



UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE INGENIERÍA

Año 2019 - 2^{do} Cuatrimestre

Simulacion (75.26)

Trabajo Final

Estudio y análisis de:

A pseudo-random numbers generator based on a novel 3D chaotic map with an application to color image encryption

Fecha de entrega: 14/09/2020

Contenido

1. Introducción	3
1.1 Metodología de trabajo	3
1.2 Organización del repositorio	3
2. Generador 3D-PLM	5
2.1 Fundamentos	5
2.2 Implementación	6
2.3 Pruebas realizadas	7
2.3.1 Análisis gráfico	7
2.3.2 Análisis de periodicidad	11
2.3.3 Tests estadísticos NIST	13
2.3.4 Tests de aleatoriedad	14
2.3.5 Entropía de la información	16
3. Aplicación a encriptación de imágenes	18
3.1 Fundamentos	18
3.2 Implementación	19
3.3 Pruebas realizadas	20
3.3.1 Análisis de espacio de claves	21
3.3.2 Análisis de sensibilidad de clave	21
3.3.3 Análisis de histograma	22
3.3.4 Correlación de píxeles adyacentes	23
3.3.5 Medición de discrepancia	24
3.3.5.1 SSIM	25
3.3.5.2 MSE	25
3.3.5.3 PSNR	25
3.3.6 Vulnerabilidad - ataques plaintext	26
3.3.7 Vulnerabilidad - ataques de oclusión	27
4. Uso en una solución real	31
4.1 Implementación	31
4.1.1 Requerimientos	31
4.1.2 Diseño	31
4.1.3 Ejecución y uso	34
4.2 Consideraciones y análisis de la solución	35
4.3 Pruebas realizadas	36
4.3.1 Pruebas con imágenes reales	37
4.3.2 Pruebas de performance	40
5. Conclusiones	43
R. Referencias	45

1. Introducción

En este trabajo se estudiará y analizará el *paper* de los autores M. L. Sahari e I. Boukemara titulado:

A pseudo-random numbers generator based on a novel 3D chaotic map with an application to color image encryption

El informe estará dividido en tres grandes secciones, que se describen en forma introductoria a continuación:

- En primera instancia, se considerarán los aspectos relacionados al generador caótico pseudo-aleatorio (CPRNG) 3D-PLM propuesto por los autores.
- Acto seguido, se trabajará con el método de encriptación y desencriptación de imágenes presentado en el *paper*.
- Por último, se aprovechará el trabajo realizado en los dos puntos anteriores para implementar un programa de consola que permita aplicar estos conceptos en una situación real.

1.1 Metodología de trabajo

La metodología de trabajo consistió en complementar lo visto en clase con investigación en diversas fuentes online y la bibliografía de la materia. Dichas fuentes se encuentran citadas a lo largo de este informe en las secciones correspondientes.

Durante el desarrollo del trabajo, se utilizó el lenguaje de programación Python para implementar los algoritmos y las pruebas realizadas. Las secciones concernientes al generador y al algoritmo de encriptación se encuentran en Jupyter *notebooks*, para que se los pueda correr por *segmentos* o *celdas*, mientras que la implementación final es un script de Python que puede ejecutarse directamente desde el *shell*.

1.2 Organización del repositorio

Todo el código generado como producto de esta investigación se encuentra en un repositorio de GitHub de acceso público, con el fin de ser descargado y ejecutado. Se puede encontrar en la siguiente URL:

<https://github.com/juanmg0511/fiuba-75.26-final>

Los *notebooks* con los distintos algoritmos y métodos ensayados, así como también el programa desarrollado se encuentran organizados en directorios. La estructura básica del repositorio es como se detalla a continuación:

Directorio	Descripción
1.3d-plm	Implementación del generador caótico pseudo-aleatorio 3D-PLM.
1.3d-plm/tests	Pruebas realizadas.
2.encryption.scheme	Implementación del algoritmo de encriptación de imágenes propuesto.
2.encryption.scheme/images	Imágenes utilizadas y generadas en las pruebas.
2.encryption.scheme/tests	Pruebas realizadas.
3.imgcrypt	Implementación de un programa aplicando los conceptos estudiados.
3.imgcrypt/extra	Cálculo de histogramas y plots de performance.
3.imgcrypt/images	Imágenes utilizadas con el programa.
LICENSE	Archivo con la licencia (GPLv3)
README.md	<i>Readme</i> para mostrar en la <i>home</i> del repositorio.

Tabla 1.2 - Organización del repositorio

Por último, también podrá encontrarse una copia del presente informe en el directorio raíz del repositorio.

2. Generador 3D-PLM

En esta sección se exponen los aspectos estudiados del generador caótico pseudo-aleatorio propuesto por los autores. Se comienza por una breve introducción sobre sus fundamentos y funcionamiento, para luego describir algunos detalles destacables de su implementación, así como también presentar las pruebas realizadas al mismo.

2.1 Fundamentos

Conceptualmente, la idea de los autores es plantear un generador de números pseudo-aleatorio a partir de la combinación de 2 sistemas caóticos separados, que no tienen tan buena performance por sí mismos, combinando las fortalezas de cada uno.

En particular, no habiendo visto esta aplicación en profundidad durante el curso, parece ser un uso más que interesante del comportamiento caótico de estos sistemas, uniendo si se quiere 2 temas que fueron vistos por separado en la materia.

De esta forma, se parte de dos sistemas conocidos, el *piecewise map* y el *logistic map*¹ para crear el *3D piecewise-logistic map*, como se define a continuación:

$$\begin{aligned} \text{3D-PLM} : T(x_k, y_k, z_k) \\ = \begin{cases} x_{k+1} = \Psi_{c_1}(x_k) + \Lambda_{c_2}(y_k, z_k) & \text{mod } 1 \\ y_{k+1} = \Psi_{c_1}(y_k) + \Lambda_{c_2}(z_k, x_k) & \text{mod } 1 \\ z_{k+1} = \Psi_{c_1}(z_k) + \Lambda_{c_2}(x_k, y_k) & \text{mod } 1 \end{cases} \end{aligned} \quad (2.1.1)$$

Siendo el *piecewise map*:

$$\Psi_{c_1}(x) = |1 - c_1 x| \quad (2.1.2)$$

Y el *logistic map*:

$$\Lambda_{c_2}(x, y) = c_2 x(1 - y) \quad (2.1.3)$$

La importancia de definir un generador que devuelva un punto en \mathbb{R}^3 se encuentra relacionado al hecho que se más adelante se lo utilizará para encriptar imágenes en formato **RGB**, donde el color de cada píxel está formado como la combinación de 3 canales, cada uno correspondiente a un color primario según la síntesis aditiva del color (rojo: **R**, verde: **G** y azul: **B**).

Los valores iniciales podrán ser tomados del cubo real definido por $[0, 1]^3$, y se cuenta con los parámetros de control reales c_1, c_2 .

¹ [3] pág 80, similar al visto en clase pero con un input en 2 dimensiones, por lo que recibe el nombre de *2D logistic map*.

Recordando lo visto en el curso, se deberá verificar que:

- Los números deben tener una distribución correcta, por ejemplo uniforme, y no deben estar correlacionados.
- La secuencia debe tener un período suficientemente largo.
- La secuencia se tiene que poder reproducir (replicar experimentos).

Adicionalmente, los autores extienden² este sistema agregando más dimensiones, para definir el 3ND-PLM. El objetivo es hacer más seguro el generador para la aplicación criptográfica, pero el planteo queda parametrizado en función del número N y no recibe tratamiento adicional. De esta forma, todo el desarrollo del *paper* y este trabajo se centran en la versión definida en (2.1.1): se entiende que por analogía los resultados obtenidos aplicarán al sistema extendido.

2.2 Implementación

Para implementar este sistema, se optó por utilizar el lenguaje Python, mediante *notebooks* de Jupyter. Esto permite ejecutar el código por segmentos, o celdas, dando así un mejor control y capacidad de *tuning* a los algoritmos.

Se tomó esta decisión para lograr que el código sea estándar, universal y que esté dotado de un buen nivel de portabilidad. Dado que el *paper* no incluye el código, se realizó toda la implementación, tanto de los algoritmos como de las pruebas, en forma original.

La implementación del generador es bastante sencilla, se realizó como tres funciones, una principal tMap, y dos auxiliares: pMap y lMap. Cada una de ellas corresponde a los sistemas definidos en (2.1.1), (2.1.2) y (2.1.3), respectivamente.

Se reproduce el código a continuación:

```
#3D piecewise-logistic map
#3D-PLM

#Definición de parámetros de control: c1, c2, reales
#Como define el paper, c1=c2=20
c1 = 20
c2 = 20

#Implementación de funciones auxiliares
#Picewise map
def pMap(x):
    "piecewise map: función que dado x y un parámetro de control real c1, calcula Ψ(x)"
    return (abs(1 - (c1 * x)))

#Logistic map
def lMap(x, y):
    "2D logistic map: función que dados x e y, y un parámetro de control real c2, calcula Λ(x, y)"
    return (c2 * x * (1 - y))
```

² [1], sección 2.1.

```
#Implementación del 3D piecewise-logistic map (3D-PLM), T
def tMap(x, y, z):
    "3D piecewise-logistic map: función que dados x, y, z, los parámetros de control reales c1 y c2,
    calcula T(x, y, z)"
    x = (pMap(x) + lMap(y, z)) % 1
    y = (pMap(y) + lMap(z, x)) % 1
    z = (pMap(z) + lMap(x, y)) % 1

    return x, y, z
```

El archivo con la implementación permite ejecutar el generador ya configurado³ y obtener los primeros 100 números de la serie. Puede encontrarse en el repositorio, bajo el directorio:

1.3d-plm

2.3 Pruebas realizadas

En esta sección se presentan en detalle las pruebas realizadas al generador. Se apunta a implementar y ejecutar las pruebas realizadas por los autores, con el fin de replicar los resultados publicados y demostrar que el generador es bueno. Adicionalmente se aplicarán algunas de las pruebas vistas en el curso.

A modo de comienzo, y en línea con la tarea a realizar, se reproduce⁴ una cita del año 1951 del Profesor D. H. Lehmer, pionero de la teoría numérica computacional:

A random sequence is a vague notion ... in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests traditional with statisticians ...

Los archivos con los tests pueden encontrarse en el repositorio, bajo el directorio:

1.3d-plm/tests

2.3.1 Análisis gráfico

Una primera batería de pruebas a efectuar sobre el sistema consiste en el análisis gráfico de diversos aspectos, tanto de las características del sistema en sí como de las series de números generados a partir de ciertas condiciones iniciales. Esto tendrá como objetivo entender el comportamiento del sistema, pero a su vez definir los valores de los parámetros de control.

El análisis comienza viendo un diagrama de bifurcación⁵ en el plano c_1-c_2 , realizado en forma de *heat map*, y contabilizando la cantidad de ciclos presentes para cada set de valores. En particular,

³ Como se verá más adelante, esto implica fijar los parámetros de control en $c_1=c_2=20$ y los valores iniciales también están puestos, con el ejemplo (0.411, 0.321, 0.631).

⁴ [4], Cap. 9, pág. 1.

⁵ [3]. Pág 132.

en el gráfico se varía el valor de estos parámetros entre 0 y 20, con el objetivo de visualizar áreas donde pudiera darse el caos, necesario para el correcto funcionamiento del generador:

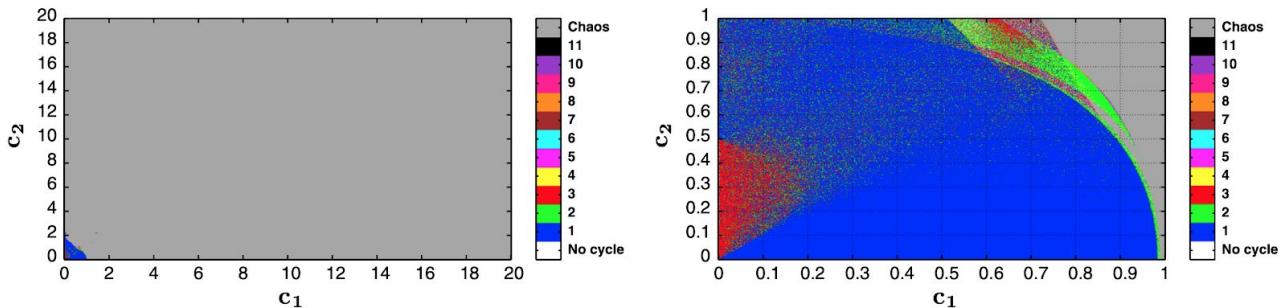


Imagen 2.3.1-1 - diagrama de bifurcación en el plano c_1 , c_2 y detalle

Como se puede apreciar, salvo por la región $[0,1]^2$, se observa un comportamiento posiblemente caótico a lo largo de la recta $c=c_1=c_2$, por lo que de aquí en adelante se trabajará con esos valores igualados.

Con esta conclusión, se pasa a trabajar con cada componente del sistema por separado, graficando sus respectivos diagramas de bifurcación y exponente de Lyapunov⁶, en función del valor de c .

