

Trabajo Práctico 2

Procesamiento de Imágenes y Visión por Computadora

16 de octubre de 2025

Descripción Bordes

- Detectar bordes consiste en encontrar los pixels correspondiente a partes salientes de la imagen.

Causas

- Los bordes son causados por **discontinuidad** en los valores de intensidad o color.
- Los bordes son causados por **discontinuidad** profundidad de la imagen.
- Los bordes son causados por**discontinuidad** en la normal a la superficie.
- Los bordes son causados por **discontinuidad** en iluminación.

Métodos de Gradiente

Puntos de borde

Son aquellos pixels de la imagen en donde hay un cambio brusco o discontinuidad en los valores de intensidad o de color.

- Los métodos de gradiente calculan estimaciones del gradiente $\nabla I = (I_x, I_y)$ para cada píxel.
- La Magnitud del Gradiente se calcula como $|\nabla I| = \sqrt{I_x^2 + I_y^2}$.

Operador de Prewitt

Descripción

- El operador de Prewitt aproxima el gradiente de la imagen utilizando diferencias en regiones de 3×3 píxeles.
- Calcula componentes del gradiente:

$$I_x = (I_{1,3} + I_{2,3} + I_{3,3}) - (I_{1,1} + I_{2,1} + I_{3,1})$$

$$I_y = (I_{3,1} + I_{3,2} + I_{3,3}) - (I_{1,1} + I_{1,2} + I_{1,3})$$

Máscara horizontal (G_y)

-1	-1	-1
0	0	0
1	1	1

Máscara vertical (G_x)

-1	0	1
-1	0	1
-1	0	1

Operador Sobel

Descripción

- Aproximan el gradiente utilizando diferencias en secciones de 3×3 píxeles. Le da más importancia a los pixeles más cercanos.
- Calcula componentes del gradiente:

$$I_x = (I_{1,3} + 2 \cdot I_{2,3} + I_{3,3}) - (I_{1,1} + 2 \cdot I_{2,1} + I_{3,1})$$

$$I_y = (I_{3,1} + 2 \cdot I_{3,2} + I_{3,3}) - (I_{1,1} + 2 \cdot I_{1,2} + I_{1,3})$$

Máscara horizontal (G_y)

-1	-2	-1
0	0	0
1	2	1

Máscara vertical (G_x)

-1	0	1
-2	0	2
-1	0	1

Implementación del operador de Prewitt

```
1 def prewitt_magnitud():
2     global imagen_actual, imagen_operativa
3     imagen_original = imagen_actual.copy().astype(np.float32)
4
5     mascara_horizontal = np.array([[1, 1, 1], [0, 0, 0], [-1, -1,
6         -1]], dtype=np.float32)
7     mascara_vertical = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]],
8         dtype=np.float32)
9
10    if len(imagen_original.shape) == 2:
11
12        prewitt_horizontal = fc.mascara(imagen_original, mascara=
13            mascara_horizontal, tipo_kernel="Prewitt Horizontal", grises=True,
14            prewitt=False)
15        prewitt_vertical = fc.mascara(imagen_original, mascara=
16            mascara_vertical, tipo_kernel="Prewitt Vertical", grises=True,
17            prewitt=False)
```

Implementación del operador de Prewitt 2

```
1 else:  
2     prewitt_horizontal = fc.mascara(imagen_original, mascara=  
3                                     mascara_horizontal, tipo_kernel="Prewitt Horizontal", grises=False,  
4                                     prewitt=False)  
5     prewitt_vertical = fc.mascara(imagen_original, mascara=  
6                                     mascara_vertical, tipo_kernel="Prewitt Vertical", grises=False,  
7                                     prewitt=False)  
8  
8 magnitud = np.sqrt(prewitt_horizontal.astype(np.float32)**2 +  
9                     prewitt_vertical.astype(np.float32)**2)  
10 magnitud = ((magnitud - np.min(magnitud)) / (np.max(magnitud) - np.min  
11         (magnitud))) * 255  
12 imagen_operativa = magnitud.astype(np.uint8)  
13 mostrar_imagen(imagen_operativa, lblOutputImage)
```

