener valores entre 0 y 1.

Listing 2: Paso 2

```
# Paso 2: Dividimos entre 255 para pasar a [0,1]
inf_original = [x / 255 for x in inf_original]
```

- Paso 3. Generar el vector Inf_cifrada de longitud LM, (L debe ser menor que LM); donde se almacenará la información codificada.

Listing 3: Paso 3

```
# Paso 3: Generamos LM y lo rellenamos de -1
inf_cifrada = np.zeros((1,LM))[0]
inf_cifrada = [-1 for i in range(LM)]
```

- Paso 4. Utilizar las llaves de cifrado r1 y x10 para resolver la ecuación 1 L veces y generar una órbita caótica de longitud L con valores entre 0 y 1 que se almacenarán en el vector logistic_mezcla.

Listing 4: Paso 4

```
# Paso 4: Encontrar una orbita de longitud L
key_1 = r_generator()
x0_1  = random.uniform(0, 1)
logistic_mezcla = np.zeros((1,L))[0] # Vector de ceros
print('Paso 4')
logistic_mezcla[0] = x0_1
for i in range(L-1):
    logistic_mezcla[i+1] = logistic(key_1,logistic_mezcla[i])
```

- Paso 5. Multiplicar cada valor del vector logistic_nezcla por LM, después redondear para obtener L valores entre 1 y LM, que servirán como posiciones para aplicar la técnica de difusión.

Listing 5: Paso 5

```
# Paso 5: Conseguir el vector posiciones
posiciones = np.zeros((1,L))[0] # Vector de ceros

for i in range(L):
    posiciones[i] = round(LM*logistic_mezcla[i])
```

- Paso 6. Asignar a ubicación el valor de la llave de cifrado ubicación_nicial.

Listing 6: Paso 6

```
# Paso 6: Guardamos la ubicacion inicial en ubicacion_inicial
ubicacion = ubicacion_inicial
```

- Paso 7. Calcular la ubicación donde se posicionará el valor de Inf_original en Inf_cifrada, también se utiliza el siguiente valor del vector posiciones.

  ubicación = ubicación + (L MOD posiciones [pos])

- Paso 8. Si ubicación LM, colocar en la ubicación del vector Inf_cifrada el valor de Inf_original, en caso de que la ubicación ya tenga algún valor, entonces se coloca en la siguiente posición vacía.

- Paso 9. Si ubicación LM excede la longitud del vector Inf_cifrada, por lo tanto, se le asigna ubicación = L MOD posiciones [pos], con la finalidad de volver a recorrer el vector Inf_cifrada; en la nueva ubicación se almacena el valor de Inf_original. En caso de ya haber acomodado un valor, entonces se coloca en la siguiente posición que este vacía.

- Paso 10. Realizar los pasos 7 y (8 o 9) hasta acomodar cada valor del vector Inf_original en alguna ubicación de Inf_cifrada.

Listing 7: Paso 7-10

```
# Pasos 7-10: Metemos inf_original en inf_cifrada desordenadamente
for i in range(0,L):
    print(f'i = {i}  L = {L}')
    ubicacion = int(ubicacion + (posiciones[i]%L))
    if(ubicacion < LM): # Paso 8
        # Buscamos un hueco vacio
        while inf_cifrada[ubicacion] != -1:
            #print('c1')
            ubicacion = ubicacion + 1
            if(ubicacion == LM): ubicacion = 0

        inf_cifrada[ubicacion] = inf_original[i]

    else: # Paso 9 ubicacion > LM
        ubicacion = int(posiciones[i]%L)

        while inf_cifrada[ubicacion] != -1:
            #print(f'c2 ubicacion = {ubicacion}')
            ubicacion = ubicacion + 1
            if(ubicacion == LM): ubicacion = 0

        inf_cifrada[ubicacion] = inf_original[i]
```

- Paso 11. En los pasos 7 a 10 se acomodan los L términos entre [0,1] del vector Inf_original en Inf_cifrada, pero como Inf_cifrada es de longitud LM, quedan LM - L ubicaciones vacías, para llenar, se resuelve la ecuación 1 con las llaves de cifrado r2 y x20, para generar otra órbita caótica con valores también entre 0 y 1, la cual se guarda en el vector denominado Relleno.