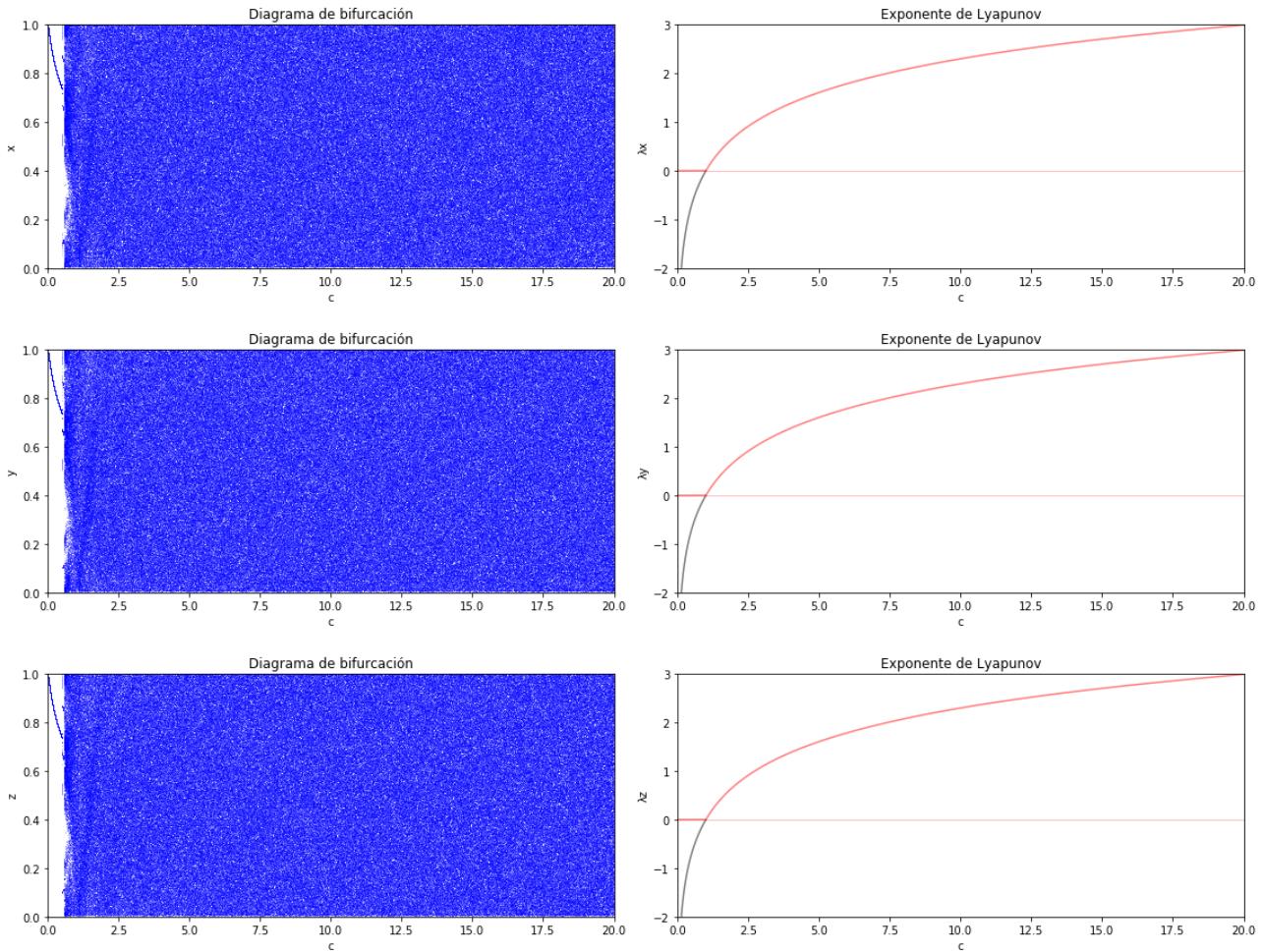
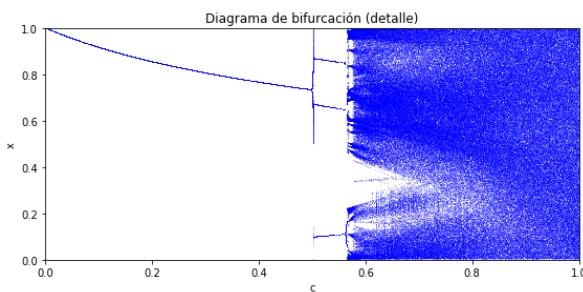


Imagen 2.3.1-2 - diagrama de bifurcación y exponente de Lyapunov para las componentes x,y,z

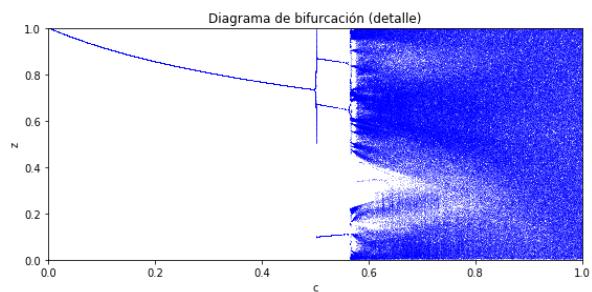
⁶ [3]. Pág 157.

Puede verse que los resultados obtenidos son consistentes con los que fueron publicados en el *paper*. Las bifurcaciones comienzan en $c < 1$, con un comportamiento totalmente caótico al alcanzar $c=20$. El valor del exponente de Lyapunov confirma lo dicho, ya que alcanza su valor máximo para dicho número.

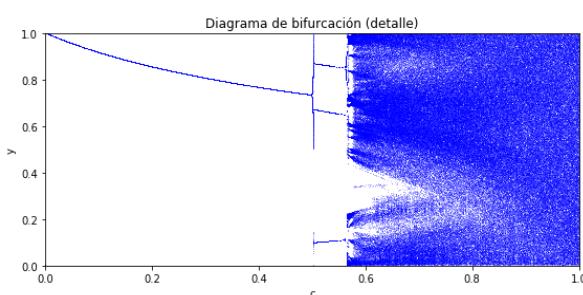
Sin embargo, al hacer un acercamiento para c entre 0 y 1, se encuentra una sorpresa, dado que los diagramas no coinciden con los publicados:



(a)



(c)



(b)

Imagen 2.3.1-2 - aumento diagrama de bifurcación para las componentes x,y,z respectivamente

Se investigaron las posibles causas de esta diferencia, y como se puede apreciar en el código entregado, las visualizaciones se realizaron en forma manual: al utilizar por ejemplo un *logistic map* se obtiene el gráfico esperado. Si bien se podría haber hecho un estudio analítico, lo importante del caso es demostrar que el comportamiento del sistema es caótico para $c=20$, y fijar esa cifra para su uso con el generador, lo cual más allá de esta discrepancia es posible afirmar.

Otro aspecto del sistema a estudiar consistió en ver la sensibilidad del sistema a pequeñas variaciones introducidas en los valores iniciales, conocido también como el *efecto mariposa*⁷. Para hacer esto, se computaron dos series de 100 elementos cada una, con las siguientes condiciones iniciales:

⁷ [3]. Pág. 155.

Serie 1

$$X_0 = 0,411$$

$$Y_0 = 0,321$$

$$Z_0 = 0,631$$

con $\delta = 10^{-6}$

Graficando para las componentes x, y,z se obtiene:

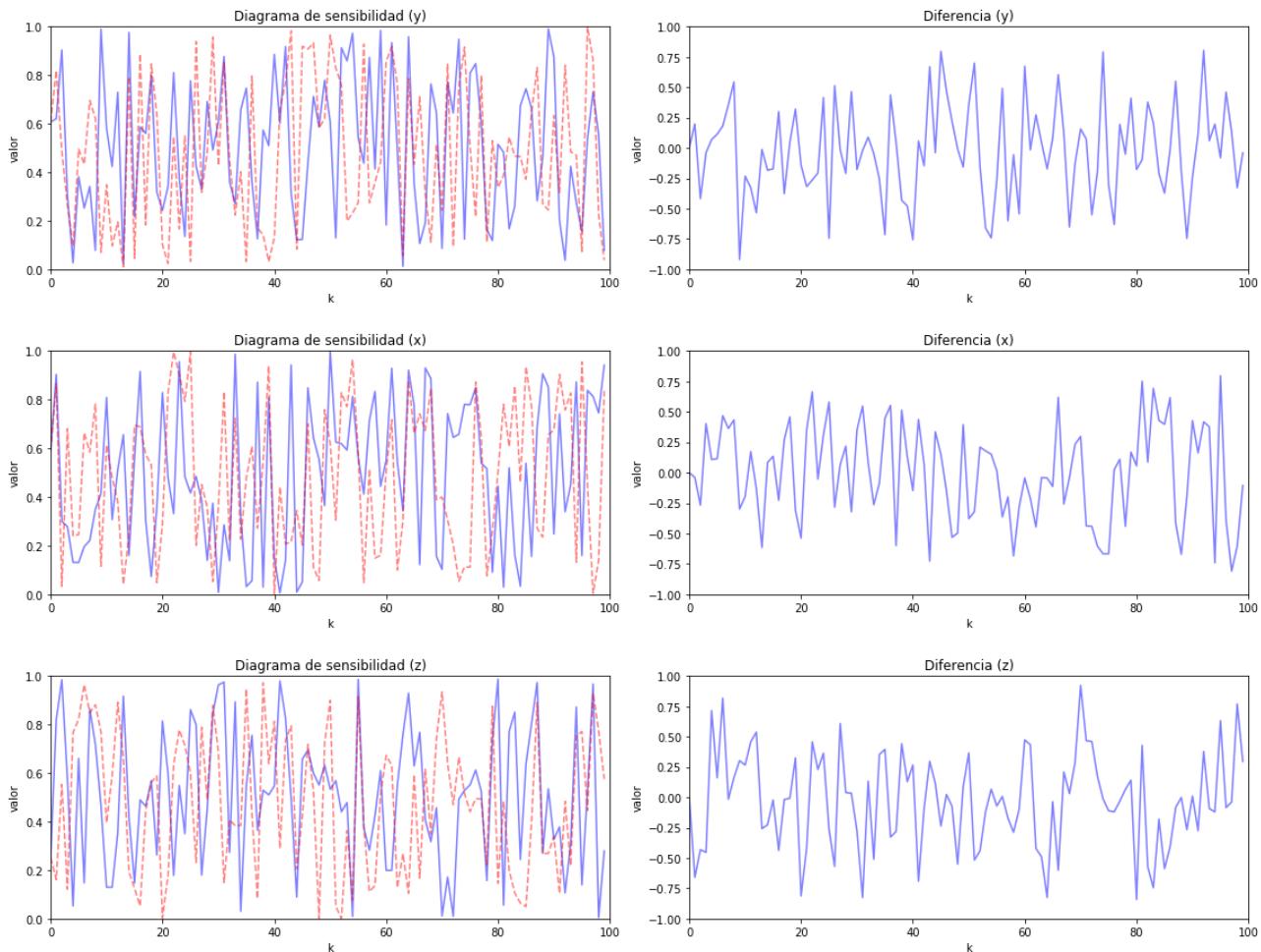


Imagen 2.3.1-3 - sensibilidad para las componentes x,y,z respectivamente

Los resultados obtenidos coinciden con los publicados, y demuestran un comportamiento típicamente caótico, donde una variación infinitesimal en los valores de entrada producen dos salidas completamente distintas. En los gráficos esto puede verse en forma superpuesta (izquierda, con la serie original en azul y la alterada en rojo), o de diferencia (derecha).

Se destaca que esta propiedad va a ser de suma importancia a la hora de utilizar el generador como piedra fundamental de un criptosistema.

Un último análisis realizado en forma gráfica consistió en un test espectral, que permite chequear en forma simple la existencia de algún patrón en las secuencias generadas.

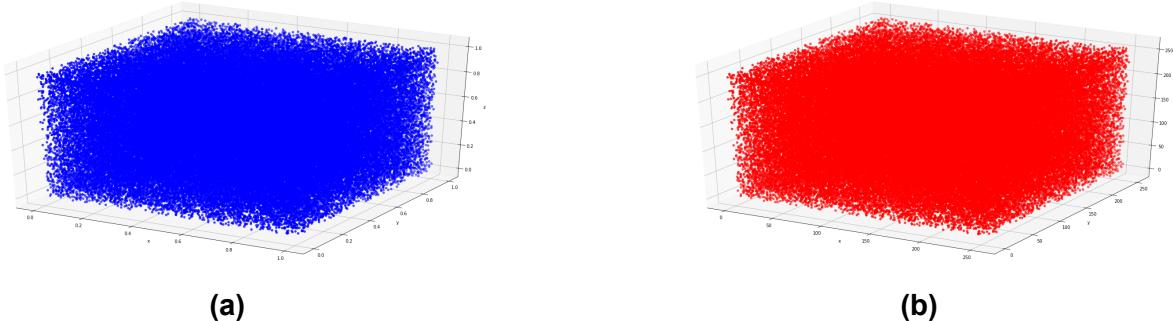


Imagen 2.3.1-4 - test espectral 3D-PLM, espacio de fases continuo (a) y discretizado (b)

Se generaron 10^6 valores y se partió de las mismas condiciones iniciales que en el test de sensibilidad. Como se ve, los gráficos no exhiben patrones, y el comportamiento es de carácter uniforme. Además de probar con el espacio continuo (Imagen 2.3.1-4 a), también se hizo la prueba con el espacio de fases del sistema discretizado mediante una función especial, que se definirá y utilizará para la construcción del criptosistema. Los resultados son análogos a los producidos en el *paper*.

Los archivos con la implementación pueden encontrarse en el repositorio, bajo el nombre:

```
3d-plm.bifurcation.lyapunov.ipynb
3d-plm.sensitivity.ipynb
3d-plm.espectral.ipynb
```

2.3.2 Análisis de periodicidad

Una vez establecidos los parámetros del sistema, y hecho el análisis básico, se pasa a estudiar la periodicidad del generador propuesto. Como se mencionó en la sección 2.1, se espera que un generador pseudoaleatorio tenga un período adecuadamente largo. Dado que el sistema objeto de estudio es una combinación de dos sistemas, se analizará en primera instancia los períodos del *piecewise map* y del *logistic map*.

Resulta fundamental comentar que este estudio se da, para los sistemas caóticos, en un contexto de aritmética de precisión finita, es decir que los números reales son representados con una secuencia de **b** bits, en particular siguiendo el estándar IEEE-754⁸.

A raíz de esto, se estudiará el período de los sistemas variando la precisión del sistema mediante una biblioteca de aritmética de precisión variable, gmpy2⁹. La búsqueda de los períodos se efectuó en forma práctica¹⁰ para una precisión de entre 2 y 32 bits, para secuencias de 2^{20}

⁸ Ver [7].

⁹ Ver [8].

¹⁰ [9], Pág. 122.

elementos. Cada secuencia se computó n veces, con n variable entre 2 y 100, partiendo de condiciones iniciales generadas pseudo aleatoriamente con el generador de Python. Como semilla para este generador se utilizó el número de padrón del autor de este trabajo.

