Assiment1

October 16, 2017

1 Trabajo 1 de Procesamiento de Imagenes

Importante: Tener las imagenes requeridas en la misma carpeta para poder ver los resultado

Datos: Alumno: Gabriel Esteban Ramirez Reategui Codigo: 201516512 Profesor: Ricardo González Valenzuela Curso: CC53 - Procesamiento de Imágenes

El objetivo de este trabajo es realizar algunos procesamientos básicos en imágenes digitales usando los conocimientos adquiridos durante la primera mitad del ciclo del curso de procesamiento de imagenes, es por esta razon que se implementara los siguientes algoritmos:

Mosaico

Combinación de Imagenes

Puntillado

Ademas se investigara y/o implementara el algoritmo de Floyd-Steinberg utilizando diferentes maneras de recorrer la imagen.

2 Codigo

Para poder ejecutar el codigo de manera correcta necesitamos una serie de librerias:

numpy(np)
pyplot(plt)
cv2
time

```
In [5]: import numpy as np
        from matplotlib import pyplot as plt
        import cv2
        import time
```

En primer lugar, el proyecto mantiene una serie de funciones que son usadas a lo largo del mismo. Estas funciones son:

2.1 Cargar y mostrar imagenes

Para mostrar las imagenes y luego de manera posterior mostrar el resultado se utiliza las siguientes funciones:

```
In [6]: def cargar(link): #Load picture
            img  = cv2.imread(link) #Lee la imagen
```

```
        img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB) #Make it RGB/La convierte a RGB
        return img

    def show(imagen):
        plt.imshow(imagen) #Cargamos la imagen en el plot
        plt.show() #Mostramos el plot
        pass
```

Ademas, necesitamos trabajar con blanco y negro, por lo tanto creamos una funcion para poder convertirlo a gris, luego necesitaremos su funcion inversa para poder mostrarla y exportarla

```
In [7]: def to_gray(image): #Genera una imagen en escala de grises desde una imagen RGB
            img = np.copy(image)
            img = cv2.cvtColor(img,cv2.COLOR_RGB2GRAY)
            return img


        def back_gray(image): #Convierte la imagen en escala de grises a una imagen RGB(seguira
            img = np.copy(image)
            img = cv2.cvtColor(img,cv2.COLOR_GRAY2RGB)
            return img
```

Con los datos adquiridos ahora si podemos empezar los programas requeridos del Trabajo Nř1

2.2 Actividad 1: Mosaico

Para esta actividad se ha pedido construir un mosaico a partir de una imagen monocromática y generar una nueva imagen en base a la nueva disposición de los bloques, el cual se obtiene aleatoriamente. El nombre del archivo es: Assigment1.py Los archivos requeridos son: demo1.png El archivo resultante sera: resultado1.jpg

2.2.1 Generar la matriz random de ordenamiento

```
In [9]: def get_shuffle(size):
            ans = []
            for i in range(0,size * size):
                ans.append(i)
            np.random.shuffle(ans)
            ans = np.reshape(ans,(size,size))
            return ans
```

En el caso del mosaico requerido se ha pedido un ordenamiento de tamaño 4 x 4, el cual daría un resultado similar al siguiente:

```
In [10]: get_shuffle(4)

Out[10]: array([[13, 14,  2,  8],
                [11,  5,  6, 10],
                [12,  3,  4,  9],
                [ 1,  0,  7, 15]])
```

Recordar que el siguiente no es el mismo orden que la matriz final, pues esta se genera dentro de la función de demostración. Luego de haber cargado la imagen esta tiene que pasar por una serie de funciones, las cuales tendran como resultado final un arreglo de 16(u otro tamaño a decidir) piezas que en total daran la imagen original.

2.2.2 Obtener el tamaño de la partición

Por medio de este algoritmo, se nos permite encontrar el tamaño de cada particion de la imagen

```
In [11]: def get_partition(image,sb): #image and number of boxes
             size = [int(np.size(image,0) /sb),int(np.size(image,1) / sb)]
             return size
```

2.2.3 Creación de una imagen vacia del tamaño de la partición

Usando el siguiente algoritmo se obtiene un arreglo vacio con el tamaño de cada partición

```
In [12]: def create_data(size):
             ar = np.zeros([size[0],size[1]],dtype = np.uint8)
             return ar
```

