Artículos de investigación

Visualización de series de tiempo en Python

Rodríguez Torres, Erika Elizabeth; Tetlalmatzi Montiel, Margarita; Villarroel Flores, Rafael

Erika Elizabeth Rodríguez Torres
erikart@uaeh.edu.mx
Universidad Autónoma del Estado de Hidalgo, México
Margarita Tetlalmatzi Montiel
tmontiel@uaeh.edu.mx
Universidad Autónoma del Estado de Hidalgo, México
Rafael Villarroel Flores
rafaelv@uaeh.edu.mx
Universidad Autónoma del Estado de Hidalgo, México

Pädi Boletín Científico de Ciencias Básicas e Ingenierías del ICBI
Universidad Autónoma del Estado de Hidalgo, México
ISSN-e: 2007-6363
Periodicidad: Semestral
vol. 6, núm. 11, 52-57, 2018
sitioweb@uaeh.edu.mx

URL: http://portal.amelica.org/ameli/journal/595/5952914012/

Resumen: Se muestra el uso del lenguaje de programación *Python* para obtener representaciones gráficas de series de tiempo. Además, se usa *Python* para estudiar el concepto de la gráfica de visibilidad de una serie de tiempo. Los ejemplos mostrados pueden ser útiles en otros contextos donde pueda aplicarse la programación en problemas científicos.

Palabras clave: series de tiempo, Python, graficación.

Abstract: English Summary
Time Series Visualization in Python Abstract
It is shown how to use the Python programming language in order to get graphical representations of time series. Furthermore, Python is used to study the concept of the visibility graph of a time series. The examples shown here could be useful in contexts where programming may be applied in scientific problems.

Keywords: time series, Python, graphing.

1. Introducción

Una serie de tiempo se define como una sucesión de datos X = {xt} , donde la variable t toma valores uniformemente espaciados en el tiempo. El valor xt se interpreta como el valor que toma una cantidad a estudiar en el tiempo t. Como se observa en (Brockwell and Davis, 2016), existen diversas aplicaciones de las series de tiempo en la economía, por ejemplo: describiendo los volúmenes de venta de un producto a lo largo del tiempo, a la demografía: estudiando cambios en una cierta población, a la sociología, meteorología, etc. En este artículo y para simplificar la discusión, supondremos sin perder generalidad que la variable t toma valores en los enteros no negativos: 0, 1, 2, 3, ...

Python es un lenguaje de programación moderno, el cual puede obtenerse legalmente de internet de manera gratuita. Esta es quizás una de sus principales ventajas, ya que los científicos y sus estudiantes pueden estar seguros de realizar sus investigaciones sin preocuparse de situaciones como adquisición de licencias de software. Además, el lenguaje *Python* se presta de manera natural a las aplicaciones a la ciencia, puesto que varios equipos de trabajo han contribuido librerías de código abierto en diversos campos del conocimiento, que extienden las capacidades del lenguaje de programación. En este trabajo mostraremos el uso de sus

librerías matplotlib (la cual sirve para producir dibujos científicos) y networkx (la cual ayuda a manipular y dibujar redes complejas).

Como primer ejemplo, en el listado 1 se muestra el código, usando matplotlib, para producir la figura 1, la cual muestra una representación de la serie de tiempo [2,3,10,2,4,5].

```
import matplotlib . pyplot as plt y=[2 ,3 ,10 ,2 ,4 ,5]
plt. grid ( True ) plt. plot( y) plt. show ()
```

