UNAHUR
UNIVERSIDAD NACIONAL DE HURLINGHAM

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 $\gamma \neq 1$.

Transformación Gamma γ

Definición

$$T(r) = c.r^{\gamma}, \quad 0 < \gamma < 2, \ \gamma \neq 1$$

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.

Código

```
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]).

Resultados

Al aplicar **la transformación gamma** con $\gamma = 1{,}5$ sobre la imagen de la Figura 1, se obtuvo la versión modificada que aparece en la Figura 2.

Figura 1: Imagen original

Figura 2: Imagen modificada

Consigna 2:

Implementar una función que devuelva el negativo de una imagen.

Negativo

Definición

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

Código

```
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]).

Resultados

Al aplicar **la transformación de negativo** sobre la imagen de la Figura 3, se obtuvo el resultado que se observa en 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.

Histograma

Definición

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

Donde

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

Resultados

Al graficar **el histograma de niveles de gris** de la imagen de la Figura 5, se obtuvo la representación que se muestra en la Figura 6.

Figura 5: Imagen original

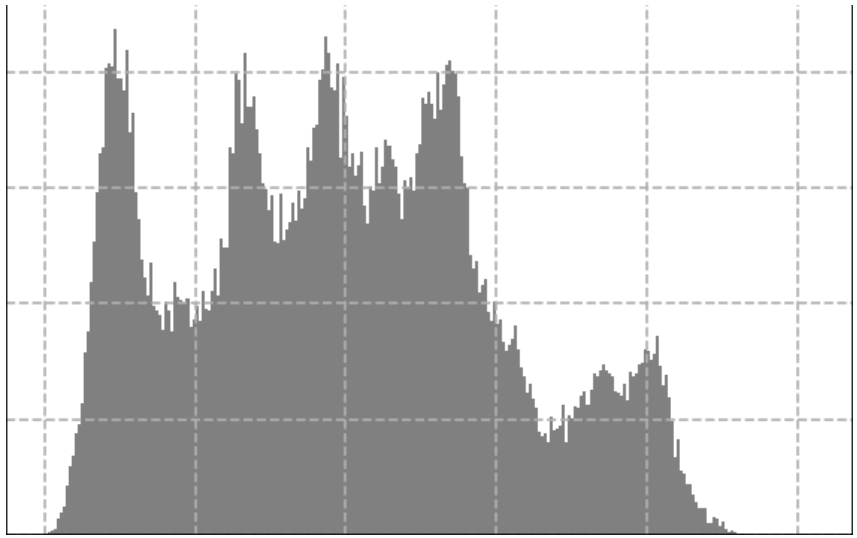

Figura 6: Histograma de la imagen

Consigna 4:

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

Umbralización

Definición

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

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

Código

```
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 [?]).

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

Resultados

Al aplicar la técnica de **umbralización** sobre la imagen de la Figura 7, con un valor de umbral de $u = 128$, se obtuvo la imagen binaria que se muestra en la Figura 8.

Figura 7: Imagen original

Figura 8: Imagen binaria

Consigna 5:

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

Ecualización del histograma

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, $i = 0, ..., 255$ es el número de pixels de la imagen con nivel de gris r_i, $i = 0, ..., 255$.
- n es el número total de pixels de la imagen.
- $\frac{n_i}{n}$, $i = 0, ..., 255$ 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:

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

Código

