Revisión de conceptos

1. Python y la Modularidad

En Python es común (y se recomienda) utilizar el diseño modular para la programación de un sistema informático. La programación modular consiste en tomar un sistema complejo y dividirlo en partes o componentes más pequeños y fáciles de manejar, es decir, **módulos**. Estos componentes pueden ser creados y probados independientemente. Además, es posible usarlos en otros sistemas.

Ahora, un módulo de Python es cualquier archivo que tiene la extensión .py y consta de un código de Python adecuado. No se requiere una sintaxis especial para convertir dicho archivo en un módulo. Igualmente, un módulo puede contener objetos arbitrarios, por ejemplo: archivos, funciones, clases o atributos. Se puede acceder a todos esos objetos después de una importación.

A medida que crece el número de módulos en un sistema, aumenta la dificultad para el manejo eficiente de estos, por lo tanto, existen los **paquetes**. Estos permiten una estructuración jerárquica del espacio de nombres de los módulos por medio de la notación de puntos. Un paquete es básicamente un directorio (carpeta) con archivos Python (módulos) y un archivo con el nombre __init__.py. Esto significa que cada directorio dentro de la ruta de Python, que contiene un archivo llamado __init__.py, será tratado como un paquete por Python. Asimismo, es posible crear un paquete principal que contiene subpaquetes.

Una de las principales ventajas de utilizar Python es el gran número de librerías o paquetes especializados en todo tipo de tareas, que facilitan el desarrollo de un sistema. Estos son creados por terceros y se encuentran disponibles en la Web. En el ámbito científico y en el campo de la ingeniería, algunos de los paquetes de código libre más utilizados son: SciPy, Numpy, Matplotlib y OpenCV. Estos facilitan el trabajo en dichas áreas al permitir un mejor manejo, procesamiento y presentación de la información.

2. Numpy

Junto con Python, NumPy es el paquete fundamental para la computación científica, garantizando la funcionalidad y velocidad de operación. Proporciona estructuras de datos vectoriales, matriciales y de mayor dimensión, las cuales acoge con el término ***array***.
Un array es similar a una lista, pero con la característica de que todos sus elementos deben ser del mismo tipo. Tiene una longitud fija, en el sentido de que no es posible agregar ni eliminar elementos a un *array* ya definido. Lo más importante de un *array* es el hecho de que cuenta con métodos y funcionalidades adicionales al paquete Numpy.

A continuación, se muestra el código básico para obtener diferentes *arrays* y comprobar algunas de sus características. El proceso es por medio de métodos propios de Numpy y una función estándar de Python:

```
import numpy as np # Se importa el paquete Numpy y se enmascara como np
# Creamos dos arrays utilizando listas
Array1 = np.array([[ 0,  1,  2,  3,  4],
           [ 5,  6,  7,  8,  9],
           [10, 11, 12, 13, 14]])
Array2 = np.array([6, 7.0, 8, 2])
# Imprimir el número de dimensiones
Ndim1 = Array1.ndim
Ndim2 = Array2.ndim
print("Numero de dimensiones Array1: "+str(Ndim1))
print("Numero de dimensiones Array2: "+str(Ndim2))
# Imprimir el número de datos en cada una de las dimensiones
Shape1 = Array1.shape
Shape2 = Array2.shape
print("Numero de datos en las dimensiones Array1: "+str(Shape1))
print("Numero de datos en las dimensiones Array2: "+str(Shape2))
# Imprimir el tamaño con la función estándar len
Len1 = len(Array1)
Len2 = len(Array2)
print("Tamaño Array1: "+str(Len1))
print("Tamaño Array2: "+str(Len2))
# Imprimir el tipo de los datos que contiene
Type1 = Array1.dtype.name
Type2 = Array2.dtype.name
print("Tipo de datos Array1: "+str(Type1))
print("Tipo de datos Array2: "+str(Type2))

>>
Numero de dimensiones Array1: 2
Numero de dimensiones Array2: 1
Numero de datos en las dimensiones Array1: (3, 5)
Numero de datos en las dimensiones Array2: (4,)
Tamaño Array1: 3
Tamaño Array2: 4
Tipo de datos Array1: int32
Tipo de datos Array2: float64
```

Nota: para conocer más métodos sobre las propiedades de un objeto de tipo numpy.ndarray, consulte la página oficial de la documentación de Numpy, referenciada en la sección Material de apoyo.

Como se observa en el resultado, el tamaño entregado por la función estándar len() solo muestra el número de datos de la primera dimensión (o eje); y a pesar de que el Array2 se creó con la mayoría de datos de tipo entero, por incluir el 7.0, Numpy asume que todo los datos son flotantes. Ahora, se muestran algunas formas para crear *arrays*:

```
import numpy as np # Se importa el paquete Numpy y se enmascara como np
# Crear un array utilizando una lista
Lista = [[3, 4, 6], [1, 0, 8]]
Array1 = np.array(Lista)
print("Array1: ")
print(Array1)
# Crear un array especificando el tipo de datos
```

```
Array2 = np.array(Lista, dtype=complex)
print("\nArray2: ")
print(Array2)
# Crear un array de ceros con dimensiones (2,2,2)
Array3 = np.zeros((2,2,2))
print("\nArray3: ")
print(Array3)
# Crear un array de valores aleatorios entre 0 y 1 con dimensiones (2,3)
Array4 = np.random.random((2,3))
print("\nArray4: ")
print(Array4)
# Crear un array (1 dimensión) con números de 10 a 30 con paso de 5
Array5 = np.arange(10, 31, 5)
print("\nArray5: ")
print(Array5)
# Crear un array (1 dimensión) con 5 números igualmente espaciados entre 0 a 2*pi
Array6 = np.linspace(0, 2*np.pi, 5)
print("\nArray6: ")
print(Array6)

>>
Array1:
[[3 4 6]
 [1 0 8]]

Array2:
[[3.+0.j 4.+0.j 6.+0.j]
 [1.+0.j 0.+0.j 8.+0.j]]

Array3:
[[[0. 0.]
  [0. 0.]]
 [[0. 0.]
  [0. 0.]]]

Array4:
[[0.29858921 0.87326326 0.40161994]
 [0.85498118 0.4526238  0.53372838]]

Array5:
[10 15 20 25 30]

Array6:
[0.         1.57079633 3.14159265 4.71238898 6.28318531]
```

Nota: para conocer más métodos dedicados a la creación de *array*, consulte la página oficial de la documentación de Numpy, referenciada en la sección Material de apoyo.