Listado 1: Listado sencillo

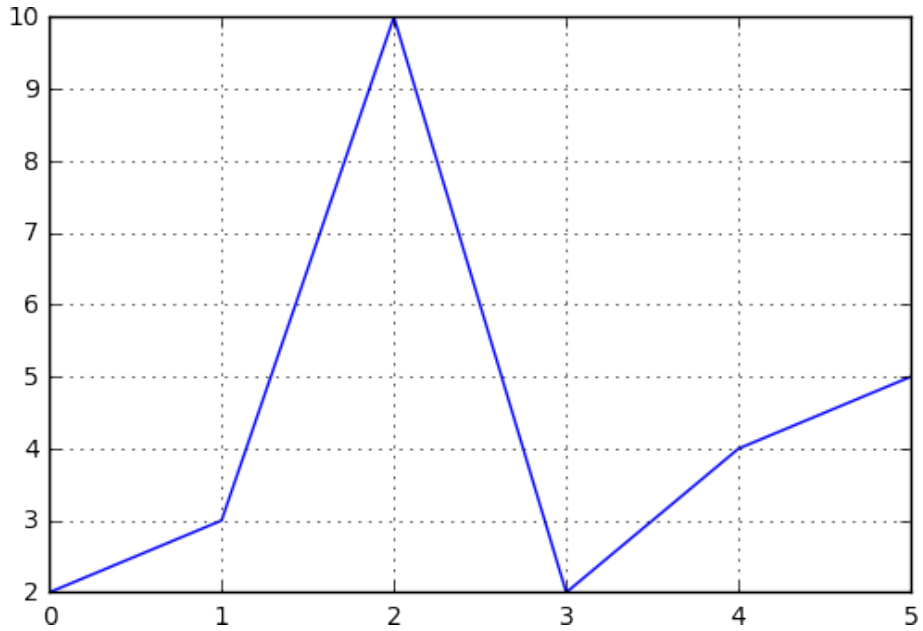

FIGURA 1:
Gráfica de una serie de tiempo

El propósito del presente artículo es mostrar que no es difícil utilizar un sistema como *Python* en el estudio de las disciplinas científicas donde se involucran las series de tiempo. En la última parte de este artículo, se utiliza el concepto de gráfica de visibilidad para proporcionar un ejemplo no trivial del uso de *Python*.

2. El lenguaje Python

El lenguaje Python puede descargarse, para diversos sistemas operativos, incluyendo Windows™ y MacOS™, de su sitio oficial: https://www.python.org. En Ubuntu y otras varias distribuciones de Linux, Python viene preinstalado, sin embargo, en los ejemplos mostrados en este artículo, además se necesitan los paquetes python-matplotlib y python-networkx. Hay excelentes textos introductorios, como ejemplo podemos citar (Langtangen, 2016).

Python tiene las estructuras de datos básicas, como las listas y los enteros. Por ejemplo, la segunda línea en el listado 1 asigna a la variable y la lista de enteros [2,3,10,2,4,5], que en este caso la estamos interpretando como una serie de tiempo. En *Python* existen muchísimas librerías que extienden la capacidad del lenguaje. Como ejemplo, mencionamos la librería matplotlib (https://matplotlib.org), la cual se invoca en la primer línea del listado 1. La instrucción import matplotlib.pyplot as plt sirve para poder usar en nuestro programa una sintaxis similar a la de MATLAB™.

Los dos siguientes renglones modifican un objeto gráfico. Con plt.grid(True) se superpone una cuadrícula en la gráfica, y con plt.plot(y) se añaden las líneas correspondientes a la serie de tiempo dada por la lista que hemos denotado con y. Finalmente, con plt.show() se muestra el dibujo así formado. En la figura 1 hemos

demostrado que, aunque *Python* es un lenguaje muy complejo, no es necesario usar toda su complejidad para empezar a usarlo. Vamos a dar un ejemplo ligeramente más complicado, el cual nos será útil en adelante. Para el pro pósito de definir la gráfica de visibilidad, será conveniente usar barras para indicar los valores de la serie de tiempo, como se muestra en la figura 2.

```
import matplotlib . pyplot as plt y=[2 ,3 ,10 ,2 ,4 ,5]
plt. grid ( True ) plt. bar( range ( len ( y)),
y, width =0.2 ,
align =' center ') plt. show ()
```

