Revista digital
Matemática, Educación e Internet
https://tecdigital.tec.ac.cr/revistamatematica/
Vol 21, No 1. Setiembre – Febrero, 2021 ISSN 1659 -0643

Ejemplos básicos de álgebra lineal con python

| Basic examples of linear algebra with python |

Byron Jiménez Oviedo
byron.jimenez.oviedo@una.ac.cr
Escuela de matemática
Universidad Nacional
Costa Rica

Katalina Oviedo Rodríguez
koviedo@utn.ac.cr
Escuela de matemática
Universidad Técnica Nacional
Universidad Nacional
Costa Rica

Recibido: 30 setiembre 2019 Aceptado: 15 Marzo 2020

Resumen. En muchas ocasiones, durante el desarrollo de un curso, es realmente difícil (por la premura de abarcar los contenidos) dar ejemplos concretos y aplicaciones de la teoría que se ha desarrollado. Este material ofrece una presentación de algunas aplicaciones o visualizaciones que se pueden desarrollar con los contenidos vistos en curso de álgebra lineal básica, pasando por operaciones matriciales, transformaciones y vectores propios. Para el desarrollo de estos temas se utilizará el lenguaje de programación python. Por lo cual se recomienda un conocimiento básico en programación o en python.

Palabras clave: Álgebra lineal, aplicaciones, transformaciones en imágenes, descomposición en valores singulares.

Abstract. Many times during the development of a course it is really difficult (due to the rush to cover the contents) to give specific examples and applications of the theory that has been developed. This material offers a presentation of some applications or visualizations that can be developed with the contents in the basic course of linear algebra, going through matrix operations, transformations and eigenvalues and eigenvectors. The python programming language will be used to develop these topics. Therefore, basic knowledge in programming or python is recommended.

KeyWords: Linear algebra, applications, transformations in images, singular values decomposition.

1.1 Introducción

Es claro que los cursos de matemática a nivel universitario se vuelven muy teóricos y usualmente están alejados de aplicaciones concretas. Un curso de álgebra lineal básica (donde se estudia: matrices y sistemas de ecuaciones lineales, espacios vectoriales, transformaciones lineales y valores y vectores propios) no se escapa de esto último, y a pesar de tener aplicaciones intrínsecas teórias, es natural que el estudiante se pregunte por la utilidad práctica.

Este artículo pretende dar ejemplos concretos donde el álgebra lineal es el eje primordial y su desarrollo es pragmático y simple. En esta primera parte hemos seleccionado dos ejemplos que son muy visuales, es decir, el lector además de darse cuenta de la aplicación como tal, también va a poder visualizarla. La primera de dichas aplicaciones está relacionada con transformaciones afines: rotación, traslación, multiplicación por escalar, entre otras. Estas transformaciones se van a aplicar a dos tipos de objetos. El primero será una imagen vectorial, es decir, una imagen digital conformada por segmentos, polígonos, arcos y otras. El segundo objeto es una imagen digital basada en pixeles. El tratamiento de ambas tienen ciertas disimilitudes que son interesantes de tratar [1, 4, 3].

La segunda aplicación está relacionada con una descomposición de matrices y con valores y vectores propios. Específicamente, vamos a tratar la factorización matricial SVD (descomposición en valores singulares [2, 5, 6, 7]), la cual tiene muchas aplicaciones tales como: el cálculo de pseudoinversas, ajuste de mínimo cuadrados, y aproximación de matrices, entre otras [2, 7]. En especial, vamos a utilizar esta descomposición para la compresión de imágenes. Ya que, como veremos una imagen es una matriz donde cada posición contiene el "color" del pixel en dicha posición. En el caso, de que el rango de la matriz es significativamente pequeño que sus dimensiones el método nos ayudará a poder generar una imagen similar (matriz aproximada) pero con la característica que se necesita menos información que la original.

Toda esta manipulación vamos a tratarla con el lenguaje de programación python. Además del propósito de exponer nuestros ejemplos, usar este lenguaje ayudará también a una introducir o mejorar las habilidades y competencias con dicho lenguaje de programación.

1.2 Visualización de operaciones matriciales

Recordemos que una transformación lineal T del espacio vectorial V al espacio vectorial W es una regla que asigna a cada vector $\mathbf{v} \in V$ un único vector $T(\mathbf{v})$ en W, tal que

- $T(\mathbf{u}+\mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v})$, para todo $\mathbf{u}, \mathbf{v} \in V$.
- $T(c \cdot \mathbf{u}) = c \cdot T(\mathbf{u})$, para todo $\mathbf{u} \in V$, $c \in \mathbb{R}$.

En esta sección queremos visualizar cómo un espacio se transforma al aplicar una transformación lineal. De hecho, vamos a trabajar con transformaciones $T : \mathbb{R}^m \to \mathbb{R}^n$, específicamente cuando $n = m = 2$.

Ahora, utilizando la propiedades de las matrices no es difícil mostrar que para una matriz A de tamaño $m \times n$ la aplicación $T(X) = AX$ con $X \in \mathbb{R}^m$ es una transformación lineal. Existen un conjunto de transformaciones lineales que son parte de este tipo de transformaciones, llamadas transformaciones afines. Estas transformaciones preservan puntos, rectas, rectas paralelas, entre otras. Todo esto puede ser visualizado al transformar (rotar, trasladar, escalar, etc.) objetos en 2D o 3D.