```
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')
```

Código +

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 [?]). np.bincount cuenta la cantidad de veces que aparece cada valor con indice igual a nivel de pixel y valor igual a frecuencia (ver [?]).

Resultados histogramas

Al aplicar **la ecualización del histograma** sobre el de la Figura 9, se obtuvo la versión ecualizada que se muestra en la Figura 10.

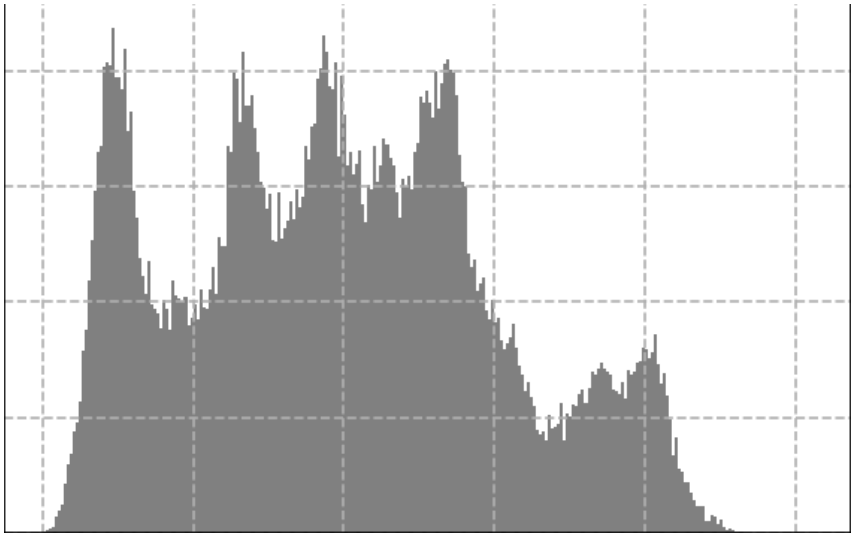

Figura 9: Histograma original

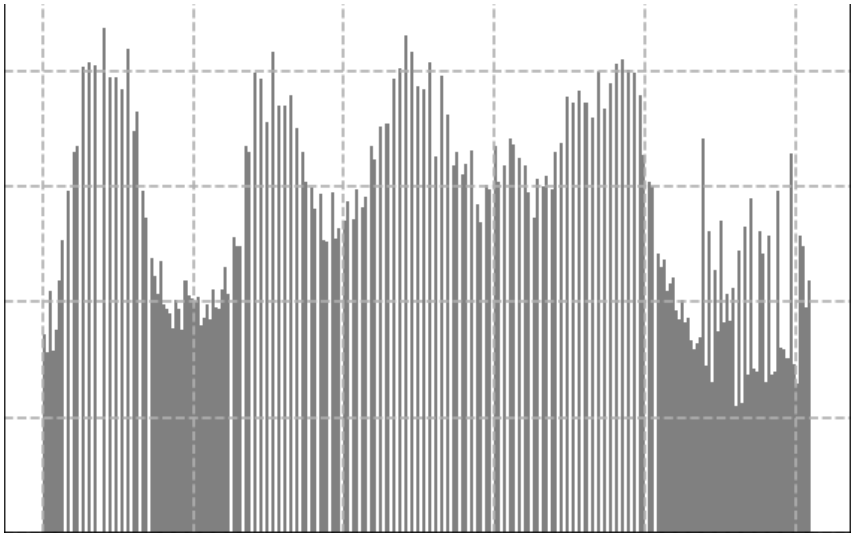

Figura 10: Histograma ecualizado

Resultados imagen

Al aplicar **la ecualización** sobre la imagen de la Figura 11, se obtuvo la versión mejorada que aparece en la Figura 12.

Figura 11: Imagen original

Figura 12: 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.

Resultados - Histogramas

En la Figura 13 se muestra el histograma original. Al aplicar la primera **ecualización** se obtuvo el histograma de la Figura 14, y al aplicar una segunda **ecualización** se llegó al de la Figura 15.

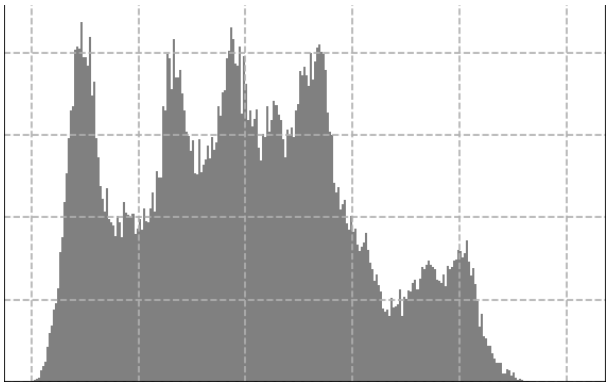

Figura 13: Histograma original

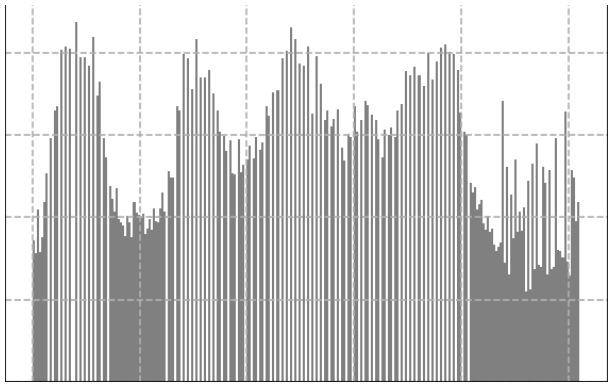

Figura 14: 1ª ecualización