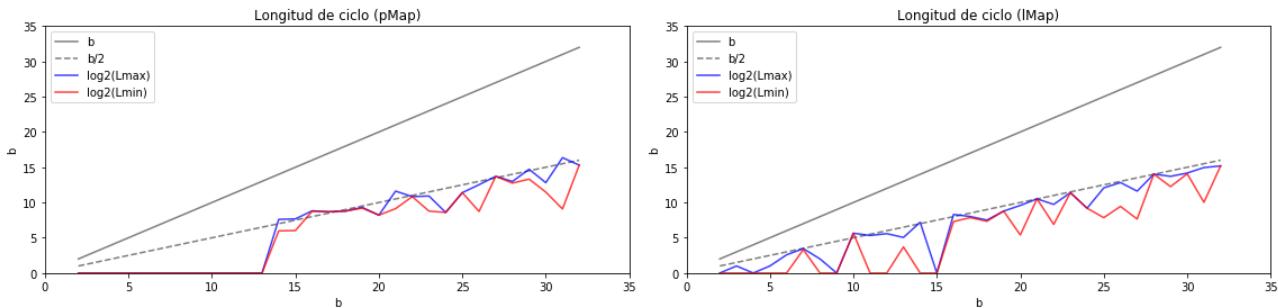


Imagen 2.3.2-1 - períodos máximo y mínimo para pMap y IMap respectivamente

Para cada conjunto de n secuencias, fueron seleccionados los períodos mínimo y máximo, y el resultado fue graficado para apreciar los resultados. En el caso del *piecewise map* y el *logistic map* se nota que los valores máximos apenas alcanzan el 50% (línea punteada) de 2^b , por lo que estos períodos no son considerados lo suficientemente largos.

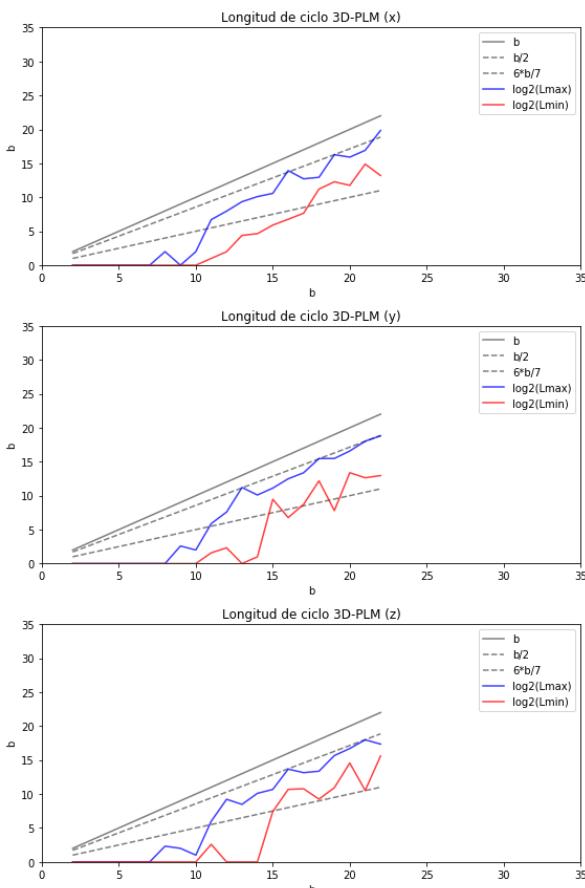


Imagen 2.3.2-2 - períodos máximos y mínimos para las componentes del 3D-PLM

Para el caso del 3D-PLM, se puede ver a simple vista que los valores son mucho más altos, y aproximan el 85% de 2^b , como lo indica la línea punteada superior. Esto permite demostrar que el generador propuesto mejora este aspecto débil de los dos sistemas estudiados anteriormente, y obtiene resultados que pueden considerarse adecuados.

Un dato interesante que se desprende de los gráficos es que estos no llegan hasta los 32 bits de precisión. Esto se debe a que la cantidad de valores generados en cada serie no alcanzó para detectar ciclos más allá de los 22 bits, por lo que estos valores se omitieron, para no ensuciar el gráfico. Todos los intentos de generar 2^{32} valores resultaron infructuosos, principalmente debido a limitaciones de memoria.

Sin embargo, es posible proyectar las curvas y estimar los valores faltantes, que son consistentes a los obtenidos por los autores en su trabajo.

Se recuerda, además, que la implementación en Python del generador utiliza el estándar IEEE-754 con 64 bits de almacenamiento, por lo que se puede estimar el período real en el 85% de 2^{52} , es decir $5,25 \cdot 10^{15}$.

Los archivos con la implementación pueden encontrarse en el repositorio, bajo el nombre:

```
pmap.lmap.periodicity.ipynb  
3d-plm.periodicity.ipynb
```

2.3.3 Tests estadísticos NIST

Para probar el generador, los autores utilizan la suite de test estadísticos del NIST¹¹. Se generaron archivos de entrada mediante un *notebook*, con una muestra de 10^6 elementos, encodeada en *arrays* con la representación IEEE-754 de 64bits de cada número, que fueron guardados tanto en formato ASCII y binario.

Todos los intentos por correr la *suite* fallaron. Si bien el paquete se pudo *buildear* y correr con los ejemplos suministrados, con cualquier archivo ajeno el programa acusó errores de *underflow*, y cuando se intentó correr para todos los bits de la secuencia de 10^6 elementos, directamente el error fue *segmentation fault*. Se probó la suite tanto en macOS como en Linux, y hasta una versión mejorada precompilados para Windows¹², con el mismo resultado.

Como alternativa se pensó en implementar los tests manualmente¹³, pero esta opción fue luego descartada por el tiempo que insumiría. Es una pena no poder someter el generador a estas pruebas: como alternativa en la sección siguiente se harán algunas de las vistas a lo largo del curso.

El archivo con la implementación para la generación de los archivos utilizados, tanto en formato ASCII como binario, puede encontrarse en el repositorio, bajo el nombre:

```
NIST.suite.input.ipynb
```

¹¹ Ver [10].

¹² Ver [11].

¹³ El manual de la *suite* incluye una descripción completa de los 15 tests, lo que permitiría realizar una implementación.

2.3.4 Tests de aleatoriedad

Dada la imposibilidad de ejecutar la suite de pruebas del NIST, como se detalló en la sección anterior, se realizarán una serie de tests vistos en el curso¹⁴, para verificar la aleatoriedad de las secuencias generadas mediante el 3D-PLM.

Se comienza mostrando la distribución obtenida al generar una secuencia de 10^6 números, partiendo de los valores iniciales (0.411, 0.321, 0.631), que aparenta ser uniforme en todas sus dimensiones.

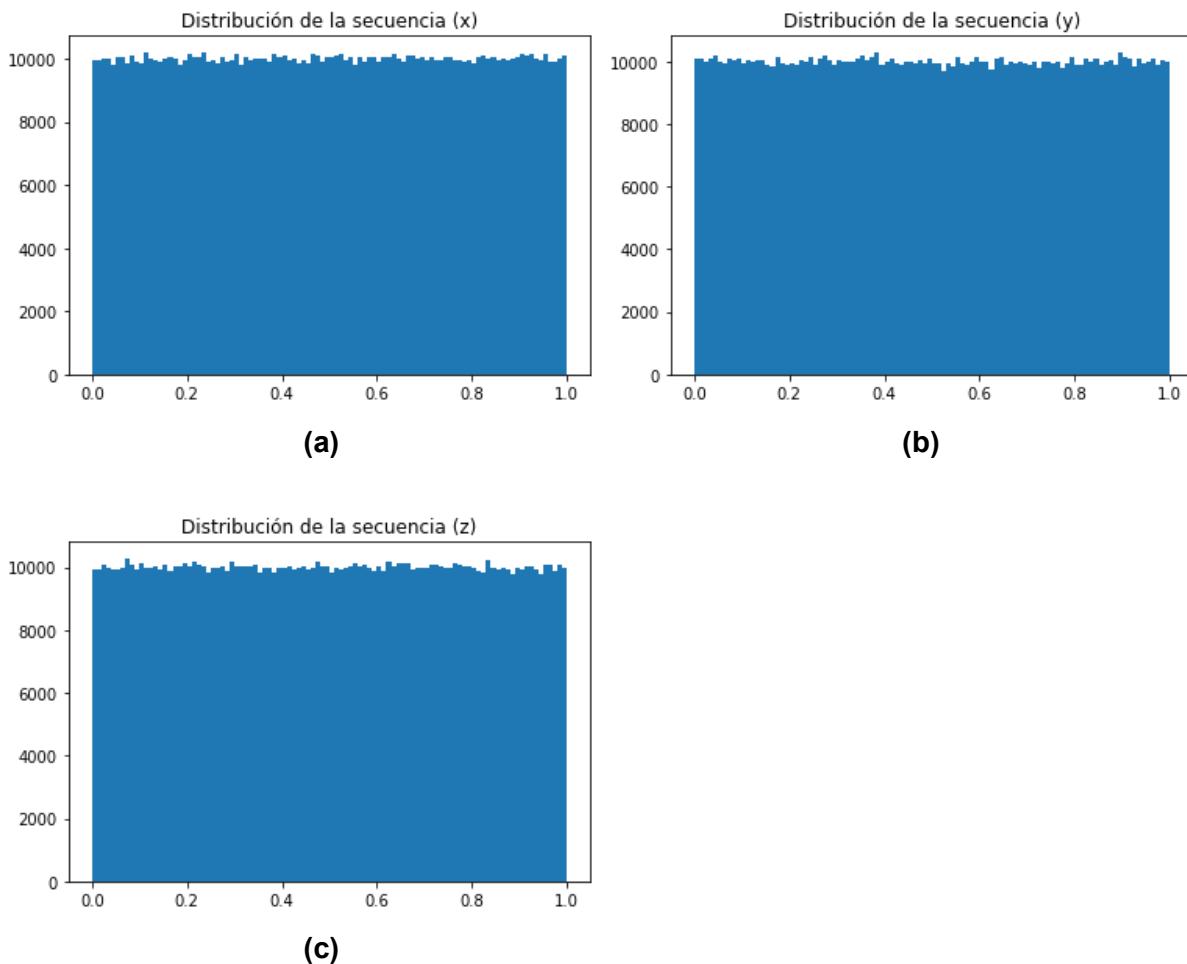


Imagen 2.3.4-1 - distribución de x, y, z sobre 100 bins, para 10^6 números generados por 3D-PLM

Se realizarán las siguientes pruebas **generales** sobre la serie:

- Test χ^2 sobre los valores de los bins del histograma.
- Test de Independencia sobre los valores de la secuencia.
- Test de Kolmogorov-Smirnov (K-S) sobre los valores de la secuencia.

¹⁴ [21], Secciones 3.3.1 y 3.3.2.

Y las siguientes pruebas **empíricas**:

- Test de frecuencia.
- Test serial.
- Test de gap, para $\alpha=0.3$ y $\beta=0.6$.
- Test de runs.
- Test espectral en 2D¹⁵, para los planos x-y, z-x y z-y.

Para esta secuencia, utilizando un nivel de confianza del 95%, los valores obtenidos son:

Test	C	E	pValue ¹⁶	Resultado
χ^2	x	530.9259	0.1560	OK
	y	501.3399	0.4621	OK
	z	509.8270	0.3589	OK
Independencia	x	109.2102	0.2269	OK
	y	116.1736	0.1144	OK
	z	91.9426	0.6794	OK
K-S	x	0.0007	0.5902	OK
	y	0.0013	0.5427	OK
	z	0.0007	0.5806	OK
Frecuencia	x	1.3735	0.9992	OK
	y	5.1458	0.8813	OK
	z	0.1536	0.9999	OK
Serial	x	5.2554	0.8114	OK
	y	16.2401	0.6230	OK
	z	6.51506	0.6874	OK
Gap	x	0.0001	1.0	OK
	y	0.0001	1.0	OK
	z	0.0001	1.0	OK
Runs	x	1.5380	-	OK
	y	0.5460	-	OK
	z	1.2620	-	OK

Tabla 2.3.4-1 - resultados de los tests de aleatoriedad para 10^6 puntos partiendo de (0.411,0.321,0.631)

¹⁵ Recordar que el test espectral en 3D ya fue realizado en la sección [2.3.1](#).

¹⁶ La implementación del runs test no devuelve el *pValue*, pero define un E_c de 1.96 para un nivel del 95%.

Finalmente, se genera un test espectral en 2 dimensiones para los planos x-y, z-x y z-y, generando unos 10^6 valores y partiendo de las mismas condiciones iniciales.

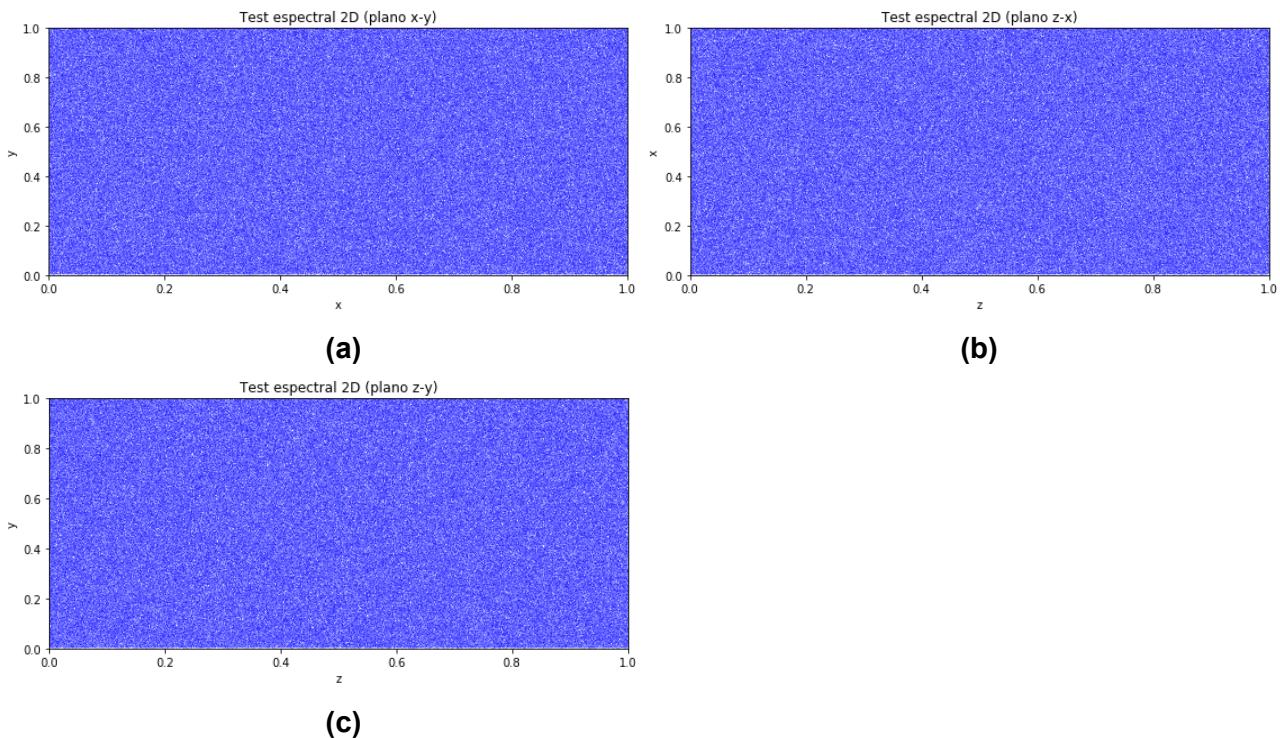


Imagen 2.3.4-2 - test espectral para los planos x-y, z-x y z-y

Como se puede ver, la secuencia generada pasa todos los test, tanto los generales como los empíricos, por lo que se puede decir que es de buena calidad.