Listing 8: Paso 11

```
# Paso 11: creacion del vector relleno

key_2 = r_generator()
x0_2  = random.uniform(0, 1)
relleno = np.zeros((1,LM-L))[0]

relleno[0] = x0_2
for i in range(LM-L-1):
    relleno[i+1] = logistic(key_2,relleno[i])
```

- Paso 12. Tomar un valor del vector Relleno y colocarlo en la siguiente ubicación vacía de Inf_cifrada. Repetir este paso LM - L veces.

Listing 9: Paso 12

```
# Paso 12: Rellenao inf_cifrada con relleno
ubicacion = 0
index = 0
for number in inf_cifrada:
    if(inf_cifrada[ubicacion] == -1):
        inf_cifrada[ubicacion] = relleno[index]
        index = index + 1
    ubicacion = ubicacion + 1
```

- Paso 13. Utilizar las llaves r3 y x30 para resolver la ecuación 1 y obtener valores caóticos que se almacenen en el vector denominado Confusión. Repetir este paso LM veces.

Listing 10: Paso 13

```
key_3 = r_generator()
x0_3  = random.uniform(0, 1)
confusion = np.zeros((1,LM))[0]

confusion[0] = x0_3
for i in range(LM-1):
    confusion[i+1] = logistic(key_3,confusion[i])
```

- Paso 14. Aplicar la técnica de confusión: sumando a cada valor del vector Inf_cifrada uno de Confusión, en forma ordenada,

Listing 11: Paso 14

```
# Paso 14: Aplicar confusion
for i in range(LM):
    inf_cifrada[i] = inf_cifrada[i] + confusion[i]
```

Una vez introducida nuestra foto tendremos la siguiente foto como output (recalcar que se ha mantenido el ancho de la foto por tomar alguna medida y poder mostrarla, pero esto es arbitrario):