Numpy permite operaciones con los *arrays*. En el siguiente ejemplo se emplean algunas de sus operaciones básicas. (Asegúrese de que los *arrays* cumplan con los requisitos específicos entre sus dimensiones para ejecutar la operación).

```
import numpy as np # Se importa el paquete Numpy y se enmascara como np
# Crear dos arrays de 1 dimensión
Array1 = np.array([20,30,40,50])
Array2 = np.arange(4)
# Crear un array de valores aleatorios entre 2 y 7 con dimensiones (2,4)
Array3 = 5*np.random.random((2,4)) + 2
print("Array1: ")
print(Array1)
print("\nArray2: ")
print(Array2)
print("\nArray3: ")
print(Array3)
print("\nArray1 + Array2: ")
print(Array1 + Array2)
print("\nArray2 / Array1: ")
print(Array2 / Array1)
print("\nArray2 ** 3: ")
print(np.power(Array2,3))
print("\nRaíz cuadrada de Array1: ")
print(np.sqrt(Array1))
print("\n2*seno(Array1): ")
print(2*np.sin(Array1))
print("\nProducto cruz entre Array3 x Array2: ")
print(np.dot(Array3, Array2))
print("\ne ** Array3: ")
print(np.exp(Array3))
```

```
>>
Array1:
[20 30 40 50]

Array2:
[0 1 2 3]

Array3:
[[4.34401519 2.83843463 2.07218778 2.51952779]
 [4.64730195 5.02283823 6.02815695 5.65159542]]

Array1 + Array2:
[20 31 42 53]

Array2 / Array1:
[0.         0.03333333 0.05       0.06      ]

Array2 ** 3:
[ 0  1  8 27]

Raíz cuadrada de Array1:
[4.47213595 5.47722558 6.32455532 7.07106781]

2*seno(Array1):
[ 1.8258905  -1.97606325  1.49022632 -0.52474971]

Producto cruz entre Array3 x Array2:
[14.54139355 34.03393838]
```

```
e ** Array3:
[[ 77.01615402  17.08899394   7.94217983  12.42272914]
 [104.30318987 151.84165455 414.94955126 284.74539136]]
```

Nota: para conocer más métodos dedicados a operar o procesar *arrays*, consulte la página oficial de la documentación de Numpy, referenciada en la sección Material de apoyo.

Al igual que con las listas, es posible cortar, extraer e ingresar elementos a un *array* utilizando los índices o posiciones. Además, a través de otros métodos, permite modificar las dimensiones del *array* mientras se conserve el número total de elementos, como se muestra en el siguiente ejemplo:

```
import numpy as np # Se importa el paquete Numpy y se enmascara como np
# Crear array de 2 dimension
Array1 = np.array([[1, 2, 3, 4, 5, 6],
                   [7, 8, 9, 10, 11, 12],
                   [13, 14, 15, 16, 17, 18],
                   [19, 20, 21, 22, 23, 24]])
print("Array1: ")
print(Array1)
print("\nfila 2: ")
print(Array1[1,:])
print("\ncolumna 3 y 6: ")
print(Array1[:,(2,5)])
print("\nvalores entre las filas 2-3 y columna 3-6: ")
print(Array1[1:3,2:6])
print("\nCambiar las dimensiones del Array a (8,3): ")
Array1.resize((8,3))
print(Array1)
print("\nObtener un Array con los mismos datos y dimensión de (2,12): ")
print(Array1.reshape(2,12))

>>
Array1:
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]

fila 2:
[ 7  8  9 10 11 12]

columna 3 y 6:
[[ 3  6]
 [ 9 12]
 [15 18]
 [21 24]]

valores entre las filas 2-3 y columna 3-6:
[[ 9 10 11 12]
 [15 16 17 18]]

Cambiar las dimensiones del Array a (8,3):
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]
 [13 14 15]
 [16 17 18]
 [19 20 21]
 [22 23 24]]
```

Obtener un Array con los mismos datos y dimensión de (2,12):

```
[[ 1  2  3  4  5  6  7  8  9 10 11 12]
 [13 14 15 16 17 18 19 20 21 22 23 24]]
```

---

3. Matplotlib

Matplotlib es un excelente paquete de figuras en 2D y 3D. Con solo unas pocas líneas de código se pueden generar todo tipo de gráficos, histogramas, espectros de potencia, gráficos de barras, gráficos de errores, diagramas de dispersión, entre otros. También permite la salida de alta calidad en muchos formatos, incluidos PNG, PDF, SVG, EPS y PGF. Todos los aspectos de la figura se pueden controlar mediante programación. Esto es importante para la reproducibilidad y es conveniente cuando se necesita regenerar la figura con datos actualizados o cambiar su apariencia.

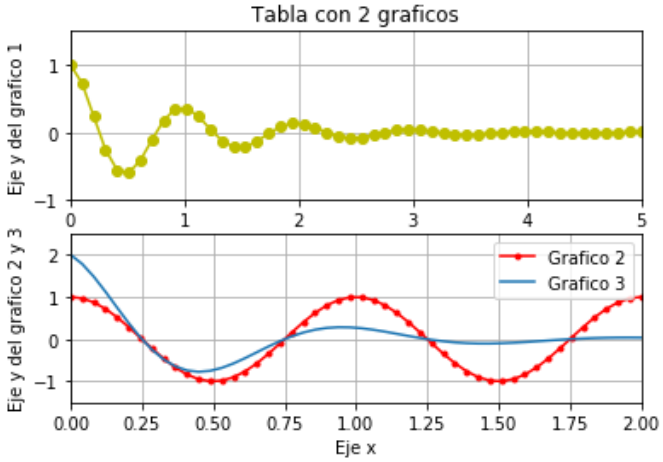
A continuación, se exponen 4 ejemplos que sirven para mostrar el funcionamiento básico de la librería. Si se desea conocer más módulos y funciones, consulte la página oficial de la documentación de Matplolib, referenciada en la sección Material de apoyo.

Se desea graficar 3 datos o señales, para esto se realizarán dos figuras. La primera consta del gráfico de la señal 1. La segunda figura está compuesta por los gráficos de la señal 2 y 3 y especifica cada una por medio de etiquetas con el fin de comparar ambas señales.

```
import numpy as np # Se importa el paquete Numpy y se enmascara como np
import matplotlib.pyplot as plt # Se importa el módulo pyplot para generar los plots
# Crear los arrays del eje x para cada grafico
x_g1 = np.linspace(0.0, 5.0)
x_g2 = np.linspace(0.0, 2.0)
# Obtener los array del eje y para cada grafico
y_g1 = np.cos(2*np.pi*x_g1)*np.exp(-x_g1)
y_g2 = np.cos(2*np.pi*x_g2)
y_g3 = 2*np.cos(2*np.pi*x_g2)*np.exp(-2*x_g2)
# Indicar el espacio del grafico en una tabla de 2 filas y 1 columna
plt.subplot(2, 1, 1) # Ubicar en la primera casilla de la tabla
# Indicar valores del eje x y y del grafico 1, color y forma de la líneas y puntos
plt.plot(x_g1, y_g1, 'yo-')
# Indicar etiquetas en el título del gráfico y en el eje y
plt.title('Tabla con 2 graficos')
plt.ylabel('Eje y del grafico 1')
# Indicar los límites de la gráfica tanto para el eje x como para y
plt.xlim(0, 5)
plt.ylim(-1, 1.5)
plt.grid() # Poner cuadricula al grafico 1
```

```
plt.subplot(2, 1, 2) # Ubicar en la segunda casilla de la tabla
# Indicar valores del eje x y y del grafico 2 y 3, color y forma de la líneas y puntos
plt.plot(x_g2, y_g2, 'r.-')
plt.plot(x_g2, y_g3) # Si no se especifica características por defecto es continua y azul
# Indicar etiquetas en el título del gráfico y en el eje y
plt.xlabel('Eje x')
# Se especifica una etiqueta para cada grafica en la figura
plt.ylabel('Eje y del grafico 2 y 3')
plt.legend(('Grafico 2', 'Grafico 3'))
# Indicar los límites de la gráfica tanto para el eje x como para el y
plt.xlim(0, 2)
plt.ylim(-1.5, 2.5)
plt.grid() # Poner cuadricula al grafico 2
# Indicar que se muestre los gráficos
plt.show()
```

>>



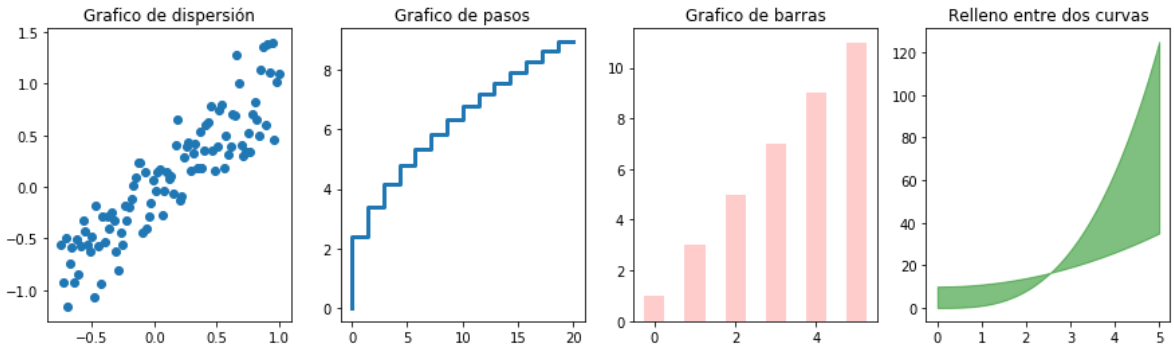
Gráfica 1. Representación de la figura 1 que contiene la señal 1 y de la figura 2 que contiene las señales 2 y 3.

