INSTITUTO POLITÉCTICO NACIONAL

ESCUELA SUPERIOR DE COMPUTO

PRÁCTICA 2. "Mejoramiento de una imagen"

GRUPO: 4BM2

Materia:

Procesamiento digital de imágenes

Presenta:

Arana Acosta Shantal Vanessa

González Estrada Carlos Yael

Herrera Monroy Abraham Andre

Ortiz Santillan Hannia Maybelline

Rojas Muciño Cecilia

Nombre de la imagen utilizada:

Trolebús

Tipo de Reto abordado :

Filtrado para reparación de imágenes

Profesor de Materia: Elena Cruz Meza

México, CDMX. a 23 de noviembre del 2025

1. Introducción

El ruido dentro de las imágenes es una variación en los píxeles; es algo natural y común que poseen casi todas las imágenes. El ruido representa un problema debido a la interrupción de la información correcta que la imagen debería ofrecer. Existen varios tipos de ruido y, para su procesamiento digital, se utiliza un tratamiento específico para cada uno. Por ello, es necesario analizar qué tipo de ruido está presente para aplicar el tratamiento adecuado y obtener la mayor información posible.

En esta práctica, el objetivo es tomar una imagen de un espacio real, capturada de la manera habitual en la vida cotidiana, revisar el ruido que posee y aplicar un filtro que se ajuste a las necesidades del tratamiento. Esto nos ayuda en el proceso de filtrado de nuestras imágenes, ya que se adapta a la propuesta del proyecto: asistir en procesos de la vida diaria (donde las imágenes suelen presentar ruido de manera natural, pues no tienen ningún tratamiento previo), como en el caso del uso del trolebús como medio de transporte para personas con discapacidad visual.

2. Marco Teórico

Ruido

Como menciona Edurne González el ruido en imagen puede referirse al ruido fotográfico, que son artefactos visuales (puntos, granulado) causados por factores técnicos como la poca luz o el alto ISO, y al ruido visual o estético, esto puede decir que el ruido puede verse como una distorsión visual identificable como un efecto de granulado o decoloración que suele reducir el impacto de una imagen, oscurece los detalles y, cuando se presenta en niveles altos, puede arruinar por completo una fotografía.

Existen varios tipos de ruido como lo son:

- Ruido Impulsional o "Sal y Pimienta": Los píxeles de la imagen son muy diferentes en color o intensidad a los píxeles circundantes. Generalmente, este tipo de ruido, afectará a una pequeña cantidad de píxeles de la imagen, encontramos puntos blancos sobre puntos negros o puntos negros sobre puntos blancos, de ahí el término sal y pimienta.

Fig. 1. Ejemplificación de ruido sal y pimienta

- Ruido Gaussiano: Todos y cada uno de los píxeles que componen la imagen cambian su valor, de acuerdo con una distribución normal o gaussiana.

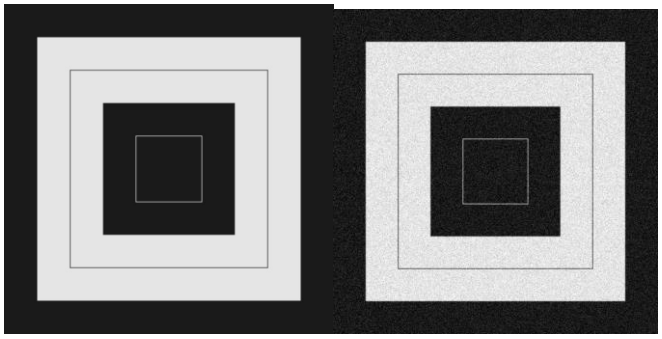

Fig. 2. Ejemplificación de ruido gaussiano

- Ruido de disparo: Es un tipo de ruido eléctrico que tiene lugar cuando el número finito de partículas que transportan energía, tales como los fotones en un dispositivo óptico, es suficientemente pequeño para dar lugar a la aparición de fluctuaciones estadísticas apreciables en una medición.

- Ruido de luminancia: El ruido de luminancia aparece como variaciones de luminosidad en la imagen, se nota sobre todo en superficies que deberían tener un color uniforme.

- Ruido de cuantificación: Errores de redondeo cuando la señal análoga es convertida en un ajuste finito de valores discretos digitales. No se aprecia comúnmente, se puede reducir usando un sensor que almacene más bits por píxel.

Filtros lineales

Es un proceso donde cada píxel de la imagen resultante se obtiene aplicando una operación lineal (normalmente una suma ponderada) sobre un conjunto de píxeles vecinos. Entre estos filtros se tiene:

- Filtro de la media: es el más simple, intuitivo y fácil de implementar para suavizar imágenes que el de la mediana, es decir, reducir la cantidad de variaciones de intensidad entre píxeles vecinos. Se visita cada píxel de la imagen y se reemplaza por la media de los píxeles vecinos. Se puede operar mediante convolución con una máscara determinada. Ofrece ciertas desventajas ya que es bastante sensible a cambios locales y puede crear nuevas intensidades de grises que no aparecían en la imagen.

Ejemplo:
Tomamos una fracción de la imagen a utilizar en esta práctica como lo es la fig. 5. pero se pasará a escala de grises y se le generará ruido (tomando la matriz de 5x5 del recuadro seleccionado):

210	215	220	215	210
220	235	240	235	220
225	245	250	245	225
220	235	240	235	220
210	215	220	215	210

Tabla 1: matriz 5x5 de un espacio de la imagen con ruido gaussiano

Para este filtro se utiliza una vecindad 8 por lo que se tomarán sólo los valores centrales, el proceso de filtrado consta en tomar un valor promedio para ello se suman estos 9 valores y se divide entre 9.