Con el fin de testear un poco más a fondo el desarrollo, se generaron 100 secuencias a partir de condiciones iniciales aleatorias (como semilla se utilizó el número de podrón del autor de este trabajo) y se repitieron todos los tests.

El archivo con la implementación puede encontrarse en el repositorio, bajo el nombre:

`3d-plm.tests.ipynb`

2.3.5 Entropía de la información

Una última prueba a realizar sobre el generador consiste en medir la entropía de la información¹⁷, lo que ayuda a ver la magnitud de la complejidad y aleatoriedad del sistema. Se espera un valor cercano a 8, que es el resultado de una señal perfectamente aleatoria.

¹⁷ [5], Pág. 51.

La entropía de la información puede calcularse como:

$$H(s) = - \sum_{i=0}^{2^n-1} P(s_i).log_2 P(s_i) \quad (2.3.6)$$

Dada la naturaleza del cálculo a realizar, estas pruebas fueron hechas discretizando el espacio de fases del sistema. Los resultados obtenidos fueron similares a los publicados:

Coordenada	Valor
x	7.9683
y	7.9685
z	7.9687

Tabla 3.2.6 - valores de entropía para el 3D-PLM

El archivo con la implementación puede encontrarse en el repositorio, bajo el nombre:

3d-plm.entropy.ipynb

3. Aplicación a encriptación de imágenes

Implementado el generador pseudo-aleatorio, se pasa a tratar el método de encriptación propuesto por los autores. Se comienza por una breve introducción sobre sus fundamentos y funcionamiento, para luego describir algunos detalles destacables de su implementación, así como también presentar las pruebas realizadas al mismo.

3.1 Fundamentos

La seguridad de un algoritmo de encriptación es determinada a partir de sus propiedades¹⁸ de *confusión* y *difusión*:

- La *confusión* puede obtenerse dada la *ergodicidad* del sistema caótico, siendo muy difícil predecir la posición final de un punto dada su posición inicial.
- La *difusión* puede obtenerse dada las propiedades del caos, donde las pequeñas variaciones amplifican y cambian el comportamiento del sistema en una forma altamente compleja.

El criptosistema planteado por los autores satisface estos dos principios, y puede utilizarse para la encriptación de imágenes, particularmente en el formato RGB.

Los pasos principales a seguir, para lograr encriptar una imagen, son:

1. *Pasos preliminares*: descomposición de la imagen en sus componentes RGB, y confección de 3 vectores (1 por canal) unidimensionales conteniendo todos los píxeles de la imagen.
2. *Generación de las condiciones iniciales*: seteando el parámetro de control c del 3D-PLM en 20, se toman 3 semillas para calcular junto con los datos de la imagen las claves de encriptación a utilizar.
3. *Obtención de la imagen mezclada*: los vectores del paso 1 son mezclados utilizando un algoritmo especial. Esta es la etapa en que se aplica *confusión*.
4. *Generación de máscaras caóticas*: se utiliza el 3D-PLM para calcular 3 vectores (llamados máscaras caóticas) y luego se utiliza una función de discretización para obtener las denominadas máscaras enteras. Esta es la etapa en que se aplica *difusión*.
5. *Producción de la imagen encriptada*: se suman los vectores mezclados a las máscaras enteras, y se toma el módulo 256. Se restablecen los componentes RGB con las dimensiones de la imagen original, generando una imagen cifrada.

La desencriptación se logra ejecutando el algoritmo en sistema inverso, con mínimas adaptaciones. Para más detalles, se puede consultar el *paper*.

¹⁸ [4], Pág. 163.

3.2 Implementación

Al igual que en el caso del generador, dado que el paper no contiene código¹⁹, ni se provee de una URL con archivos fuentes a alguna implementación, se diseñó y codificó el algoritmo propuesto desde cero, siguiendo los pasos expuestos por los autores.

Para ello se utilizó el lenguaje Python, y el código se dividió en un Jupyter *notebook* de acuerdo a los pasos mencionados en la sección anterior, a fin de poder ejecutarlo por partes y con la mayor claridad posible (el notebook permite seleccionar una imagen, encriptarla y recuperarla, verificando los resultados al terminar la operación).

En la primer celda, se encuentra la sección de configuración, donde puede seleccionarse un archivo y un *initial guess* para realizar la encriptación:

```
#SECCION DE CONFIGURACION
#Path del archivo original
path_original = "../images/lena_std.tif"
#Initial guess
xd0 = 0.411
yd0 = 0.321
zd0 = 0.631
#FIN SECCION DE CONFIGURACION
```

Además de esto, también se definen las funciones auxiliares, que son la implementación del 3D-PLM, una función para discretizar el espacio de fases y finalmente la función encargada de generar las máscaras caóticas. Al ejecutar esta y el resto de las celdas para los pasos de encriptación, se generará el archivo de salida y se informará la clave utilizada:

```
Imagen encriptada con clave K:
(0.4110000047244551, 0.3210000025965682, 0.6310000027632665)
```

Esta es la clave que debe utilizarse para recuperar el archivo original. En cuanto a la salida, se escribe con la misma extensión y el texto “_encrypted” en el nombre. En este caso sería: “lena_std_encrypted.tif”.

De continuar ejecutando el notebook, se recuperará la imagen, que se guardará de manera similar al caso anterior como “lena_std_decrypted.tif”. Finalmente, se comparan ambas imágenes para verificar que son en efecto las mismas:

```
Comparando '../images/lena_std.tif' y '../images/lena_std_decrypted.tif':
Las imágenes son iguales :)
```

El código se planteó para ser lo más portable posible, sólo se requiere de un paquete adicional a la *standard library* de Python:

¹⁹ El algoritmo de mezcla y su inversa se proveen en pseudocódigo en la sección 3.

Paquete	Descripción
PIL	Pillow ²⁰ , un fork de la Python Imaging Library. Utilizada para leer y escribir los archivos de entrada y salida.

Tabla 3.2 - requerimientos para la implementación

Esta implementación trajo aparejados ciertos desafíos, ya que fue necesario armar el algoritmo en base a una secuencia de pasos explicados más descriptivamente que en forma explícita, y para el caso del pseudocódigo suministrado, con algunas ambigüedades que hubo que resolver (principalmente en base a prueba y error).

Un ejemplo de esto son los valores de inicialización de las estructuras utilizadas, ya que en varias oportunidades se plantean fórmulas u operaciones como sumatorias especificando qué hacer con una estructura, pero no se mencionan los valores iniciales.

Otro caso es en el paso 4 (*generate the chaotic masks*) del algoritmo de encriptación, donde las instrucciones indican: “iterate the map n_t stages to obtain the chaotic orbit”. Si se siguen estas instrucciones al pie de la letra, habría que iterar el sistema al realizar el cálculo de cada píxel, desde el comienzo. Durante las pruebas con esta implementación se descubrió que el tiempo de ejecución era exageradamente elevado, y el uso de *profiling* reveló que el tiempo de procesamiento estaba siendo consumido en este paso.

Dado que se está iterando un sistema caótico y agregando un paso a medida que se avanza en el procesamiento de los píxeles, se optimizó este paso guardando el resultado del sistema para el píxel actual, y sólo calculando el próximo número de la secuencia en cada paso. Evidentemente esta es la solución que usaron los autores, dado que con esta mejora la solución corre en un tiempo razonable y es del orden publicado en el *paper*. En el notebook se encuentran ambas implementaciones y se puede verificar que el resultado de usar una o la otra es exactamente el mismo.

El archivo con la implementación puede encontrarse en el repositorio, bajo el directorio:

2.encryption.scheme

3.3 Pruebas realizadas

En esta sección se presentan las pruebas realizadas al desarrollo. Al igual que los autores del *paper*, se trabajó con la imagen Lena²¹, un conocido archivo y de particular historia, que se ha convertido en un estándar de los testeos realizados con imágenes en el mundo digital.

Una aclaración que resulta pertinente realizar es que existen muchas versiones de este archivo, principalmente en lo que respecta al formato. El original²² es en formato TIFF, con dimensiones de

²⁰ Ver [12].

²¹ Ver [13]. El nombre de la modelo es Lena, pero lo americanizó como Lenna, para evitar que fuese mal pronunciado.

²² Disponible en: http://www.lenna.org/lena_std.tif.

512x512x3. Los autores mencionan que se trata de la versión de ese tamaño, pero omiten la fuente del archivo y no hacen referencia al formato, por lo que puede haber pequeñas discrepancias en los valores de algunos píxeles respecto a la utilizada en este trabajo. Por lo tanto, producto de esto y alguna otra variable propia de la implementación (tampoco sabemos en qué lenguaje se trabajó en el *paper*), se buscará obtener resultados coherentes con los publicados, y no exactamente los mismos.

Al igual que la implementación del algoritmo principal, fue necesario codificar todas las pruebas realizadas, para lo que se utilizó nuevamente el lenguaje Python. El objetivo de esta sección es entonces desarrollar y someter la implementación propia a las pruebas realizadas por los autores, y replicar los resultados por ellos obtenidos.

La demostración empírica de la complejidad computacional se efectuará en la sección [4.3.2](#). Para todas estas pruebas se fijará el parámetro de control c en 20, n_t en 10^5 y el *initial guess* (x_0, y_0, z_0) en (0.411, 0.321, 0.631).

Los archivos con los tests pueden encontrarse en el repositorio, bajo el directorio:

2.encryption.scheme/tests

3.3.1 Análisis de espacio de claves

Un primer análisis concierne el espacio de claves de un algoritmo de encriptación, es decir el universo posible de claves a utilizar. Esto es muy importante, a fin de proteger los datos frente a un ataque por fuerza bruta. Según los autores, un valor de 2^{128} resulta aceptable. En el caso de su propuesta, el espacio de claves se basa en la representación estándar de los números reales en base 2 (punto flotante). Por lo tanto, si se usan s bits para representar los números (se resta el bit del signo), la siguiente relación permite determinar un valor de s para el cual el sistema tendrá un espacio de claves lo suficientemente grande:

$$2^{3 \cdot (s-1)}$$

Al tener 3 valores iniciales, codificados con el estándar IEEE-754 en $s = 64$ bits, se ve que se supera el valor de $s = 44$, por lo que el espacio de claves del criptosistema será adecuado para resistir un ataque por fuerza bruta.

Una última mención la tiene el uso del generador 3D-NPLM. En este caso, los autores plantean que con el uso de dicho generador en reemplazo del 3D-PLM se podría mejorar aún más el espacio de claves.

Asimismo, se esboza a modo de *remark* en la sección correspondiente cómo adaptar el algoritmo propuesto para que funcione con dicho generador. Sin embargo, queda planteado como una mejora, ya que por ejemplo el valor de N queda parametrizado y no se dan demasiados detalles.

3.3.2 Análisis de sensibilidad de clave

Esta es una prueba importante, y consiste en validar la capacidad del criptosistema en recuperar resultados completamente diferentes a partir de una pequeña variación en la clave utilizada. Se introduce un *delta* del orden de 10^{-16} en la componente x_0 , de la forma:

Imagen encriptada con clave K:
 $(0.411000047244551, 0.321000025965682, 0.631000027632665)$

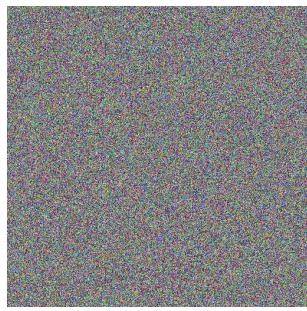
Clave K modificada:
 $(0.411000047244552, 0.321000025965682, 0.631000027632665)$

Como se puede apreciar, la diferencia es mínima, sin embargo al ejecutar el algoritmo, la verificación falla:

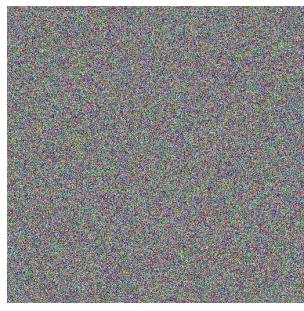
Comparando `'../images/lena_std.tif'` y `'../images/lena_std_decrypted_k2.tif'`:
 Las imágenes son distintas :(



(a) imagen original



(b) imagen encriptada



(c) imagen recuperada

Imagen 3.3.2-1 - encriptación y recuperación con clave alterada

En forma gráfica, se ve que el resultado es completamente incoherente (**c**), similar a la imagen encriptada (**b**). Este resultado es consistente con los resultados presentados en el *paper*.