Algunas de estas transformaciones en 2D no corresponden a una multiplicación directa de matrices 2×2, sino que debemos introducir una nueva representación para poder utilizar matrices 3×3, a esto se le llama coordenadas homogéneas. Por ejemplo, un punto $(x, y) \in \mathbb{R}^2$ puede ser identificado con el punto $(x, y, 1) \in \mathbb{R}^3$, así el punto $(x, y, 1)$ es la coordenada homogénea de (x, y). En general, para un vector $\mathbf{v} \in \mathbb{R}^n$ la forma homogénea de $\mathbf{v}$ es $\hat{\mathbf{v}} = (\mathbf{v}, 1) \in \mathbb{R}^{n+1}$.

Algunas de estas transformaciones afines en $\mathbb{R}^3$ se presentan en el cuadro 1.2, donde el efecto de la transformación en un punto $X = (x, y, z)^T$ es determinado al multiplicar la matriz A por X, dando un nuevo punto que vamos a representar por $X' = (x', y', z')^T$.

Transformación	Matriz A	$X' = AX$
Identidad	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$
Reflexión respecto al eje x	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x \\ -y \\ z \end{pmatrix}$

Reflexión respecto a eje y	$\begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} -x \\ y \\ z \end{pmatrix}$
Rotación en ángulo θ	$\begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x\cos(\theta) + y\sin(\theta) \\ -x\sin(\theta) + y\cos(\theta) \\ z \end{pmatrix}$
Cambio de escala	$\begin{pmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x \cdot k_x \\ y \cdot k_y \\ z \end{pmatrix}$
Traslación	$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ z \end{pmatrix}$
Deformación (vertical/horizontal)	$\begin{pmatrix} 1 & h & 0 \\ v & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x + y \cdot h \\ x \cdot v + y \\ z \end{pmatrix}$

Tabla 1.2: Transformaciones

En cada transformación, dada en la cuadro 1.2, se puede observar que la variable z no es modificada. Por ejemplo, la rotación que se presenta es una rotación alrededor del eje z. Ahora, vamos a aprovechar que un punto en $\mathbb{R}^2$ se puede representar en python como una tripleta (x, y, b) donde x, y representan las coordenadas del punto y b es un índice o bandera de cada punto. Como para efectos de este ejemplo no vamos a necesitar dicho índice vamos a considerarlo convenientemente como 1, es decir, cada punto (x, y) en el plano lo vamos a representar en python con $(x, y, 1)$ (lo tomamos como su coordenada homogénea).

Vamos a dibujar una figura simple la cual puede ser fácilmente representada con una matriz y con la cual vamos a poder visualizar el efecto de cada transformación.

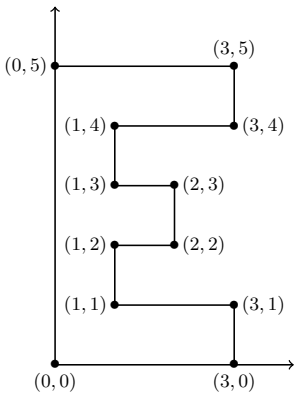


Figura 1.1: Letra E

En la Figura 1.1 vemos dibujado la letra E. Podemos representar esta figura matricialmente al guardar coordenadas estratégicas, en este caso los puntos esquinas tal y como se ponen en evidencia en la Figura 1.1. La matriz que representa esta letra es $E \in \mathbb{R}^{12\times 3}$ definida por

$$E = \begin{pmatrix} 0 & 0 & 1 \\ 3 & 0 & 1 \\ 3 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 1 \\ 2 & 2 & 1 \\ 2 & 3 & 1 \\ 1 & 3 & 1 \\ 1 & 4 & 1 \\ 3 & 4 & 1 \\ 3 & 5 & 1 \\ 0 & 5 & 1 \end{pmatrix}.$$

Cada fila de la matriz anterior representa una coordenada homogenea de un punto en $\mathbb{R}^2$, así que si en

ese orden unimos cada punto por medio de un segmento de recta vamos a obtener el dibujo deseado. Por ejemplo, si queremos reflejar la Figura 1.1 con respecto al eje y debemos realizar la multiplicación de la matriz E y la matriz de reflexión correspondiente, es decir,

$$E \cdot \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 3 & 0 & 1 \\ 3 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 1 \\ 2 & 2 & 1 \\ 2 & 3 & 1 \\ 1 & 3 & 1 \\ 1 & 4 & 1 \\ 3 & 4 & 1 \\ 3 & 5 & 1 \\ 0 & 5 & 1 \end{pmatrix} \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ -3 & 0 & 1 \\ -3 & 1 & 1 \\ -1 & 1 & 1 \\ -1 & 2 & 1 \\ -2 & 2 & 1 \\ -2 & 3 & 1 \\ -1 & 3 & 1 \\ -1 & 4 & 1 \\ -3 & 4 & 1 \\ -3 & 5 & 1 \\ 0 & 5 & 1 \end{pmatrix}.$$

De lo anterior obtenemos la letra E de la Figura 1.1 reflejada con respeto al eje x (vea la Figura 1.2).