Realice un gráfico de dispersión, un diagrama de pasos, un diagrama de barras y una gráfica donde se rellene el área entre dos curvas. Cada grafica se realiza en una figura independiente.

```
import numpy as np
import matplotlib.pyplot as plt
# Crear la figura indicando que consiste en una tabla de 1 fila y 4 columnas
# y tamaño de 15 de ancho y 4 de alto
fig, axes = plt.subplots(1, 4, figsize=(15,4))
# Crear el grafico de dispersión en la posición 0
x1 = np.linspace(-0.75, 1., 100)
y1 = x1 + 0.25*np.random.randn(len(x1))
axes[0].scatter(x1, y1)
axes[0].set_title("Grafico de dispersión")
# Crear el grafico de pasos en la posición 1
x2 = np.linspace(0, 20, 15)
y2 = 2*np.sqrt(x2)
axes[1].step(x2, y2, lw=3)
# Con grosor de línea de 3
axes[1].set_title("Grafico de pasos")
# Crear el grafico de barras en la posición 2
```

```
x3 = np.array([0,1,2,3,4,5])
y3 = 2*x3 + 1
axes[2].bar(x3, y3, align="center", color="red", width=0.5, alpha=0.2)
# barras centradas, de 0.5 de ancho, color rojo y una transparencia de 0.2
axes[2].set_title("Grafico de barras")
# Crear el grafico donde se rellena el área entre dos curvas en la posición 3
x4 = np.linspace(0.0, 5.0)
y4_1 = (x4**2)+10
y4_2 = x4**3
axes[3].fill_between(x4, y4_1, y4_2, color="green", alpha=0.5)
# Relleno de color verde y una transparencia de 0.5
axes[3].set_title("Relleno entre dos curvas")
plt.show() # Mostrar la figura
```

>>

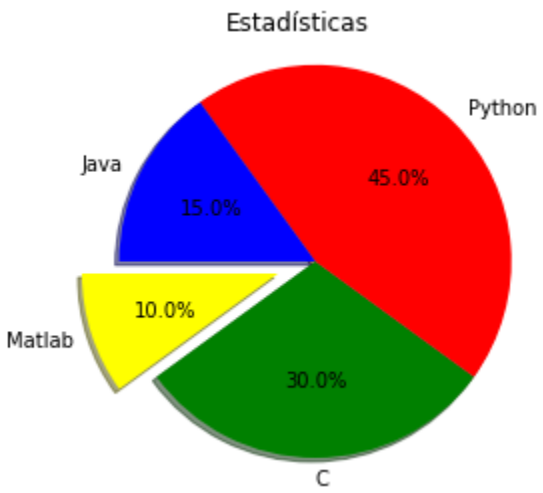


Gráfica 2. Representación de cómo se vería cada gráfica.

Realice un gráfico estadístico basado en sectores circulares y guarde el diagrama en un archivo de formato png.

```
import matplotlib.pyplot as plt
import numpy as np
# Etiquetas de elementos a comparar
labels = ['Matlab','C','Python','Java']
# Valores (en porcentaje) de cada uno de los elementos respectivos
values = [10,30,45,15]
# Colores para cada elemento
colors = ['yellow','green','red','blue']
# distancia de cada elemento con respecto al centro
explode = [0,0,0,0]
# Alejar del centro el elemento con el valor menor
explode[np.argmin(values)] = 0.2
# Crear el grafico de sectores circulares
plt.pie(values,
        labels=labels,
        colors=colors,
        explode=explode,
        shadow=True,
        autopct='%1.1f%%',
        startangle=180)
plt.title('Estadísticas')
plt.axis('equal') # Mantener una forma circular
```

```
plt.savefig('Estadisticas.png') # Guardar el grafico en un archivo .png

>>
```

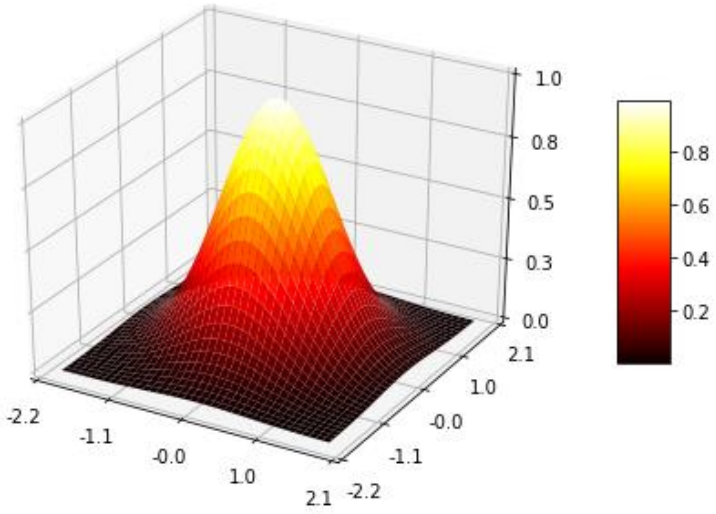


Gráfica 3. Representación de sectores circulares.

Realice un gráfico 3D de la siguiente función: $Z = e^{-X^2-Y^2}$, con los valores de X y Y entre [-2, 2]. Además, agregue una escala de color a la superficie con respecto a su valor máximo y mínimo.

```
# Importar la función que permite el grafico en 3D
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure() # Crear el espacio para la grafica
ax = Axes3D(fig) # Indicar que es de tipo 3D
X = np.arange(-2,2,0.1) # Valores del eje x
Y = np.arange(-2,2,0.1) # Valores del eje y
X,Y = np.meshgrid(X,Y) # Crear la matriz de coordenadas con respecto a x, y

def f(x,y):
    # Funcion de para obtener los valores de z con respecto a x , y
    return np.exp(-x**2-y**2)

# Crear la gráfica 3D de la superficie, con mapa de colores tipo plt.cm.hot
surf = ax.plot_surface(X, Y, f(X,Y), cmap=plt.cm.hot)
# Indicar que solo es necesario 5 divisiones de la cuadricula en cada eje
ax.zaxis.set_major_locator(LinearLocator(5))
ax.xaxis.set_major_locator(LinearLocator(5))
ax.yaxis.set_major_locator(LinearLocator(5))
# Indicar que solo es necesario mostrar hasta 1 número decimal en los ejes
ax.zaxis.set_major_formatter(FormatStrFormatter('%.01f'))
ax.xaxis.set_major_formatter(FormatStrFormatter('%.01f'))
ax.yaxis.set_major_formatter(FormatStrFormatter('%.01f'))
# Agregar una barra para la escala de los colores en la superficie
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.show()
```

>>



Gráfica 4. Representación 3D de la función $Z = e^{-X^2-Y^2}$.