Suma= (235+240+235) + (245+250+245) + (235+240+235) = 2105
Promedio = 2105/9 = 233.88 qué se redondea a 234
Ahora el valor del píxel central será 234.
Se realiza el mismo proceso para cada píxel obteniendo

210	215	220	215	210
220	224	228	224	220
225	230	234	230	225
220	224	228	224	220
210	215	220	215	210

Tabla 2. filtro promedio

Filtro gaussiano: se usa para emborronar imágenes y eliminar ruido. Es similar al filtro de media, pero se usa una máscara diferente, modelizando la función gaussiana mostrada en la Fig.3. Las ventajas del filtro gaussiano frente al filtro de media es que es separable (es decir, en lugar de realizar una convolución bidimensional, podemos realizar dos convoluciones unidimensionales una en sentido horizontal y otra en sentido vertical) además que el filtro gaussiano produce un suavizado más uniforme que el filtro de media.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Fig. 3. Función gaussiana

Ejemplo:
Tomamos la tabla 1 propuesta anteriormente, como es un matriz de 5x5 se debe trabajar con un espacio de 3x3 dado que:
La fórmula para determinar el tamaño de la matriz de salida O cuando se aplica un kernel de tamaño K (vecindad 8 por lo tanto es de 3x3) a una matriz de entrada de tamaño Y (sin relleno o *padding*) es: O= Y - K + 1

- **Filas:** 5 - 3 + 1 = 3
- **Columnas:** 5 - 3 + 1 = 3
- Si elegimos sigma = 1

Tomando como punto (0,0) el centro y multiplicando por un factor de 4 para optimizar el resultado (por 4 ya que es el punto más central de la campana de gauss)

$$G(0,0) \propto e^{-\frac{(0)^2+(0)^2}{2(1)^2}} = e^0 = \mathbf{1.0000}$$

Hacemos eso para sacar el K de cada espacio de la vecindad ya que luego se aplica la convolución. donde I es la matriz original y K es el kernel Gaussiano.

$$\mathbf{I'_{i,j}} = \sum_{m,n} \mathbf{I_{i+m,j+n}} \cdot \mathbf{K_{m,n}}$$

Aplicando eso obtenemos.

210	215	220	215	210
220	224	230	224	220
225	237	243	237	225
220	224	230	224	220
210	215	220	215	210

Tabla 3. filtro gaussiano

- Filtro promediado: Llamado así debido a que promedia las muestras de la entrada y por lo tanto suprime variaciones rápidas, característica que le otorga el carácter de paso bajo. La ecuación de un filtro paso bajo digital de primer orden es:

$$y[n] = \frac{x[n] + x[n-1]}{A}$$

Fig. 4. Función de un filtro paso bajo

Donde A ha de ser mayor que uno.

- Filtrado bilineal: Es un método de procesamiento que calcula el color de un píxel promediando los valores de los cuatro píxeles más cercanos en una vecindad, aunque puede hacer que las texturas aparezcan un poco borrosas. Se puede entender como la aplicación de un filtro lineal en una dirección (horizontal), y luego aplicar otro filtro lineal en la dirección perpendicular (vertical)

Ejemplo:
Se puede volver a tomar la propuesta de la tabla 1 una matriz de 5x5. Para aplicar el filtrado bilineal, se trabaja con una vecindad de 2x2, porque este filtro se interpola usando los cuatro píxeles más cercanos al punto donde queramos calcular un nuevo valor.
Supóngase que se busca estimar el valor del píxel ubicado en una posición intermedia entre los siguientes valores de la matriz original:

210	215	220	215	210
220	235	240	235	220
225	245	250	245	225
220	235	240	235	220
210	215	220	215	210

Tabla 4. filtro bilineal

El filtro bilineal realiza primero un promedio horizontal y luego un promedio vertical:

1. Interpolación horizontal:

$$H_1 = \frac{235 + 240}{2} = 237.5$$

$$H_2 = \frac{245 + 250}{2} = 247.5$$

2. Interpolación vertical:

Ahora se promedian ambos resultados:

$$V = \frac{H_1 + H_2}{2} = \frac{237.5 + 247.5}{2} = 242.5$$

Redondeando el valor, obtenemos:

Valor interpolado = 243

Este será el nuevo valor asignado al píxel en la ubicación analizada.
El proceso se repite para todos los puntos necesarios dentro de la imagen, produciendo un suavizado más progresivo que el filtro de media, aunque con una ligera pérdida de nitidez debido a la interpolación.

Filtros no lineales

Estos filtros no se pueden representar de forma de ecuación matricial y tampoco tienen un formato general, sino que cada tipo de filtrado es un algoritmo determinado distinto del resto.

- Filtro de máximo: También llamado erosión, debido a la propiedad que posee de "adelgazar" líneas. Si nuestra imagen posee líneas negras, al elegir el valor máximo de la vecindad de cada pixel, los valores más oscuros serán sustituidos por valores más altos con la consiguiente reducción de los pixeles cercanos al negro. Una ventaja es que elimina el ruido pimienta (píxeles negros) pero tiene ciertos inconvenientes puesto que sólo funciona cuando el ruido es exclusivamente tipo pimienta y tiende a aclarar la imagen

Ejemplo:
Tomamos una fracción de la imagen a utilizar en esta práctica como lo es la fig. 5. pero se pasará a escala de grises y se le generará ruido (tomando la matriz de 5x5 del recuadro seleccionado):

210	0	220	215	255
255	235	240	0	220
225	0	250	245	225
220	235	255	235	0
0	215	220	255	210

Tabla 5. matriz 5x5 de un espacio de la imagen con ruido salpimienta

Se toma el valor máximo que se encuentra dentro de su vecindad 8 y se cambian todos los pixeles a ese valor

210	0	220	215	255
255	255	255	255	220
225	255	255	255	225
220	255	255	255	0
0	215	220	255	210

Tabla 6. filtro máximo

- Filtro de mínimo: De forma contraria al filtro de máximo este filtro tiende a ensanchar las líneas negras de la imagen, por esta razón también es conocido como filtro de dilatación. Selecciona el menor valor dentro de una ventana ordenada de valores de nivel de gris. Una ventaja es que elimina el ruido sal (píxeles blancos) pero tiene ciertos inconvenientes, sólo funciona cuando el ruido es exclusivamente tipo sal y tiende a oscurecer la imagen.