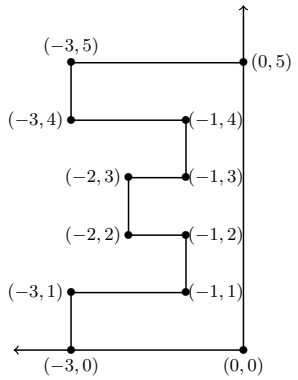


Figura 1.2: reflexión de la Letra E

Para realizar todo esto en python vamos a trabajar con el paquete NumPy, el cual es una biblioteca que proporciona una gran cantidad de operaciones sobre arreglos multidimensionales. Además, también vamos a trabajar con la biblioteca matplotlib la cual nos ayudará a la hora de graficar datos en listas. Para llamar a estas bibliotecas debemos escribir

```
import numpy as np
import matplotlib.pyplot as plt
```

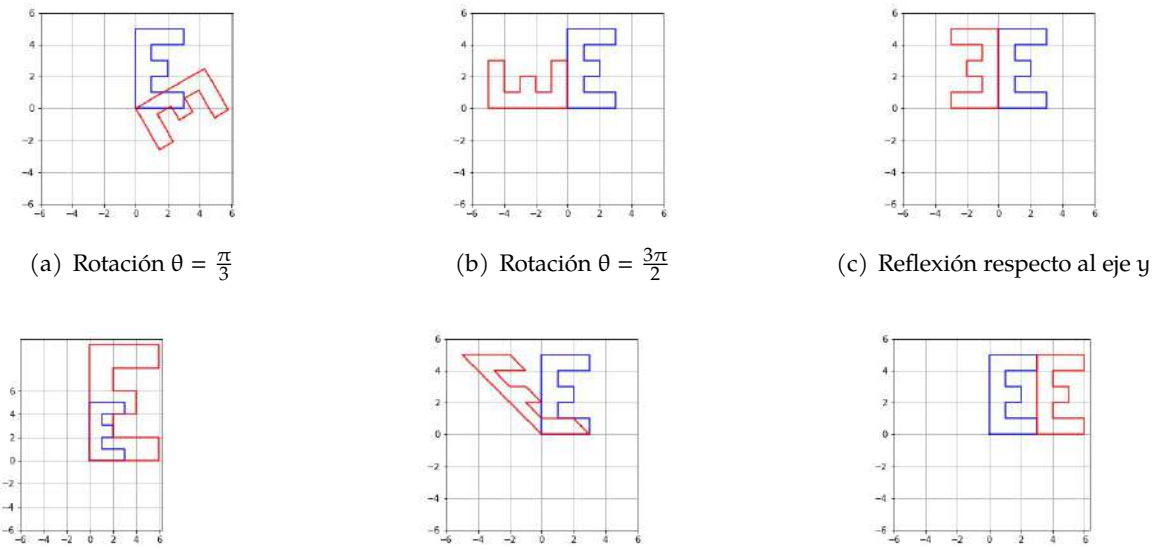
Para poder definir la matriz que contiene las coordenadas de la figura que queremos dibujar utilizamos el siguiente código (una vez importadas las las bibliotecas)

```
import numpy as np
import matplotlib.pyplot as plt

E = np.array([(0,0,1),(3,0,1),(3,1,1),(1,1,1),(1,2,1),(2,2,1),
(2,3,1),(1,3,1),(1,4,1),(3,4,1),(3,5,1),(0,5,1)])
```

El resto del código (con comentarios en verde) corresponde a la definición de las matrices que determinan la transformación, el uso de dichas transformaciones y la gráfica correspondiente.

```
import numpy as np
import matplotlib.pyplot as plt

E = np.array([(0,0,1),(3,0,1),(3,1,1),(1,1,1), (1,2,1),(2,2,1),
(2,3,1),(1,3,1),(1,4,1),(3,4,1),(3,5,1),(0,5,1)])

# Matriz Identidad
Ic = np.eye(3)

#Matriz de Reflexión con respecto al eje x
Refx=np.array([[1., 0, 0],[0, −1., 0],[0., 0., 1.]])

#Matriz de Reflexión con respecto al eje y
Refy=np.array([[−1., 0, 0],[0, 1., 0],[0., 0., 1.]])

# Matriz de Rotación
theta= np.pi/3 #Ángulo de rotación deseado,en este caso usamos pi/3.
R=np.array([[np.cos(theta), np.sin(theta), 0.],
[−np.sin(theta), np.cos(theta), 0.],
[0., 0., 1.]])

# Matriz de cambio de escala
s=2 #Escalar
S=np.array([[s, 0, 0.], [0., s, 0.],[0., 0., 1.]])

# Matriz de deformación horizontal/vertical
h=−1,v= 2
D=np.array([[1., h, 0.],[v, 1., 0.],[0., 0., 1.]])

# Matriz de traslación
tx=−3
ty=−5
T=np.array([[1., 0, tx], [0, 1., ty], [0., 0., 1.]])

#transformación y plot
ax = plt.gca()  #gca= get current axes, obtiene los ejes actuales
Ex = []  #Lista de primeras componentes
Ey = []  #Lista de segundas componentes
for row in E:
output_rown = (R.dot(row)) #Multiplicación de la matriz por el punto respectivo
x, y= output_rown #Coordenadas ya transformadas
Ex.append(x) #Se guardan las coordenadas
Ey.append(y)

plt.plot(E[:,0], E[:,1], color="blue")  #Se grafica E sin transformar
plt.plot(Ex, Ey, color="Red")  #Se grafica E  transformada

ax.set_xticks(np.arange(−6, 8, 2))  # Fija el rango del eje x
ax.set_yticks(np.arange(−6, 8, 2))  # Fija el rango del eje y
plt.gca().set_aspect('equal', adjustable='box') #Establece misma escala
plt.grid()  #Habilita la cuadrícula
```

En la Figura 1.3 se presenta algunas transformaciones ya ejecutadas (para la rotación se ha usado $\theta = \frac{\pi}{3}$ y $\theta = \frac{3\pi}{2}$).