4. SciPy

SciPy se basa en la estructura de los *arrays* de NumPy y lleva la programación científica a un nivel completamente nuevo al proporcionar módulos con funciones matemáticas avanzadas, tales como: integración, solucionador de ecuaciones diferenciales, funciones especiales, optimizaciones, manipulación de diferentes tipos de archivos, procesamiento de señales y más.

A continuación, se realizan ejemplos utilizando algunos de los módulos del paquete SciPy. Para conocer más módulos y funciones, consulte la página oficial de la documentación de SciPy, referenciada en la sección Material de apoyo.

Módulo optimize: hay varias formas de ajustar los datos con una regresión lineal. En este ejemplo se utilizará la función curve_fit del módulo. Se generan datos de una función conocida y se agrega ruido a estos. Por último, se realiza el ajuste a los datos ruidosos. La función que se modela en el ejemplo es una ecuación lineal simple, f (x) = ax + b, donde a y b son constantes.

```
import numpy as np # Se importa el paquete Numpy y se enmascara como np
from scipy.optimize import curve_fit # Importamos la función del módulo optimize de scipy

def Funcion(x, a, b):
    # Funcion que se encarga de modelar y crear los datos
    # F(x) = a*x + b -> a y b son constantes
    return (a*x)+b

# Generar los datos limpios
x = np.linspace(0, 10, 100)
a = 2
b = 15
y = Funcion(x, a, b)
# Adicionar ruido a los datos
```

```
y_ruido = y + 0.9*np.random.normal(size=len(x))
# Ejecutar el ajuste a los datos ruidosos
ajuste, pcov = curve_fit(Funcion, x, y_ruido)
# Imprimimos los valores de ajuste
print("Los valores originales (limpios) para a y b son:")
print("El valor de a: "+ str(a))
print("El valor de b: "+ str(b))
print("\nLos valores obtenidos del ajuste para a y b son:")
print("El valor de a: "+ str(ajuste[0]))
print("El valor de b: "+ str(ajuste[1]))

>>
Los valores originales (limpios) para a y b son:
El valor de a: 2
El valor de b: 15

Los valores obtenidos del ajuste para a y b son:
El valor de a: 1.9663202533477777
El valor de b: 15.23450468382866
```

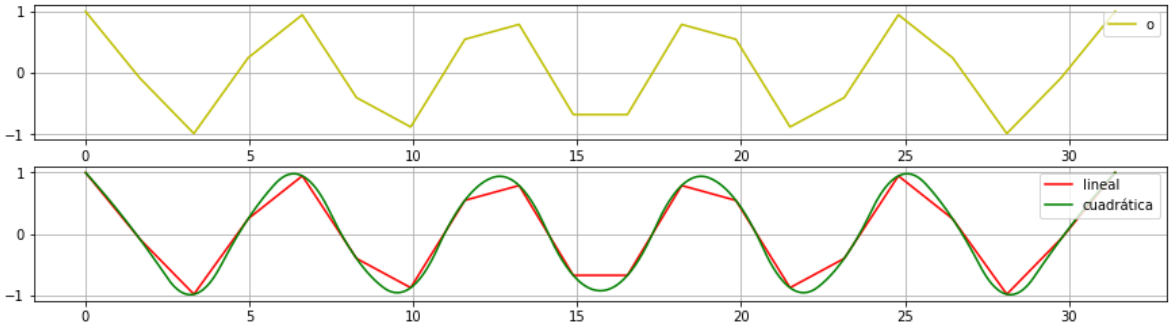
Como se puede observar en los resultados, los valores de ajuste obtenidos al utilizar la función curven_fit con los datos ruidosos, son muy cercanos a los valores originales.

Módulo interpolation: como su nombre lo indica es utilizado para objetos en interpolación. Tanto en el campo científico como en la ingeniería se presentan datos o señales que tienden a una forma funcional; nuestro trabajo es el de modelar dichos datos para poder manejarlos con mayor facilidad y predecir su comportamiento temporal o espacial.
SciPy ofrece varias funciones diferentes para la interpolación, que van desde los casos simples univariados hasta los complejos multivariados. El siguiente ejemplo interpola una función sinusoidal con diferentes parámetros de ajuste. El primer parámetro es un ajuste "lineal" y el segundo es un ajuste "cuadrático".

```
import numpy as np
from scipy.interpolate import interp1d #Importamos la función interp1d del módulo interpolate de Scipy
import matplotlib.pyplot as plt
# Crear los datos a interpolar
x = np.linspace(0, 10*np.pi, 20)
y = np.cos(x)
# Interpolar los datos
f_lineal = interp1d(x, y, kind='linear') # Ajuste lineal
f_cuadratica = interp1d(x, y, kind='quadratic') # Ajuste cuadrático
# Crear los datos de la interpolación
x_int = np.linspace(x.min(), x.max(), 1000)
y_intl = f_lineal(x_int) # Lineal
y_intq = f_cuadratica(x_int) # Cuadrática
# Graficar los datos originales y en otra los de la interpolación
fig, axes = plt.subplots(2, 1, figsize=(15,4))
axes[0].plot(x, y, 'y')
axes[0].legend('original', loc='upper right')
axes[0].grid()
axes[1].plot(x_int, y_intl, 'r')
axes[1].plot(x_int, y_intq, 'g')
```

```
axes[1].legend(('lineal', 'cuadrática'), loc='upper right')
axes[1].grid()
plt.show()
```

>>



Gráfica 5. Representación de sinusoidal con ajustes lineal y cuadrático.

Como se puede observar en los resultados, las graficas de los datos obtenidos de la interpolación son semejantes o tienden a la original. La cuadrática es mas cercana a la función sinusoidal que la lineal, pero esto en gran parte se debe a la falta de datos en la original.

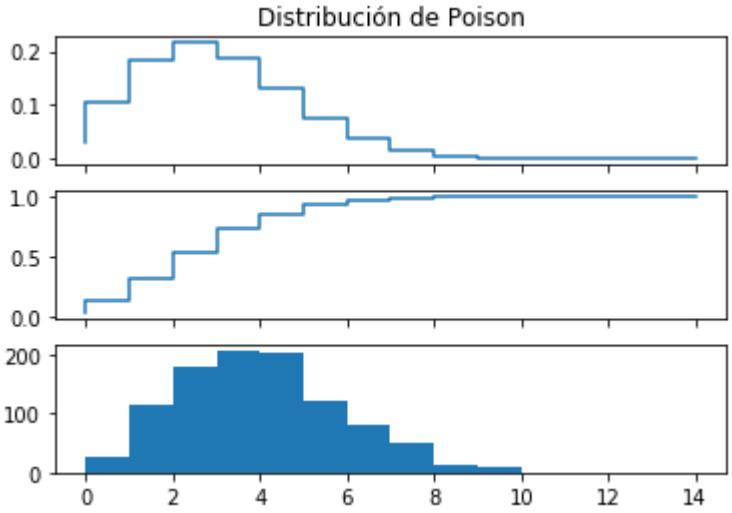
Módulo stats: este módulo contiene una gran cantidad de distribuciones de probabilidad, así como una creciente biblioteca de funciones estadísticas. En NumPy hay funciones estadísticas básicas como la media, desviación estándar, mediana, máximo, mínimo, entre otras. Para cálculos rápidos, estos métodos son útiles, pero en el caso de investigación cuantitativa y el manejo de grandes cantidades de datos se queda corto y deficiente.