2.2.4 Conseguir una partición de imagen basada en la posición:

Luego de tener el tamaño, y utilizando la función de creacion de datos:

```
In [14]: def part(size,image,total,part): #retorna una imagen nueva, la cual 1/16 de la imagen o
             img  = create_data(size)
             sy = int(part/ np.sqrt(total))
             sx = int(part - sy * np.sqrt(total))

             for n in range(0,size[0]):
                 for m in range(0,size[1]):
                     img[n][m] = image[n + sy*size[0]][m + sx*size[1]]
             return img
```

2.2.5 Obtener el arreglo de imagenes:

Por medio del siguiente algoritmo se obtiene un arreglo de tamaño 16 el cual contiene cada uno de los sectores de la imagen:

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 9 \end{pmatrix}$$

```
In [21]: def get_pictures(image,ar,size,asize): #an array with the new shuffle
             img = np.copy(image)
             asize = get_partition(image,4)
             pictures = []
             for i in range(0,size*size):
```

```
            temp = part(asize,img,size*size,i)
            pictures.append(temp)

        return pictures
```

2.2.6 Juntar las imagenes basado en el arreglo de posiciones random

Luego unimos la imagen usando la función concatenate() de numpy y retornamos la imagen unida

```
In [23]: def join(pictures,ar): #Asuming all images are the same size:
            rep = np.size(ar,0)
            #for each block
            temp = []
            for i in range(0,rep):
                img = pictures[ar[0][i]]
                for j in range(1,rep):
                    img = np.concatenate((img,pictures[ar[j][i]]),axis=1)
                temp.append(img)
            img = temp[0]
            for i in range(1,rep):
                img = np.concatenate((img,temp[i]),axis=0)
            return img
```

Eso seria toda la implementación del programa de la actividad 1.

2.2.7 Demostración

La siguiente función es usada para mostrar el resultado requerido de manera rapida

```
In [37]: def demo():
            stime = time.time() #Inicio el contador de tiempo
            otest = cargar("demo1.png")
            btest = to_gray(otest)
            ar = get_shuffle(4)
            size = get_partition(btest,4)
            pictures = get_pictures(btest,ar,4,size)
            ans = join(pictures,ar)
            ans = back_gray(ans)
            cv2.imwrite("resultado1.jpg",ans)
            show(ans)
            ftime = time.time() - stime #Obtengo el tiempo demorado
            print("El tiempo de ejecucion es:",round(ftime,3),"segundos") #Muestro el tiempo qu
            return
```

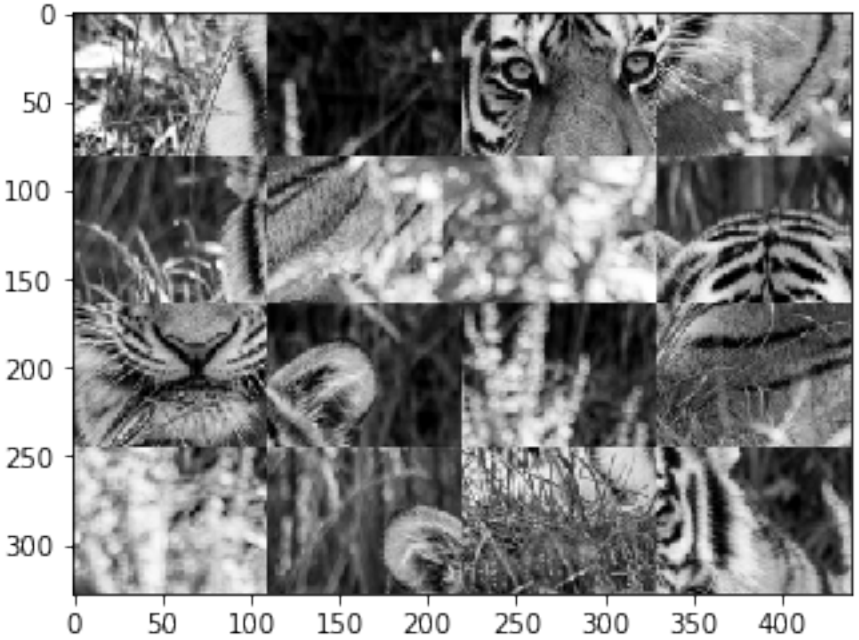
2.2.8 Resultados:

Imagen a mostrar:

Resultado:

title

In [38]: demo()