(a) Rotación $\theta = \frac{\pi}{3}$ (b) Rotación $\theta = \frac{3\pi}{2}$ (c) Reflexión respecto al eje y

(d) Escala $k_x = k_y = 2$ (e) Deformación $h = -1, v = 0$ (f) Traslación $t_x = 3, t_y = 0$

Figura 1.3: Transformaciones afines

1.3 Transformaciones en imágenes

Con la información que tenemos podemos realizar transformaciones directamente a imágenes digitales. Recordemos que todo esto está ya estudiado y que lo que pretendemos es dar ejemplos de aplicaciones del álgebra lineal. Es claro que el lector a partir de esto podrá investigar las formas más óptimas tanto a nivel teórico como a nivel de programación de cómo realizar los ejemplos que aquí se brindan.

Es probable que el lector esté familiarizado con la palabra pixel (picture element), la cual definimos como el elemento gráfico más pequeño de una imagen, que puede tomar solo un color a la vez.

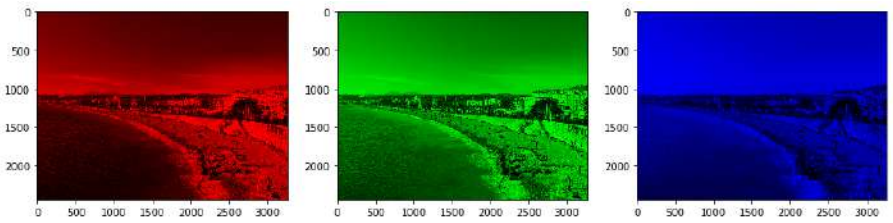


Figura 1.4: Formato RGB

En otras palabras, el pixel es un cuadro pequeño que contiene un color y que forma parte de una imagen. Al número de pixeles de una imagen se le llama resolución (el tamaño físico depende de la resolución que se tome). Ahora, una imagen a color puede ser interpretada como como una matriz 3D de tamaño $px \times py \times 3$, cada entrada se puede referenciar por medio de (i, j, k) donde $i \in \{0, 1, 2, \cdots, px\}$, $j \in \{0, 1, 2, \cdots, py\}$ y $k \in \{0, 1, 2\}$. Cada matriz va a representar un color en el modelo de color RGB, esto es un acrónimo de Red, Green, Blue (existen otros tipos de modelos de color tales como: CMYK, HSL y HSV). La intensidad de cada color va de 0 a 255 (ver Figura 1.4).

La imagen a color es una combinación de las tres matrices, por ejemplo, la imagen dada en la Figura 1.5 de tamaño $p_x \times p_y = 2448 \times 3264$ pixels es la combinación de las imágenes en la Figura 1.4.

Figura 1.5: Imagen de tamaño 2448×3264

Vamos a utilizar de nuevo las bibliotecas numpy y matplotlib. En python podemos cargar una imagen en la matriz "img" con el comando "img=plt.imread(path)", donde "path" es la ubicación de la imagen. Ahora, para acceder a la información de la intensidad de los colores escribiendo "img[i,j]", por ejemplo, para $i = j = 1$ obtenemos "array([108, 147, 214], dtype=uint8)". Esto quiere decir que la intesidad del rojo (R) es 108, la de verde (G) es 147 y la de azul (B) 214. En el caso que se quiere acceder a un color en particular, digamos verde, basta escribir "img[i,j,1]".

```
import numpy as np
import matplotlib.pyplot as plt
path= "tu_directorio//imagen.jpg"
img = plt.imread(path)

a, l, r =img.shape

# Matriz Identidad
Ic = np.eye(3)

#Matriz de Reflexión con respecto al eje x
Refy=np.array([[1., 0, 0],[0, -1., 0],[0., 0., 1.]])

#Matriz de Reflexión con respecto al eje y
Refx=np.array([[-1., 0, 0],[0, 1., 0],[0., 0., 1.]])

# Matriz de Rotación
theta=3*np.pi/4 #Ángulo de rotación
R=np.array([[np.cos(theta), np.sin(theta), 0.],[-np.sin(theta), np.cos(theta), 0.],[0.,
    0., 1.]])

# Matriz de cambio de escala
s=2 #Escalar
S=np.array([[s, 0, 0.], [0., s, 0.],[0., 0., 1.]])

# Matriz de deformación vertical/horizontal
h=-1,v= 2
D=np.array([[1., h, 0.],[v, 1., 0.],[0., 0., 1.]])

# Matriz de traslación
tx=a/2
ty=-l/2
T=np.array([[1., 0, tx], [0, 1., ty], [0., 0., 1.]])
```

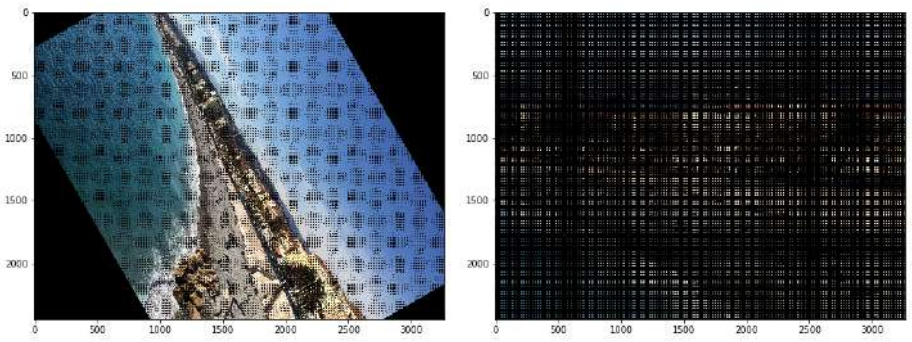
La estrategia a seguir para la transformación de una imagen es la siguiente:

1. Vamos a crear una nueva imagen en negro img N.

2. Luego iteramos sobre cada pixel de la imagen original img.
3. Trasformamos las coordenadas de (i, j) y obtenemos las nuevas coordenadas (i', j').
4. Si (i', j') no son coordenadas enteras tomamos la parte entera y obtenemos $([\![i]\!], [\![j]\!])$.
5. Colocamos lo información en (i, j) de la imagen original en la coordenada $([\![i]\!], [\![j]\!])$, i.e.,

$$\text{img N}([\![i]\!], [\![j]\!]) = \text{img}(i, j).$$

Ahora, recuerde que las entradas de una imagen están conformadas por números enteros. Sin embargo, algunas de las transformaciones pueden generar índices no enteros. Si tomamos la decisión de redondear o utilizar la función parte entera traerá problemas pues puede generar pixeles sin informaición nueva y por lo tanto quedarán en negro (color asignado por defecto a la imagen nueva). Existen también otras situaciones que podrían generar este mismo problema aún cuando las coordenadas transformadas sean enteras. Por ejemplo, si hace un cambio de escala al doble. Es claro que todas los índices generados son enteros, sin embargo, solo se colocará información en las coordenadas de la forma $(2i, 2j)$ y las coordenadas de la forma $(2i + 1, 2j + 1)$ quedarán con la infomarción antigua. Observe que este tipo de problema se presenta en la Figura 1.6.



(a) Rotación $\theta = \frac{\pi}{3}$ (b) Escala $k_x = k_y = 2$

Figura 1.6: Transformaciones a la imagen

El código para realizar esta transformación es el siguiente (adicional al previo):

```
img_N = np.empty((a,l, 3), dtype=np.uint8) #Crea una nueva imagen del mismo tamaño que la
    original, en negro.

for i in range(0,img.shape[0]):  # filas
for j in range(0,img.shape[1]):# columnas
pixel_data = img[i, j, :] # Obtiene la información de color en el pixel (i,j)
input_coords = np.array([i, j, 1]) # Establece las coordenadas (i,j)
i_out, j_out,_ = S.dot(np.array((input_coords))) # Transforma las coordenadas
i_o=np.int(i_out) # Parte entera (o redondeo)
j_o=np.int(j_out) # Parte entera (o redondeo)
if a>i_o >0 and l>j_o>0: # En este caso nos interesa los pixeles que quedan dentro del
   tamaño original
img_N[i_o, j_o, :] = pixel_data  # Se coloca el color en la imagen nueva
```

Este tipo de problemas se puede solucionar fácilmente cambiando un par de líneas del código anterior. En el código anterior, la transformación se realiza de la imagen original a la imagen nueva (negro), basta cambiar el paradigma y pensar hacia la otra "dirección". Es decir,

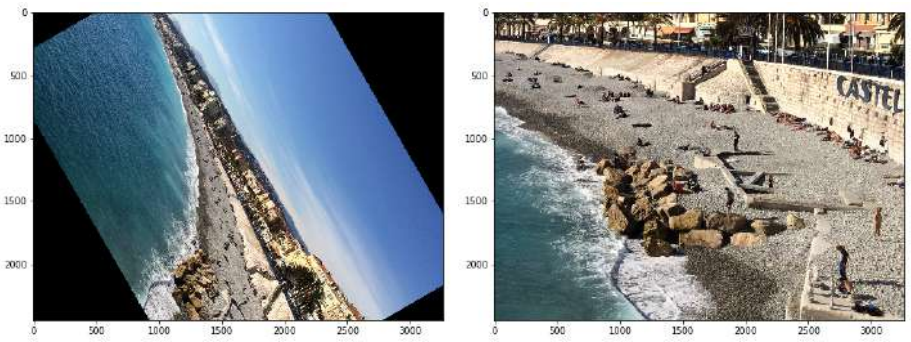
1. Iteramos sobre cada pixel de la imagen nueva img N.

2. Trasformamos las coordenadas de (i, j) y obtenemos las coordenadas (i', j').
3. Si (i', j') no son coordenadas enteras tomamos la parte entera y obtenemos $([\![i]\!], [\![j]\!])$.
4. Colocamos la información en $([\![i]\!], [\![j]\!]$ de la imagen original en la coordenada (i, j), i.e.,

$$\text{img } N(i, j) = \text{img}([\![i]\!], [\![j]\!]).$$

El único detalle se presenta es que la trasformación obtenida va a ser la transformación inversa. Por ejemplo, la rotación es en contra de la manecillas del reloj. El código es el siguiente y los resultados se pueden ver en la Figura 1.7.

```
img_N = np.empty((a,l, 3), dtype=np.uint8) #Crea una nueva imagen del mismo tamaño que la
    original, en negro.

for i in range(0,img_N.shape[0]):
for j in range(0,img_N.shape[1]):
input_coords = np.array([i, j, 1])
i_out, j_out,_ = S.dot(np.array((input_coords)))
i_o=np.int(i_out)
j_o=np.int(j_out)
if a>i_o >0 and l>j_o>0:
pixel_data = img[i_o, j_o, :]
img_N[i, j, :] = pixel_data
```