Implementación del operador de Sobel

```
1 def sobel_magnitud():
2     global imagen_actual, imagen_operativa
3     imagen_original = imagen_actual.copy().astype(np.float32)
4
5     mascara_horizontal = np.array([[1, 2, 1], [0, 0, 0], [-1, -2,
6         -1]], dtype=np.float32)
7     mascara_vertical = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], 
8         dtype=np.float32)
9
10    if len(imagen_original.shape) == 2:
11        sobel_horizontal = fc.mascara(imagen_original, mascara=
12            mascara_horizontal, tipo_kernel="Sobel Horizontal", grises=True,
13            prewitt=False)
14        sobel_vertical = fc.mascara(imagen_original, mascara=
15            mascara_vertical, tipo_kernel="Sobel Vertical", grises=True,
16            prewitt=False)
17
18
19
```

Implementación del operador de Sobel 2

```
1 else:  
2     prewitt_horizontal = fc.mascara(imagen_original, mascara=  
3                                     mascara_horizontal, tipo_kernel="Prewitt Horizontal", grises=False,  
4                                     prewitt=False)  
5     prewitt_vertical = fc.mascara(imagen_original, mascara=  
6                                     mascara_vertical, tipo_kernel="Prewitt Vertical", grises=False,  
7                                     prewitt=False)  
8  
8 magnitud = np.sqrt(prewitt_horizontal.astype(np.float32)**2 +  
9                     prewitt_vertical.astype(np.float32)**2)  
10 magnitud = ((magnitud - np.min(magnitud)) / (np.max(magnitud) - np.min  
11 (magnitud))) * 255  
12 imagen_operativa = magnitud.astype(np.uint8)  
13 mostrar_imagen(imagen_operativa, lblOutputImage)
```

Resultados del Detector de Bordes



Figura: Lena Prewitt



Figura: Girl2 Prewitt



Figura: Barco Prewitt

Resultados del Detector de Bordes



Figura: Lena Sobel



Figura: Girl2 Sobel



Figura: (Barco Sobel)