Ejemplo: tomando los valores de la tabla 5 se busca el valor mínimo dentro de vecindad y se coloca en todos los pixeles.

210	0	220	215	255
255	0	0	0	220
225	0	0	0	225
220	0	0	0	0
0	215	220	255	210

Tabla 7. filtro mínimo

- Filtro de la mediana: Se visita cada píxel de la imagen y se reemplaza por la mediana de los píxeles vecinos. La mediana se calcula ordenando los valores de los píxeles vecinos en orden y seleccionado el que queda en medio. Da muy buenos resultados en caso de ruido sal y pimienta.

Ejemplo:

Para el siguiente ejemplo se toma en base a la tabla 5 anterior siendo la matriz de la imagen a probar con un ruido salpimienta.

Para el cálculo, se toma la vecindad 3x3 del pixel central:

210	0	220	215	255
255	235	240	0	220
225	0	250	245	225
220	235	255	235	0
0	215	220	255	210

Tabla 8. filtro medio

1. Orden de valores

[235, 240, 0, 0, 250, 245, 235, 255, 235]

2. Valor medio (posición 5)

250

Resultado: el píxel central queda reemplazado por 250.

- Filtro de la moda: Es un filtro donde se toma una ventana alrededor de cada píxel, y el valor que aparece con mayor frecuencia dentro de esa ventana reemplaza al píxel central. Es parecido al filtro de la mediana, pero en vez de tomar la mediana toma la moda.

Ejemplo:
Se usa la misma vecindad 3x3 que usamos en el ejemplo del filtro de mediana.

1. Frecuencias de valores:
   Anotamos la frecuencia que tiene cada valor:

Valor	Frecuencia
0	2
235	3
240	1
245	1
250	1
255	1

2. Valor con mayor repetición:

235

Resultado: el píxel central queda reemplazado por 235.

3. Desarrollo de la Práctica

3.1 Preparación

Para iniciar la práctica, fue necesario instalar algunas librerías (considerando que la práctica se desarrolla en el lenguaje de programación Python). Las librerías utilizadas fueron las siguientes:

Librería OpenCV (cv2):
Permite abrir una imagen mediante la función `cv2.imread()`, la cual recibe dos argumentos: el nombre del archivo de la imagen y la ruta donde se encuentra (Cruz Meza, 2025).

Librería NumPy:
Se utiliza para trabajar con arreglos (arrays) y matrices numéricas, ofreciendo velocidad y funciones avanzadas para operaciones matemáticas.

También se seleccionó una fotografía que se empleará en esta propuesta, mostrada en la Fig. 5, con el fin de analizar la imagen en el entorno de estudio al que se dirige la práctica, pero desde perspectivas distintas.

Fig 5. Imágen de trabajo en la práctica

3.2 Generación de ruido

Aunque la imagen cuenta naturalmente con ruido, se aplicaron las funciones **add_salt_pepper_noise** y **add_gaussian_noise** para agregar más ruido de tipo sal y pimienta o ruido gaussiano, y así resaltarlo en nuestra imagen.

```
def add_salt_pepper_noise(self):
    if self.original_image is None:
        messagebox.showwarning("Advertencia", "Primero carga una imagen.")
        return
    gray = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
    salida = np.copy(gray)
    cantidad = 0.05
    num_pixeles = int(cantidad * gray.size)
    # sal
    coords = [np.random.randint(0, i, num_pixeles) for i in gray.shape]
    salida[coords[0], coords[1]] = 255
    # pimienta
    coords = [np.random.randint(0, i, num_pixeles) for i in gray.shape]
    salida[coords[0], coords[1]] = 0
    self.processed_image = salida
    self.display_image(self.processed_image, self.processed_canvas)
    self.show_histogram_auto()
    self.set_message("Ruido sal y pimienta aplicado.")
```

Como se puede observar en la sección de código, se agrega la función que provoca el ruido sal-pimienta en píxeles aleatorios de la imagen cargada. El resultado es una imagen en escala de grises con el ruido añadido, tal como se muestra en la Fig. 6.

Fig 6. Imágen con ruido sal y pimienta generado

Ahora en la siguiente sección de código, se agrega la función que provoca el ruido gaussiano en píxeles aleatorios de la imagen cargada. El resultado es una imagen en escala de grises con el ruido añadido, tal como se muestra en la Fig. 7.

```python
def add_gaussian_noise(self):
    if self.original_image is None:
        messagebox.showwarning("Advertencia", "Primero carga una imagen.")
        return
    gray = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
    media, sigma = 0, 25
    gauss = np.random.normal(media, sigma, gray.shape).astype(np.int16)
    salida = gray.astype(np.int16) + gauss
    salida = np.clip(salida, 0, 255).astype(np.uint8)
    self.processed_image = salida
    self.display_image(self.processed_image, self.processed_canvas)
    self.show_histogram_auto()
    self.set_message("Ruido gaussiano aplicado.")
```

Fig 7. Imágen con ruido gaussiano generado

3.3 Aplicación de filtros

Filtros lineales

El primer fragmento de código muestra el filtro de media o promedio. Este filtro se aplica calculando el promedio de los 25 píxeles contenidos en una ventana de 5×5 mediante la función `cv2.blur()`. Su propósito principal es tratar el ruido de tipo gaussiano.

```
def filter_mean(self):
    if self.original_image is None:
        messagebox.showwarning("Advertencia", "Primero carga una imagen.")
        return
    gray = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
    salida = cv2.blur(gray, (5,5))
    self.processed_image = salida
    self.display_image(self.processed_image, self.processed_canvas)
    self.show_histogram_auto()
    self.set_message("Filtro Promediador aplicado.")
```