En el siguiente ejemplo se crean dos distribuciones aleatorias, una discreta de tipo Poisson y otra continua de tipo Normal. A ambas se les aplican dos funciones de probabilidad y cada una de estas es graficada en una figura independiente. Se realiza un histograma de cada distribución. Por último, se obtienen los valores de media, desviación estándar y varianza.

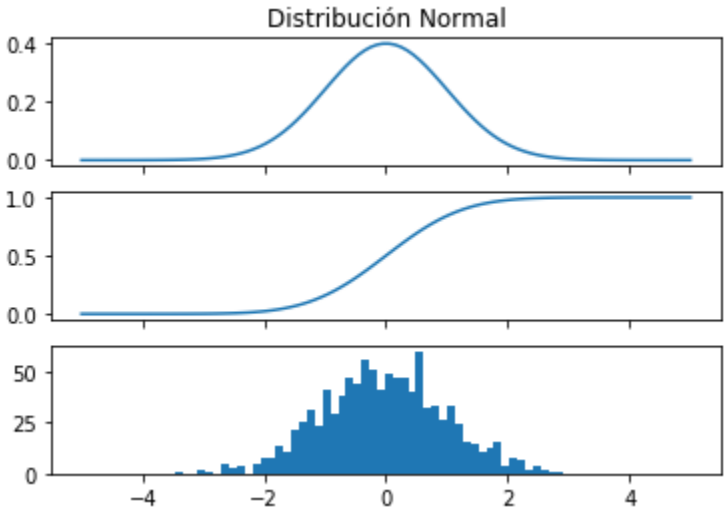
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
# crear una variable aleatoria (discreta) con distribución de poisson
X = stats.poisson(3.5)
n = np.arange(0,15)
fig, axes = plt.subplots(3,1, sharex=True)
# graficar la Función de masa de probabilidad (PMF)
axes[0].step(n, X.pmf(n))
axes[0].set_title("Distribución de Poison")
# graficar la Función de Distribución Acumulada (CDF)
axes[1].step(n, X.cdf(n))
# graficar histograma de 1000 realizaciones aleatorias de la variable estocástica X
axes[2].hist(X.rvs(size=1000))
plt.show()
# crear una variable aleatoria (continua) con distribución normal
Y = stats.norm()
x = np.linspace(-5,5,100)
```

```
fig, axes = plt.subplots(3,1, sharex=True)
# graficar la Función de distribución de probabilidad (PDF)
axes[0].plot(x, Y.pdf(x))
axes[0].set_title("Distribución Normal")
# graficar la Función de Distribución Acumulada (CDF)
axes[1].plot(x, Y.cdf(x));
# graficar histograma de 1000 realizaciones aleatorias de la variable estocástica Y
axes[2].hist(Y.rvs(size=1000), bins=50)
plt.show()
# imprimir valores estadísticos
print('Valores Estadísticos: ')
print('Distribución de Poisson:')
print('Media:'+str(X.mean())+
    'Desviación estándar:'+str(X.std())+
    'Varianza:'+str(X.var()))
print('Distribución Normal:')
print('Media:'+str(Y.mean())+
    'Desviación estándar:'+str(Y.std())+
    'Varianza:'+str(Y.var()))
```

>>



Gráfica 6. Representación de la distribución de Poison.



Gráfica 7. Representación de la distribución Normal.

Valores estadísticos:

Distribución de Poisson

Media:3.5
Desviación estándar:1.8708286933869707
Varianza:3.5

Distribución Normal

Media:0.0
Desviación estándar:1.0
Varianza:1.0

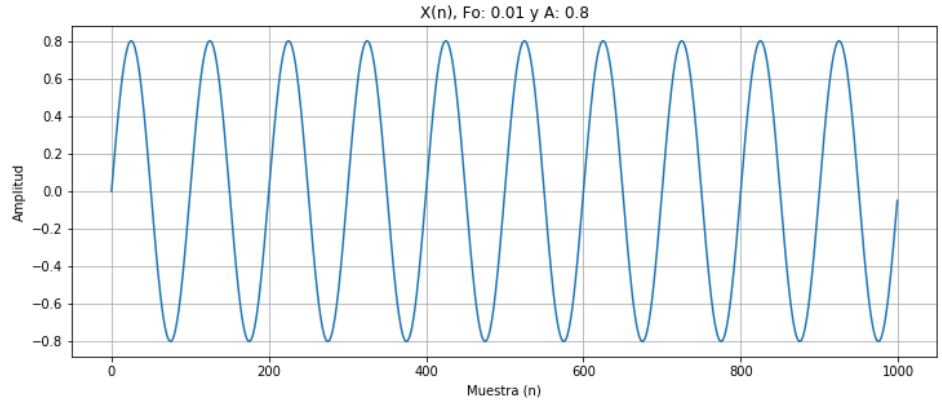
Módulo signal: este módulo contiene una gran cantidad de herramientas dedicadas al procesamiento y análisis de señales, como lo son las operaciones de convolución, diseño y aplicación de filtros, análisis de sistemas lineales continuos y discretos, análisis espectrales, entre otras.

En el siguiente ejemplo se genera una secuencia ficticia x(n) y se les aplica w(n) como una secuencia de ruido Gaussiano de características media cero y varianza σ2=0.01, α=0.8 y un retraso D=0 muestras, para obtener una señal ruidosa y(n). Luego, se diseña un filtro FIR pasa bajas para la señal y(n), con una frecuencia de corte de 1.25 muestras/s y orden 31. Por último, se analiza la respuesta del filtro en frecuencia de magnitud y fase.

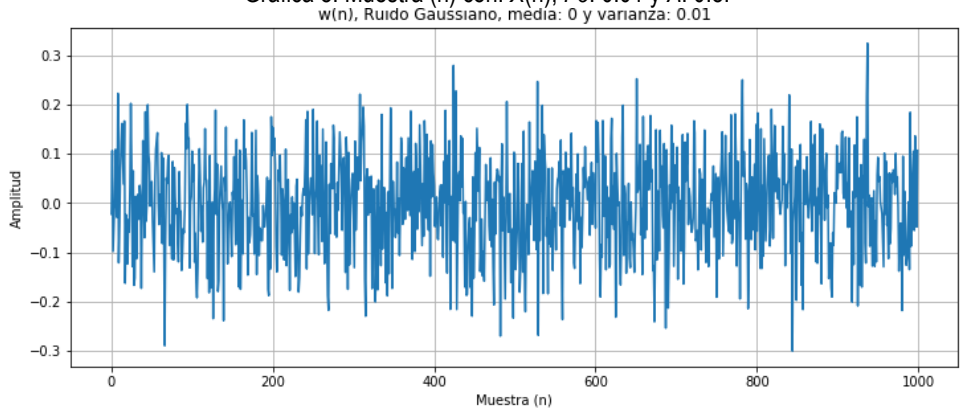
```
import scipy.signal as signal;
import matplotlib.pyplot as plt;
import numpy as np;
import math as mt
# Generación de la secuencia ficticia X(n)
Fo = 0.01     # Frecuencia de la señal
Tp = 1/Fo     # Periodo de la señal
Fs = 25
num_samples = 1000  # número de muestras
n = np.arange(0,num_samples);   # Muestras
A = 0.8;            # Amplitud
x = A*np.sin(2*mt.pi*Fo*n); # Señal biológica
```