(a) Rotación $\theta = \frac{\pi}{3}$ (b) Escala $k_x = k_y = 2$

Figura 1.7: Transformaciones a la imagen sin problemas

1.4 Compresión de imágenes

Otro ejemplo interesante que aplica el álgebra lineal es el de compresión de datos. El objetivo de la compresión es disminuir la redundancia de datos para poder almacenar o transmitir dichos datos de manera eficiente. En esta sección vamos a considerar una imagen y vamos a utilizar la descomposición en valores singulares (SVD, del inglés singular values decomposition) para su compresión. De hecho, la descomposición SVD tiene un sinnúmero de aplicaciones tales como: resolución de sistemas lineales, aproximación de rango bajo, procesamiento de señales, compresión de información, entre otros.
La descomposición SVD consiste en factorizar una matriz real (o compleja). Consideremos una matriz M de tamaño $m \times n$ la descomposición SVD encuentra tres matrices: U de tamaño $m \times m$ unitaria $(UU^T = I_m)$, Σ de tamaño $m \times n$ diagonal y V de tamaño $n \times n$ unitaria $(V^TV = I_n)$ tales que

$$M_{m\times n} = U_{m\times m}\Sigma_{m\times n}V^T_{n\times n}.$$

Las matrices M y Σ tiene el mismo rango (dimensión del espacio columna). Ahora si suponemos que $M = U\Sigma V^T$, entonces note que

$$\begin{aligned} MM^T &= (U\Sigma V^T)(U\Sigma V^T)^T \\ &= (U\Sigma V^T)(V^T\Sigma^T U^T) \\ &= U\Sigma\Sigma^T U^T, \end{aligned} \tag{1.1}$$

donde la matriz $\Sigma\Sigma^T$ es una matriz diagonal cuyos elementos de la diagonal son los elementos Σ elevados al cuadrado. Así que, para hallar U y Σ basta trabajar con la matriz MM^T la cual es simétrica. Entonces, para determinar las matrices U, Σ, V seguimos los siguientes pasos (para más detalles ver [2, 7]):

1. Se calcula MM^T la cual es una matriz simétrica.

2. Se resuelve el problema de valores propios $(MM^T)\mathbf{u} = \lambda\mathbf{u}$. Dado que la matriz MM^T es simétrica y por lo tanto definida positiva (siempre tiene solución). Sea el espectro $\sigma = \{\lambda_1, \lambda_2, \cdots, \lambda_m\} \subset \mathbb{R}^+ \cup \{0\}$ ordenado de manera decreciente. Por lo tanto podemos definir

$$\sigma_i = \sqrt{\lambda_i}, \ \forall\, i \in \{1, 2, \cdots, m\}.$$

3. Formamos la matriz Σ. Para esto definimos $r = \min\{m, n\}$ y definimos

$$\Sigma = \begin{pmatrix} D_{r\times r} & O_{r\times n-r} \\ O_{m-r\times r} & O_{m-r\times n-r} \end{pmatrix}$$

   donde

$$D = \text{Diag}[\sigma_1, \sigma_2, \cdots, \sigma_r],$$

   es decir, una matriz diagonal cuyos elementos en la diagonal son los indicados. Y el resto de matrices O son matrices nulas del tamaño indicado.

4. Para formar U utilizamos los correspondientes vectores propios $\mathbf{u}_i$ y los colocamos como columnas, es decir $U = (\mathbf{u}_1\mathbf{u}_2\cdots\mathbf{u}_m)$.

5. Para hallar V, usamos la relación $M = U\Sigma V^T \Leftrightarrow MV = U\Sigma$. Es decir debemos resolver el sistema (para más detalles ver [2, 7])

$$MV = U\Sigma.$$

Utilicemos los pasos anteriores en el siguiente ejemplo.

Ejemplo 1.1

Considere la matriz

$$M = \begin{pmatrix} 1 & 1 \\ 2 & -2 \\ 1 & 1 \end{pmatrix},$$

de donde se tiene que

$$MM^T = \begin{pmatrix} 1 & 1 \\ 2 & -2 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 2 & 1 \\ 1 & -2 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 2 \\ 0 & 8 & 0 \\ 2 & 0 & 2 \end{pmatrix}.$$

Ahora, vamos a resolver el problema de valores y vectores propios $MM^T\mathbf{u} = \lambda\mathbf{u}$, de donde

obtenemos $\lambda_1 = 8, \lambda_2 = 4, \lambda_3 = 0$, con vectores asociados

$$\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}.$$

Al normalizar obtenemos,

$$\mathbf{u}_1 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \mathbf{u}_2 = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \mathbf{u}_3 = \frac{1}{\sqrt{2}}\begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}.$$

Así tenemos que

$$U = \begin{pmatrix} 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ 1 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}$$

y además $\sigma_1 = \sqrt{8}, \sigma_2 = 2$ de donde se tiene que

$$\Sigma = \begin{pmatrix} \sqrt{8} & 0 \\ 0 & 2 \\ 0 & 0 \end{pmatrix}$$

Luego, para obtener V basta resolver el sistema $MV = U\Sigma$, es decir

$$\begin{pmatrix} 1 & 1 \\ 2 & -2 \\ 1 & 1 \end{pmatrix} V = \begin{pmatrix} 0 & \frac{2}{\sqrt{2}} \\ \sqrt{8} & 0 \\ 0 & \frac{2}{\sqrt{2}} \end{pmatrix},$$

de la cual obtenemos

$$V = \begin{pmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}.$$

■

Ahora, observe que podemos escribir la matriz Σ como

$$\Sigma = \Sigma_1 + \Sigma_2 + \cdots + \Sigma_r = \sum_{i=1}^{r} \Sigma_i,$$

donde $\Sigma_i = \text{Diag}[0, \cdots, \sigma_i, \cdots, 0]$ para todo $i \in \{1, 2, \cdots, r\}$. Luego, sabemos que

$$M = U\Sigma V^T = \sum_{i=1}^{r} u_i \sigma_i v_i^T = u_1\sigma_1 v_1^T + u_2\sigma_2 v_2^T + \cdots + u_r\sigma_r v_r^T.$$