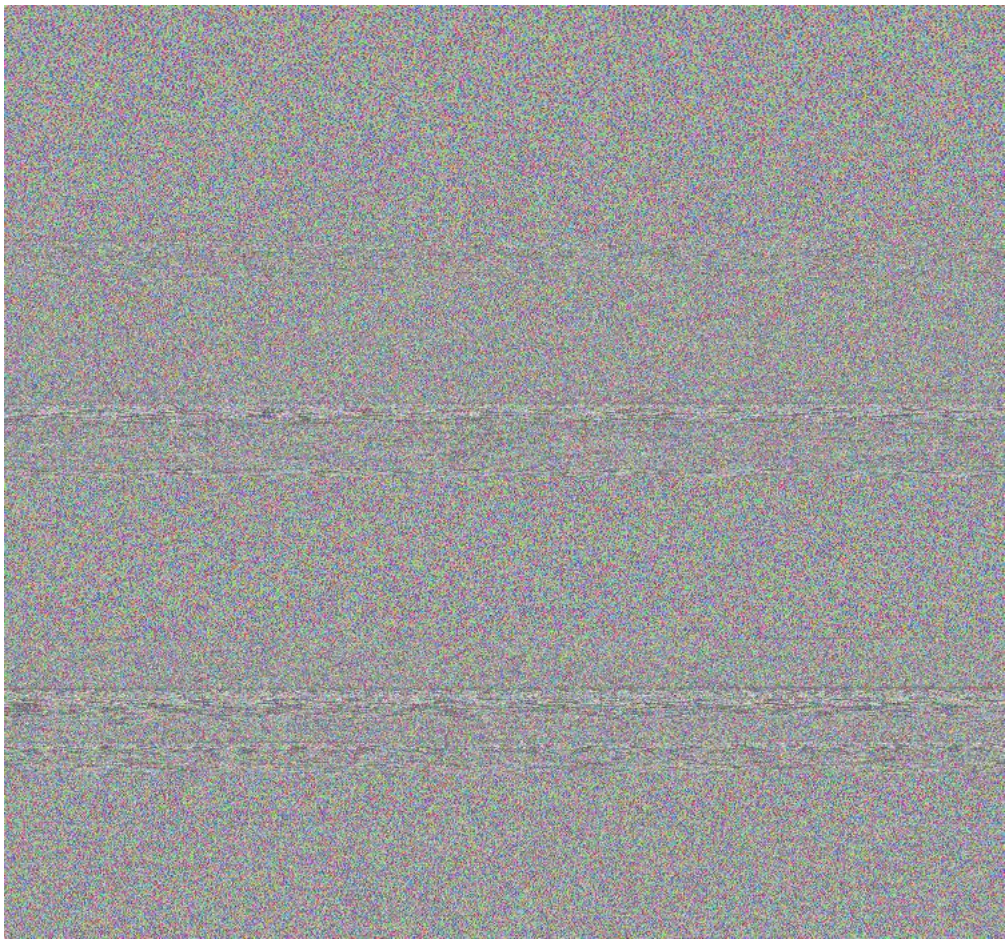

Figure 3: Campus.jpg encriptado

Ahora se mostrará la implementación de la desincriptación:

- Paso 1. Utilizar las llaves r3 y x30 para generar el vector Confusión como se indica en el paso 13 del algoritmo para cifrar.

Listing 12: Paso 1

```
# Paso 1: recrear confusion
confusion = np.zeros((1,LM))[0]

confusion[0] = x0_3
for i in range(LM-1):
    confusion[i+1] = logistic(key_3, confusion[i])
```

- Paso 2. Restar a cada valor del vector Inf_cifrada el respectivo elemento del vector Confusión de manera ordenada.

Listing 13: Paso 2

```
# Paso 2: restar confusion
print('Paso 2')
for i in range(LM):
    inf_cifrada[i] = inf_cifrada[i] - confusion[i]
```

- Paso 3. Utilizar las llaves r1 y x10 para generar el vector posiciones, realizando los pasos 4 y 5 del algoritmo para cifrar.

Listing 14: Paso 3

```
# Paso 3: Recrear logistic_mezcla y posiciones

logistic_mezcla = np.zeros((1,L))[0] # Vector de ceros

logistic_mezcla[0] = x0_1
for i in range(L-1):
    logistic_mezcla[i+1] = logistic(key_1, logistic_mezcla[i])

posiciones = np.zeros((1,L))[0] # Vector de ceros
for i in range(L):
    posiciones[i] = round(LM*logistic_mezcla[i])
```

- Paso 4. Asignar ubicación jnicial como en el paso 6 del algoritmo para cifrar.

Listing 15: Paso 3

```
# Paso 4: Reasignar ubicacion_inicial

ubicacion = ubicacion_inicial
```

- Paso 5. Calcular ubicación como en el paso 7 del algoritmo para cifrar.
- Paso 6. Si ubicación LM, se toma el valor almacenado en esta ubicación dentro del vector Inf_cifrada, en caso de no encontrarse, se toma el de la siguiente posición que sí contenga valor.

- Paso 7. Si ubicación ¿ LM, se asigna ubicación = L MOD posiciones [pos], para recorrer de nuevo el vector Inf_cifrada, enseguida se toma el valor de la ubicación en Inf_cifrada, en caso de no encontrarse, se toma el de la siguiente posición que sí contenga valor.

- Paso 8. El valor que se obtuvo en el paso 6 ó 7, se multiplica por 255 y se redondea. Posteriormente se almacena el resultado en la siguiente posición vacía de Inf_original y se elimina el valor de la ubicación que se tomó en Inf_cifrada.

  Inf_original[pos] = redondear (Inf_cifrada[ubicación]*255)

- Paso 9. Repetir los pasos 5 a 8 L veces para eliminar la técnica de difusión y reacomodar todos los datos del vector Inf_original con los valores de los subpixeles entre 0 y 255

Listing 16: Paso 5-9

```
# Paso 5: Volver a crear inf_original

inf_original = np.zeros((1,L))[0]
for i in range(0,L):
    print(f'i = {i}  L = {L}')
    ubicacion = int(ubicacion + (posiciones[i]%L))

    if(ubicacion < LM): # Paso 6
        # Buscamos un hueco vacio
        while inf_cifrada[ubicacion] == -1:
            ubicacion = ubicacion + 1
            if(ubicacion == LM): ubicacion = 0

        inf_original[i] = round(255*inf_cifrada[ubicacion])
        inf_cifrada[ubicacion] = -1

    else: # Paso 7 ubicacion > LM
        ubicacion = int(posiciones[i]%L)

        while inf_cifrada[ubicacion] == -1:
            ubicacion = ubicacion + 1
            if(ubicacion == LM): ubicacion = 0

        inf_original[i] = int(round(255*inf_cifrada[ubicacion]))
        inf_cifrada[ubicacion] = -1

inf_original = inf_original.astype(int)
```

La imagen que devuelve, como era de esperar es exactamente la misma que la original.