```
# Generacion del ruido gaussiano w(n)
mean = 0           # Media
var = 0.01         # Varianza
std = np.sqrt(var)  # Desviación estándar
Noise = np.random.normal(mean, std, size=num_samples)
# Generacion de la secuencia ficticia Y(n)
y = x + Noise
# Graficar las tres señales (X, w, Y)
fig = plt.figure('Señales generadas xe, w y ye',(12,16))
graph1_1 = fig.add_subplot(311)
graph1_1.plot(n, x)
graph1_1.set(xlabel='Muestra (n)', ylabel='Amplitud', title='X(n), Fo: '+ str(Fo)+ ' y A: ' + str(A));
graph1_1.grid(which='both', axis='both')
graph1_2 = fig.add_subplot(312)
graph1_2.plot(n, Noise)
graph1_2.set(xlabel='Muestra (n)', ylabel='Amplitud', title='w(n), Ruido Gaussiano, media: '+ str(mean)+ ' y varianza: ' +
str(var));
graph1_2.grid(which='both', axis='both')
graph1_3 = fig.add_subplot(313)
graph1_3.plot(n, y)
graph1_3.set(xlabel='Muestra (n)', ylabel='Amplitud', title='Y(n) = A*X(n) + w(n)');
graph1_3.grid(which='both', axis='both')
plt.show()
# Craer el filtro pasa bajas de tipo FIR
Fcutoff = 1.25         # Frecuencia de corte
Order = 31             # Orden del filtro
Fnyquist = Fs/2        # Frecuencia de Nyquist
LowpassfilterFIR = signal.firwin(Order, Fcutoff/Fnyquist)
# Analizar la respuesta del filtro
w, h = signal.freqz(LowpassfilterFIR)
h_dB = 20 * np.log10 (abs(h));
wn = w/np.max(w)
# Grafica de la magnitud
fig1 = plt.figure('Filtro pasa bajas FIR',(12,8))
graph2_1 = fig1.add_subplot(211)
graph2_1.plot(wn, h_dB)
graph2_1.set(ylabel='Magnitud [dB]', xlabel='Frecuencia normalizada [rad/sample]')
graph2_1.grid(which='both',axis='both')
graph2_1.set(title='Respuesta del filtro pasa bajas FIR, de orden: '+ str(Order)+ ' y Frec. corte: ' + str(Fcutoff))
# Grafica de la fase
graph2_2 = fig1.add_subplot(212)
angles = np.unwrap(np.angle(h))
graph2_2.plot(wn, angles, 'g')
graph2_2.set(ylabel='Angle (radians)', xlabel='Frecuencia normalizada [rad/sample]')
graph2_2.grid(which='both',axis='both')
plt.show()
# Aplicar el filtro utilizando dos funciones distintas del modulo signal
filteredLowpassFIRlfilt = signal.lfilter(LowpassfilterFIR, 1.0, y)
filteredLowpassFIRfiltfilt = signal.filtfilt(LowpassfilterFIR, 1.0, y)
# Graficar la señal ruidosa y las dos obtenidas al aplicar el filtro
fig7 = plt.figure('Filtrado del Ye con pasa bajas FIR',(12,6))
graph8_1 = fig7.add_subplot(111)
graph8_1.plot(n, y, 'y-', n, filteredLowpassFIRlfilt, 'b-',n, filteredLowpassFIRfiltfilt, 'r-')
graph8_1.set(xlabel='Muestra (n)', ylabel='Amplitud', title='Aplicando el filtro pasa bajas FIR');
graph8_1.grid(which='both', axis='both')
graph8_1.legend(('ye(n)','lfilter', 'filtfilt'))
```

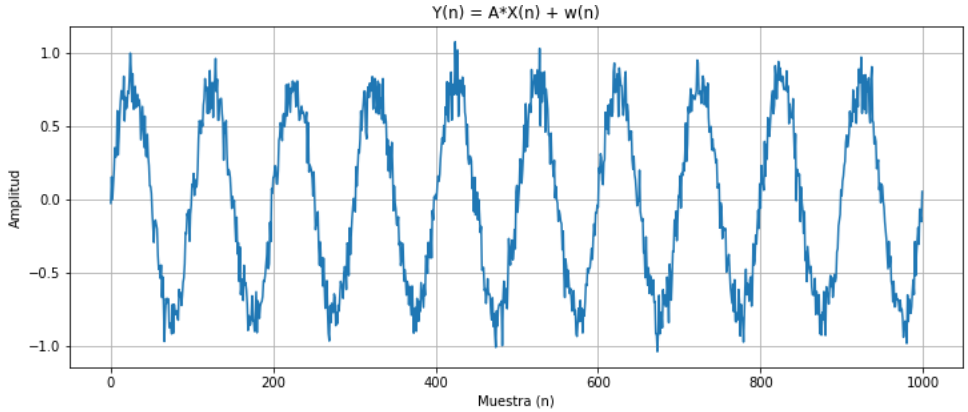
```
plt.show()

>>
```



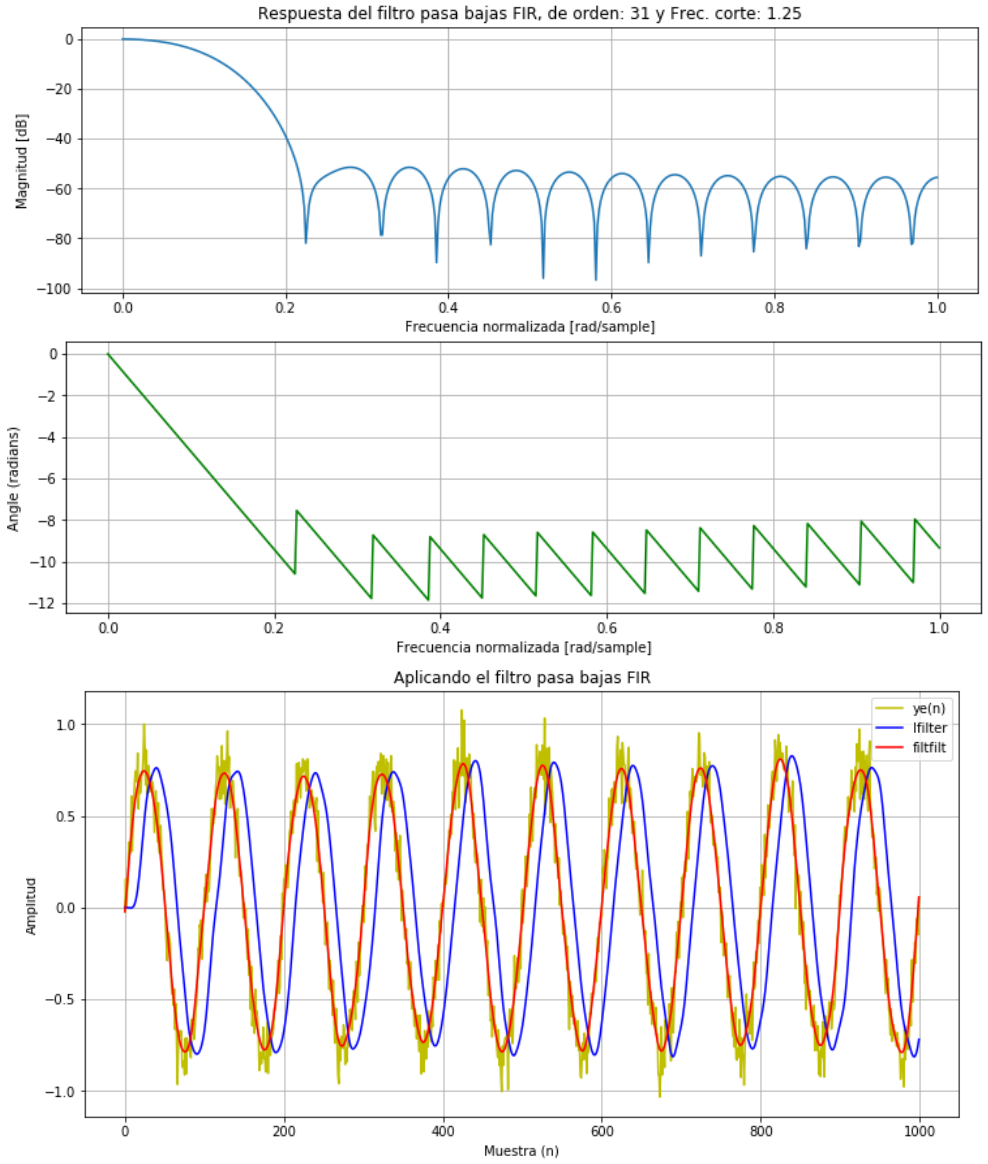
Gráfica 8. Muestra (n) con: *X*(n), *Fo*: 0.01 y *A*: 0.8.



Gráfica 9. Muestra (n) con: W(n), Ruido Gaussiano, media: 0 y varianza 0.01.



Gráfica 10. Muestra (n) con: Y(n) = A*X(n) + W(n).



Gráfica 11. Frecuencia normalizada con: respuesta del filtro pasa bajas FIR, de orden: 31 y Frec. Corte: 1.25.

Como se observa en las curvas obtenidas al aplicar el filtro a la señal ruidosa y(n), se nota que se eliminó el Ruido Gaussiano. En el caso de la función lfilt (azul) se presenta un desfase con respecto a la señal original y en el filtfilt (roja) se mantienen mejor las características de la señal original.

5. OpenCV

OpenCV (Open Source Computer Vision) es una librería de código libre para la visión por computadora, está escrita principalmente en C y C ++, pero existe un desarrollo activo para Python, Java, MATLAB y otros lenguajes. El principal objetivo de OpenCV es proporcionar una infraestructura de visión computarizada fácil de usar, que permita a las personas crear rápidamente aplicaciones de visión bastante sofisticadas.

Esta librería cuenta con más de 500 funciones que abarcan muchas áreas de la visión, incluida la inspección de productos de fábrica, imágenes médicas, seguridad, interfaz de usuario, calibración de la cámara, visión estéreo y robótica.

A continuación, se realizan dos ejemplos. En el primer ejemplo, se muestran las funciones básicas para cargar, cambiar el formato y guardar una imagen.

Para conocer más módulos y funciones, consulte la página oficial de la documentación de OpenCV, referenciada en la sección Material de apoyo.