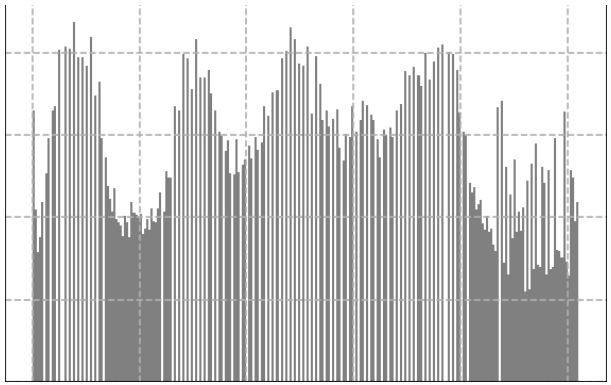

Figura 15: 2ª ecualización

Resultados - Imágenes

En la Figura 16 se observa la imagen original. Tras la primera **ecualización** se obtuvo la versión de la Figura 17, y al aplicar la **ecualización** una segunda vez se llegó a la Figura 18.

Figura 16: Imagen original

Figura 17: 1ª ecualización

Figura 18: 2ª ecualización

Consigna 7:

Implementar generadores de números aleatorios con las siguientes distribuciones:

1. Gaussiana con desviación estándar σ y valor medio μ.
2. Rayleigh con parámetro ξ.
3. Exponencial con parámetro λ.

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

Código

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.

Resultados

A continuación se muestran los histogramas generados para distintas distribuciones: Gaussiana, Rayleigh y Exponencial.

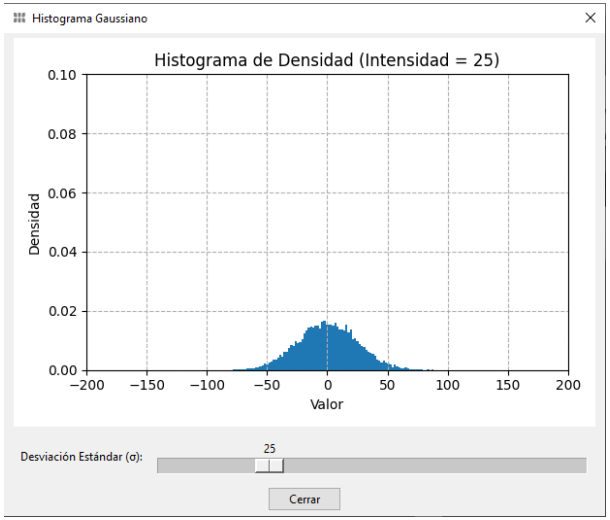

Figura 19: Dist gaussiana

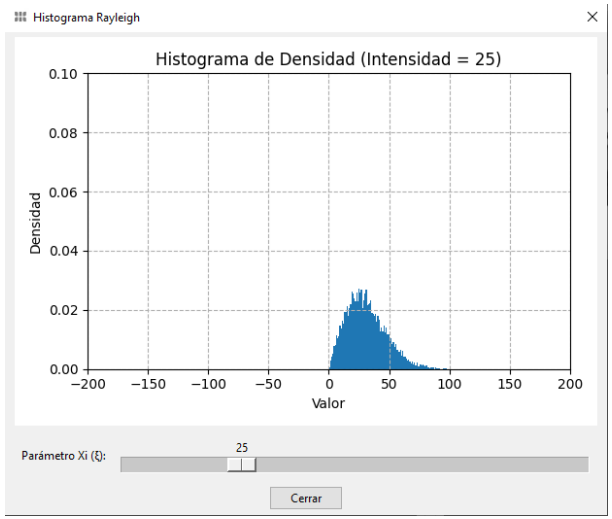

Figura 20: Dist rayleigh

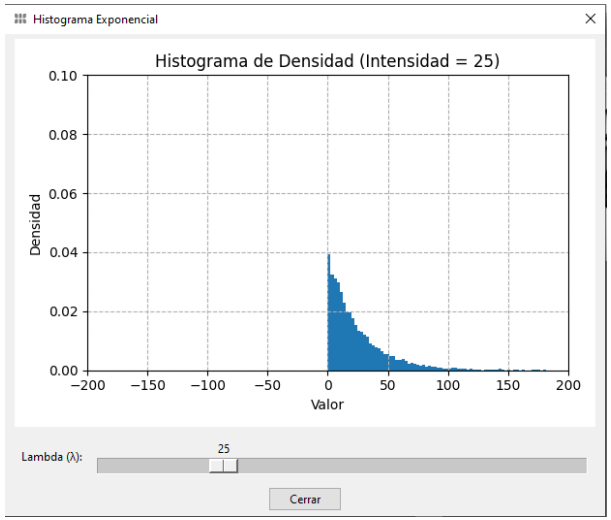

Figura 21: Dist exp

(*) 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.

Agregar ruido

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 λ, μ o ξ (se toma como parámetro).
