

# Trabajo Práctico 1

## Procesamiento de Imágenes y Visión por Computadora

Matías Cisnero

11 de agosto de 2025

### Consigna 1:

Implementar la función de potencia  $\gamma$ ,  $0 < \gamma < 2$  y  $\gamma \neq 1$ .



## Definición

$$T(r) = cr^\gamma, \quad 0 < \gamma < 2, \gamma \neq 1$$

Donde  $c$  es una constante apropiada, de tal forma que  $T(255) = 255$ .

## Constante $c$

$$c = 255^{1-\gamma}$$

Y  $r$  los niveles de gris de una imagen  $f$ .



```
def _aplicar_gamma(self, imagen, gamma):  
    # Convierto en un array de (m, n, 3)  
    imagen_np = np.array(imagen)  
  
    c = (255)**(1-gamma)  
    resultado_np = c*(imagen_np**gamma)  
  
    self.imagen_procesada = Image.fromarray(resultado_np.  
        astype('uint8'))
```

(\*) Se puede operar directamente con la imagen gracias al Broadcasting de Numpy (ver [1]).

Al aplicar la transformación con  $\gamma = 1,5$  en la Figura 1 obtenemos la imagen de la Figura 2:



Figura 1: Imagen original

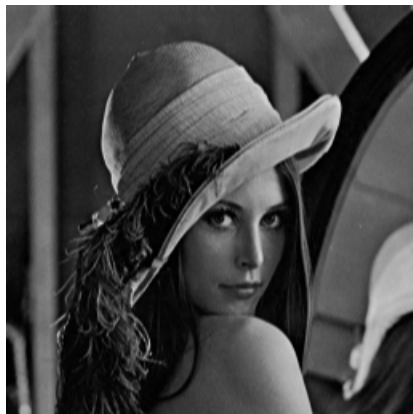


Figura 2: Imagen modificada

## Consigna 2:

Implementar una función que devuelva el negativo de una imagen.



## Definición

$$T(r) = 255 - r$$



```
def _aplicar_negativo(self):  
    # Convierto en un array de (m, n, 3)  
    imagen_np = np.array(self.imagen_procesada)  
  
    resultado_np = 255 - imagen_np  
    self.imagen_procesada = Image.fromarray(resultado_np.  
        astype('uint8'))
```

(\*) Nuevamente se puede operar directamente gracias al Broadcasting de Numpy (ver [1]).



Al aplicar el negativo en la Figura 3 obtenemos la imagen de la Figura 4:



Figura 3: Imagen original



Figura 4: Imagen en negativo

### Consigna 3:

Implementar una función que devuelva el histograma de niveles de gris de una imagen.



## Definición

$$h_i = \frac{n_i}{N.M}, i = 0, \dots, 255$$

Donde  $n_i$  : cantidad de ocurrencias del nivel de gris  $i$  dentro de la imagen.  $N.M$  : cantidad total de píxeles de la imagen,  $M$  filas y  $N$  columnas.

- **Consigna 4:**

Implementar una función que aplique un umbral a una imagen, devolviendo una imagen binaria. El umbral debe ser un parámetro de entrada.



## Definición

Dado un umbral  $u \in [0, \dots, 255]$

$$T(r) = \begin{cases} 255, & \text{si } r \geq u \\ 0, & \text{si } r < u \end{cases}$$



```
def _aplicar_umbralizacion(self, imagen, umbral):  
    # Convierto en un array de (m, n, 3)  
    imagen_np = np.array(imagen)  
  
    resultado_np = np.where(imagen_np >= umbral, 255, 0)  
  
    self.imagen_procesada = Image.fromarray(resultado_np.  
        astype('uint8'))
```

(\*) np.where es una forma vectorizada de hacer un if-else para los píxeles de la imagen (ver [2]).

(\*) El umbral se asigna mediante un Scale (slider) de tkinter.