Listado 2: Código para gráfica con barras

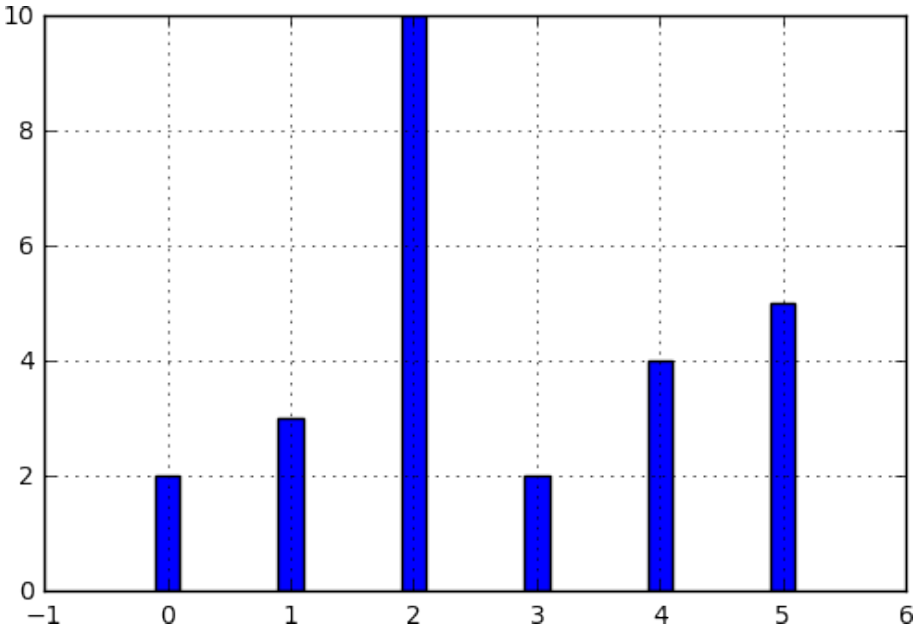

FIGURA 2:
Serie de tiempo con barras

3. Gráficas

3.1. El concepto matemático

Una **gráfica***G*, como concepto matemático, se compone de un conjunto de puntos, denotado con *V(G)*, a cuyos elementos se les llama **vértices**; y de un conjunto de parejas no ordenadas de vértices, denotado con *E(G)*, a cuyos elementos se les llama **aristas**. En las aplicaciones, se le suele llamar **nodos** a los vértices, y a las gráficas se les llega a llamar **grafos** y también **redes complejas**. Si los vértices v_1, v_2 son tales que $\{v_1, v_2\} \epsilon E(G)$, , decimos que v_1 y v_2 son **adyacentes**.

Si *v* es un vértice de la gráfica *G*, a la cantidad de aristas que contienen a *v* se le llama el **grado** de *v*. La cantidad $\Delta(G) = \{máx\ grado(v) \mid v \epsilon V(G)\}$ es el **grado máximo** de la gráfica. Se dice que una gráfica es **regular** si todos los vértices tienen el mismo grado. La sucesión $(d_0, d_1, ..., d_{\Delta(G)})$, donde $d_i$es la cantidad de vértices de *G* de grado *i*, es el **histograma de grados** de G. El histograma de grados ha sido empleado como un parámetro que permite detectar propiedades de la gráfica *G*. Existen muchas otros parámetros de las gráficas dignos de estudiarse, que se pueden consultar en un texto como (Harary (1969)).

Como ejemplo, podemos definir una gráfica *P*, cuyo con junto de vértices es $V(P) = \{a, b, c\}$, y cuyo conjunto de aristas es $E(P) = \{\{a, b\}, \{b, c\}\}$. La gráfica *P* no es regular, pues los vértices *a* y *c* tienen grado 1,

mientras que el vértice b tiene grado 2. De este modo, $\Delta(P) = 2$ y el histograma de grados es (0, 2, 1), pues hay cero vértices de grado 0, dos de grado 1 y uno de grado 2.

3.2. Gráficas en Python

En la figura 3 se muestra un dibujo de la gráfica P, que definimos en el párrafo anterior, hecho con Python y las librerías matplotlib y networkx (la página web de networkx se encuentra en https://networkx.github.io/). Como se aprecia en el código, mostrado en el listado 3, después de declarar que queremos usar networkx con la instrucción import networkx as nx, podemos usar las funciones nx.Graph para crear una gráfica, y funciones que modifican los diversos elementos de una gráfica (vértices, aristas y etiquetas de los vértices), para crear un dibujo, que como antes, se muestra con plt.show().

Notemos que, en este caso, hemos creado una gráfica por medio de la función nx.Graph, dando como argumento la lista de aristas de la gráfica a considerar. Existen otras maneras de crear gráficas, las cuales se pueden consultar en la documentación de networkx.

```
import matplotlib . pyplot as plt import networkx as nx P=nx. Graph ([('a','b') ,('b','c')]) pos=nx. spectral_layout ( P)
nx. draw_networkx_nodes (P, pos , node_size =800) nx. draw_networkx_edges (P, pos) nx. draw_networkx_labels (P, pos , font_size =25) cut = 0.07
plt. xlim (- cut ,1+ cut) plt. ylim (- cut ,1 - cut)
plt. axis(' off') plt. show ()
```