El tiempo de ejecucion es: 0.307 segundos

2.3 Actividad 2: Combinación de Imágenes

Para esta actividad se ha pedido combinar dos imágenes del mismo tamaño por medio de la sumatoria de sus niveles de gris. Los porcentajes se dan como parametro de la función El nombre del archivo es: Assigment2.py Los archivos requeridos son: demo2a.jpg y demo2b.jpg El archivo resultante sera: resultado2.jpg

2.3.1 Juntando imagenes del mismo tamaño:

En caso las imagenes sean del mismo tamaño se utilizara el siguiente algoritmo, recibiendo como parametros las dos imagenes en escala de grises.

```
In [25]: def join_same_size(image1,image2,a,b): #both images and the % of each one
             sx, sy = np.size(image1,0),np.size(image1,1)
             img = np.copy(image1)
             for i in range(0,sx):
                 for j in range(0,sy):
                     img[i][j] = max(0,min(255,image1[i][j]*a + image2[i][j]*b))

             return img
```

2.3.2 Selector de opciones:

El presente codigo funciona en las situaciones cuando:
Ambas imagenes son del mismo tamaño
una imagen es mas grande que la otra

```
In [31]: def join2(image1,image2,a,b):
             s1x,s1y = np.size(image1,0),np.size(image1,1)
             s2x,s2y = np.size(image2,0),np.size(image2,1)
             print(s1x,s2x)
             if(s1x == s2x and s1y == s2y):
                 print(1)
                 img = join_same_size(image2,image1,b,a)
             if(s1x > s2x and s1y > s2y):
                 print(2)
                 img = join_same_size(image2,image1,b,a)
             if(s1x < s2x and s1y < s2y):
                 print(3)
                 img = join_same_size(image1,image2,a,b)
             return img
```

2.3.3 Demostración

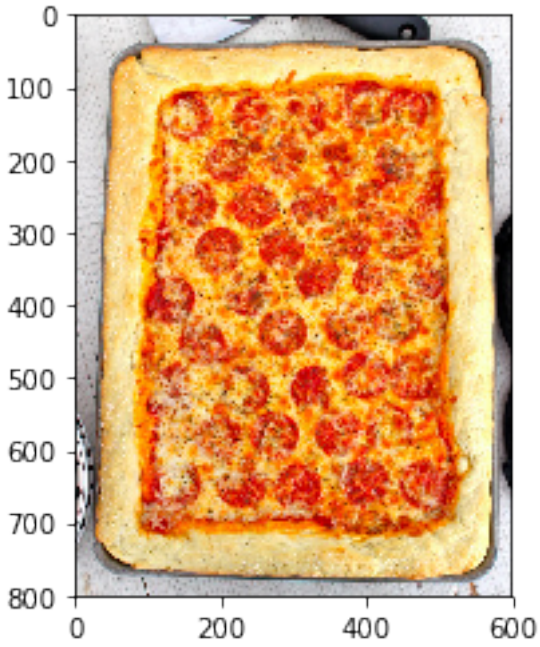
La siguiente función es usada para mostrar el resultado requerido de manera rapida

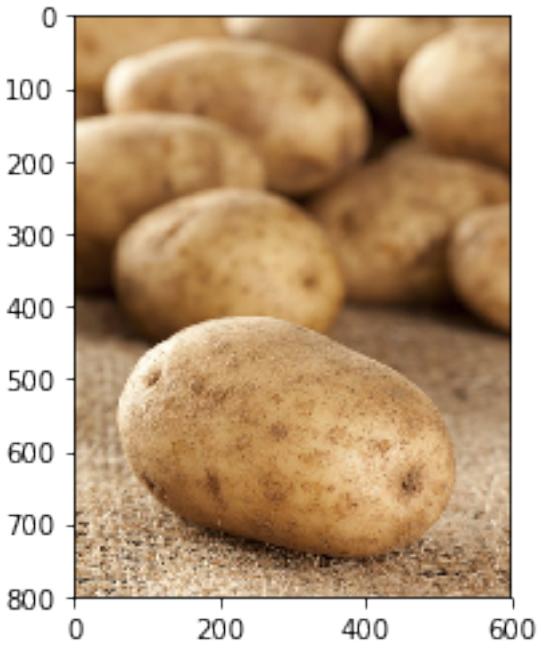
```
In [35]: def demo2():
             stime = time.time()
             img1 = cargar("demo2a.jpg")
```

```
    img2 = cargar("demo2b.jpg")
    img1 = to_gray(img1)
    img2 = to_gray(img2)
    img = join2(img1,img2,0.5,0.5) #Se puede cambiar los valores
    img = back_gray(img)
    show(img)
    cv2.imwrite("resultado2.png",img)
    ftime = time.time() - stime #Obtengo el tiempo demorado
    print("El tiempo de ejecucion es:",round(ftime,3),"segundos") #Muestro el tiempo qu
    return
```