El archivo con la implementación puede encontrarse en el repositorio, bajo el nombre:

`key.sensitivity.ipynb`

3.3.3 Análisis de histograma

Se verificaron los histogramas tanto de la imagen original como de la encriptada, de manera de ver cómo el algoritmo distribuye los valores píxeles en los canales individuales como en su conjunto:

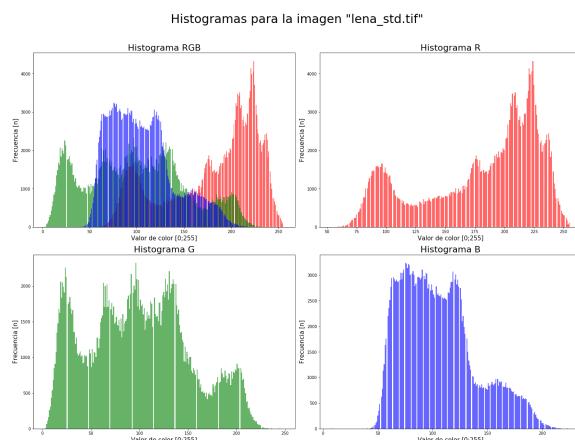


Imagen 3.3.3-1 - histograma de la imagen original

Puede verse que el algoritmo redistribuye los valores de modo que siguen una distribución aparentemente uniforme en la imagen encriptada.

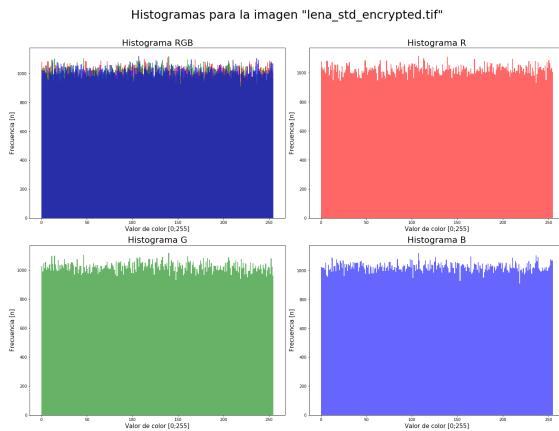


Imagen 3.3.3-2 - histograma de la imagen encriptada

Si bien el análisis del histograma es necesario para ver esto en forma gráfica, también se realizó un test χ^2 para asegurar la uniformidad, de acuerdo a la fórmula:

$$\chi^2 = \sum_{i=0}^{255} \frac{(o_i - e_i)^2}{e_i} \quad (3.3.3)$$

Para que el criptosistema sea seguro, los valores de la imagen encriptada deben ser mucho menores que los de la imagen original. Los resultados obtenidos fueron los siguientes, en concordancia con los publicados:

Imagen	R	G	B
Original	254333	113929	344338
Encriptada	274	284	231

Tabla 3.3.3-1 - valores de χ^2

Los archivos con la implementación pueden encontrarse en el repositorio, bajo los nombres:

```
hist.corr.original.ipynb  
hist.corr.encrypted.ipynb
```

3.3.4 Correlación de píxeles adyacentes

Otro análisis realizado es calcular los coeficientes de correlación de ambas imágenes, por canal y en promedio. Se hicieron pruebas tomando 2 píxeles adyacentes, en las direcciones diagonal, horizontal y vertical. Sobre esto último, los autores no aclaran el criterio a aplicar en los bordes, por lo que se tomó el de hacer la imagen “circular” (es decir que, por ejemplo, los píxeles

adyacentes en sentido horizontal al los de la última columna son los de la primera). La relación para calcular el coeficiente, dados (u, v) adyacentes, es:

$$\rho = \frac{\frac{1}{m.n} \sum_{i=1}^{m.n} (u_i - \mu(u))(v_i - \mu(v))}{\sqrt{\frac{1}{m.n} \sum_{i=1}^{m.n} (u_i - \mu(u))^2} \cdot \sqrt{\frac{1}{m.n} \sum_{i=1}^{m.n} (v_i - \mu(v))^2}} \quad (3.3.4)$$

Donde m y n son las dimensiones de la imagen, y μ representa el promedio. Los resultados obtenidos fueron:

Dirección	R	G	B	Promedio
Diagonal	0.9661	0.9519	0.9152	0.9444
Horizontal	0.9775	0.9662	0.9304	0.9580
Vertical	0.9880	0.9817	0.9568	0.9755

Tabla 3.3.4-1 - valores de correlación, imagen original

Dirección	R	G	B	Promedio
Diagonal	0.0028	0.0015	0.0003	0.0015
Horizontal	-0.0019	-0.0044	-0.0026	-0.0030
Vertical	-0.0013	0.0038	-0.0003	0.0007

Tabla 3.3.4-2 - valores de correlación, imagen encriptada

La idea es ver que el coeficiente para la imagen original es cercano a 1, y para la encriptada tiende a 0 (por lo que se destruye la correlación existente en la imagen original), valores consistentes con los resultados obtenidos y los publicados por los autores.

Los archivos con la implementación pueden encontrarse en el repositorio, bajo los nombres:

```
hist.corr.original.ipynb
hist.corr.encrypted.ipynb
```

3.3.5 Medición de discrepancia

La degradación de la imagen original es clave para garantizar la calidad de un sistema criptográfico. Con el objetivo de medir la distancia entre la imagen original y la encriptada, se realizaron mediciones con tres herramientas estadísticas:

- El índice de similaridad estructural (SSIM)
- El error cuadrático medio (MSE)
- El pico de la relación entre señal y ruido (PSNR)

Estas pruebas fueron implementadas utilizando el paquete **metrics** de la biblioteca de Python **skimage**²³, que incluye los tres tests.

El archivo con la implementación puede encontrarse en el repositorio, bajo el nombre:

`discrepancy.measurements.ipynb`

3.3.5.1 SSIM

Este índice varía entre -1 y 1, y el valor 1 se obtiene cuando las imágenes coinciden perfectamente. Para realizar esta pruebas se generaron 3 copias de la imagen, con sus componentes R, G y B. Cabe aclarar que estas imágenes son monocromáticas, y no tienen el tinte del canal asociado como se muestran en el *paper*. Los autores no ahondan en el método utilizado para calcular el índice, ni cómo generaron las imágenes; probablemente esto pudo haber sido hecho para que la representación sea más clara. En las pruebas realizadas en este trabajo, se utilizan las versiones monocromáticas.

Los resultados obtenidos son cercanos a 0, consistentes con los publicados, lo que indica la máxima distancia entre las 2 imágenes.

	R	G	B	Promedio
SSIM	0.0097	0.0087	0.0010	0.0010

Tabla 3.3.5.1-1 - valores de SSIM entre las dos imágenes

3.3.5.2 MSE

El MSE representa otra forma de medir la distancia. En este caso, los valores presentados en el *paper* son erróneos, probablemente por un error de transcripción, dado que las cifras mostradas para el PSNR son correctas y éstas utilizan el MSE en el cálculo (adicionalmente, el cálculo del PSNR con los valores de MSE publicados no da los resultados de PSNR que figuran en el *paper*).

Por lo general, se esperará un MSE alto, los resultados obtenidos son:

	R	G	B	Promedio
MSE	10626	9033	7098	8919

Tabla 3.3.5.2-1 - valores de MSE entre las dos imágenes

3.3.5.3 PSNR

Por último, el PSNR se puede explicar como la relación entre el valor de intensidad máximo en la imagen y el MSE. Para una buena encriptación se recomienda un valor de PSNR pequeño, menor a 10.

²³ Ver [14].

La fórmula para calcularlo es la siguiente:

$$PSNR = 20 \cdot \log_{10} \frac{\max_{i,j} I(i,j)}{\sqrt{MSE}} \quad (3.3.5.3)$$

Existen varias fórmulas equivalentes para calcular el PSNR²⁴, en particular la que se muestra en el paper contiene un error, ya que de utilizar 10 como constante de proporcionalidad debe elevarse al cuadrado el valor de intensidad máximo. A pesar de ello, los resultados mostrados son los correctos, y coinciden con los calculados con la implementación propia:

	R	G	B	Promedio
PSNR	7.8671	8.5725	9.6192	8.6862

Tabla 3.3.5.3-1 - valores de PSNR entre las dos imágenes

3.3.6 Vulnerabilidad - ataques plaintext

Un criptosistema resistirá adecuadamente los ataques de known-plaintext, chosen-plaintext y known-ciphertext si la imagen encriptada resulta significativamente diferente con un pequeño cambio en la imagen original, por lo general de un un píxel.

Para hacer esta prueba, se calculan el NPCR y el UACI entre dos imágenes encriptadas, generadas a partir de dos imágenes planas que varían en un píxel. Estas dos métricas son generalmente utilizadas para testear la influencia de un cambio de este tipo en imágenes. Como no se especifica en el paper, se tomó la imagen de Lena y se modificó a color rojo (255,0,0) la coordenada (309,241).

$$NPCR = \frac{\sum_{i,j} D(i,j)}{m \cdot n} \quad (3.3.6-1)$$

con

$$D(i,j) = \begin{cases} 0 & \text{si } C_1(i,j) = C_2(i,j) \\ 1 & \text{si } C_1(i,j) \neq C_2(i,j) \end{cases}$$

$$UACI = \frac{1}{m \cdot n} \cdot \sum_{i,j} \left[\frac{|C_1(i,j) - C_2(i,j)|}{255} \right] \quad (3.3.6-2)$$

La fórmula de UACI mostrada en el paper contiene un error, dado que se debe aplicar el valor absoluto a la diferencia entre los píxeles en el numerador.

²⁴ Ver [15].

Los resultados obtenidos muestran una buena performance, y son similares a los publicados por los autores:

	R	G	B	Promedio
NPCR	0.9961	0.9960	0.9960	0.9961
UACI	0.3353	0.3346	0.3354	0.3351

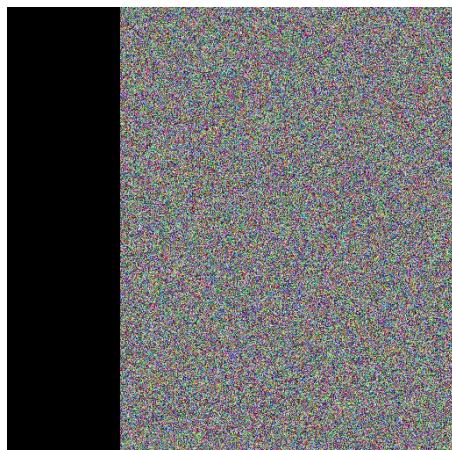
Tabla 3.3.6-1 - valores de NPCR y UACI entre las dos imágenes

El archivo con la implementación puede encontrarse en el repositorio, bajo el nombre:

`plaintext.attk.ipynb`

3.3.7 Vulnerabilidad - ataques de oclusión

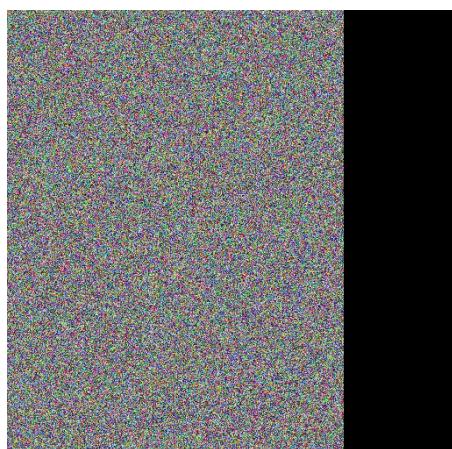
Por último, se hicieron pruebas de oclusión, que miden la capacidad de un criptosistema de recuperar imágenes dañadas. Para ello, se modificaron las imágenes encriptadas quitando un 25%, 50% y un 75% de los píxeles, y reemplazandolos con el valor (0,0,0), representativo del mayor daño posible. Los resultados se muestran a continuación:



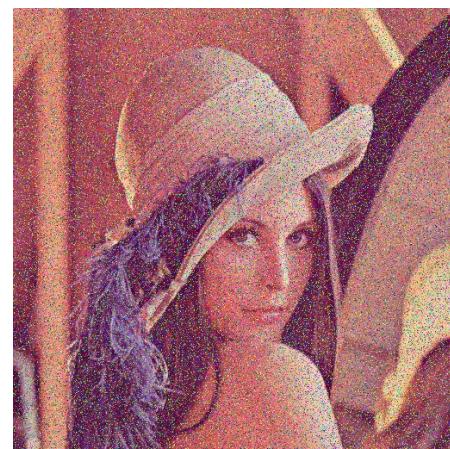
(a)



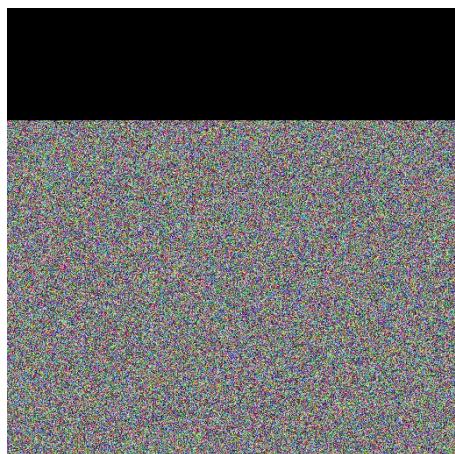
(b)



(c)



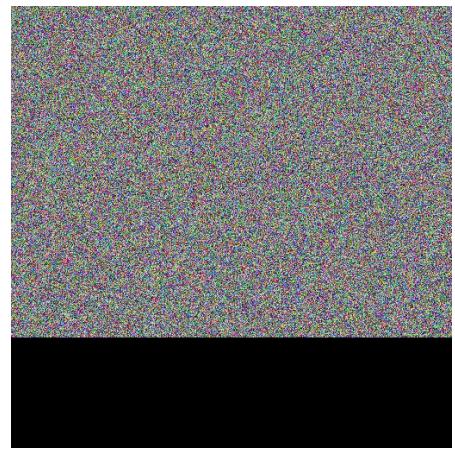
(d)



(e)



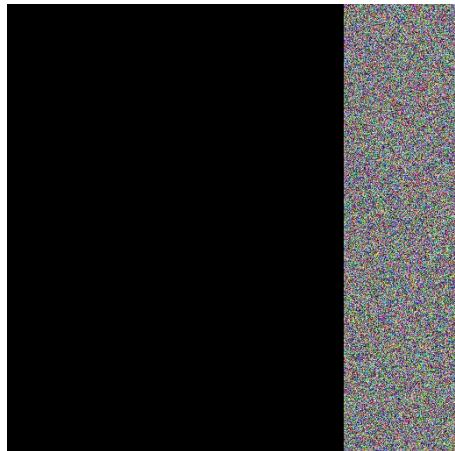
(f)