Los valores de σ_i más pequeños están relacionados con la parte de la matriz de menos interés y por lo cual aportarán poca información a la matriz. Entonces, podemos aproximar la matriz M con una matriz de menor rango, digamos de rango $p \leq r$. Para esto basta definir

$$M_p = \sum_{i=1}^{p} u_i \sigma_i v_i^T = u_1\sigma_1 v_1^T + u_2\sigma_2 v_2^T + \cdots + u_p\sigma_p v_p^T.$$

De hecho la matriz M_p va a ser la mejor la aproximación de M en la norma 2 (Euclideana)[1] de las matrices de rango menor a p. Es decir

$$\inf_{\{A_{m\times n}:\text{rang}(A)\leq p\}} \|M - A\|_2 = \|M - M_p\|_2 = \sigma_{p+1}.$$

Esto es la clave para poder realizar la compresión, por ejemplo, si consideramos los primeros $p \leq r$ valores propios tendremos

$$M_p = \tilde{U}_{m\times p}\tilde{\Sigma}_{p\times p}\tilde{V}^T_{p\times n}.$$

De donde podremos obtener una aproximación de M utilizando menos información. Solo se necesitaría almacenar $(m \cdot p) + p + (n \cdot p)\nu = p \cdot (m + n + 1)$ datos en lugar de $m \times n$.

Usando python para hallar la descomposición SVD

Vamos a aprovecharnos de que en python podemos obtener la descomposición SVD simplemente llamando la función svd ( ), la cual toma la matriz M y nos devolverá la matriz U, un vector cuyos elementos son los elementos de la diagonal de Σ y la matriz V^T. Por ejemplo, si tomamos los datos de la matriz del ejemplo anterior es decir

$$M = \begin{pmatrix} 1 & 1 \\ 2 & -2 \\ 1 & 1 \end{pmatrix},$$

el código en python es

```
import numpy as np
M=np.array([[1,1],[-2,2],[1,1]]) #Se define la matriz
U, Sigma, VT = svd(M) # Llamamos a la función que hace la descomposición
# imprimimos las matrices para verificar con los datos obtenidos en el ejemplo anterior
print(U)
print(Sigma)
print(VT)
```

Las matrices dadas por python son:

```
[[ 1.11022302e-16 -7.07106781e-01 -7.07106781e-01]
 [ 1.00000000e+00  2.22044605e-16  5.37546367e-17]
 [ 1.11022302e-16 -7.07106781e-01  7.07106781e-01]]

[2.82842712 2.        ]

[[-0.70710678  0.70710678]
 [-0.70710678 -0.70710678]].
```

Ahora veamos esto aplicado a una imagen. Como la imagen está en formato RGB, sabemos que estamos trabajando con tres matrices, a las cuales vamos a tener que factorizar separadamente, es decir, aplicar la descomposición SVD a cada una de ellas. El código que hace la descomposición y devuelve la imagen aproximada es el siguiente.

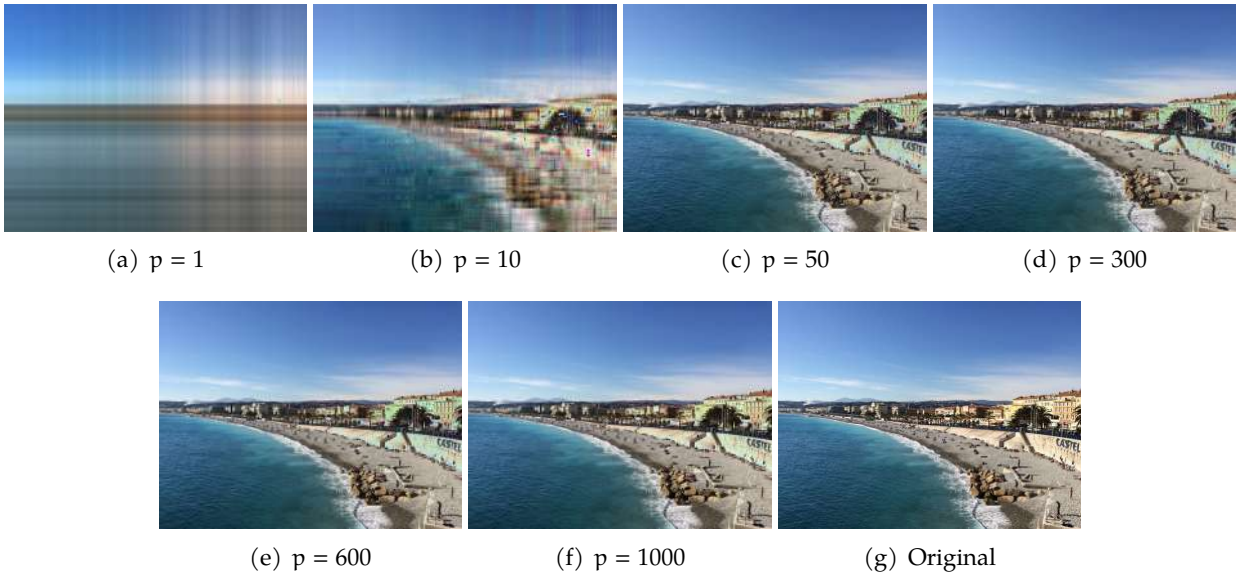
[1] Definimos la norma 2 de una matriz $A_{m\times n}$ como