Imagenes utilizadas:

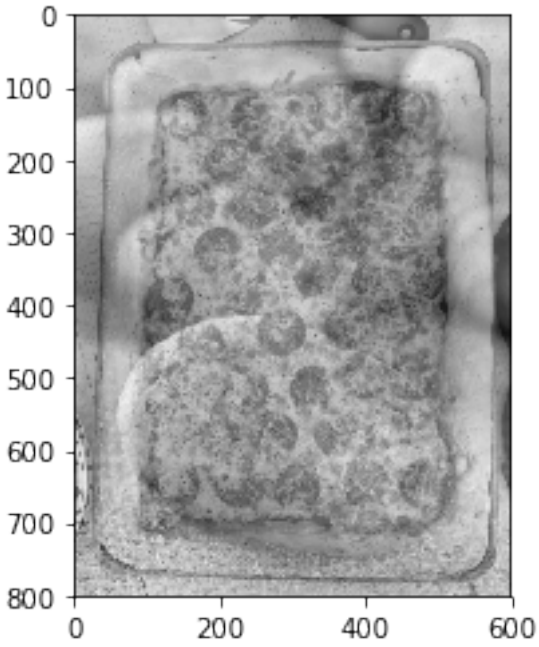
```
In [41]: show(cargar("demo2a.jpg"))
         show(cargar("demo2b.jpg"))
```





In [36]: demo2()

800 800
1



```
El tiempo de ejecucion es: 2.714 segundos
```

2.4 Actividad 3: Puntillado

Para esta actividad se ha pedido convertir una imagen de escala de grises a blanco y negro utilizando el concepto de puntillado, remplazando cada pixel por una matriz 3x3. En primer lugar, se debe normalizar los valores y luego, dependiendo del valor obtenido, se debe implementar la informacion de la siguiente manera:

$$\begin{pmatrix} 6 & 8 & 4 \\ 1 & 0 & 3 \\ 5 & 2 & 7 \end{pmatrix}$$

El nombre del archivo es: Assigment3.py Los archivos requeridos son: demo3.jpg El archivo resultante sera: resultado3.jpg

2.4.1 Datos necesarios:

Se necesita una serie de matrices para cada uno de los posibles valores del pixel normalizado:

```
In [32]: #Matrix
         m0 = [[0,0,0],
               [0,0,0],
               [0,0,0]]
         m1 = [[0, 0, 0],
               [0,255,0],
               [0, 0 ,0]]
         m2 = [[0  , 0 ,0],
               [255,255,0],
               [0  , 0 ,0]]
         m3 = [[0  , 0 ,0],
               [255,255,0],
               [0  ,255,0]]
         m4 = [[0  , 0 , 0 ],
               [255,255,255],
               [0  ,255, 0 ]]
         m5 = [[0  , 0 ,255],
               [255,255,255],
               [0  ,255, 0 ]]
         m6 = [[0  , 0 ,255],
               [255,255,255],
               [255,255, 0 ]]
         m7 = [[255, 0 ,255],
               [255,255,255],
               [255,255, 0 ]]
         m8 = [[255, 0 ,255],
```

```
            [255,255,255],
            [255,255,255]]
m9 = [[255,255,255],
      [255,255,255],
      [255,255,255]]
```

Luego se crea un diccionario con los mismos:

```
In [33]: mat_dic = np.array([m0,m1,m2,m3,m4,m5,m6,m7,m8,m9],dtype = np.uint8) #Dictionary of pix
```

2.4.2 Normalizado de Datos:

Tambien es necesario obtener los valores de cada pixel(0-255) y luego convertirlo a una normalizacion de 0-9, usando las siguientes funciones:

```
In [34]: def normalize(value): #Normalizacion del pixel
             ans = int(value/(256/9))
             return ans

         def normalize_matrix(): #Diccionario del normalizado
             ans = np.zeros(256,dtype = np.uint8)
             for i in range(0,np.size(ans)):
                 ans[i] = normalize(i)
             return ans
```