El segundo fragmento de código muestra el filtro por promedio pesado (este no se mencionó anteriormente). Este filtro aplica una máscara con pesos (kernel) donde el centro pesa más usa `cv2.filter2D().` (convolución). Su propósito principal es tratar el ruido de tipo gaussiano.

```
def filter_weighted_mean(self):
    if self.original_image is None:
        messagebox.showwarning("Advertencia", "Primero carga una imagen.")
        return
    gray = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
    kernel = np.array([[1,1,1],[1,5,1],[1,1,1]], dtype=np.float32) / 13
    salida = cv2.filter2D(gray, -1, kernel)
    self.processed_image = salida
    self.display_image(self.processed_image, self.processed_canvas)
    self.show_histogram_auto()
    self.set_message("Filtro Promediador Pesado aplicado.")
```

El tercer fragmento de código muestra el último filtro lineal que es el filtro gaussiano este suaviza usando una máscara Gaussiana, que es una convolución con pesos según la campana de Gauss. Su propósito principal es tratar el ruido de tipo gaussiano.

```
def filter_gaussian(self):
    if self.original_image is None:
        messagebox.showwarning("Advertencia", "Primero carga una imagen.")
        return
    gray = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
    salida = cv2.GaussianBlur(gray, (5,5), 1)
    self.processed_image = salida
    self.display_image(self.processed_image, self.processed_canvas)
    self.show_histogram_auto()
    self.set_message("Filtro Gaussiano aplicado.")
```

Filtros no lineales

El primer fragmento de código de la sección de filtros no lineales muestra el filtro de mediana. Toma los valores de la ventana 5×5 y escoge la mediana (el valor del medio ordenado). Sirve para tratar el ruido sal-pimienta.

```
def filter_median(self):
    if self.original_image is None:
        messagebox.showwarning("Advertencia", "Primero carga una imagen.")
        return
    gray = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
    salida = cv2.medianBlur(gray, 5)
    self.processed_image = salida
    self.display_image(self.processed_image, self.processed_canvas)
    self.show_histogram_auto()
    self.set_message("Filtro Mediana aplicado.")
```

El siguiente fragmento de código muestra el filtro de moda. Toma los 9 píxeles de una ventana 3×3 y elige la moda, el que más se repite (esto puesto que se usan las vecindades). Sirve para tratar el ruido sal-pimienta.

```
def filter_mode(self):
    if self.original_image is None:
        messagebox.showwarning("Advertencia", "Primero carga una imagen.")
        return
    gray = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
    h, w = gray.shape
    salida = np.copy(gray)
    pad = 1
    imagen_padded = np.pad(gray, pad, mode='constant', constant_values=0)
    for i in range(h):
        for j in range(w):
            window = imagen_padded[i:i+3, j:j+3].flatten()
            moda = stats.mode(window, keepdims=True)

            if hasattr(moda, "mode"):
                valor = int(moda.mode[0])
            else:
                valor = int(moda[0][0])
            salida[i,j] = valor
    self.processed_image = salida.astype(np.uint8)
    self.display_image(self.processed_image, self.processed_canvas)
    self.show_histogram_auto()
    self.set_message("Filtro Moda aplicado.")
```

Tenemos en el siguiente fragmento de código el filtro bilateral funciona usando `cv2.bilateralFilter()`. Sirve para tratar el ruido gaussiano.

```
def filter_bilateral(self):
    if self.original_image is None:
        messagebox.showwarning("Advertencia", "Primero carga una imagen.")
        return
    gray = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
    salida = cv2.bilateralFilter(gray, 9, 75, 75)
    self.processed_image = salida
    self.display_image(self.processed_image, self.processed_canvas)
    self.show_histogram_auto()
    self.set_message("Filtro Bilateral aplicado.")
```

Tenemos en el siguiente fragmento de código el filtro de máximos, funciona tomando una matriz de 3×3 (definida por la vecindad) y toma el valor máximo de ellos. Sirve para tratar el ruido de la pimienta.

```
def filter_maximum(self):
    if self.original_image is None:
        messagebox.showwarning("Advertencia", "Primero carga una imagen.")
        return

    gray = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
    fil, col = gray.shape

    # Inicialización de matrices
    w_max = np.zeros((fil + 2, col + 2), dtype=np.uint8)
    w_max[1:fil + 1, 1:col + 1] = gray
    img_max = np.zeros((fil, col), dtype=np.uint8)

    # Recorrer píxel por píxel
    for j in range(1, fil + 1):
        for i in range(1, col + 1):
            ventana_max = w_max[j - 1:j + 2, i - 1:i + 2].flatten()
            img_max[j-1, i-1] = np.max(ventana_max)
```

Por último, tenemos el filtro de mínimos; funciona tomando una matriz de 3×3 (definida por la vecindad) y toma el valor mínimo de ellos. Sirve para tratar el ruido sal pimienta.

```
def filter_minimum(self):
    if self.original_image is None:
        messagebox.showwarning("Advertencia", "Primero carga una imagen.")
        return

    gray = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
    fil, col = gray.shape

    # Inicialización de matrices
    w_min = np.ones((fil + 2, col + 2), dtype=np.uint8) * 255
    w_min[1:fil + 1, 1:col + 1] = gray
    img_min = np.zeros((fil, col), dtype=np.uint8)

    # Recorrer píxel por píxel
    for j in range(1, fil + 1):
        for i in range(1, col + 1):
            ventana_min = w_min[j-1:j+2, i-1:i+2].flatten()
            img_min[j - 1, i - 1] = np.min(ventana_min)
```

Filtros Paso Alta

En la siguiente sección de código se define la función apply_sobel, el proceso del filtro Sobel se centra en la detección de bordes al calcular el gradiente de intensidad de la imagen. Inicialmente, la imagen se convierte a escala de grises para que el cálculo se base únicamente en la intensidad lumínica.

```python
def apply_sobel(self):
    if self.original_image is None:
        messagebox.showwarning("Advertencia", "Primero carga una imagen.")
        return

    self.save_processing_state("Filtro Sobel")

    # Convertir a escala de grises si es necesario
    if len(self.original_image.shape) == 3:
        image = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
    else:
        image = self.original_image

    sobel_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
    sobel_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)
    bordes_sobel = cv2.magnitude(sobel_x, sobel_y)
    self.processed_image = np.uint8(np.clip(bordes_sobel, 0, 255))
    self.display_image(self.processed_image, self.processed_canvas)
    self.show_histogram_auto()
    self.set_message("Filtro Sobel aplicado")
```