Resultados del Detector de Bordes con Ruido Gaussiano

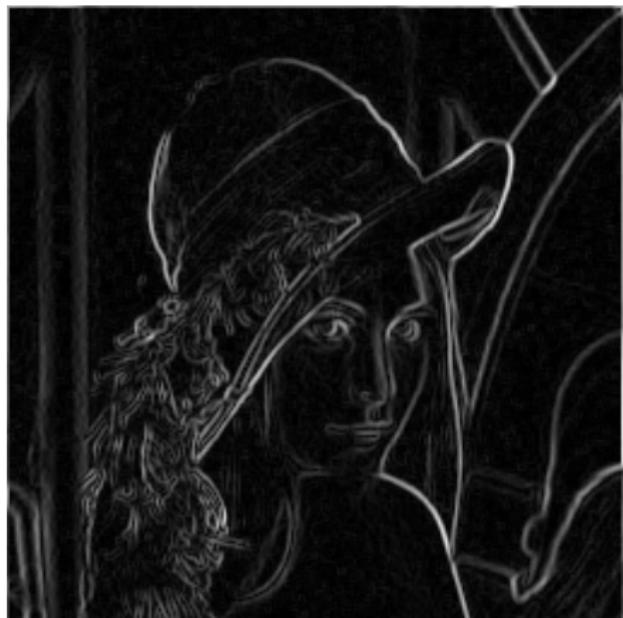


Figura: Lena Gauss 10 % con desvío 5
/Prewitt



Figura: Lena Gauss 10 % con desvío 5
/Sobel

Resultados del Detector de Bordes con Ruido Gaussiano



Figura: Lena Gauss 30 % con desvío 10
/Prewitt

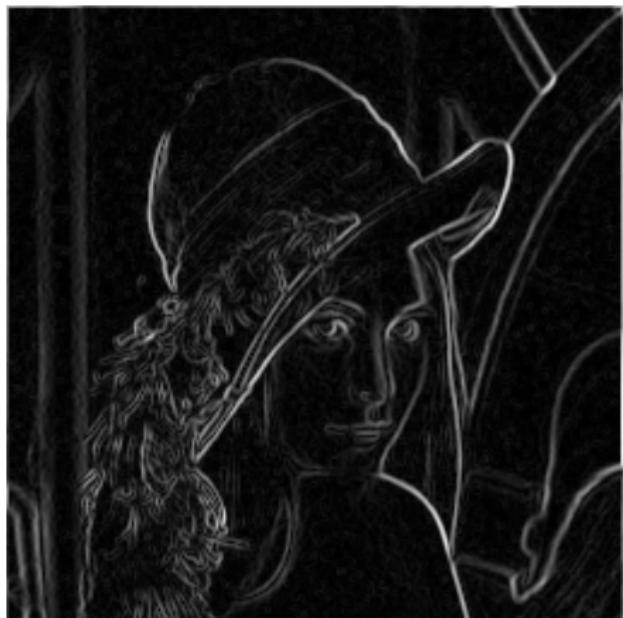


Figura: Lena Gauss 30 % con desvío 10
/Sobel

Resultados del Detector de Bordes con Ruido Gaussiano



Figura: Lena Gauss 50 % con desvío 20
/Prewitt



Figura: Lena Gauss 50 % con desvío 20
/Sobel

Resultados del Detector de Bordes con Ruido Rayleigh



Figura: Girl 2 Rayleigh 10 % con parámetro 0.1 /Prewitt



Figura: Girl 2 Rayleigh 10 % con parámetro 0.1 /Sobel

Resultados del Detector de Bordes con Ruido Rayleigh



Figura: Girl 2 Rayleigh 30 % con parámetro 0.2 /Prewitt



Figura: Girl 2 Rayleigh 30 % con parámetro 0.2 /Sobel

Resultados del Detector de Bordes con Ruido Rayleigh



Figura: Girl 2 Rayleigh 50 % con parámetro 0.5 /Prewitt



Figura: Girl 2 Rayleigh 50 % con parámetro 0.5 /Sobel