2.4.3 Creación de la nueva matriz

Usando la informacion anterior, se genera una nueva imagen(del triple de tamaño de la original) ahora unicamente confirmada por blancos y negros:

```
In [39]: def point_process(image):
             stime = time.time()
             sx,sy = np.size(image,0),np.size(image,1)
             k = normalize_matrix();
             temp = []
             for i in range(0,sx):
                 line = mat_dic[k[image[i][0]] - 1]
                 for j in range(1,sy):

                     line = np.concatenate((line,mat_dic[k[image[i][j]]]),axis=1)
                 temp.append(line)
             img = temp[0]
             for i in range(1,np.size(temp,0)):
                 img = np.concatenate((img,temp[i]),axis=0)
             ftime = time.time() - stime
             print(int(ftime))
             return img
```

2.4.4 Resultados:

Imagen utilizada:

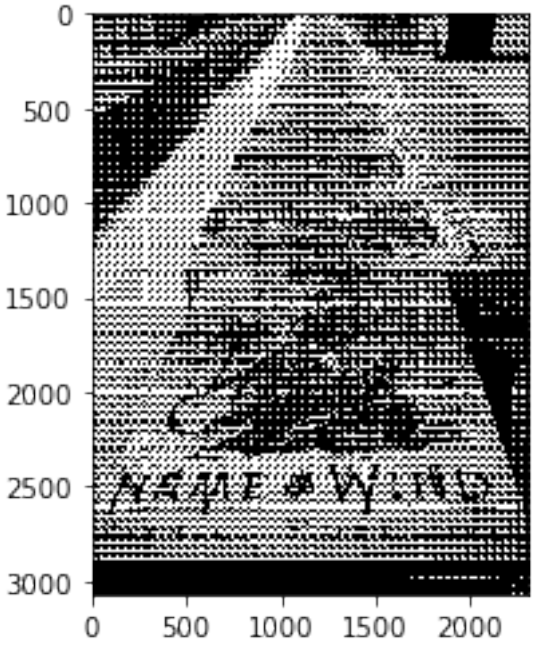
```
In [42]: show(cargar("demo3.jpg"))
```

La siguiente función es usada para mostrar el resultado requerido de manera rapida

```
In [50]: def demo3():
             stime = time.time()
             trial = cargar("demo3.jpg")
             ktrial = to_gray(trial)
             nasd = point_process(ktrial)
             final = back_gray(nasd)
             show(final)
             cv2.imwrite("resultado3.jpg",final)
             ftime = time.time() - stime
             print("El tiempo de ejecucion es:",round(ftime,3),"segundos") #Muestro el tiempo qu
             pass

In [51]: demo3()

3
```




```
El tiempo de ejecucion es: 4.15 segundos
```

El resultado se ve puede apreciar mejor en el archivo creado resultado3.jpg

2.5 Actividad Extra: Floyd-Steinberg

Para esta actividad se ha pedido construir una imagen en blanco y negro usando como input una imagen en escala de grises, para esot se utilizara el algoritmo de Floyd-Steinberg. El nombre del archivo es: AssigmentExtras.py Los archivos requeridos son: demo1.png El archivo resultante sera: resultado3f.jpg y resultado3f2.jpg

2.5.1 Un poco de Teoria:

El algoritmo de Floyd-Steinberg se creo en 1976 por Robert Floyd y Louis Steinberg el cual utiliza el concepto de difusion de errores para minimizar la cantidad de bits utilizada en un pixel.

2.5.2 £Como el tipo de recorrido afecta el codigo?