La función clave es *cv2.Sobel*, que se invoca dos veces: una para el gradiente horizontal y otra para el vertical. La primera llamada, sobel_x, aplica un kernel de convolución que está diseñado para medir los cambios de intensidad en la dirección X, lo que resulta en la detección y el resaltado de los bordes verticales. La segunda llamada, sobel_y, realiza la misma operación, pero en la dirección Y, detectando así los bordes horizontales. Ambas operaciones utilizan el tipo de dato cv2.CV_64F para trabajar con valores de gradiente que pueden ser positivos o negativos y que, posteriormente, se convierten a sus valores absolutos y se reescalan a 8 bits usando cv2.convertScaleAbs.

Finalmente, las dos imágenes resultantes, se combinan para producir la imagen, que representa la magnitud total del gradiente y muestra todos los bordes detectados, tanto horizontales como verticales, resaltando las transiciones de píxeles más abruptas.

La siguiente sección define el filtro apply_prewitt, se implementa el Filtro Prewitt, al igual que el filtro Sobel, se basa en la medición del gradiente de intensidad de la imagen para identificar contornos.

```python
def apply_prewitt(self):
    if self.original_image is None:
        messagebox.showwarning("Advertencia", "Primero carga una imagen.")
        return

    self.save_processing_state("Filtro Prewitt")

    # Convertir a escala de grises si es necesario
    if len(self.original_image.shape) == 3:
        image = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
    else:
        image = self.original_image

    # Definir kernels de Prewitt
    kernel_prewitt_x = np.array([[1, 0, -1], [1, 0, -1], [1, 0, -1]], dtype=np.float32)
    kernel_prewitt_y = np.array([[1, 1, 1], [0, 0, 0], [-1, -1, -1]], dtype=np.float32)

    # Aplicar filtros
    bordes_prewitt_x = cv2.filter2D(image, -1, kernel_prewitt_x)
    bordes_prewitt_y = cv2.filter2D(image, -1, kernel_prewitt_y)
    bordes_prewitt = cv2.addWeighted(bordes_prewitt_x, 0.5, bordes_prewitt_y, 0.5, 0)

    self.processed_image = bordes_prewitt
    self.display_image(self.processed_image, self.processed_canvas)
    self.show_histogram_auto()
    self.set_message("Filtro Prewitt aplicado")
```

Primero, la imagen de entrada se procesa y se convierte a escala de grises. La característica definitoria de este método es la definición manual de sus propios kernels de convolución utilizando NumPy, que son arreglos de 3x3 que definen la forma en que se miden los cambios de intensidad.

El kernel horizontal se define con valores de 1 y -1 en sus columnas, diseñado para detectar cambios abruptos de intensidad en la dirección horizontal, lo que subraya los bordes verticales de la imagen. Inversamente, el kernel vertical se define con valores de 1 y -1 en sus filas, diseñado para detectar cambios en la dirección vertical, lo que resalta los bordes horizontales.

Para aplicar estos kernels, el código utiliza la función general de convolución cv2.filter2D de OpenCV, que aplica cada máscara a la imagen para obtener dos resultados separados: bordes_prewitt_x y bordes_prewitt_y. Finalmente, para generar una imagen se combinan como en el filtro sobel.

En el siguiente segmento de código, se implementa el Operador de Roberts, que es el método de detección de bordes más simple y uno de los primeros, caracterizado por utilizar los kernels de convolución más pequeños de 2x2.

```python
def apply_roberts(self):
    if self.original_image is None:
        messagebox.showwarning("Advertencia", "Primero carga una imagen.")
        return

    self.save_processing_state("Filtro Roberts")

    # Convertir a escala de grises si es necesario
    if len(self.original_image.shape) == 3:
        image = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
    else:
        image = self.original_image

    # Definir kernels de Roberts
    kernel_roberts_x = np.array([[1, 0], [0, -1]], dtype=np.float32)
    kernel_roberts_y = np.array([[0, 1], [-1, 0]], dtype=np.float32)

    # Aplicar filtros
    bordes_roberts_x = cv2.filter2D(image, -1, kernel_roberts_x)
    bordes_roberts_y = cv2.filter2D(image, -1, kernel_roberts_y)
    bordes_roberts = cv2.addWeighted(bordes_roberts_x, 0.5, bordes_roberts_y, 0.5, 0)

    self.processed_image = bordes_roberts
    self.display_image(self.processed_image, self.processed_canvas)
    self.show_histogram_auto()
    self.set_message("Filtro Roberts aplicado")
```

La principal distinción del filtro Roberts es el uso de dos kernels de 2x2 definidos manualmente que miden las diferencias de píxeles a lo largo de las diagonales. el primer kernel es diagonal positiva y está diseñado para medir el gradiente que ocurre en esa dirección, el segundo kernel (kernel_roberts_y) es diagonal negativa (de arriba a la derecha a abajo a la izquierda), midiendo el gradiente en la dirección opuesta.

Estos dos kernels se aplican a la imagen de grises usando la función general de convolución cv2.filter2D, generando dos imágenes intermedias: bordes_roberts_x y bordes_roberts_y, que resaltan los bordes detectados en sus respectivas orientaciones diagonales, y como los 2 ultimos filtros, se combinan las imágenes.