Listado 3: Listado que produce gráfica

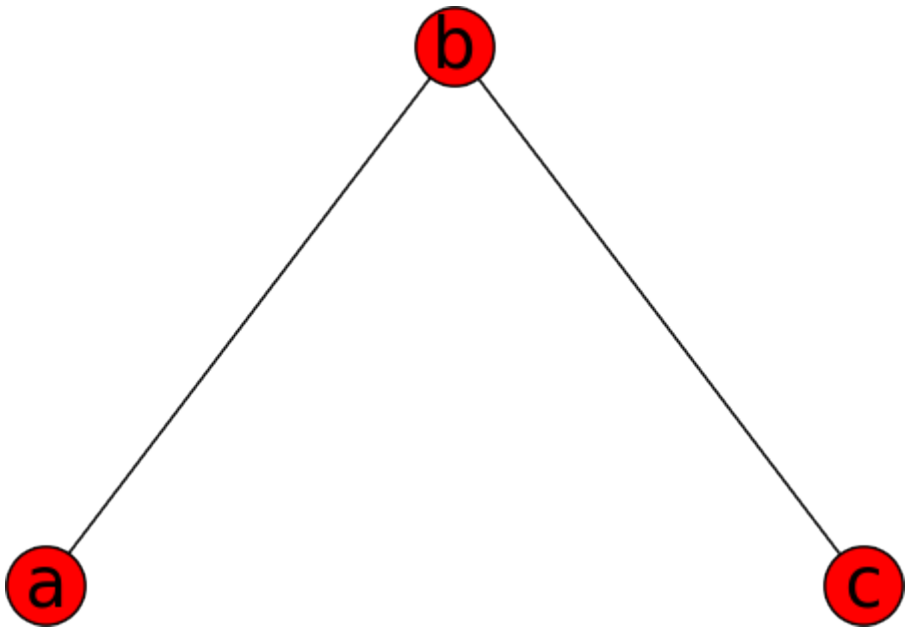

FIGURA 3:
Un dibujo de una gráfica

Una vez que la gráfica ha sido creada en Python, es posible usar la computadora para obtener propiedades de la gráfica. Por ejemplo:

```
P. nodes ()
produce:
['a', 'c', 'b']
```

mientras que:

P. degree ('b')
produce:
2

3.3. Nuevas funciones en Python

En networkx no existe ya definida una función para calcular el grado máximo de una gráfica. Sin embargo, no es difícil obtener tal parámetro si tenemos en cuenta que:

- P.nodes es la lista de vértices de la gráfica P, como usamos antes.
- Si l es una lista de vértices, P.degree(l) nos da una estructura de datos que incluye la información de los grados de cada vértice en l.
- Con P.degree(P.nodes()) obtenemos la información de todos los grados de todos los vértices de P. Pero para obtener una lista con únicamente los grados, usamos P.degree(P.nodes()).values().
- El grado máximo de la gráfica P se obtiene entonces con: max(P.degree(P.nodes()).values()).

Lo anterior se puede aplicar de manera práctica definiendo una nueva función, digamos grado_max, del siguiente modo:

def grado_max (G):
return max (G. degree (G. nodes ()). values ())
y entonces
grado_max (P)
produce:
2

4. Aplicación a la gráfica de visibilidad

4.1. Gráfica de visibilidad

En el artículo (Lacasa et al., 2008), los autores introducen la gráfica de visibilidad asociada a una serie de tiempo como una herramienta para analizar diversas propiedades de la serie, usando las técnicas y la terminología de la teoría de las gráficas. Por otro lado, también observan que la gráfica de visibilidad permanece invariante bajo ciertos cambios no esenciales de la serie de tiempo, como traslación o reescalamientos.

La gráfica de visibilidad tiene como vértices los datos de la serie de tiempo, de tal modo que por la suposición que hicimos al principio, en el caso de que haya n datos, los vértices pueden tomarse como $V = \{0, 1, 2, \ldots, n-1\}$. Se declaran adyacentes entre sí aquellos datos tales que, en el dibujo de la serie de tiempo, las partes superiores de sus barras correspondientes sean «visibles» entre sí considerando a las barras como «paredes». En las figuras 4 y 5 se muestran ejemplos.