(g)



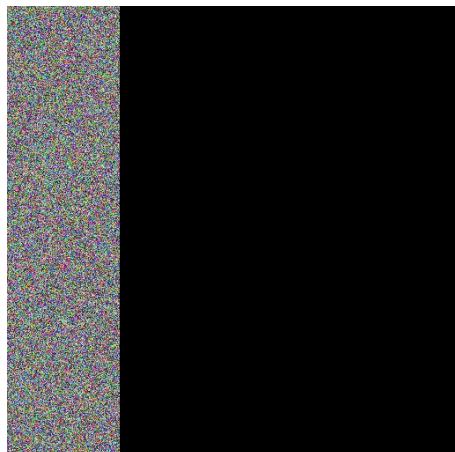
(h)



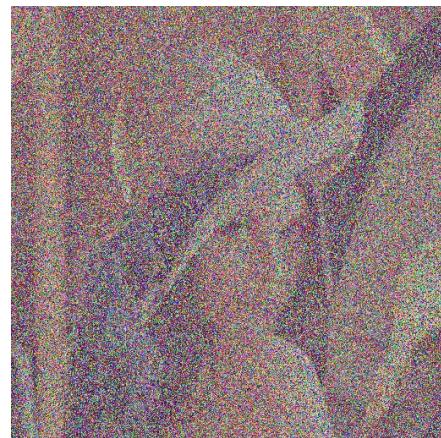
(i)



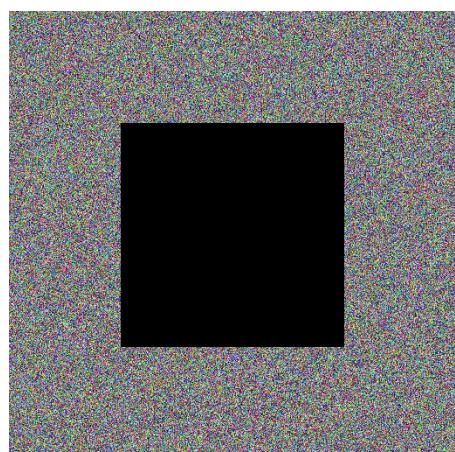
(j)



(k)



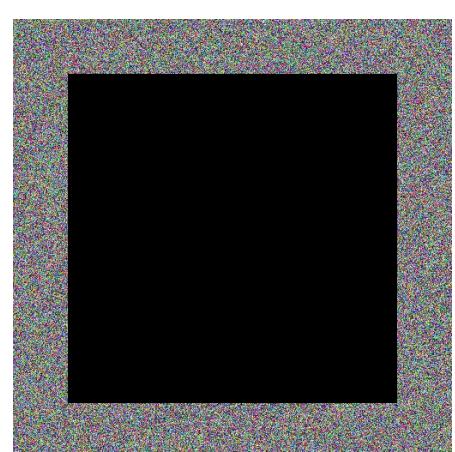
(l)



(m)



(n)



(o)



(p)

Imagen 3.3.7-1 - Resultados de los ataques de oclusión.

- Para los casos a, c, e, g se quitaron bandas de 128 píxeles, es decir el 25% de la imagen.
- Para el caso m, se removieron el 50% de los píxeles del medio de la imagen.
- Para los casos i, k, el número elegido fue de 384 píxeles o el 75%.
- Finalmente, para el caso o se removieron el 50% de los píxeles del medio de la imagen.

Como se puede apreciar, los resultados son equivalentes a los publicados por los autores. Aunque se ha realizado un ataque fuerte, las imágenes recuperadas pueden ser de todas formas reconocidas. Adicionalmente, el hecho de que la distribución de los histogramas sea uniforme en las imágenes encriptadas da inmunidad a este tipo de ataques.

El archivo con la implementación puede encontrarse en el repositorio, bajo el nombre:

`occlusion.attk.ipynb`

4. Uso en una solución real

Luego de completada la investigación sobre los dos temas centrales que trata el *paper*, se decidió implementar una solución real aplicando los algoritmos estudiados. El programa creado, **imgcrypt**, fue pensado como una aplicación de consola, completo y para uso productivo.

En esta sección se expone la implementación realizada, así como también las pruebas realizadas, más algunas consideraciones del uso de solución con imágenes reales.

4.1 Implementación

A fin de aprovechar el código ya realizado, el programa fue codificado en su totalidad en Python, pero agregando la funcionalidad necesaria.

La solución se compone enteramente de un programa para ejecutar por consola, cuenta con mensaje de ayuda y versión, más algunas opciones para permitir su ejecución en modo *batch* (supresión de salida por pantalla o *stdout*), y operar con formatos con compresión *lossy*, como JPEG, uno de los temas a solucionar a la hora de utilizar el método de encriptación ensayado.

4.1.1 Requerimientos

Para ejecutar el programa se requiere de un ambiente que cuente con Python 3, y los siguientes paquetes instalados:

Paquete	Descripción
PIL	Pillow, un fork de la Python Imaging Library. Utilizada para leer y escribir los archivos de entrada y salida.
tqdm	Barra de progreso.

Tabla 4.1.1-1 - Requerimientos de imgcrypt

El resto de los módulos utilizados pertenecen a la Python *standard library*, no existiendo el requerimiento de instalar paquetes adicionales.

4.1.2 Diseño

La solución se diseñó como una aplicación de consola, teniendo como premisa que pueda ser altamente portable y *batchable*. Para lograr lo primero, se trabajó en lo posible utilizando la biblioteca estándar de Python, por lo que el programa podrá ejecutarse en cualquier máquina que pueda correr Python 3 y cuente con los paquetes necesarios. Adicionalmente, se utilizó el idioma inglés para todas las salidas. En cuanto a lo segundo, el hecho de que se trate de una aplicación de consola facilita la ejecución en modo *batch*, pero para que realmente puede ser programada como parte de otro script o como tarea programada se decidió incluir una opción para suprimir

todas las salidas del programa (excepto en la encriptación, donde el programa solamente informa la clave utilizada).

Las opciones que fueron implementadas son (en **negrita** las obligatorias):

Opción	Argumento	Descripción
Ayuda	-h, --help	Imprime el mensaje de ayuda.
Versión	-V, --version	Imprime la versión del programa.
Silencio	-s, --silent	Oculta toda la salida del programa, para ejecución <i>batch</i> .
Sin pérdida	-l, --lossless	Utiliza un formato binario sin compresión para guardar las imágenes encriptadas.
Encriptar	-e, --encrypt	Ejecutar operación de encriptar.
Desencriptar	-d, --decrypt	Ejecutar operación de desencriptar.
Lento	-u, --unhurried	Utiliza la implementación original para la generación de las máscaras caóticas.
Clave	-k, --key	Clave a utilizar.

Tabla 4.1.2-1 - Opciones de configuración de imgcrypt

De no ingresar una operación explícitamente, el programa encriptará el *input*. En cuanto al diseño general del programa, la secuencia de una ejecución puede resumirse de la siguiente forma (se muestra una encriptación):

1. Parseo de la línea de comando (se utiliza la biblioteca getopt).
2. Lectura de la imagen de entrada (se utiliza la biblioteca PIL).
3. Encriptación de la imagen, con el algoritmo seleccionado.
4. Escritura de salida (se utiliza la biblioteca PIL o un archivo binario si se elige la opción *lossless*).
5. Impresión de la clave a utilizar para la desencriptación.
6. Escritura del código de retorno (0 para éxito o 1 para error) y vuelta al *shell*.

Los formatos soportados son todos los soportados por Pillow. Puede consultarse la lista completa en Internet.

Toda entrada a la aplicación es validada, para evitar que el programa falle: por ejemplo olvidar un argumento o especificar un archivo inexistente o inválido, reportará el error correspondiente, y en modo *verbose* se diseñó la solución para informar al usuario en todo momento del progreso de la operación. En los pasos más críticos en cuanto a performance, se muestra una barra de progreso que informa al usuario la velocidad de procesamiento (en píxeles por segundo) y una predicción de tiempo:

100% | ██████████ | 262144/262144 [00:00<00:00, 366181.88px/s]

Un tema no contemplado en el *paper* es qué hacer con dos aspectos importantes que constituyen una imagen: la *metadata* y el *perfil de color*. De existir en la imagen fuente, se entiende que deberían ser encriptados, pero se deja planteado como una mejora para una versión futura. En la versión actual, el programa descarta la *metadata* y el *perfil* al realizar la encriptación.

Dado que la solución produce como salida una imagen, existe un último problema a resolver, y es cómo manejar la operación para imágenes que usan compresión con pérdida, o *lossy*, por ejemplo JPEG. En este caso, la imagen encriptada no representará exactamente el resultado del algoritmo, ya que sin importar el nivel de fidelidad seleccionado, siempre aplicará compresión. Para estos casos, se creó un modo de operación especial, que escribe un archivo binario con la información RGB de la imagen y alguna *metadata* crítica en un diccionario, que luego es serializado. La estructura es la siguiente:

```
#Lossless mode, generate a custom dictionary with the image data to be saved
#TODO: compress dictionary to reduce size of dump
path_final = path_encrypted_lossless
im_original_data = {
    "imageFormat": im_original.format,
    "imageMode": im_original.mode,
    "imageSize": im_original.size,
    "ImageData": If
}
```

Esta modalidad aumenta significativamente el tamaño en bytes del resultado de la operación. Se podría mejorar, sin embargo, aplicando compresión *lossless* a este archivo.

A pesar de lo planteado se permite, sin embargo, utilizar cualquier modo con cualquier archivo. Se deja como responsabilidad del usuario entender los efectos de que tendrá el proceso de encriptación en su archivo. A modo de ayuda, se imprime un mensaje a modo de advertencia, si se detecta un formato de imagen que utiliza compresión *lossy*:

```
Opened file "images/lena_std.jpg", 55001 bytes read.
WARNING: file format uses lossy compression. Results may be unexpected, please use lossless '-l' flag.
```

Para todos estos casos, además, se guarda el grado de compresión para guardar la imagen con el nivel de compresión original.

4.1.3 Ejecución y uso

Para ejecutar el programa basta con llamarlo desde el *shell* del sistema operativo, proporcionando los argumentos necesarios y las opciones deseadas. Esto puede consultarse imprimiendo el mensaje de ayuda del programa, como se ve a continuación:

```
% python ./imgcrypt.py -h
imgcrypt: an RGB color image encryption/decryption scheme, using a pseudo-random number generator based on
a novel 3D chaotic map.

imgcrypt [options] -k key file

-h, --help      display this help and exit.
-V, --version   display version information and exit.
-s, --silent    mute all output.
-l, --lossless  save/read encrypted data in an alternative, lossless format.
-e, --encrypt   run in encrypt mode (default).
-d, --decrypt   run in decrypt mode.
-u, --unhurried use the unoptimized version of the chaotic masks generation algorithm.

-k, --key       initial guess for encryption or key for decryption.

imgcrypt -e -k "0.411;0.321;0.631" "lena_std.tif"
```

Asimismo, puede obtenerse información sobre la versión, invocando la opción correspondiente:

```
% python ./imgcrypt.py -V
imgcrypt v1.00 - 15/04/2020

Based on "A pseudo-random numbers generator based on a novel 3D chaotic map with an application to color
image encryption" by Mohamed Lamine Sahari & Ibtissem Boukemara (https://doi.org/10.1007/s11071-018-4390-z)

Created by Juan Manuel Gonzalez (juanmg0511@gmail.com)
FIUBA - 75.26 Simulación - 2c2019
```

A continuación se muestra, a modo de ejemplo, la *salida* en modo verbose para una encriptación y luego desencriptación.

Encriptación:

```
% python ./imgcrypt.py -e -k "0.411;0.321;0.631" "images/lena_std.tif"
Opened file "images/lena_std.tif", 786572 bytes read.

Encrypting image, please wait...
Step 1/5, preliminary steps: done!
Step 2/5, generation of the initial conditions: done!
Step 3/5, obtain the shuffled image:
100%|██████████| 262144/262144 [00:14<00:00, 18498.53px/s]
Step 4/5, generate the chaotic masks:
100%|██████████| 262144/262144 [00:00<00:00, 366181.88px/s]
Step 5/5, produce the cipher image: done!

Encrypted image "images/lena_std.tif" with key K:
```

```
[0.411000047244551;0.321000025965682;0.631000027632665]
```

```
Saved file "images/lena_std_encrypted.tif", 786572 bytes written.
```

Desencriptación:

```
% python ./imgcrypt.py -d -k "0.411000047244551;0.321000025965682;0.631000027632665"
"images/lena_std_encrypted.tif"
Opened file "images/lena_std_encrypted.tif", 786572 bytes read.

Decrypting image, please wait...
Step 1/4, preliminary stage: done!
Step 2/4, generate the chaotic masks:
100%|██████████| 262144/262144 [00:00<00:00, 372247.41px/s]
Step 3/4, obtain the shuffled image: done!
Step 4/4, retrieve the original image:
100%|██████████| 262144/262144 [00:14<00:00, 18686.69px/s]

Decrypted image "images/lena_std_encrypted.tif" with key K:
[0.411000047244551;0.321000025965682;0.631000027632665]

Saved file "images/lena_std_encrypted_decrypted.tif", 786572 bytes written.
Done.
```

Como se observa, el programa maneja en forma automática los nombres de archivo, para evitar un *re-write* accidental de alguno de los archivos involucrados en la ejecución.

4.2 Consideraciones y análisis de la solución

Habiendo hecho una evaluación del programa desarrollado desde la perspectiva de un estudiante de la carrera de Ingeniería en Informática y de un profesional de imagen²⁵, es posible destacar las siguientes consideraciones:

En primer lugar, se menciona que el método de encriptación propuesto trabaja exclusivamente con la información de color de los píxeles, en formato RGB²⁶. Por lo tanto, no aplica para una amplia gama de formatos altamente usados, como por ejemplo CMYK, usado principalmente en la industria de la impresión, o RGBA, donde se agrega un *alpha channel* (o transparencia) a un esquema RGB tradicional, ya prácticamente de uso universal en la web, en formatos como PNG. Podría argumentarse que puede transformarse a RGB cualquier imagen, pero esta operación no siempre es exacta, por lo que se pueden introducir variaciones en las imágenes (en el caso de las transparencias directamente se perderían).