De foma consiguiente, se aplica el Algoritmo de Canny, que sigue un proceso de múltiples etapas diseñado para encontrar la mayoría de los bordes reales y evitar los falsos.

```python
def apply_canny(self):
    if self.original_image is None:
        messagebox.showwarning("Advertencia", "Primero carga una imagen.")
        return

    self.save_processing_state("Filtro Canny")

    # Convertir a escala de grises si es necesario
    if len(self.original_image.shape) == 3:
        image = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
    else:
        image = self.original_image

    threshold1 = self.get_numeric_input("Canny", "Umbral inferior:", 100)
    threshold2 = self.get_numeric_input("Canny", "Umbral superior:", 200)

    if threshold1 is None or threshold2 is None:
        return

    self.processed_image = cv2.Canny(image, threshold1, threshold2)
    self.display_image(self.processed_image, self.processed_canvas)
    self.show_histogram_auto()
    self.set_message(f"Filtro Canny aplicado (umbrales: {threshold1}, {threshold2})")
```

A diferencia de los métodos Sobel o Prewitt, el filtro Canny no requiere la definición manual de kernels, ya que la función cv2.Canny encapsula los cinco pasos completos del algoritmo.

Antes de la detección de bordes, el código requiere la entrada del usuario para dos parámetros cruciales, el umbral superior y el umbral inferior . Estos umbrales controlan la etapa final de la detección de bordes, la detección se realiza con una única llamada a la función: self.processed_image = cv2.Canny(image, threshold1, threshold2). Dentro de esta función, se ejecutan las siguientes etapas en secuencia.

Primero la reducción de Ruido, se aplica un filtro Gaussiano a la imagen de grises para suavizarla y eliminar el ruido, luego se calculan los gradientes de la imagen para determinar la fuerza y la dirección de los bordes, de forma similar a los operadores Sobel/Prewitt. Luego la Supresión de No Máximos, esta etapa adelgaza los bordes, asegurando que solo los picos más fuertes de gradiente se mantengan, lo que resulta en líneas de contorno de un solo píxel de ancho, luego viene la aplicación de umbrales Aquí es donde entran en juego los valores. Los píxeles de gradiente superior se clasifican como bordes fuertes. Los píxeles entre los 2 umbrales se clasifican como bordes débiles. Los píxeles por debajo del umbral inferior se eliminan inmediatamente. Finalmente, el algoritmo examina los bordes débiles, un borde débil solo se mantiene y se clasifica como borde real si está conectado a un borde fuerte. Los bordes débiles que no están conectados a bordes fuertes se descartan.

Este proceso produce una imagen que contiene contornos limpios y bien definidos, que es el resultado del algoritmo Canny.

En la función apply_kirsch, que viene en el siguiente segmento de código , se implementa el Operador de Kirsch, que, a diferencia de Sobel o Prewitt, no solo detecta bordes, sino que también pretende determinar su orientación con alta precisión.

```
def apply_kirsch(self):
    if self.original_image is None:
        messagebox.showwarning("Advertencia", "Primero carga una imagen.")
        return

    self.save_processing_state("Filtro Kirsch")

    # Convertir a escala de grises si es necesario
    if len(self.original_image.shape) == 3:
        image = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
    else:
        image = self.original_image

    # Definir kernels de Kirsch (algunos ejemplos)
    kirsch_kernels = [
        np.array([[5, 5, 5], [-3, 0, -3], [-3, -3, -3]], dtype=np.float32),  # Norte
        np.array([[-3, 5, 5], [-3, 0, 5], [-3, -3, -3]], dtype=np.float32),  # Noreste
        np.array([[-3, -3, 5], [-3, 0, 5], [-3, -3, 5]], dtype=np.float32),  # Este
        np.array([[-3, -3, -3], [-3, 0, 5], [-3, 5, 5]], dtype=np.float32),  # Sureste
        np.array([[-3, -3, -3], [-3, 0, -3], [5, 5, 5]], dtype=np.float32),  # Sur
        np.array([[-3, -3, -3], [5, 0, -3], [5, 5, -3]], dtype=np.float32),  # Suroeste
        np.array([[5, -3, -3], [5, 0, -3], [5, -3, -3]], dtype=np.float32),  # Oeste
        np.array([[5, 5, -3], [5, 0, -3], [-3, -3, -3]], dtype=np.float32)  # Noroeste
    ]

    # Aplicar cada kernel y tomar el máximo
    responses = [cv2.filter2D(image, -1, kernel) for kernel in kirsch_kernels]
    bordes_kirsch = np.max(responses, axis=0)

    self.processed_image = np.uint8(np.clip(bordes_kirsch, 0, 255))
    self.display_image(self.processed_image, self.processed_canvas)
    self.show_histogram_auto()
    self.set_message("Filtro Kirsch aplicado")
```

El proceso inicia, como siempre, convirtiendo la imagen a escala de grises. La clave del Operador de Kirsch reside en el uso de ocho kernels de 3x3. Estos kernels están definidos manualmente y cada uno está diseñado para detectar una dirección específica del borde: Norte, Noreste, Este, Sureste, Sur, etc. Cada kernel asigna un peso alto a los píxeles que están en la dirección del borde que intenta detectar y un peso bajo o negativo a los píxeles en la dirección opuesta, lo que amplifica la respuesta en la orientación deseada.

La etapa de aplicación es un proceso repetitivo, el código usa una comprensión de lista para aplicar cada uno de los ocho kernels a la imagen de grises mediante la función cv2.filter2D. Esto resulta en una lista llamada responses, que contiene ocho imágenes de salida, donde cada imagen resalta los bordes que coinciden con una de las ocho direcciones.

Finalmente, para obtener el resultado completo del filtro Kirsch, la imagen de salida final se calcula tomando el valor máximo de respuesta para cada píxel a través de las ocho imágenes generadas: bordes_kirsch = np.max(responses, axis=0). Esto significa que, para cada punto de la imagen, se selecciona la respuesta más fuerte de cualquiera de los ocho kernels direccionales, lo que garantiza que el borde sea detectado sin importar su ángulo, y el resultado es una imagen self.processed_image que resalta los contornos con alta definición y sensibilidad a la orientación.