```
import cv2 # Importar la librería de OpenCV
# Cargar la imagen utilizando la dirección del archivo en formato jpg
img = cv2.imread("C:/Users/User/Desktop/Imagen.jpg")
cv2.imshow('Imagen Original', img) # Mostrar la imagen en una ventana
# Cargar la imagen en escala de grises
gray_img = cv2.imread("C:/Users/User/Desktop/Imagen.jpg", cv2.IMREAD_GRAYSCALE)
cv2.imshow('Escala de Grises desde imread', gray_img)
cv2.imwrite('C:/Users/User/Desktop/Imagen_Grises.jpg', gray_img) # Guardar la imagen
# Guardar en otro formato .png
cv2.imwrite('C:/Users/User/Desktop/Imagen_Nuevo_Formato.png', img, [cv2.IMWRITE_PNG_COMPRESSION])
# Cargar otra vez la imagen garantizando el formato RGB
img2 = cv2.imread("C:/Users/User/Desktop/Imagen.jpg", cv2.IMREAD_COLOR)
gray_img2 = cv2.cvtColor(img2, cv2.COLOR_RGB2GRAY) # Cambiar a escalas de grises
yuv_img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2YUV) # Cambiar a formato YUV
y,u,v = cv2.split(yuv_img2) # Separar cada uno de los canales de la imagen en formato YUV
# Mostrar la imagen en formato de grises y los 3 canales del formato YUV por separado
cv2.imshow('Escala de Grises desde cvtColor', gray_img2)
cv2.imshow('Canal Y', y)
cv2.imshow('Canal U', u)
cv2.imshow('Canal V', v)
# Fusionar canales para formar nuevas imágenes
g,b,r = cv2.split(img2)  # Extrar los tres canales del formato RGB
gbr_img = cv2.merge((g,b,r)) # Fusionar los canales en el orden de G B R
rbr_img = cv2.merge((r,b,r)) # Fusionar los canales en el orden de R B R
cv2.imshow('Fusión en G-R-B', gbr_img)
cv2.imshow('Fusión en R-B-R', rbr_img)
# Indicar que las ventanas se cierran cuando se presiona cualquier tecla
cv2.waitKey(0)
cv2.destroyAllWindows()

>>
```

Figura 1. Imagen original de El hombre creador de energía. Adaptada de Urán, O. (2009).

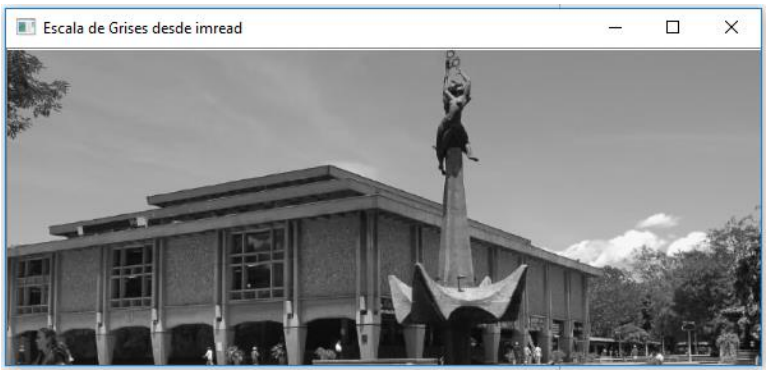


Figura 2. Escala de grises (imread) de El hombre creador de energía. Adaptada de Urán, O. (2009).

Figura 3. Escala de grises (cvtColor) de El hombre creador de energía. Adaptada de Urán, O. (2009).

Figura 4. Canal Y de El hombre creador de energía. Adaptada de Urán, O. (2009).

Figura 5. Canal U de El hombre creador de energía. Adaptada de Urán, O. (2009).

Figura 6.Canal V de El hombre creador de energía. Adaptada de Urán, O. (2009).

Figura 7. Fusión en G-R-B de El hombre creador de energía. Adaptada de Urán, O. (2009).

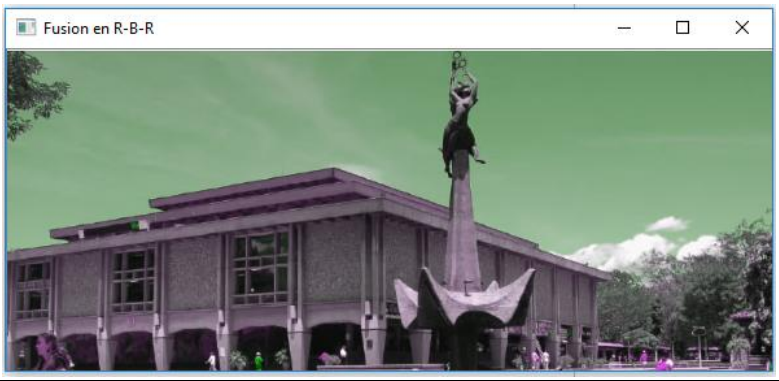


Figura 8. Fusión en R-B-R de El hombre creador de energía. Adaptada de Urán, O. (2009).

El siguiente ejemplo se muestran algunas funciones básicas para mover y ajustar el tamaño de una imagen.