4. Generar d valores aleatorios con la distribución correspondiente (se toma como vector_ruido).
5. Luego en los indices D de la imagen I multiplicar o sumar el escalar k del vector ruido al valor del pixel según corresponda.

Código

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 [?]). np.unravel_index (ver [?]).

Resultados

Al contaminar la imagen con **ruido gaussiano aditivo** y $\sigma = 20$, se obtuvo el resultado que se muestra en la Figura 23.

Figura 22: Imagen original

Figura 23: Imagen con ruido

Resultados

Al contaminar la imagen con **ruido Rayleigh multiplicativo** y $\xi = 2$, se obtuvo el resultado que se observa en la Figura 25.

Figura 24: Imagen original

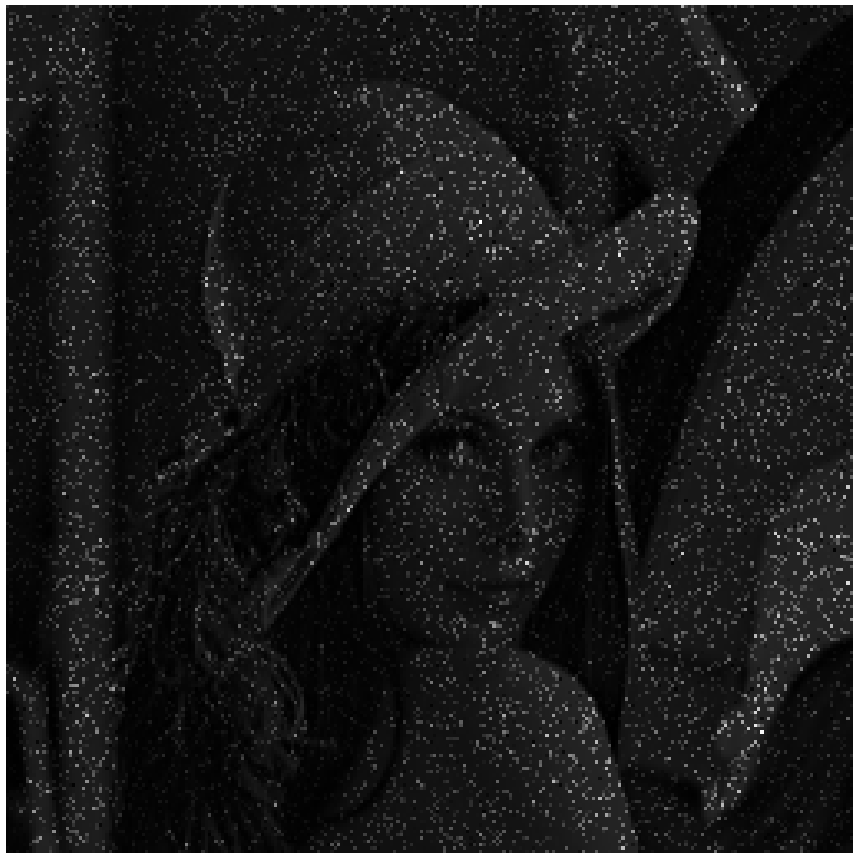

Figura 25: Imagen con ruido

Resultados

Al contaminar la imagen con **ruido exponencial multiplicativo** y $\frac{1}{\lambda} = 1$, se obtuvo la versión que aparece en la Figura 27.

Figura 26: Imagen original

Figura 27: Imagen con ruido

Consigna 9:

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

Ruido Sal y Pimienta

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 $I(i, j) = 255$

Código