Ahora se aplica el Filtro Laplaciano, que es un operador de segundo orden utilizado para la detección de bordes y la acentuación de detalles, a diferencia de los operadores como Sobel yPrewitt que miden la fuerza del gradiente. El Laplaciano mide la tasa de cambio del gradiente, o de otra forma, detecta los puntos donde el gradiente cambia de dirección, lo que corresponde a un borde.

```python
def apply_laplacian(self):
    if self.original_image is None:
        messagebox.showwarning("Advertencia", "Primero carga una imagen.")
        return

    self.save_processing_state("Filtro Laplaciano")

    # Convertir a escala de grises si es necesario
    if len(self.original_image.shape) == 3:
        image = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)
    else:
        image = self.original_image

    laplacian = cv2.Laplacian(image, cv2.CV_64F)
    self.processed_image = np.uint8(np.clip(np.abs(laplacian), 0, 255))
    self.display_image(self.processed_image, self.processed_canvas)
    self.show_histogram_auto()
    self.set_message("Filtro Laplaciano aplicado")
```

La detección del Laplaciano se realiza con una sola llamada a la función cv2.Laplacian: laplacian = cv2.Laplacian(image, cv2.CV_64F). La función aplica un kernel que calcula la segunda derivada de la intensidad de la imagen, los bordes en la imagen Laplaciana se caracterizan por el fenómeno de "cruce por cero": el valor del píxel en el Laplaciano cambia de positivo a negativo justo en el borde.

Dado que la imagen de resultado laplacian contiene valores negativos y positivos, se aplica una operación de post-procesamiento para que sea visible como una imagen de 0 a 255, esta etapa consiste en tomar el valor absoluto del Laplaciano ,lo que convierte todos los cruces por cero en picos positivos, y luego escalar estos valores para que encajen en el rango de 0 a 255. El resultado final, self.processed_image, muestra los contornos resaltados donde hubo un cruce por cero.

4. Análisis de Resultados

En el primer caso, con el filtro promediado (Fig. 8), tenemos la imagen original y la procesada, donde podemos ver un claro resultado de cómo se suaviza la imagen y se consigue una imagen más clara y mejor visualmente, por lo cual se toma como una buena opción al resolver el problema de ruido.

Fig. 8. Resultado del filtro promediado

En el segundo caso, con el filtro promediado pesado (Fig. 9), tenemos la imagen original y la procesada, donde podemos ver cómo se suaviza la imagen, pero no se corrige completamente como con el filtro anterior. Por lo tanto, se concluye que, aunque ayuda a reducir el problema, no es la solución más eficiente.

Fig. 9. Resultado del filtro promediado pesado

En el tercer caso, con el filtro gaussiano (Fig. 10), tenemos la imagen original y la procesada, donde podemos ver cómo se suaviza la imagen; el ruido es menor y se corrige prácticamente todo, aunque aún se observan algunos detalles. Aun así, es una buena opción para tratar este tipo de ruido.

Fig. 10. Resultado del filtro gaussiano

En el último caso, con el filtro bilateral (Fig. 11), tenemos la imagen original y la procesada, donde podemos ver cómo se suaviza la imagen; el ruido se reduce en su mayoría, obteniendo una imagen clara pero no completamente limpia.

Fig. 11. Resultado del filtro bilateral

Ahora, teniendo ruido salpimienta, se usaron los filtros mediana, máximo y mínimo, obteniendo los siguientes resultados.

En la Fig. 12 tenemos el filtro mediana, donde podemos notar el cambio de una imagen a otra, resolviendo el problema del ruido y ofreciendo una imagen más clara y detallada.

Fig 12. Resultado del filtro mediana

En la Fig. 13 tenemos el filtro máximo; usando este filtro se obtiene una imagen más clara y con menos ruido pimienta, pero no se logra mejorar significativamente la calidad de la imagen.

Fig 13. Resultado del filtro máximo

En la Fig. 14 tenemos el filtro de mínimo; usando este filtro se obtiene una imagen más oscura y con menos ruido sal, pero no se logra mejorar significativamente la calidad de la imagen.

Fig 14. Resultado del filtro mínimo

En la Figura 15 tenemos el filtro sobel, observamos que el filtro Sobel ha resaltado eficazmente los bordes y contornos del trolebús y su entorno, transformando la imagen original en un mapa de gradientes de intensidad.

Los elementos con cambios abruptos en el brillo , como la carrocería del autobús, las ventanas, las líneas de la carretera y las estructuras como arboles y cables, aparecen ahora como líneas blancas brillantes sobre un fondo predominantemente oscuro o negro.

Fig 15. Resultado del filtro sobel

En la figura 16, observamos que cumple su función principal de aislar las regiones donde la intensidad de los píxeles cambia drásticamente, todos los contornos principales y secundarios, como las líneas de la carrocería del autobús, los marcos de las ventanas, las líneas de la carretera y los detalles finos de los cables se manifiestan como líneas blancas o brillantes sobre un fondo que es predominantemente oscuro.

Fig 16. Resultado del filtro Prewitt

Las áreas que eran uniformes en la imagen original, como grandes secciones del cielo o superficies lisas, aparecen en negro o con muy poca intensidad, debido a que el cambio de gradiente en esas zonas es insignificante.

En la figura 17, se muestra el resultado del filtro Roberts, observamos que el filtro ha capturado exitosamente las principales líneas de alto contraste, como los contornos de la carrocería, los bordes de las ventanas y las líneas de demarcación en el pavimento.

Fig 17. Resultado del filtro Roberts

sin embargo, debido a que el filtro Roberts opera sobre una pequeña vecindad de 2x2 píxeles y no incluye un paso de suavizado inherente como Sobel o Prewitt, es particularmente sensible al ruido. En la imagen, esto se traduce en una apariencia más "ruidosa, granulada o delgada de los bordes y en la aparición de puntos aislados de alta intensidad en áreas que deberían ser uniformes

La figura 18 son los resultados del filtro Canny, el resultado es extremadamente sucio. Canny es un algoritmo que depende críticamente de los dos umbrales para definir qué se considera un borde real, si los umbrales son muy altos, el filtro será muy selectivo. Solo los cambios de intensidad muy fuertes (bordes evidentes) serán clasificados como bordes fuertes, y solo los bordes débiles muy prominentes conectados a esos pocos fuertes serán retenidos, y si son valores bajos, sera muy permisivo, por lo tanto, al tener una imagen con tantos detalles y diferencias de valores, le es complicado al algoritmo detallar los objetos correctamente.