$$\|A\|_2 = \max_{\|x\|_2=1} \|Ax\|_2.$$

```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image

path1="C://Users//Byron//Desktop//CS1.jpg"# Direcci\'on  de la imagen


img = Image.open(path1)#Abrimos la imagen y la guardamos en img

\obtenemos los tres canales RGB
imgmatrizR = np.array(list(img.getdata(band=0)), float)
imgmatrizG = np.array(list(img.getdata(band=1)), float)
imgmatrizB = np.array(list(img.getdata(band=2)), float)


#Damos forma de matriz al arreglo

imgmatrizR.shape = (img.size[1], img.size[0])
imgmatrizG.shape = (img.size[1], img.size[0])
imgmatrizB.shape = (img.size[1], img.size[0])


#Aplicamos a cada matriz la descomposición SVD

UR, sigmaR, VR = np.linalg.svd(imgmatrizR)
UG, sigmaG, VG = np.linalg.svd(imgmatrizG)
UB, sigmaB, VB = np.linalg.svd(imgmatrizB)

p=500 # n\'umero de valores propios que vamos tomar, en este caso 500

#Se reconstruye las matrices con la informaci\'on comprimida

imgreconstruida = np.matrix(U[:, :60]) * np.diag(sigma[:60]) * np.matrix(V[:60, :])
imgreconstruidaR = np.matrix(UR[:, :p]) * np.diag(sigmaR[:k]) * np.matrix(VR[:p, :])
imgreconstruidaG = np.matrix(UG[:, :p]) * np.diag(sigmaG[:k]) * np.matrix(VG[:p, :])
imgreconstruidaB = np.matrix(UB[:, :p]) * np.diag(sigmaB[:k]) * np.matrix(VB[:p, :])



rgbArray[..., 0] = imgreconstruidaR
rgbArray[..., 1] = imgreconstruidaG
rgbArray[..., 2] = imgreconstruidaB

plt.imshow(rgbArray);
```

Las figuras en 1.8 representan algunas de las imágenes reconstruidas a partir de la descomposición SVD. Definimos p como el número de valores propios que vamos tomar y hemos corrido el programa con $p = 1, 10, 50, 300, 600, 1000$. Esto nos genera seis imágenes que aproximan la original con la ventaja de poder reconstruirla con matrices de menor tamaño, aún para la matriz diagonal Σ basta solo guardar los datos de la diagonal. De este modo, para la imagen de tamaño 2448×3264 (7990272 datos) tenemos que

p	Matriz U	Matriz Σ	Matriz V^T	Total de datos	Porcentaje
1	2448×1	1	1×3264	5713	0,07
10	2448×10	10	10×3264	57130	0,71
50	2448×50	50	50×3264	285650	3,57
300	2448×300	300	300×3264	1713900	21,45
600	2448×600	600	600×3264	3427800	42,90
1000	2448×1000	1000	1000×3264	5713000	71,50



(a) p = 1 (b) p = 10 (c) p = 50 (d) p = 300

(e) p = 600 (f) p = 1000 (g) Original

Figura 1.8: Imágenes reconstruidas con diferentes valores k

1.5 Conclusión

Sin duda alguna, el álgebra lineal ha dado grandes contribuciones al desarrollo de la matemática y de sus aplicaciones. Los ejemplos que hemos dado en este artículo están muy lejos de ser los únicos, pues existen una enorme cantidad de aplicaciones que podrían mencionarse.

Sabemos que en muchas ocasiones existe una una sensación por parte del estudiantado de que si lo que está aprendiendo (la teoría) tiene alguna aplicación práctica o si simplemente se convertirá en conocimiento que no utilizarán en su vida profesional. Y uno de los objetivos principales de este artículo es romper con este paradigma.

Los ejemplos que se han seleccionado para presentar en este artículo se han elegido debido a que son muy visuales y de uso diario, sobre todo en esta era tecnológica. Hemos visto que usando matrices y transformaciones lineales logramos entender cómo podemos visualizar transformaciones afines y sus aplicaciones directas en el procesamiento digital de imágenes. Además, la descomposición SVD nos permite la compresión de imágenes digitales, lo cual es útil para lograr ahorrar memoria de almacenamiento o para que dichas imágenes puedan ser transmitidas de una forma más eficiente.

Esperamos que con este artículo el lector pueda visualizar algunas aplicaciones del álgebra lineal. Pero aún más, esperamos que propicie la motivación del estudio en esta área y la utilización de lenguajes de programación tales como python y, que en lugar de la adquisición de un conocimiento vano en el curso de álgebra lineal este se pueda convertir en una herramienta poderosa.

Bibliografía

[1] Gonzalez, C., Woods, E.(2007) Digital Image Processing. Third Edition. Prentice Hall.

[2] Lay, D., Lay, S., McDonald,J. (2015)*Linear algebra and its applications* .Fifth Edition.Pearson.

[3] McQuistan, A. (2009).Affine Image Transformations in python with Numpy, Pillow and OpenCV. Recuperado de https://stackabuse.com/affine-image-transformations-in-python-with-numpy-pillow-and-opencv/.

[4] Pesco, U.; Bortolossi,J. Matrix and digital images. 2012 . Recuperado de http://dmuw.zum.de/images/6/6d/Matrix-en.pdf.

[5] Schlegel, A. (s.f.).*Image Compression with Singular Value Decomposition.* Recuperado de https://aaronschlegel.me/image-compression-singular-value-decomposition.html.

[6] Verma, S., Krishna, P. (2013). Image Compression and Linear Algebra. Recuperado de https://www.cmi.ac.in/~ksutar/NLA2013/imagecompression.pdf.

[7] Puntanen, S. (2011). Projection matrices, generalized inverse matrices, and singular value decomposition by haruo yanai, kei takeuchi, yoshio takane. *International Statistical Review, 79*(3), 503-504.

[8] Putalapattu, R.(s.f.). *Jupyter, python, Image compression and svd-An interactive exploration.* Recuperado de https://medium.com/@rameshputalapattu/jupyter-python-image-compression-and-svd-an-interactive-exploration-703c953e44f6.