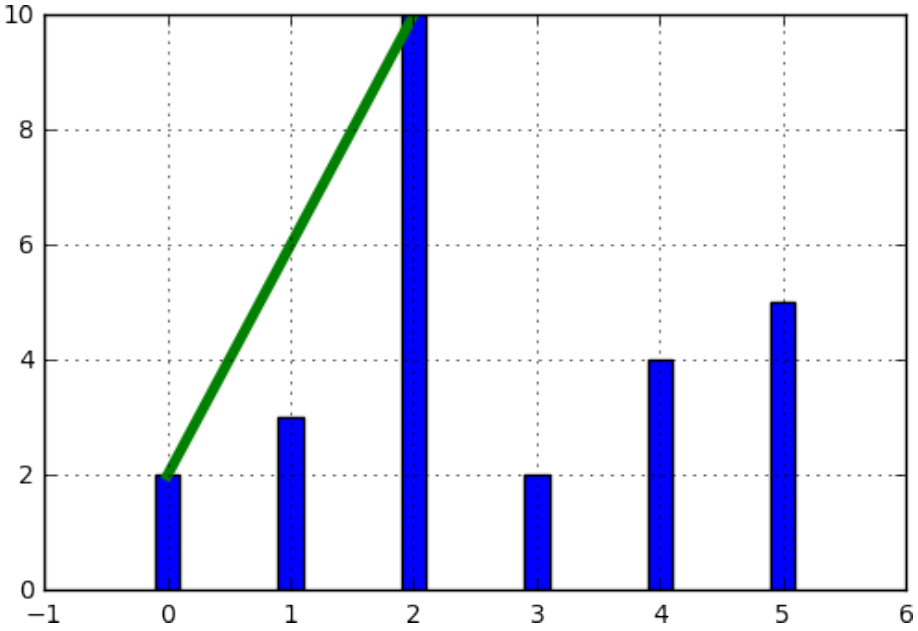

FIGURA 4:
Datos adyacentes

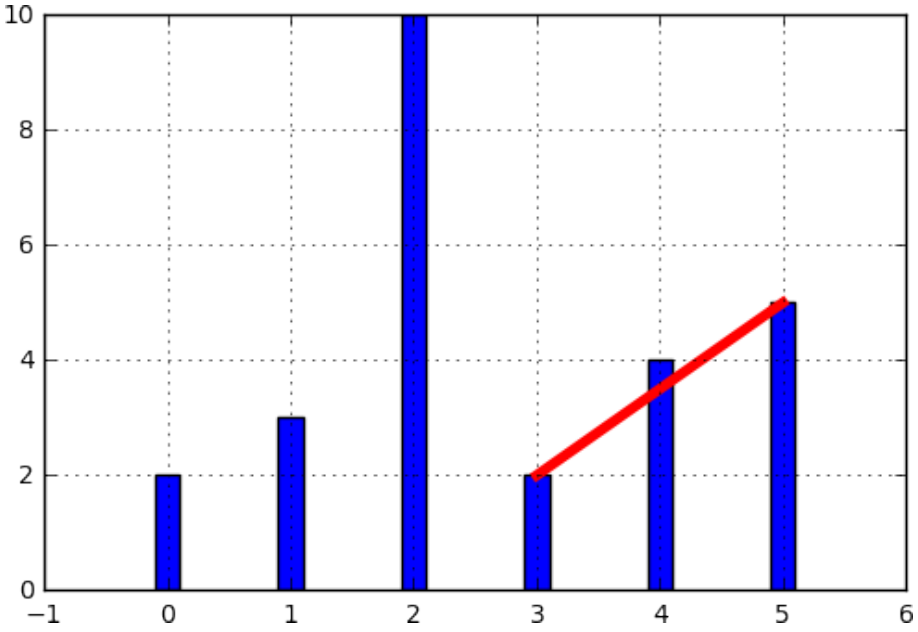

FIGURA 5:
Datos no adyacentes

Es inmediato entonces determinar la lista completa de las aristas de la gráfica de visibilidad de una serie de tiempo, como, por ejemplo: [2,3,10,2,4,5], y usando networkx, realizar un dibujo de la gráfica.

El dibujo se muestra en la figura 6.

En este caso, puesto que la serie de tiempo considerada tiene pocos datos, es factible enumerar explícitamente las aristas de la gráfica de visibilidad simplemente usando el dibujo de la serie de tiempo. En las siguientes secciones mostraremos el modo en que *Python* nos puede ayudar para estudiar series de tiempo más complicadas.

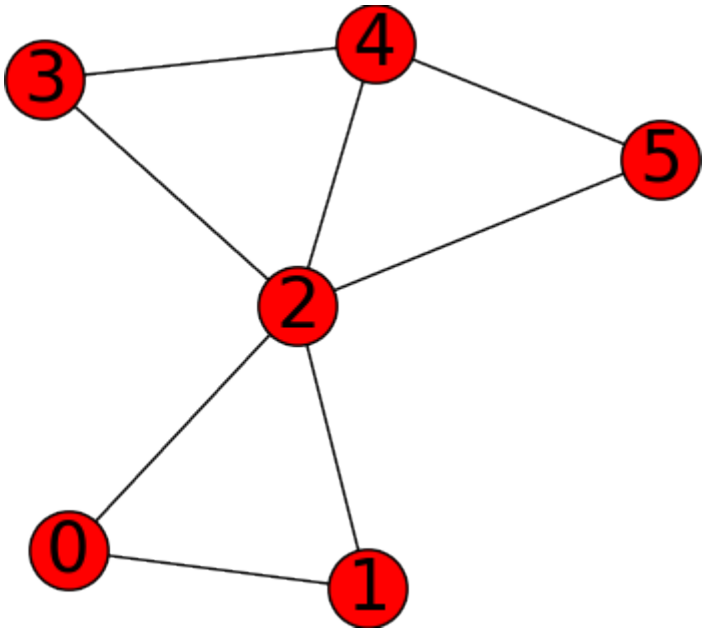

FIGURA 6:
Gráfica de visibilidad sencilla

4.2. Definición formal de la gráfica de visibilidad

Determinar si dos datos son adyacentes en la gráfica de visibilidad de manera formal es un ejercicio de geometría analítica. Por ejemplo, en (Lacasa et al., 2008) se describe que los datos $(t_a, y_a), (t_b, y_b)$ se declaran adyacentes en la gráfica de visibilidad siempre y cuando se tenga que para todos los t_c con $t_a < t_c < t_b$ se cumple que:

$$y_c < y_b + (y_a - y_b)\frac{t_b - t_c}{t_b - t_a}. \tag{1}$$

En *Python*, podemos definir una función que determine si, dada una serie de tiempo y dos datos, tales datos son adyacentes en la gráfica de visibilidad de la serie de tiempo. En el listado 4 se define tal función. La función is_visible regresa True si los datos a, b son adyacentes y False si no.

```
def is_visible (y,a, b): isit = True
c = a+1
while isit and c < b:
isit = y[ c] <y[ b ]+( y[ a]- y[ b]) *(( b- c)/ float(b- a)) c = c+1
return isit
```

Listado 4: Función de adyacencia

Por otro lado, en el listado 5 se muestra el código para definir la gráfica de visibilidad de una serie de tiempo, usando la función is_visible del listado 4. En este caso, se usa una variable llamada eds para colectar las aristas en una lista. Para cada dato a en la serie de tiempo y cada b que sea mayor que a se determina si b es visible desde a, y solo en el caso de que así sea se añade la arista (a,b) a la lista eds. La función visibility_graph regresa finalmente la gráfica de visibilidad de la serie de tiempo ts.

```
def visibility_graph ( ts): eds = []
for a in range ( len ( ts)):
for b in range ( a+1 , len ( ts)): if is_visible ( ts ,a, b):
eds. append (( a, b)) return nx. Graph ( eds)
```

Listado 5: Gráfica de visibilidad

Recientemente se han definido, (y estudiado y aplicado) variantes de la gráfica de visibilidad, como la gráfica de visibilidad horizontal (Luque et al. (2009)), la gráfica de visibilidad con pesos (Supriya et al. (2016)), y la gráfica de visibilidad paramétrica (Bezsudnov and Snarskii (2014)). El código que hemos presentado en el listado 4 que determina si dos datos son visibles para la gráfica de visibilidad «natural» podría adaptarse sin muchos problemas para considerar las otras definiciones de visibilidad.

En la siguiente sección aplicaremos este código a una serie de tiempo mucho más complicada.

5. El mapeo logístico

Una manera sencilla de obtener una serie de tiempo en un conjunto de números reales X, es por medio de las iteraciones de una función $f : X \to X$. Para construir una serie de tiempo de tal modo, se toma una condición inicial $x_0 \in X$, y los datos subsiguientes se definen recursivamente para t > 0 como $x_t = f(x_{t-1})$.

Uno de los casos más estudiados, el cual incluimos aquí únicamente como ejemplo, es el **mapeo logístico**, en el cual se toma X = [0, 1], es decir, el intervalo de números reales entre 0 y 1, y la función $f(x) = \mu x(1 - x)$, donde μ satisface $0 \leq \mu \leq 4$.

Remitimos al lector interesado en las propiedades del mapeo logístico a (Devaney (1989)), en donde se demuestra que para ciertos valores de μ las iteraciones muestran el fenómeno de **caos**.

En el listado 6 se muestra cómo obtener las iteraciones del mapeo logístico para producir la figura 7, donde se dibuja la serie de tiempo obtenida con condición inicial $x_0 = 0{,}3$, valor del parámetro $\mu = 3{,}8$, y con 50 iteraciones.

```
import matplotlib . pyplot as plt
def f( x):
return 3.8* x*(1 - x)
vals =[0 .3 ]
iteraciones =50
for i in range ( iteraciones):
new = vals [-1]
vals. append ( f( new ))
plt. figure ( figsize =(10 ,5))
plt. axis([-1 , iteraciones +1 ,0 ,1])
plt. grid ( True )
plt. bar( range ( len ( vals)),
vals ,
width =0.2 ,
align =' center ')
plt. show ()
```

Listado 6: Listado para obtener serie de tiempo

Observemos que en el listado 6, además de producir el dibujo, se crea una lista vals, que contiene los datos de la serie de tiempo. A continuación se puede usar tal lista para obtener la figura 8, que muestra la gráfica de visibilidad de la serie de tiempo que se originó con las iteraciones del mapeo logístico. En este caso, notamos que resulta difícil derivar propiedades de la gráfica únicamente del dibujo, debido a la complejidad del mismo, ya que no siempre es claro determinar si una arista une