Figure 4: Imagen desencriptada

Por si se quiere consultar con detalle el código está disponible en el siguiente enlace.

4 Conclusiones

Una vez realizado el código y hecho algunas pruebas podemos sacar algunas conclusiones. Lo primero de todo es que es una idea ingeniosa utilizar el caos determinista del modelo logarítmico para algo supuestamente tan alejado como es la encriptación pero quitando que su implementación es relativamente sencilla este método tiene varias desventajas.

- Contras encontrados:
  - Lentitud: Aunque en el propio paper dicen que este método es relativamente rápido, al menos con Python esto no es cierto, la imagen que hemos usado es de un tamaño extremadamente pequeño a día de hoy (750x350 píxeles) pero por desgracia en el pc que se ha usado ($i5 - 6500$) que aunque no es nuevo para nada está desfasado, tarda aproximadamente unos $7 - 8min$ en encriptar la imagen.

    Esto es debibo principalmente a los pasos 7-10 del proceso de encriptación, los relativos al desordenamiento del vector que cada vez le cuesta más encontrar un hueco libre en el vector y tarda bastante, es posible que esto se palie con un LM bastante superior a L pero a costa de una cantidad de RAM significativa.

- Precisión: Usar funciones caóticas es un arma de doble filo, para que este método se pueda exportar a cualquier pc habría que limitar bastante la precisión que se usa para que en todos los sistemas se recuperara exactamente los mismos números ya que un pequeño error de redondeo (que puede ser distinto incluso usando la misma versión de python en otra CPU más antigua o moderna que la usada) puede desencadenar en una órbita totalmente diferente y por tanto la información sería irrecuperable.
- No paralelizable: No se puede paralelizar mucho el proceso ya que es necesario calcular un punto detrás de otro de la órbita, a lo sumo se podrían calcular las tres órbitas simultaneamente pero el principal problema (pasos 7-10 de la encriptación) no se resolvería así.
- Información erronea en el paper: En el paper se especifica que las claves necesarias son $r_1, r_2, r_3, x_1, x_2, x_3$ y *ubicacion_inicial*, a no ser que no esté entendiendo bien el proceso las verdaderas claves serían r_1, r_3, x_1, x_3,*ubicacion_inicial*, L junto con el *ancho* o el *largo* de la imagen encriptada.

  En concreto r_2, x_2 ni siquiera se nombra en el papel en el proceso de desincriptación. Además sino se me escapa nada L es necesaria y no se puede deducir de ninguna manera.
- Imagen encriptada: Todo el proceso que se ha descrito tiene el objetivo de que la imagen encriptada tenga la apariencia de aleatoria pero en algunos casos, como en el descrito en este informe, aparecen algunos patrones en la imagen encriptada. Esto puede deberse a simplemente a fluctuaciones aleatorios puras y ser nuestra mente la que ve esos patrones pero sería interesante investigarlos por si suponen alguna debilidad del sistema.

- Propuestas:
  - Dos órbitas y usar $U(0,1)$: Como se ha comentado antes el uso de la segunda órbita parece innecesario, se podría usar perfectamente la implementación de una uniforme $U(0,1)$ la cual sí se puede paralelizar.
  - Velocidad: En general el método parece bastante bueno pero como se comentó antes los pasos 7-10 son demasiado lentos, si se consigue buscar un método que no dependa de encontrar un hueco al azar en el vector (o al menos que la probabilidad de encontrar el hueco vacío a la primera sea cercana a 1) se podría ganar muchísima velocidad.
  - Ocultar L, *ancho* y *ubicacion_inicial*: Hemos comentado antes que es necesario estas tres claves, se podría usar un vector $LM = L * n + 3$ siendo n entero en vez del propuesto originalmente y una tercera órbita para acomodar primero esas tres constantes y luego seguir con el proceso igual que antes.

    Esto permitiría ocultar más aún la información de la foto original.