Resultados del Detector de Bordes con Ruido Exponencial

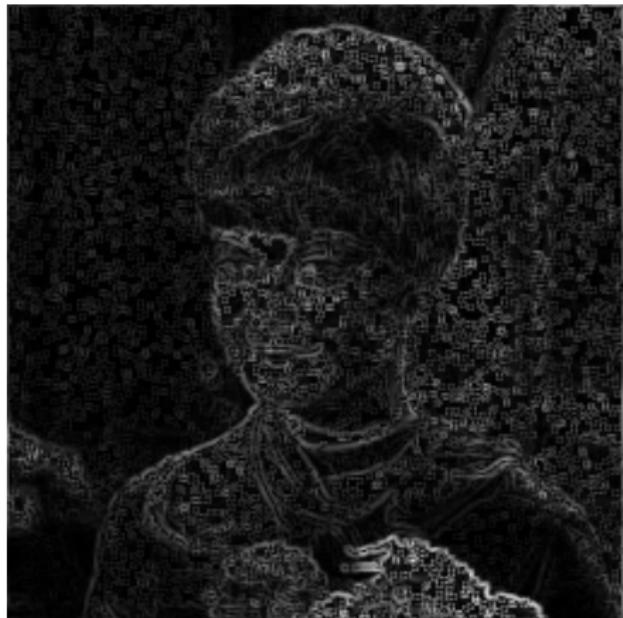


Figura: Girl 2 Exponencial 10 % con parámetro 0.01 /Prewitt



Figura: Girl 2 Exponencial 10 % con parámetro 0.01 /Sobel

Resultados del Detector de Bordes con Ruido Exponencial



Figura: Girl 2 Exponencial 30 % con parámetro 0.05 /Prewitt



Figura: Girl 2 Exponencial 30 % con parámetro 0.05 /Sobel

Resultados del Detector de Bordes con Ruido Exponencial



Figura: Girl 2 Exponencial 50 % con parámetro 0.1 /Prewitt



Figura: Girl 2 Exponencial 50 % con parámetro 0.1 /Sobel

Resultados del Detector de Bordes con Ruido Sal y Pimienta

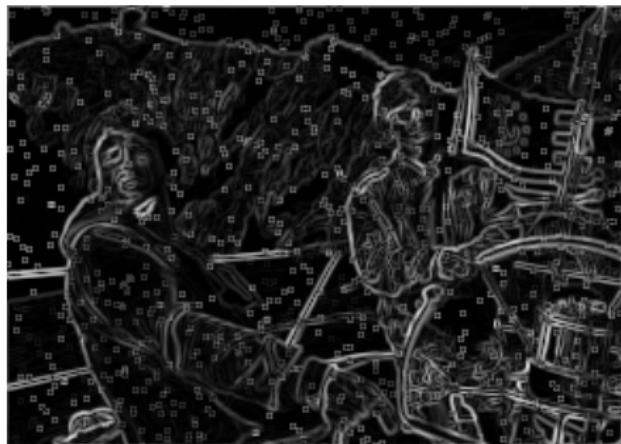


Figura: Barco Sal y Pimienta con parámetro 0.01 /Prewitt

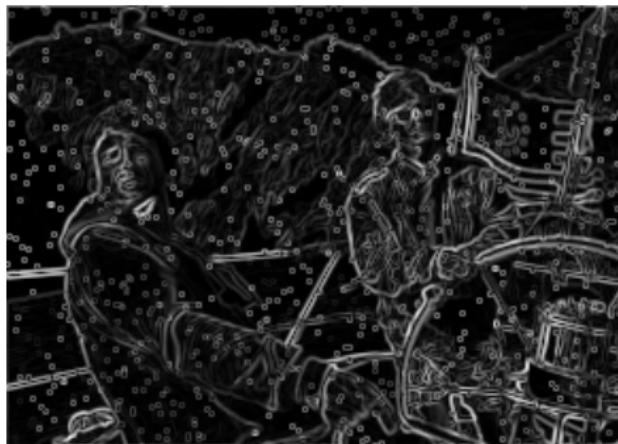


Figura: Barco Sal y Pimienta con parámetro 0.01 /Sobel

Resultados del Detector de Bordes con Ruido Sal y Pimienta

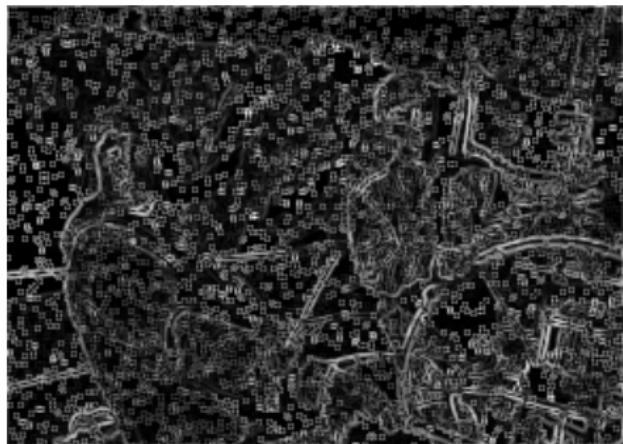


Figura: Barco Sal y Pimienta con parámetro 0.05 /Prewitt

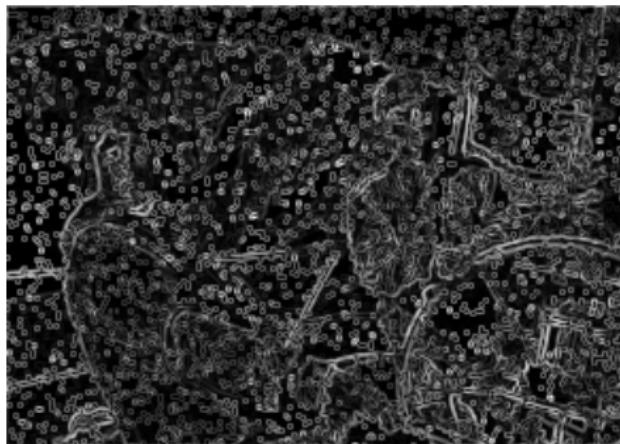


Figura: Barco Sal y Pimienta con parámetro 0.05 /Sobel