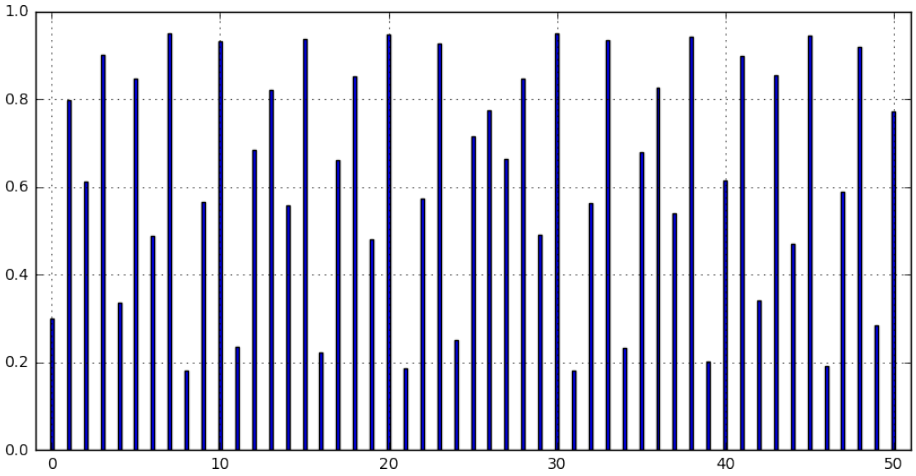

FIGURA 7:
Serie de tiempo obtenida del mapeo logístico

dos determinados vértices. De tal modo que se necesitan otras herramientas, como el histograma de grados, para estudiar una gráfica compleja.

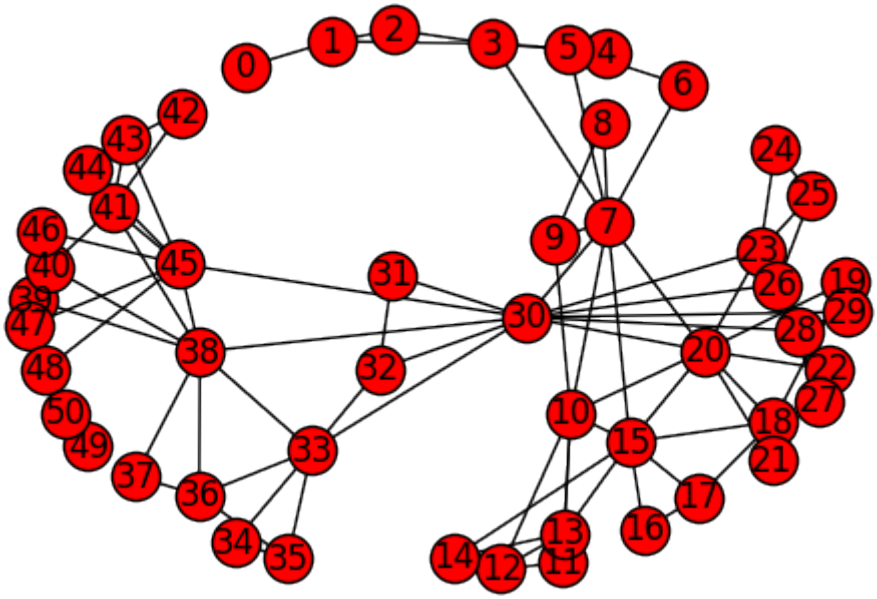

FIGURA 8:
Gráfica de visibilidad

En el listado 7, se muestra el código que se puede usar para producir un histograma y en la figura 9 se muestra el histograma de los grados de la gráfica anterior.

En la literatura, la distribución de los grados de los vértices es la principal herramienta para estudiar la gráfica de visibilidad, en especial en casos como este, en que la gráfica tiene demasiados vértices. Por ejemplo, los autores (Luque et al. (2009)) distinguen el caso en que los datos de la serie de tiempo se obtienen de manera aleatoria, del caso caótico como el que consideramos en la presente sección.

```
import numpy as np
degs = list( G. degree (). values ()) dmax = max ( degs)
bins = np. arange ( dmax ) - 0.5
plt. xlim ( xmin = -1)
plt. hist( degs , bins)
```

```
plt. xticks( range ( dmax ))
plt. xlabel(
u" Grados de los vé rtices")
plt. ylabel(
u" Cantidad de vé rtices")
plt. grid ( True )
plt. show ()
```