Fig 18. Resultado del filtro Canny

El filtro que sigue es Kirsch mostrado en la figura 19, Al igual que los filtros Sobel y Prewitt, se ha generado un mapa de contornos donde las principales estructuras del trolebús y del entorno están claramente resaltadas con líneas blancas brillantes, El filtro parece ser muy

sensible a los bordes, y en esta imagen se puede notar que ha capturado una gran cantidad de detalles estructurales, incluso en su textura.

Fig 19. Resultado del filtro Kirsch

La figura 20 es el resultado del filtro laplaciano, se puede observar que ha extraído de manera efectiva los contornos de la imagen, mostrando todos los detalles y discontinuidades. Sin embargo, ha amplificado considerablemente el ruido y las texturas finas de la escena, lo que hace que los bordes, aunque claros, parezcan más gruesos y menos limpios que los producidos por un algoritmo optimizado como el Canny.

Fig 20. Resultado del filtro laplaciano

4.1 Comparación visual

¿Qué diferencias se observan entre las imágenes filtradas?

R.- Las imágenes filtradas presentan variaciones claras dependiendo del tipo de filtro aplicado. En los filtros lineales (promediado, promediado pesado, gaussiano y bilateral), la imagen se suaviza de manera progresiva: el promediado genera un desenfoque notable, pero elimina gran parte del ruido; el promediado pesado suaviza, pero no corrige tan bien; el filtro gaussiano ofrece un balance adecuado entre suavizado y conservación de detalles; y el filtro bilateral reduce el ruido manteniendo bordes más definidos. Para los filtros no lineales (mediana, máximo y mínimo), la diferencia es más puntual: el filtro mediana elimina eficazmente el ruido salpimienta manteniendo detalles importantes; el filtro máximo elimina puntos negros, pero aclara demasiado la imagen; y el filtro mínimo elimina puntos blancos, pero oscurece la imagen.

¿Qué filtro conserva mejor los bordes?

R.- El mejor filtro con el que se cuenta por ahora capaz de conservar mejor los bordes es el bilateral, es el que mejor conserva los bordes, ya que reduce el ruido sin difuminar los contornos, a diferencia de los filtros lineales tradicionales que tienden a suavizar excesivamente.

¿Qué filtro elimina más eficazmente el ruido?

R.- Para ruido gaussiano, los filtros más efectivos son el gaussiano y el promediado, siendo el gaussiano el que ofrece un mejor equilibrio entre suavizado y nitidez.

Para ruido sal y pimienta, el filtro mediana es el más eficaz al eliminar valores extremos sin perder detalles importantes.

4.2 Evaluación técnica

¿Qué parámetros fueron clave en cada filtro?
R.- El parámetro más importante en todos los filtros es el tamaño del kernel, pues determina cuántos píxeles participan en el cálculo del valor final. Además:

- En el filtro gaussiano, el parámetro *sigma* (desviación estándar) es esencial para el grado de suavizado.

- En el filtro bilateral, los parámetros *d*, *sigmaColor* y *sigmaSpace* influyen directamente en la preservación de bordes.

- En la generación de ruido, la cantidad de ruido y la desviación estándar para el ruido gaussiano fueron fundamentales.

¿Qué dificultades enfrentaron al implementar los algoritmos?
R.- El principal problema surgió con el filtro de moda, ya que su implementación es más compleja de lo esperado. La función no logra generar correctamente la matriz resultante debido a la forma en que se calcula la moda dentro de cada vecindad, causando errores y deteniendo el procesamiento. El resto de los filtros funcionó sin mayores complicaciones.

5. Conclusiones

El análisis realizado muestra que la selección del filtro adecuado depende directamente del tipo de ruido presente en la imagen. Para el ruido sal y pimienta, el filtro de mediana resulta el más eficiente al eliminar valores extremos sin comprometer detalles importantes. En el caso del ruido gaussiano, los filtros gaussiano y bilateral ofrecen los mejores resultados al reducir el granulado y mantener la nitidez. Cuando el ruido corresponde exclusivamente a sal o únicamente a pimienta, los filtros mínimo y máximo, respectivamente, proporcionan la eliminación más adecuada.

El estudio evidencia que cada filtro cumple una función específica y que una elección inadecuada puede deteriorar la calidad de la imagen. Se identificó la relevancia del tamaño del kernel, de la preservación de bordes y de la estructura del ruido, así como la conveniencia de aplicar técnicas lineales y no lineales según las características del problema.

La práctica demuestra que, para el segundo prototipo, es indispensable trabajar con imágenes depuradas antes de su procesamiento. La aplicación del filtro correcto permite obtener información visual más clara y confiable, reduce errores y mejora la precisión del sistema dentro del entorno planteado.

6. Referencias

Martínez, L. (2020, November 3). ¿Qué es el ruido en una foto? Domestika.

https://www.domestika.org/es/blog/3922-que-es-el-ruido-en-una-foto?exp_set=1

colaboradores de Wikipedia. (2025, April 25). *Ruido en la fotografía digital*. Wikipedia, La Enciclopedia Libre.

https://es.wikipedia.org/wiki/Ruido_en_la_fotograf%C3%ADa_digital#Filtros_lineales

Curso de Procesado de Imagen (c)GPI. (n.d.).

https://www.uv.es/gpoei/eng/Pfc_web/realzado/fnl/rank.htm

colaboradores de Wikipedia. (2025, March 14). *Filtro paso bajo*. Wikipedia, La Enciclopedia Libre. https://es.wikipedia.org/wiki/Filtro_paso_bajo