Resultados del Detector de Bordes con Ruido Sal y Pimienta

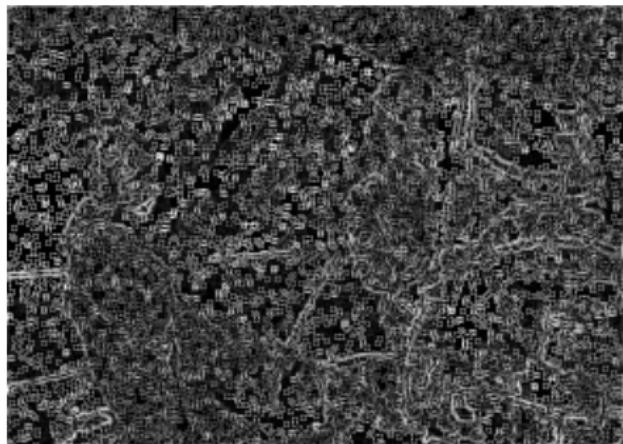


Figura: Barco Sal y Pimienta con parámetro 0.1 /Prewitt

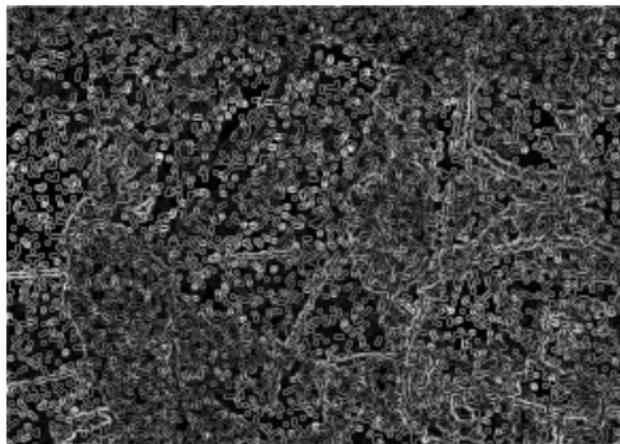


Figura: Barco Sal y Pimienta con parámetro 0.1 /Sobel

Resultados del Detector de Bordes con Canales



Figura: Baboon/Prewitt



Figura: Baboon/Sobel

Resultados del Detector de Bordes con Canales



Figura: Flores/Prewitt

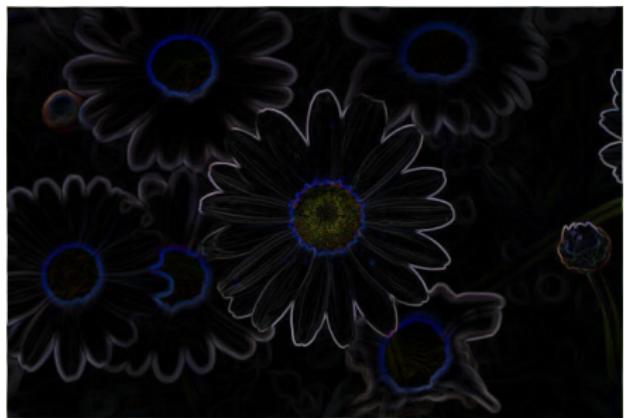


Figura: Flores/Sobel

Laplaciano

- Los operadores de segunda derivada requieren:
- 1) Pasar la máscara.
- 2) Detectar los cruces por cero.

Método del Laplaciano

- El operador ΔI se aproxima por:

$$\Delta I(x, y) \approx 4I(x, y) - I(x \pm 1, y) - I(x, y \pm 1)$$

- Los bordes se detectan buscando **cruces por cero** (cambios de signo).

Máscara Laplace

0	-1	0
-1	4	-1
0	-1	0

Laplaciano

Laplaciano con Evaluación de la Pendiente

- Se define un umbral de borde u_b .
- Solo se marca un píxel como borde si la **pendiente del cruce con cero es mayor que u_b** .