```
import cv2
import numpy as np
img = cv2.imread("C:/Users/User/Desktop/Imagen.jpg") # Cargar la imagen
num_rows, num_cols = img.shape[:2] # Obtener el número de filas y columnas (pixeles)
# Mover la imagen 70 pixeles en x (hacia la derecha) y 110 en y (hacia abajo)
translation_matrix = np.float32([[1,0,70],[0,1,110]])
img_translation = cv2.warpAffine(img, translation_matrix, (num_cols , num_rows))
# El tercer argumento garantiza que la ventana permanezca con las mismas dimensiones del original
cv2.imshow('Imagen original', img)
cv2.imshow('Imagen trasladad', img_translation)
# Rotar la imagen
# Se necesita primero trasladar hacia la derecha para garantizar que se pueda mostrar completa
translation_matrix = np.float32([[1,0,120],[0,1,0]])
# Indicar el giro de la imagen
rotation_matrix = cv2.getRotationMatrix2D((num_cols, num_rows), 30, 1)
# Aplicar primero la traslación y luego la rotación
img_translation = cv2.warpAffine(img, translation_matrix, (num_cols, num_rows))
img_rotation = cv2.warpAffine(img_translation, rotation_matrix, (num_cols+70, num_rows+250))
# Se aumenta el tamaño de la ventana para observar la imagen
cv2.imshow('Imagen Rotada', img_rotation)
# Escalar la imagen
# Escalonamiento lineal de la imagen con 1.2 en x y 1.2 en y
img_scaled_linear = cv2.resize(img,None,fx=1.2, fy=1.2, interpolation = cv2.INTER_LINEAR)
cv2.imshow('Escalonar - interpolacion lineal', img_scaled_linear)
# Escalonamiento cubico de la imagen con 1.2 en x y 1.2 en y
img_scaled_cubic = cv2.resize(img,None,fx=1.2, fy=1.2, interpolation = cv2.INTER_CUBIC)
cv2.imshow('Escalonar - interpolacion cubica', img_scaled_cubic)
# Escalonamiento por área, indicando tamaño en alto y ancho
img_scaled_area = cv2.resize(img,(450, 400), interpolation = cv2.INTER_AREA)
cv2.imshow('Escalonar - Area', img_scaled_area)
cv2.waitKey(0)
cv2.destroyAllWindows()

>>
```

Figura 8. Imagen original de El hombre creador de energía. Adaptada de Urán, O. (2009).

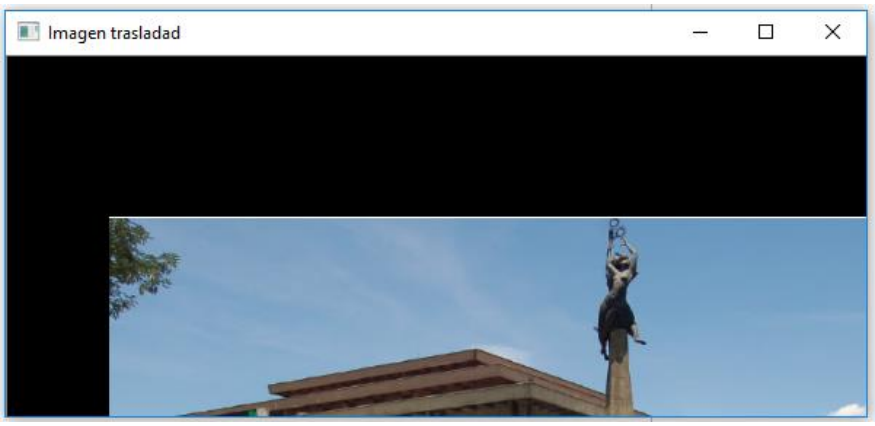


Figura 9. Trasladada de El hombre creador de energía. Adaptada de Urán, O. (2009).

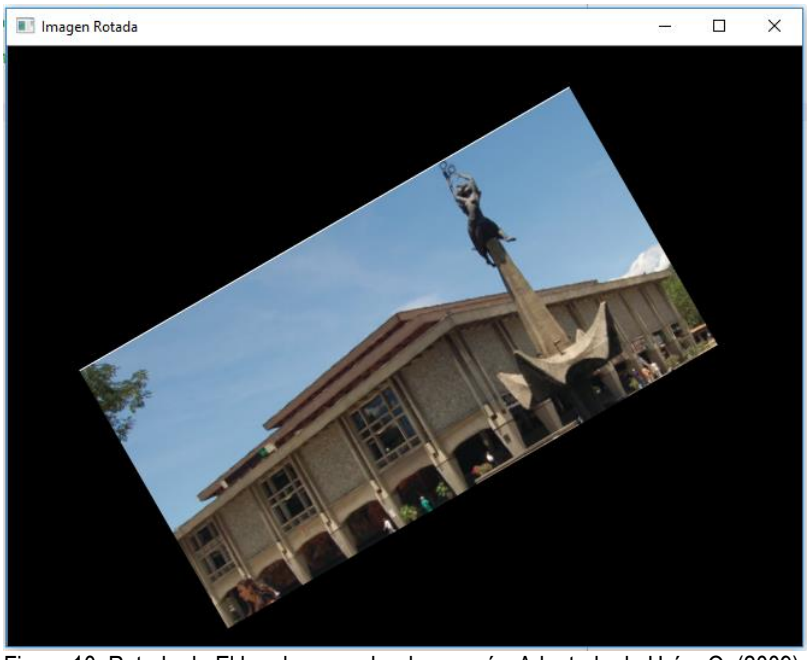


Figura 10. Rotada de El hombre creador de energía. Adaptada de Urán, O. (2009).

Figura 11. Interpolación lineal de El hombre creador de energía. Adaptada de Urán, O. (2009).

Figura 12. Interpolación cúbica de El hombre creador de energía. Adaptada de Urán, O. (2009).

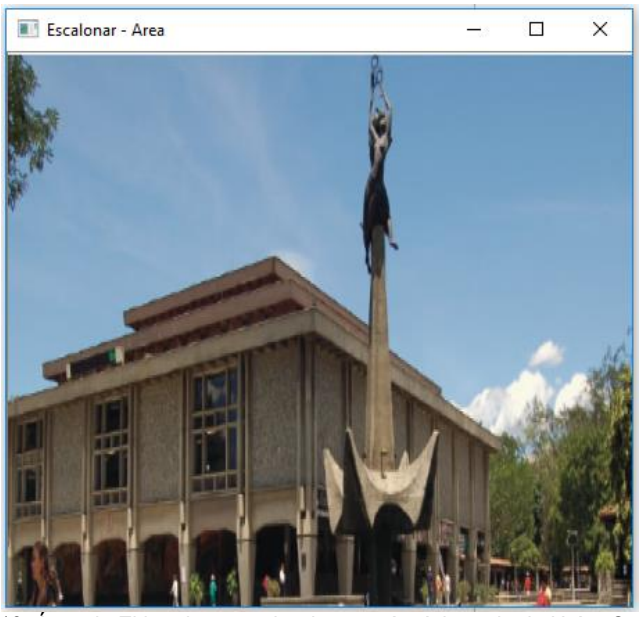


Figura 13. Área de El hombre creador de energía. Adaptada de Urán, O. (2009).

Referencias

Uran, O. (2009). *Biblioteca central UdeA* [Fotografía]. Recuperado de https://www.flickr.com/photos/uranomar/3788254830/in/album-72157621940454280/