Es importante aclarar que en este algoritmo, el cual modifica continuamente la imagen y los pixeles previamente recorridos, la manera de recorrer la imagen varia:

2.5.3 Raster:

Este tipo de recorrido es el más usual, el cual es de Izquierda a derecha y de arriba abajo, seguiendo la matriz, esta se recorre de la siguientes maneras:

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 9 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 8 \end{pmatrix}$$

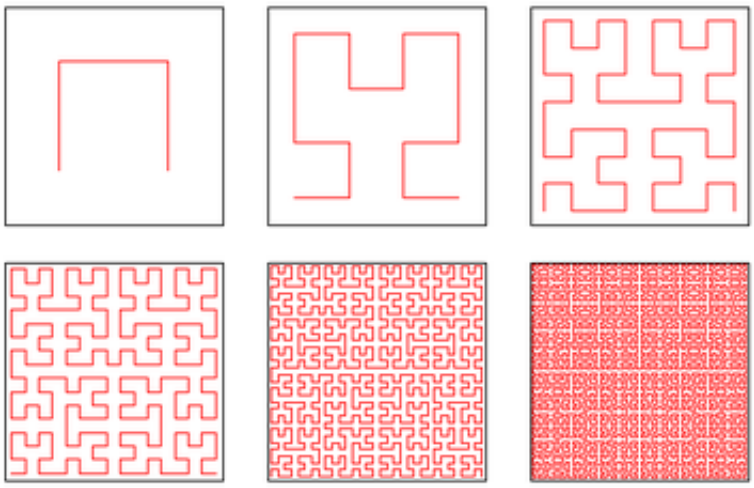
2.5.4 Serpenteado:

Este tipo de recorrido nos dara un resultado mas uniforme, el cual recorre la matriz como si fuera una serpiente, yendo de manera inversa al recorrido previo en la siguiente linea de la imagen para explicarlo mejor, esta se recorre de la siguientes maneras:

$$\begin{pmatrix} 0 & 1 & 2 \\ 5 & 4 & 3 \\ 6 & 7 & 9 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 6 & 7 \\ 2 & 5 & 8 \\ 3 & 4 & 8 \end{pmatrix}$$

Hilbert: Uno de los conceptos matematicos mas utilizados a la hora de llenar espacios, este permite recorrer toda la imagen de manera adecuada siguiendo un patron similar al siguiente:



Gracias a esto se puede llenar cualquier imagen, ya que esta es una matriz. ### Codigo: Para poder efectual esta transformación se ha creado un codigo que permita utilizar:

Raster
Serpenteado

```
In [53]: def floyd_steinberg(image,tipo): #1 normal, 2 serpent
             img = np.copy(image)
```

```
            sx,sy = np.size(image,0) - 1,np.size(image,1) - 1
            ran = [0,sy-1]
            for i in range(0,sx):
                for j in range(ran[0],ran[1]):
                    temp_pixel = img[i][j]
                    npixel = round(temp_pixel/255)*255 #Threshold
                    img[i][j] = npixel
                    error = temp_pixel - npixel
                    img[i+1][j] = min(255,img[i + 1][j] + error * 7/16)
                    img[i-1][j+1] = min(255,img[i-1][j+1] + error * 3/16)
                    img[i][j+1] = min(255,img[i][j+1] + error * 5 / 16)
                    img[i+1][j+1] = min(255,img[i+1][j+1] + error / 16)
                if(tipo == 2): ran.reverse()
            return img
```