El método de encriptación propuesto tampoco no contempla el tratamiento del perfil de color asociado a las imágenes. Esto es sumamente importante a la hora de hacer trabajos de precisión, ya que dicha información se usa para mostrar la imagen correctamente, ya sea en una pantalla o

²⁵ El autor de este trabajo es egresado de la escuela de la A.F.P.R.A. y se desempeña actualmente en tareas tanto de fotografía como de diseño.

²⁶ Los autores manifiestan en el *paper* que el algoritmo “puede ser implementado para cualquier tipo de imágenes, en especial imágenes RGB”. Sin embargo, el algoritmo se desarrolla exclusivamente para el caso RGB. No vuelve a mencionarse el tema, ni cómo adaptar o extender la propuesta a otros formatos de representación.

en papel una vez impresa. Una imagen sin perfil de color es asignado uno por default, y dado que no todos los espacios de color contienen los mismos colores, pueden producirse errores de interpretación (el concepto es análogo al *encoding* de un archivo de texto) y por lo tanto mostrarse una imagen con variaciones de color respecto a la original. Este problema podría solucionarse copiando la información del perfil de color a la imagen resultante, sin encriptar, dado que no revela datos sensibles, aunque dejaría una parte de la imagen sin encriptar. Se aclara que no todos los formatos permiten adjuntar un perfil de color, lo que complica la operación.

Asimismo, tampoco se establece el tratamiento de la *metadata* de los archivos. La metadata contiene un sinfín de información útil, que debe conservarse. A diferencia del perfil de color, en este caso si sería necesario encriptarla en caso de existir: por citar un ejemplo simple, una foto cuyo tag GPS revele que fue sacada en un astillero podría contener información de defensa sensible. Para resolver este punto debería recurrirse a un esquema de encriptación distinto, y encontrar la forma de guardar todo el producto junto, en cual caso probablemente sea más práctico encriptar todo el archivo como una tira de bytes con un método tradicional.

Finalmente, no se especifica qué hacer con archivos que utilizan compresión con pérdida. Si bien se entiende que este punto se deja a criterio de cada implementación y no tiene que ver con el método propuesto en sí, fue un punto a resolver a la hora de desarrollar una *utility* real.

Como punto positivo, se puede mencionar que el archivo encriptado sigue siendo legible como una imagen, lo que permitiría realizar encriptaciones *in-place*. Se agrega además que en un uso no tan crítico desde el punto de vista técnico de las imágenes (por ejemplo para encriptar planos o esquemas), su uso sería viable.

Sobre la implementación realizada, y el pseudocódigo presentado en el paper, una mejora interesante sería la paralelización de las operaciones sobre matrices, a fin de acelerar la ejecución de los algoritmos. La compresión del archivo de salida para el caso *lossless* también es una optimización a tener en cuenta para el desarrollo aquí presentado.

Como comentario final, se destaca que con un sistema de encriptación tradicional **ninguno** de los puntos anteriormente expuestos representa un problema, dado que la mayoría de las soluciones trata los archivos como una tira de bytes, independientemente de su contenido.

4.3 Pruebas realizadas

Además de las diversas pruebas de funcionamiento básico, como ser el parser de argumentos o validación de entrada, se realizaron pruebas con dos objetivos principales:

1. Probar la solución con imágenes reales.
2. Verificar empíricamente la performance del algoritmo y su complejidad computacional, y compararlo con una solución de encriptación de amplio uso en el mundo real.

Los archivos con las pruebas y sus resultados pueden encontrarse en el directorio del repositorio correspondiente al programa **imgcrypt**.

4.3.1 Pruebas con imágenes reales

Se hicieron pruebas con dos fotografías reales, propiedad del autor de este trabajo, una de ellas en formato JPG y la otra en formato PNG:

Nombre de archivo	Resolución (WxH)	Resolución (MP)	Tamaño (KiB)	Formato	Descripción
DSC_8376p.jpg	1600x1070	1,7	496	JPG	Foto de la Luna, procesada para web.
DSC_8922n.png	5568x3712	20,6	21094	PNG	Douglas A4, directo de cámara sin procesar.

Tabla 4.3.1-1 - características de las imágenes probadas

Para la primer imagen, “**DSC_8376p.jpg**”, podemos ver los histogramas, por canal, que tienen la forma característica de una imagen con exposición correcta y muchas zonas oscuras:

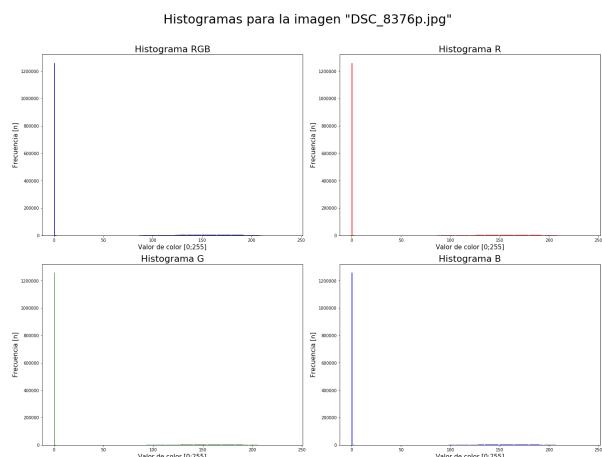


Imagen 4.3.1-1 - imagen “DSC_8376p.jpg” original y sus histogramas

Al correr el proceso de encriptación, se obtiene el archivo “**DSC_8376p_encrypted.jpg**”. Al observar los histogramas de la imagen encriptada, se descubre que no tienen la forma esperada, lo que sugiere que el resultado de la encriptación no es el correcto. Desencriptar esta imagen lo confirma, “**DSC_8376p_encrypted_decrypted.jpg**”:



Imagen 4.3.1-2 - imagen “DSC_8376p_encrypted_decrypted.jpg”

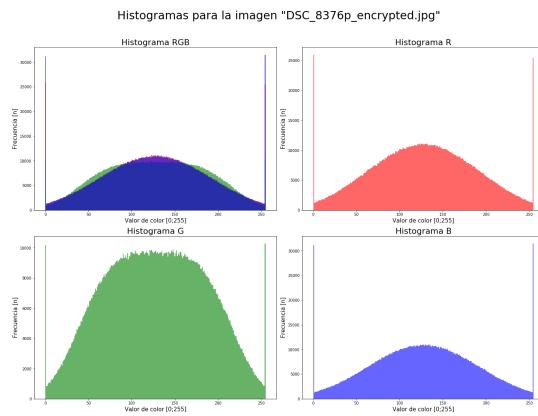


Imagen 4.3.1-3 - imagen “DSC_8376p_encrypted.jpg” y sus histogramas

Esto se debe a la aplicación del algoritmo de compresión de JPEG, que altera los píxeles en la imagen encriptada, al momento de escribirla. Es decir, el algoritmo de encriptación trabaja en forma correcta, pero el hecho de que la salida sea una imagen con compresión *lossy* arruina el resultado. Es por esto que se codificó un modo de operación especial para trabajar archivos con este tipo de compresión.

Al repetir la operación partiendo del archivo original y usando el flag *lossless*, se obtiene el archivo binario “**DSC_8376p_encrypted_lossless.jpg**”. Como el formato es conocido, es posible graficar el histograma, que es:

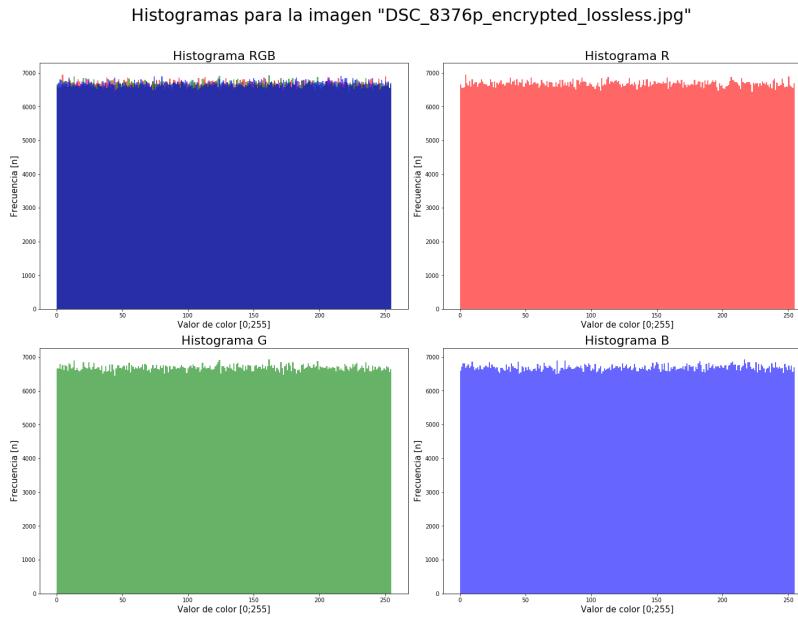


Imagen 4.3.1-4 - histogramas de la imagen “DSC_8376p_encrypted_lossless.jpg”

Como se observa tiene la forma esperada y con este archivo si es posible recuperar la imagen original. Se puede consultar a tal fin el archivo:

“**DSC_8376p_encrypted_lossless_decrypted.jpg**” en el directorio “images” del repositorio.

Se repite el análisis para la segunda imagen, “**DSC_8922n.png**”:

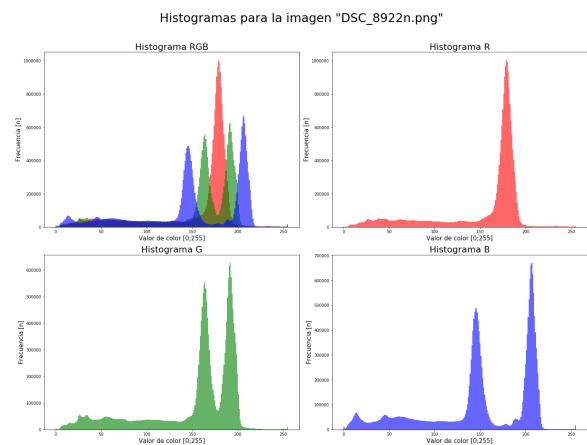


Imagen 4.3.1-5 - imagen “DSC_8922n.png” y sus histogramas

Al correr el proceso de encriptación sobre este archivo obtenemos la imagen encriptada “**DSC_8922n_encrypted.png**”:

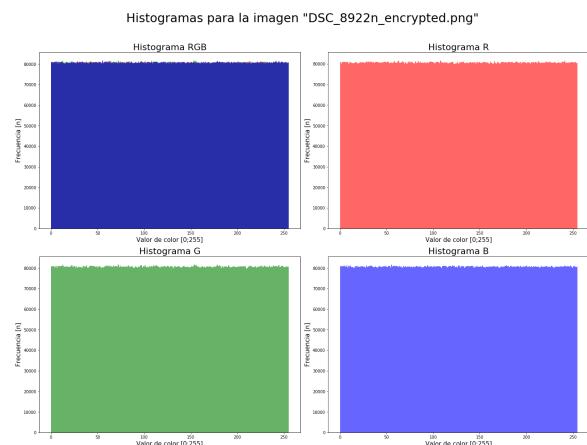
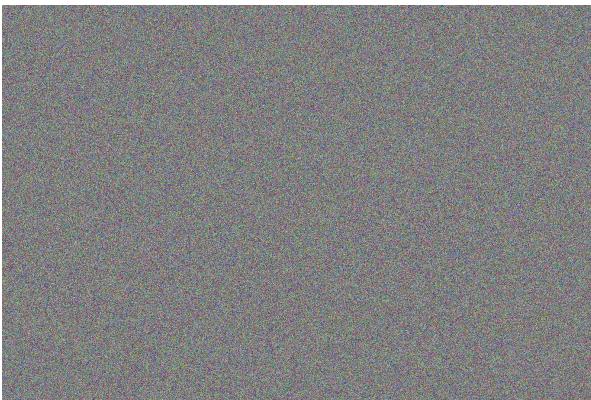


Imagen 4.3.1-6 - imagen “DSC_8922n_encrypted.png” y sus histogramas

Como se observa tiene la forma esperada y es posible recuperar la imagen original. Se puede consultar a tal fin el archivo:

“**DSC_8922n_encrypted_decrypted.png**” en el directorio “images” del repositorio.

A modo de cierre, se ha logrado correr la solución para encriptar y luego recuperar dos imágenes reales, una editada para web en formato JPEG y otra sin editar, con resolución de 20MP, en formato PNG.

Asimismo, se ha podido corroborar que para casos con compresión *lossy* debe utilizarse un formato especial de almacenamiento para los datos encriptados, y que en ambos casos, teniendo ese cuidado, tanto la solución como el algoritmo funcionaron como era esperado.

4.3.2 Pruebas de performance

En esta sección se muestran los resultados obtenidos de la medición de performance del algoritmo. Para ejecutar las pruebas fueron generadas 9 copias de la imagen Lena, en formato TIFF, a fin de contar con una amplia gama de resoluciones, y de representar de la mejor forma posible los casos de uso. El archivo original fue sometido a un *resampling bicúbico* para alterar su tamaño.

Para la implementación del trabajo, se intentó verificar el orden de complejidad en forma empírica, encriptando todas las imágenes, y a su vez compararlo con un algoritmo moderno de uso común: la implementación AES-256²⁷ de openSSL²⁸.

Los tiempos de ejecución fueron tomados con el profiler cProfile²⁹ para el caso de imgcrypt y el comando time³⁰ de UNIX para el caso de openSSL. En ambos casos, se realizaron 3 corridas con cada archivo y se tomó el promedio para informar los resultados.