Al aplicar la umbralización con  $u = 128$  en la Figura 5 obtenemos la imagen de la Figura 6:

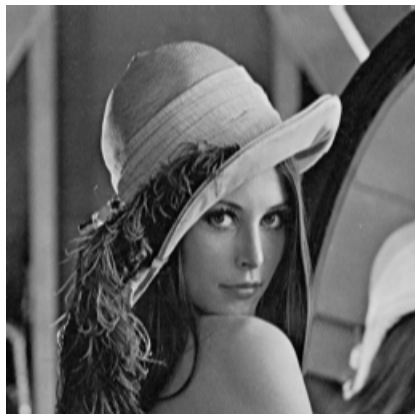


Figura 5: Imagen original



Figura 6: Imagen en negativo

### Consigna 5:

Implementar una función que realice la ecualización del histograma para mejorar la imagen.





## Definición

$$s_k = T(r_k) = \sum_{i=0}^k \frac{n_i}{n}$$

donde:

- $r_k$  es el  $k$ -ésimo nivel de gris dentro del intervalo  $[0, 255]$ .
- $n_i$ ,  $i = 0, \dots, 255$  es el número de pixels de la imagen con nivel de gris  $r_i$ ,  $i = 0, \dots, 255$ .
- $n$  es el número total de pixels de la imagen.
- $\frac{n_i}{n}$ ,  $i = 0, \dots, 255$  es la frecuencia relativa del  $i$ -ésimo nivel de gris.

Actualmente  $s_{min} \leq s_k \leq 1$  y queremos que  $s_k \in [0, 255]$ , para eso aplicamos la siguiente transformación para discretizar los valores:

$$\hat{s}_k = 255 * \left\lceil \frac{s_k - s_{min}}{1 - s_{min}} \right\rceil$$



```
def _aplicar_ecualizacion_histograma(self):
    imagen_np_gris = np.array(self.imagen_procesada.convert(
        'L')) # Array de la forma (m. n).
    datos_gris = imagen_np_gris.flatten()

    n_r = np.bincount(datos_gris, minlength=256) # Freq abs(
        ni)
    NM = datos_gris.size # Pixels totales(n)
    h_r = n_r / NM # Freq relativa(ni/n)

    sk = np.zeros(256) # Hacemos la suma acumulada
    for k in range(len(sk)):
        sk[k] = np.sum(h_r[0:k+1])

    sk_sombrero = self._escalar_255(sk) # Discretizamos
    resultado_np = sk_sombrero[imagen_np_gris]

    self.imagen_procesada = Image.fromarray(resultado_np.
        astype('uint8')).convert('RGB')
```



Una forma más optimizada de hacerlo aprovechando los métodos de Numpy.

```
def _aplicar_ecualizacion_histograma(self):
    imagen_np_gris = np.array(self.imagen_procesada.convert(
        'L'))
    datos_gris = imagen_np_gris.flatten()

    n_r = np.bincount(datos_gris, minlength=256) # Freq abs
    h_r = n_r / np.sum(n_r) # Freq relativa

    sk = np.cumsum(h_r)
    sk_sombrero = self._escalar_255(sk) # Discretizamos

    resultado_np = sk_sombrero[imagen_np_gris]
    self.imagen_procesada = Image.fromarray(resultado_np.
        astype('uint8')).convert('RGB')
```

(\*) np.cumsum hace la suma acumulada desde el principio hasta la posición del valor (ver [5]). np.bincount cuenta la cantidad de veces que aparece cada valor con índice igual a nivel de pixel y valor igual a frecuencia (ver [3]).



Al realizar la ecualización del histograma de la Figura 7 obtenemos el de la Figura 8:

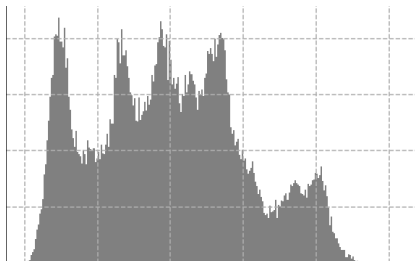


Figura 7: Histograma original

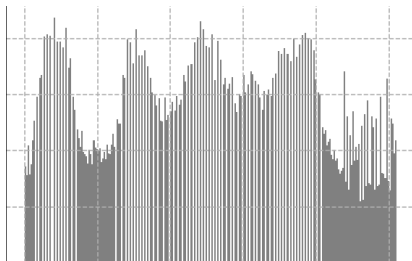


Figura 8: Histograma ecualizado

La ecualización de la imagen se puede apreciar al ver como al ecualizar la Figura 9 se obtiene la Figura 10:



Figura 9: Imagen original



Figura 10: Imagen ecualizada

### Consigna 6:

Aplicar la ecualización del histograma por segunda vez a la misma imagen. Observar el resultado y dar una explicación de lo sucedido.