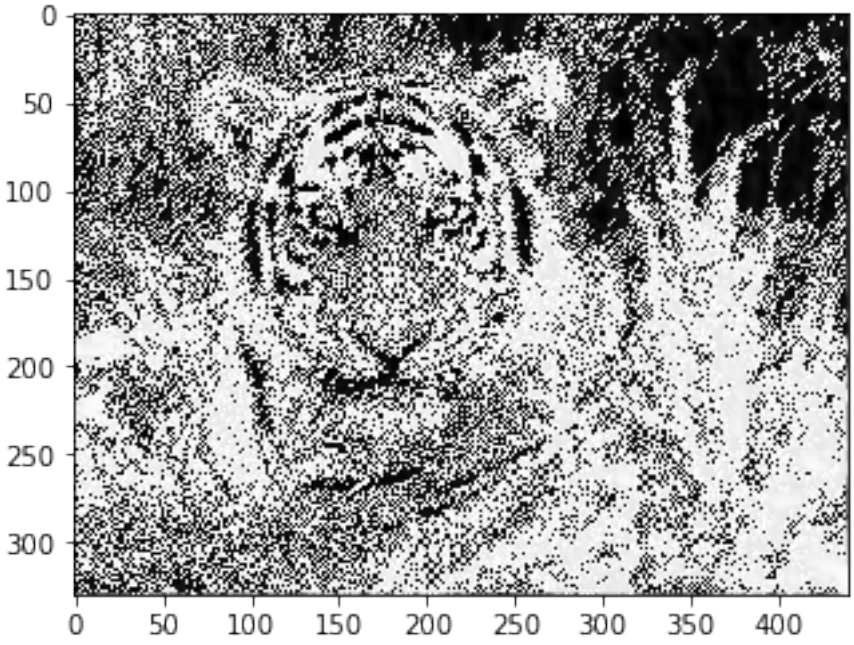
2.5.5 Resultados

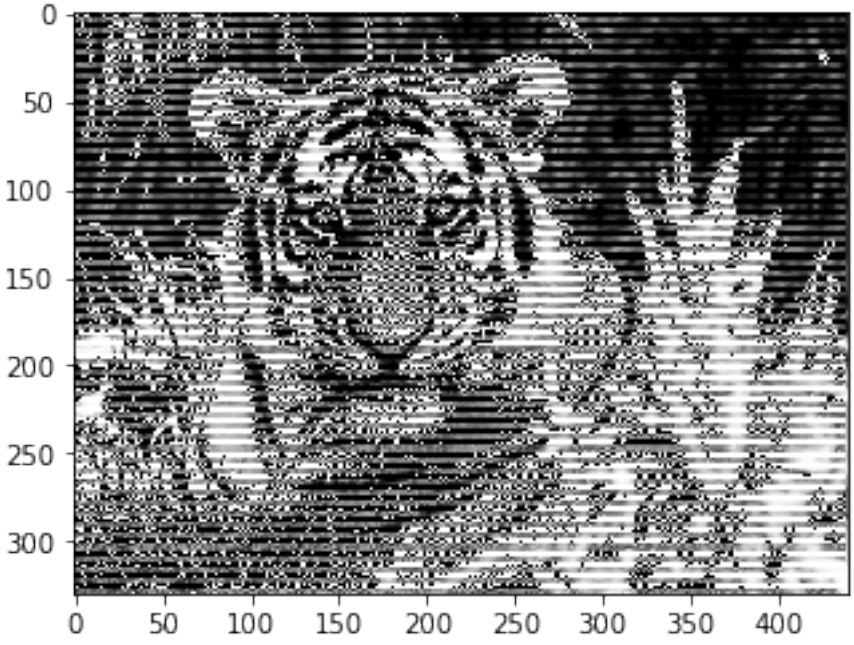
A continuacion se presenta un codigo que genera los resultados de manera facil:

```
In [56]: def demo3f():
             stime = time.time()
             demo = cargar("demo1.png")
             kdemo = to_gray(demo)
             r_floyd = floyd_steinberg(kdemo,1)
             r_final = back_gray(r_floyd)
             ftime = time.time() - stime
             print("El tiempo de ejecucion en raster es:",round(ftime,3),"segundos")
             stime = time.time()
             s_floyd = floyd_steinberg(kdemo,2)
             s_final = back_gray(s_floyd)
             ftime = time.time() - stime
             print("El tiempo de ejecucion en serpenteado es:",round(ftime,3),"segundos")
             show(r_final)
             show(s_final)
             cv2.imwrite("result3f.jpg",r_final)
             cv2.imwrite("result3f2.jpg",s_final)
             pass

In [57]: demo3f()

El tiempo de ejecucion en raster es: 3.17 segundos
El tiempo de ejecucion en serpenteado es: 1.627 segundos
```





2.6 Bibliografia:

Las siguientes fuentes fueron consultadas para la implementación del proyecto: Para entender el concepto de Floyd-Steinberg Tanner Helland (dot) com. (2017). Image Dithering: Eleven Algo-

rithms and Source Code. [online] Available at: http://www.tannerhelland.com/4660/dithering-eleven-algorithms-source-code/ [Accessed 15 Oct. 2017]. Para entender el concepto de Hilbert: Velho, L. and Gomes, J. (1991). Digital halftoning with space filling curves. Proceedings of the 18th annual conference on Computer graphics and interactive techniques - SIGGRAPH '91. Para conceptos de la libreria numpy y referencias: Docs.scipy.org. (2017). NumPy Reference — NumPy v1.13 Manual. [online] Available at: https://docs.scipy.org/doc/numpy-1.13.0/reference/index.html [Accessed 16 Oct. 2017].

2.7 Github

El codigo previamente presentado se encuentra disponible en el repositorio publico de Github con dirección:

https://github.com/GaEsRaRe/PDI/tree/master/Assigment%20TB1