Listado 7: Código para producir histograma

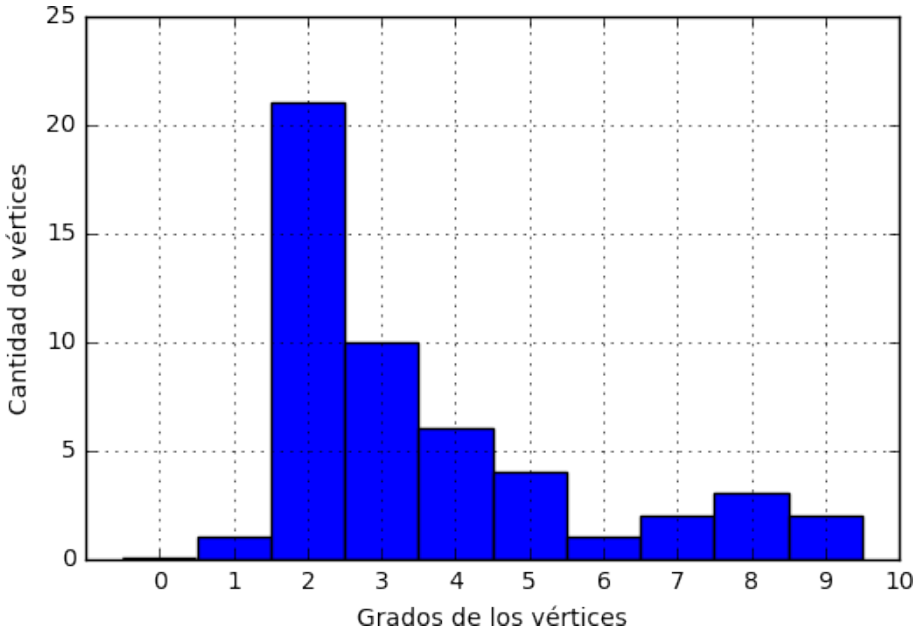

FIGURA 9:
Histograma de grados

6. Conclusiones

En el presente artículo se han mostrado ejemplos del uso de Python en el estudio de series de tiempo. Los autores desean que sean cada vez más los científicos que se decidan a usar herramientas computacionales, como las mostradas aquí, en sus investigaciones y en su trabajo didáctico.

Existen dos posibles direcciones de investigación respecto a los conceptos aquí mostrados. Uno es teórico, y se refiere a estudiar las propiedades de las diferentes gráficas de visibilidad que se han definido. Por ejemplo, en (Gutin et al. (2011)) se caracterizan las gráficas de visibilidad horizontales por medio de propiedades combinatorias.

Otro problema teórico surge al considerar que las series de tiempo que se pueden estudiar en la computadora están limitadas a que tengan una cantidad finita de términos. Sin embargo, el concepto de la gráfica de visibilidad se extiende de mane ra natural al caso infinito y conduce al estudio de propiedades de gráficas infinitas. Por ejemplo, un problema interesante es el considerar la manera en que la convergencia de una serie de tiempo se refleja en la gráfica. Tal problema es no trivial y será considerado en un trabajo futuro

Otra dirección que los autores del presente artículo considerarán en el futuro es aplicada, y se refiere a estudiar el efecto de perturbaciones en series de tiempo que provienen de fenómenos biológicos, por medio de su gráfica de visibilidad.

Referencias

Bezsudnov, I. V., Snarskii, A. A., 2014. From the time series to the complex networks: the parametric natural visibility graph. Phys. A 414, 53–60.

Brockwell, P. J., Davis, R. A., 2016. Introduction to time series and forecasting, 3rd Edition. Springer Texts in Statistics. Springer, [Cham]. URL http://dx.doi.org/10.1007/978-3-319-29854-2

Devaney, R. L., 1989. An introduction to chaotic dynamical systems, 2nd Edi- tion. Addison-Wesley Studies in Nonlinearity. Addison-Wesley Publishing Company, Advanced Book Program, Redwood City, CA.

Gutin, G., Mansour, T., Severini, S., 2011. A characterization of horizontal vi- sibility graphs and combinatorics on words. Phys. A 390 (12), 2421–2428. URL http://dx.doi.org/10.1016/j.physa.2011.02.031

Harary, F., 1969. Graph theory. Addison-Wesley Publishing Co., Reading, Mass.-Menlo Park, Calif.-London.

Lacasa, L., Luque, B., Ballesteros, F., Luque, J., Nuño, J. C., 2008. From time series to complex networks: the visibility graph. Proc. Natl. Acad. Sci. USA 105 (13), 4972–4975. URL http://dx.doi.org/10.1073/pnas.0709247105

Langtangen, H. P., 2016. A primer on scientific programming with Python, 5th Edition. Vol. 6 of Texts in Computational Science and Engineering. Sprin- ger, Heidelberg, for the fourth edition see [MR3237666].

Luque, B., Lacasa, L., Ballesteros, F., Luque, J., 2009. Horizontal visibility graphs: Exact results from random time series. Physical Review E 80.

Supriya, S., Siuly, S., Wang, H., Cao, J., Zhang, Y., 2016. Weighted visibility graph with complex network features in the detection of epilepsy. IEEE Ac- cess 4, 6554 – 6566.