## Consigna 7:

Implementar generadores de números aleatorios con las siguientes distribuciones:

- 1 Gaussianiana con desviación estándar  $\sigma$  y valor medio  $\mu$ .
- 2 Rayleigh con parámetro  $\xi$ .
- 3 Exponencial con parámetro  $\lambda$ .

Luego graficar los histogramas correspondientes. Puede utilizarse una librería que genere números aleatorios. Los parámetros del generador deben ser parámetros de entrada.



Para generar los números aleatorios (1000) con las distribuciones correspondientes se realizó la siguiente función:

```
def _generar_vector_ruido(self, distribucion, intensidad,
    cantidad):
    # distribucion = np.random.normal, np.random.rayleigh,
    np.random.exponential
    vector_aleatorio = distribucion(scale=intensidad, size=(
        cantidad, 1))

    return vector_aleatorio
```

(\*) en `np.random.normal` el parámetro "loc" predeterminadamente es 0.





Los histogramas obtenidos fueron:

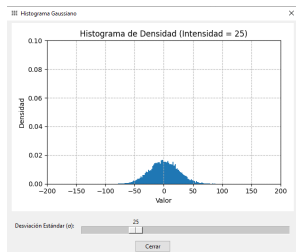


Figura 11: Dist gaussiana

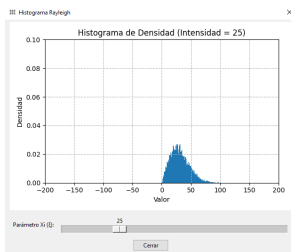


Figura 12: Dist rayleigh

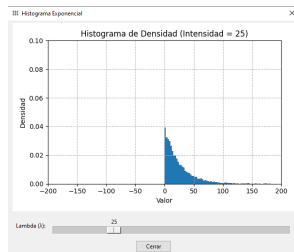


Figura 13: Dist exp

(\*) Es conveniente probarlo en el programa para mayor interactividad.

## Consigna 8:

Utilizando los generadores del punto anterior, implementar los siguientes puntos para agregar ruido a una imagen.

- 1 Contaminar un porcentaje de una imagen con ruido Gaussiano aditivo.
- 2 Contaminar un porcentaje de una imagen con ruido Rayleigh multiplicativo.
- 3 Contaminar un porcentaje de una imagen con ruido exponencial multiplicativo.

El porcentaje de contaminación y los parámetros del generador deben ser parámetros de entrada.



Dada una imagen  $I$  de dimensión  $m \times n$ . La estrategia para contaminar  $I$  y obtener  $I_C$  la imagen contaminada es:

## Estrategia

- 1 Definir el porcentaje de contaminación  $d$  (se toma como parámetro).
- 2 Elegir aleatoriamente  $d$  píxeles de la imagen y construir el conjunto  $D = \{(i, j)\}$  el conjunto de píxeles seleccionados.
- 3 Definir el parámetro de escala  $\lambda$ ,  $\mu$  o  $\xi$  (se toma como parámetro).
- 4 Generar  $d$  valores aleatorios con la distribución correspondiente (se toma como vector\_ruido).
- 5 Luego en los índices  $D$  de la imagen  $I$  multiplicar o sumar el escalar  $k$  del vector ruido al valor del pixel según corresponda.

La ecualización de la imagen se puede apreciar al ver como al ecualizar la Figura 14 se obtiene la Figura 15:

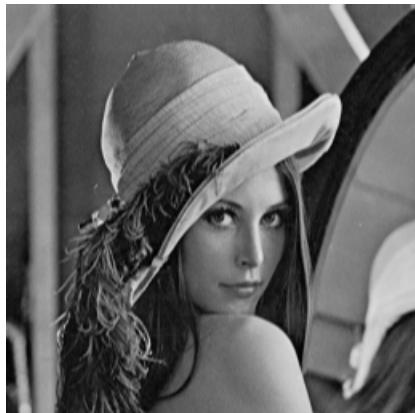


Figura 14: Imagen original



Figura 15: Imagen ecualizada



Para aplicar el ruido en la imagen utilizaremos la siguiente función:

```
def _aplicar_ruido(self, imagen, tipo, vector_ruido, d):
    imagen_np = np.array(imagen).astype(float)
    m, n, _ = imagen_np.shape

    # Cantidad de píxeles contaminados
    num_contaminados = int((d * (m * n)) / 100)
    # Indices (i, j) de los píxeles contaminados
    D = np.unravel_index(np.random.choice(m * n,
        num_contaminados, replace=False), (m, n))

    if tipo == "Aditivo": imagen_np[D] += vector_ruido
    elif tipo == "Multiplicativo": imagen_np[D] *=
        vector_ruido

    resultado_np = self._escalar_255(imagen_np)
    self.imagen_procesada = Image.fromarray(resultado_np)
```

(\*) np.random.choice (ver [7]). np.unravel\_index (ver [6]).

### Consigna 9:

Implementar un generador de ruido Sal y Pimienta de densidad variable, aplicarlo una imagen.



## Definición

Dada una imagen  $I$ .

- Elegir  $p \in (0, 0,5)$
- Recorrer cada pixel  $(i, j)$  de  $I$ .
  - Tomar  $x \sim U(0, 1)$
  - Si  $x \leq p$  entonces  $I(i, j) = 0$
  - Si  $x > 1 - p$  entonces  $I(i, j) = 255$



```
def _aplicar_ruido_sal_y_pimienta(self, imagen, p):  
    imagen_np = np.array(imagen.convert('RGB'))  
  
    m, n, _ = imagen_np.shape  
  
    for i in range(m):  
        for j in range(n):  
            x = np.random.rand()  
            if x <= p:  
                imagen_np[i, j, :] = 0 # pimienta (negro)  
            elif x > (1-p):  
                imagen_np[i, j, :] = 255 # sal (blanco)  
  
    self.imagen_procesada = Image.fromarray(imagen_np)
```

(\*) Aunque en el slider uno pone la probabilidad, a la función le llega  $p = (\text{porcentaje} / 2) / 100$ .



Al aplicar el ruido Sal y Pimienta a una imagen con probabilidad = 10 en la Figura 16 obtenemos la imagen de la Figura 17:



Figura 16: Imagen original



Figura 17: Imagen con ruido

## Consigna 10:

Implementar una ventana deslizante que pueda aplicarse a una imagen con máscaras de tamaño variable, cuadrada y aplicar a una imagen las siguientes máscaras:

- 1 Filtro de la media.
- 2 Filtro de la mediana.
- 3 Filtro de la mediana ponderada.
- 4 Filtro de Gauss para diferentes valores de  $\sigma$  y  $\mu = 0$ .
- 5 Realce de Bordes.

## Consigna 11:

Repetir el punto anterior aplicándolo a las mismas imágenes contaminadas con:

- 1 Ruido Gaussiano aditivo para varios de  $\sigma$  y  $\mu = 0$ .
- 2 Ruido Rayleigh multiplicativo para varios valores de  $\xi$ .

## Consigna 12:

Contaminar con ruido Sal y Pimienta con diferentes densidades y aplicarle el filtro de la media y de la mediana. Observar los resultados.



NumPy Documentation: Broadcasting.

<https://numpy.org/doc/stable/user/basics.broadcasting.html>



NumPy Documentation: `numpy.where`.

<https://numpy.org/doc/stable/reference/generated/numpy.where.html>



NumPy Documentation: `numpy.bincount`.

<https://numpy.org/doc/stable/reference/generated/numpy.bincount.html>



NumPy Documentation: `numpy.sum`.

<https://numpy.org/doc/stable/reference/generated/numpy.sum.html>



NumPy Documentation: `numpy.cumsum`.

<https://numpy.org/doc/stable/reference/generated/numpy.cumsum.html>



NumPy Documentation: `numpy.unravel_index`.

[https://numpy.org/doc/stable/reference/generated/numpy.unravel\\_index.html](https://numpy.org/doc/stable/reference/generated/numpy.unravel_index.html)



NumPy Documentation: `numpy.random.choice`.

<https://numpy.org/doc/stable/reference/random/generated/numpy.random.choice.html>



NumPy Documentation: `numpy.pad`.

<https://numpy.org/doc/stable/reference/generated/numpy.pad.html>

# ¡Gracias!