```
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 = (porcentaje / 2) / 100.

Resultados

Al aplicar **ruido Sal y Pimienta** a una imagen con probabilidad = 10 sobre la Figura 28, se obtuvo la versión ruidosa que aparece en la Figura 29.

Figura 28: Imagen original

Figura 29: 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 σ y $\mu = 0$.
5. Realce de Bordes.

Aplicación de filtros

Estrategia

1. Transformar la imagen a un `np.array` en formato `RGB`.
2. Obtener sus dimensiones (m, n, c).
3. Calcular las dimensiones del filtro (k, l) y el *padding* necesario.
4. Generar una versión *padded* de la imagen.
5. Inicializar una matriz vacía para la imagen filtrada.
6. Recorrer cada píxel y canal de color:
   - Extraer la región correspondiente al filtro.
   - Multiplicar y sumar con la máscara del filtro.
   - Escalar el valor por el factor definido.
7. Escalar el resultado al rango $[0, 255]$.
8. Convertir el resultado nuevamente a imagen.

Código - Aplicar filtro

```
def _aplicar_filtro(self, imagen, filtro, factor):
  imagen_np = np.array(imagen.convert('RGB')).astype(float
    )
  m, n, _ = imagen_np.shape
  k, l = filtro.shape
  pad_h, pad_w = k//2, l//2

  imagen_padded = np.pad(imagen_np, ((pad_h, pad_h), (
    pad_w, pad_w), (0, 0)), mode='constant')
  imagen_filtrada = np.zeros_like(imagen_np)

  for i in range(m):
    for j in range(n):
      for c in range(3):
        region = imagen_padded[i:i+k, j:j+l, c]
        valor = np.sum(region * filtro) * factor
        imagen_filtrada[i, j, c] = valor

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

Filtro de la media

El filtro de la media se obtiene de la siguiente manera:

```
def _filtro_media(self, k):
  filtro = np.ones((k, k))

  factor = 1 / np.sum(filtro)
  return (filtro, factor)
```

(*) np.pad (ver [8]).

(*) np.repeat (ver [9]).

Resultados - Filtro Media

Al aplicar **el filtro Media** sobre la imagen de la Figura 30 con $k = 3$, se obtuvo la versión suavizada que aparece en la Figura 31.

Figura 30: Imagen original

Figura 31: Imagen con filtro Media

Código - Aplicar filtro Mediana

```
def _aplicar_filtro_mediana(self, imagen, filtro):
    imagen_np = np.array(imagen.convert('RGB')).astype(float
        )
    m, n, _ = imagen_np.shape
    k, l = filtro.shape
    pad_h, pad_w = k//2, l//2
    imagen_padded = np.pad(imagen_np, ((pad_h, pad_h), (
        pad_w, pad_w), (0, 0)), mode='constant')
    imagen_filtrada = np.zeros_like(imagen_np)
    indices_repeticion = filtro.flatten().astype(int)

    for i in range(m):
        for j in range(n):
            for c in range(3):
                region = imagen_padded[i:i+k, j:j+l, c]
                valores = np.repeat(region.flatten(),
        indices_repeticion) # Indica rep de cada indice
                mediana = np.median(valores)
                imagen_filtrada[i, j, c] = mediana
    self.imagen_procesada = Image.fromarray(imagen_filtrada.
        astype(np.uint8))
```

Resultados - Filtro Mediana

Al aplicar **el filtro Mediana** sobre la imagen de la Figura 32 con $k = 3$, se obtuvo la versión suavizada que aparece en la Figura 33.

Figura 32: Imagen original

Figura 33: Imagen con filtro Mediana

Filtro Mediana Ponderada

```
def _filtro_mediana_ponderada(self, k):
  filtro_gauss, _ = self._filtro_gaussiano(k)
  filtro = (filtro_gauss * 50).astype(int)

  factor = 1
  return (filtro, factor)
```

Resultados - Filtro Mediana Ponderada

Al aplicar **el filtro Mediana Ponderada** sobre la imagen de la Figura 34 con $k = 3$, se obtuvo la versión suavizada que aparece en la Figura 35.

Figura 34: Imagen original

Figura 35: Imagen con filtro Mediana Ponderada

Filtro Gaussiano

```
def _filtro_gaussiano(self, k):
filtro = np.ones((k, k)).astype(float)
u = k // 2 # Centro donde el valor debe ser máximo (
son iguales ya que es cuadrada)
sigma = (k-1) / 2

for x in range(k):
for y in range(k):
filtro[x, y] = (1 / (2 * np.pi * sigma**2)) * np.exp
(-((x - u)**2 + (y - u)**2)/(2 * sigma**2))

factor = 1 / np.sum(filtro)
return (filtro, factor)
```

Resultados - Filtro Gaussiano

Al aplicar **el filtro Gaussiano** sobre la imagen de la Figura 36, se obtuvieron las versiones suavizadas con distintos valores de sigma, que se muestran en las Figuras 37 y 38.

Figura 36: Imagen original

Figura 37: Filtro Gaussiano, $\sigma = 1$

Figura 38: Filtro Gaussiano, $\sigma = 2$

Filtro Realce de Bordes

```
def _filtro_realce(self, k):
  filtro = -1 * np.ones((k, k))
  filtro[k//2, k//2] = k**2 - 1

  factor = 1
  return (filtro, factor)
```

Resultados - Filtro Realce de Bordes

Al aplicar **el filtro Realce de Bordes** sobre la imagen de la Figura 39, se obtuvo la versión resaltada que aparece en la Figura 40.

Figura 39: Imagen original

Figura 40: Imagen con filtro Realce de Bordes

Consigna 11:

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

1. Ruido Gaussiano aditivo para varios de σ y $\mu = 0$.
2. Ruido Rayleigh multiplicativo para varios valores de ξ.

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.

Referencias

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

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

NumPy Documentation: numpy.repeat.
https://numpy.org/doc/2.3/reference/generated/numpy.repeat.html

¡Gracias!