A continuación se expone una tabla con la información de las imágenes utilizadas:

Nombre de archivo	Resolución (WxH)	Resolución (MP)	Tamaño (KiB)	Formato
lena_std_16_16.tif	16x16	0,0003	19	TIFF
lena_std_32_32.tif	32x32	0,0010	21	TIFF
lena_std_64_64.tif	64x64	0,0041	32	TIFF
lena_std_128_128.tif	128x128	0,0164	72	TIFF
lena_std_256_256.tif	256x256	0,0655	222	TIFF
lena_std_512_512.tif	512x512	0,2621	810	TIFF
lena_std_1024_1024.tif	1024x1024	1,0486	3277	TIFF
lena_std_2048_2048.tif	2048x2048	4,1943	12902	TIFF
lena_std_4096_4096.tif	4096x4096	16,7772	51610	TIFF

Tabla 4.3.2-1 - características de las imágenes probadas

En un primer gráfico se presenta el tiempo de ejecución para el promedio de las corridas realizadas con ambos algoritmos. Cabe destacar que en el eje de las abscisas se utiliza la resolución lineal de las imágenes, es decir la cantidad de megapíxeles (MP), que representa de la mejor forma la dimensión de la entrada del algoritmo objeto de estudio. Repasando, este opera

²⁷ Ver [16].

²⁸ Ver [17].

²⁹ Ver [19].

³⁰ Ver [18].

sobre un array unidimensional que contiene unidos en secuencia los componentes R, G y B de la imagen que se desea encriptar.

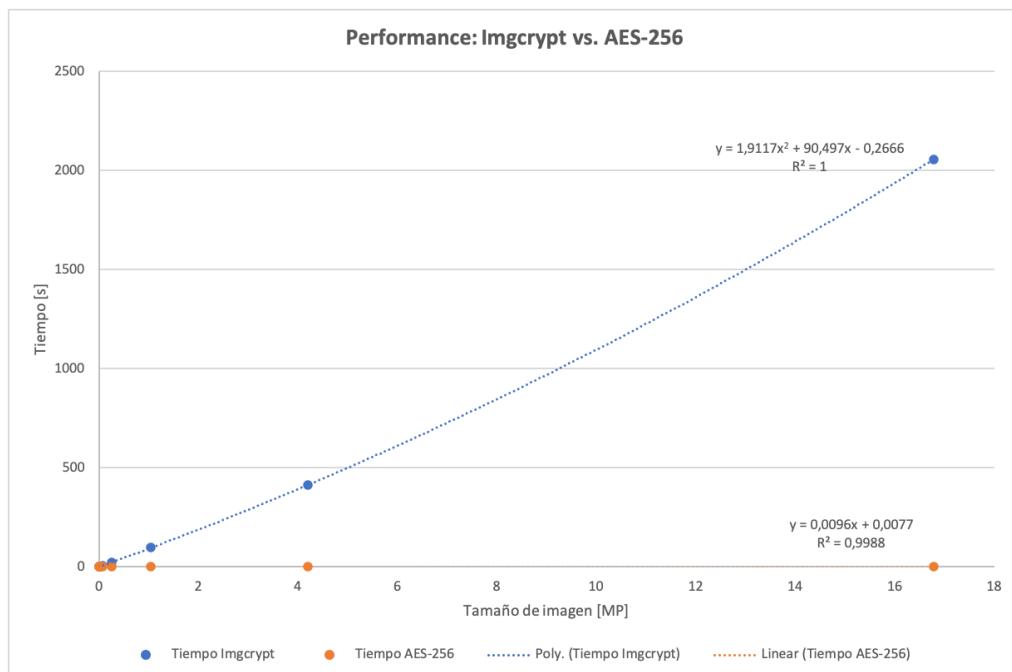


Imagen 4.3.2-1 - performance de Imgrypt vs. AES-256

Una primera conclusión es que se puede corroborar en forma empírica el resultado obtenido por los autores del *paper*, y es que el orden del algoritmo propuesto es $O(n^2)$. Por otro lado, también se puede observar que la curva de AES-256 es invisible. Se muestra a continuación un *zoom*, para poder hacer un análisis más claro:

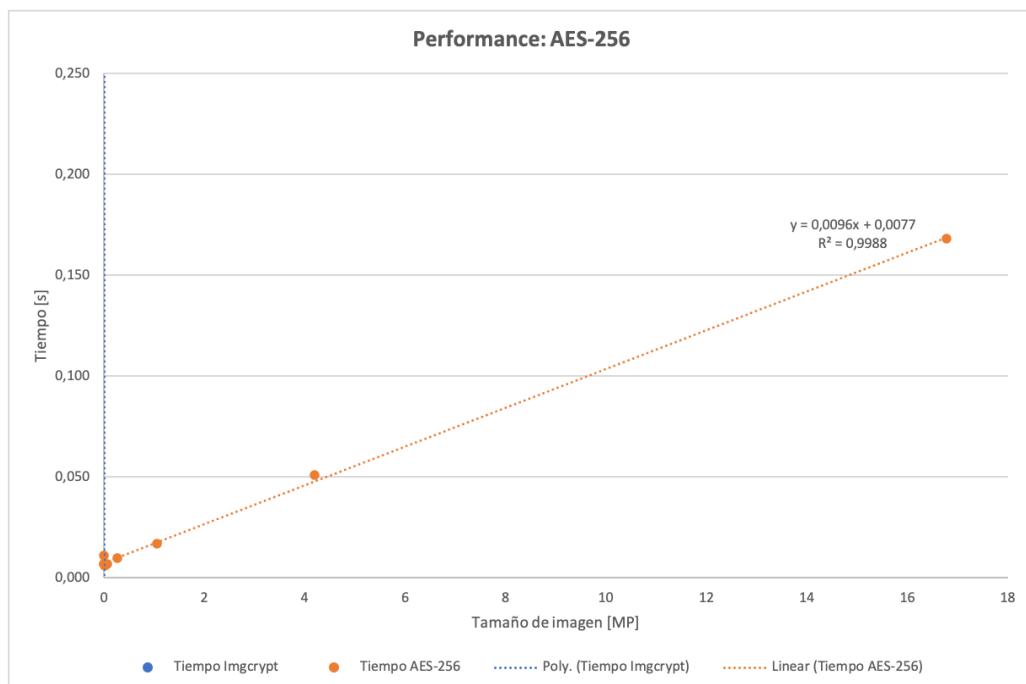


Imagen 4.3.2-2 - performance de Imgrypt vs. AES-256, ampliación

La ampliación permite ver que AES es un algoritmo *lineal*³¹, es decir que tiene orden O(n). Asimismo, para la la imagen más grande el tiempo de ejecución es inferior a 0,2s! Más allá del orden de cada algoritmo la diferencia en performance resulta abrumadora. Si bien es cierto que podría optimizarse el algoritmo implementado, como se discutió anteriormente, esto no está planteado en el *paper*.

Esta ventaja no puede ser despreciada, dado que una imagen de 16MP representa un valor relativamente bajo de resolución, habiendo imágenes que pueden superar fácilmente los 20MP. Adicionalmente, AES trata al archivo como una tira de bytes, no exhibiendo los problemas tratados en la sección [4.2](#). Como comentario final, AES acepta cualquier tipo de *string* como clave, mientras que en el algoritmo propuesto debe ser una tripla numérica. Por lo tanto, se puede afirmar que tanto desde el punto de vista de la practicidad como de performance AES es una opción superior. Sobre su seguridad³², las fuentes consultadas indican que no ha sido vulnerado, y la única opción para descifrar un criptograma es un ataque por fuerza bruta.

Tal vez la comparación sea un poco injusta, dado que AES es un algoritmo que fue diseñado y adoptado por el NIST (el National Institute of Standards and Technology de Estados Unidos), y al ser de uso corriente sus implementaciones se encuentran ampliamente optimizadas y representan el estado del arte. Citando a Katz-Lindell³³:

We conclude that, as of today, AES constitutes an excellent choice for almost any cryptographic implementation that relies on a pseudo-random permutation. It is free, standardized, efficient, and highly secure.

³¹ Ver [20].

³² [6], Pág 187.

³³ [6], Pág 187.

5. Conclusiones

A lo largo de este trabajo se ha realizado un análisis exhaustivo del *paper* estudiado: en primer instancia, se trabajó con el generador 3D-PLM, y luego con el criptosistema propuesto por los autores. Finalmente, se implementó un programa que permite aplicar todo lo visto en un escenario real.

Respecto al generador caótico pseudo aleatorio (CPRNG), se pudo implementar sin problemas en Python, un lenguaje estándar y de código abierto, y además realizarle una batería de pruebas que permitieron validar su performance.

Si bien se encontró una discrepancia en el diagrama de bifurcación respecto a los resultados publicados, se pudo constatar la presencia de caos en el sistema, lo que permitió definir el valor de los parámetros de control del sistema y llegar a la misma conclusión que los autores.

El análisis de sensibilidad también permitió probar la presencia de caos, y visualizar el *efecto mariposa*.

Los tests de aleatoriedad permitieron chequear la calidad de las secuencias generadas, que resultó también ser buena: en este caso se debió utilizar una implementación propia dada la imposibilidad de correr la *suite* de tests del NIST.

El análisis de periodicidad permitió calcular el período estimado del generador en máquinas de precisión finita, que resultó ser adecuadamente largo para los 64 bits que usa Python.

Finalmente, se calculó el valor de la entropía de la información para el generador, que dió un muy buen valor, y consistente con lo presentado en el *paper*.

En síntesis, se pudo comprobar que se trata de un buen generador pseudo aleatorio, que toma y combina las bondades de los dos sistemas en los que se basa, y por otro lado llegar a las mismas conclusiones que los autores.

En cuanto al criptosistema propuesto, también fue posible implementarlo en su totalidad, y replicar las pruebas presentadas en el *paper*, obteniendo resultados análogos a los publicados.

Como primera medida, se analizó el espacio de claves, y se verificó la sensibilidad de las mismas, obteniendo resultados satisfactorios y en línea con los obtenidos para el CPRNG.

El estudio de los histogramas de las imágenes encriptadas permitió ver el correcto funcionamiento del algoritmo, destruyendo la correlación entre los píxeles, hecho que también fue verificado analíticamente calculando un coeficiente de correlación.

Para medir la diferencia entre la imagen original y la encriptada, fue necesario calcular 3 métricas: el índice de similitud entre imágenes (SSIM), el error cuadrático medio (MSE) y el pico de la relación entre señal y ruido (PSNR), obteniendo excelentes resultados en todos ellos.

Finalmente, se calculó la resistencia del algoritmo a ciertos ataques, como ser *known*, *chosen plaintext* y ataques de oclusión, obteniendo también buenos resultados.

A modo de resumen, fue posible implementar y verificar la buena performance del criptosistema propuesto, en línea con lo publicado en el *paper*.

Por último, se pudo diseñar y construir una solución de tipo comercial, aplicando los conceptos presentados por los autores en su trabajo. Se trata del programa **imgcrypt**, el cual fue probado con imágenes reales, demostrando la viabilidad del algoritmo de encriptación.

Si bien el programa se desempeña correctamente, no se puede dejar de mencionar que en términos de performance resulta muy perjudicado respecto a una solución como AES-256, que representa el estado del arte en encriptación. En cuanto a su practicidad, también se detectaron algunos problemas, como su aplicación exclusiva a imágenes en formato RGB, el no contemplar formatos con compresión *lossy*, o el almacenamiento y encriptación de la metadata de las imágenes, todos temas que no aparecen en soluciones que tratan el objeto a encriptar como un *stream* de bytes, cual el caso de AES.

Como comentario final, se destaca que esta ha resultado una experiencia enriquecedora e integradora, tanto de varios temas vistos en el curso, como así también de temas vistos en otras asignaturas.

R. Referencias

En esta sección se listan las fuentes consultadas para la concreción del trabajo:

- [1]. Sahari, M.L., Boukemara, I. A pseudo-random numbers generator based on a novel 3D chaotic map with an application to color image encryption. *Nonlinear Dyn* 94, 723–744 (2018).
<https://doi.org/10.1007/s11071-018-4390-z>
- [2]. Leon-Garcia, Albet. Probability, Statistics, and Random Processes for Electrical Engineering. 3rd Edition, Pearson (2008).
- [3]. Sayama, Hiroki. Introduction to the Modeling and Analysis of Complex Systems. Open SUNY Textbooks (2015).
- [4]. Moller, Cleve. Numerical Computing with Matlab Mathworks (S/R año).
https://www.mathworks.com/moler/index_ncm.html
- [5]. Salomon, David. Data Compression. 4th Edition, Springer (2007).
- [6]. Katz, J., Lindell, I. Introduction to Modern Cryptography. Chapman & Hall/CRC (2008).
- [7]. The IEEE-754-2019 standard.
<https://webstore.ansi.org/Standards/IEEE/IEEE7542019>
- [8]. Gmpy2 library reference.
<https://gmpy2.readthedocs.io/en/latest/mpfr.html>
- [9]. Michael François, David Defour, Christophe Negre. A Fast Chaos-Based Pseudo-Random Bit Generator Using Binary64 Floating-Point Arithmetic. *Informatica, Slovene Society Informatika, Ljubljana* 38 (2) (2014).
<http://www.informatica.si/index.php/informatica/article/view/691>
- [10]. NIST SP 800-22: Download Documentation and Software.
<https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software>
- [11]. Faster randomness testing.
<https://randomness-tests.fi.muni.cz>
- [12]. Pillow. A Friendly fork of the Python Imaging Library.
<https://pypi.org/project/Pillow/>
- [13]. The Lenna Story.
<http://www.lenna.org>

[14]. Scikit-image. Image Processing for Python.
<https://scikit-image.org/docs/dev/api/skimage.html>

[15]. Peak signal-to-noise ratio. Wikipedia.
https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio

[16]. Advanced Encryption Standard (AES). *Federal Information Processing Standards*. (2001).
<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf>

[17]. OpenSSL. Cryptography and SSL/TLS Toolkit.
<https://www.openssl.org>

[18]. Time. Linux manual page.
<https://man7.org/linux/man-pages/man1/time.1.html>

[19]. The Python Profilers.
<https://docs.python.org/3/library/profile.html#module-cProfile>

[20]. Computational complexity class of decryption of AES
<https://crypto.stackexchange.com/questions/26151/computational-complexity-class-of-decryption-of-aes>

[21]. Knuth, Donald. The Art of Computer Programming. 3rd edition, Addison-Wesley (1998).