Máscara Laplace

0	-1	0
-1	4	-1
0	-1	0

Laplaciano del Gaussiano (LOG/Marr-Hildreth)

- **Concepto:** Primero se aplica un Filtro Gaussiano ($S = G_\sigma * I$), y luego se calcula el Laplaciano del resultado (ΔS).
- **Aporte Clave:** Es equivalente a calcular el Laplaciano de la Gaussiana (ΔG_σ) y convolucionarlo con la imagen ($\Delta S = \Delta G_\sigma * I$).
- **Máscara LOG:** Se construye con la fórmula:

$$\Delta G_\sigma(x, y) = \frac{1}{2\pi\sigma^3} e^{-\frac{x^2+y^2}{2\sigma^2}} \left(\frac{x^2 + y^2}{\sigma^2} - 2 \right)$$

- **Tamaño:** El tamaño de la máscara debe ser al menos $n = 4\sigma + 1$.

Laplaciano cruce por símbolo (Caso Gaussiano)

```
1 def aplicar_cruces_simbolos():
2     global imagen_actual, imagen_operativa, laplace_gauss
3     sigma = laplace_gauss if laplace_gauss else 0.0
4     imagen_original = imagen_actual.copy().astype(np.float32)
5
6     if sigma > 0:
7         t = int((4 * sigma) + 1)
8         ax = np.linspace(-(t - 1) / 2, (t - 1) / 2, t)
9         xx, yy = np.meshgrid(ax, ax)
10        r2 = xx**2 + yy**2
11
12        mascara = ((r2 - 2 * sigma**2) / sigma**4) * np.exp(-r2 / (2 *
13 sigma**2))
14        tipo = "Marr Hildreth"
15        set_status(f"Aplicando Laplaciano del Gaussiano con sigma={sigma:.2f}")
```

Cruce por símbolo (Grises)

```
1 def aplicar_cruces(imagen,grises=True):
2     H, W = imagen.shape
3     imagen_cruces = np.zeros_like(imagen, dtype=np.uint8)
4
5     for i in range(H):
6         for j in range(W - 1):
7             if cambio_signo(imagen[i, j], imagen[i, j + 1]):
8                 imagen_cruces[i, j] = 255
9             else:
10                imagen_cruces[i, j] = 0
11
12     for j in range(W):
13         for i in range(H - 1):
14             if cambio_signo(imagen[i, j], imagen[i + 1, j]):
15                 imagen_cruces[i, j] = 255
```

Cruce por símbolo (Color)

```
1 H, W, C = imagen.shape
2 imagen_cruces = np.zeros_like(imagen, dtype=np.uint8)
3 for i in range(H):
4     for j in range(W - 1):
5         for c in range(C):
6             if cambio_signo(imagen[i, j, c], imagen[i, j + 1, c]):
7                 imagen_cruces[i, j, c] = 255
8             else:
9                 imagen_cruces[i, j, c] = 0
10    for j in range(W):
11        for i in range(H - 1):
12            for c in range(C):
13                if cambio_signo(imagen[i, j, c], imagen[i + 1, j, c]):
14                    imagen_cruces[i, j, c] = 255
15    return imagen_cruces
16
```

Cruce por umbral (Grises)

```
1 def aplicar_cruces_umbral(imagen, umbral, grises=True):
2     if grises is True:
3         H,W = imagen.shape[:2]
4         imagen_cruces = np.zeros_like(imagen)
5         for i in range(H - 1):
6             for j in range(W - 1):
7
8             ## Esta de esta forma para verse bien
9             if np.abs(imagen[i, j]+imagen[i, j + 1])> umbral or np.abs(imagen[i, j]
10                 +imagen[i+1, j]) > umbral:
11                 imagen_cruces[i, j] = 255
12             else:
13                 imagen_cruces[i, j] = 0
14
```

Cruce por umbral (Color)

```
1 H,W, C = imagen.shape
2 imagen_cruces = np.zeros_like(imagen)
3     for i in range(H - 1):
4         for j in range(W - 1):
5             for c in range(C):
6                 ## Esta de esta forma para verse bien
7                 if np.abs(imagen[i, j, c]+imagen[i, j + 1, c]) > umbral or np.abs(
8                     imagen[i, j, c]+imagen[i+1, j, c]) > umbral:
9                     imagen_cruces[i, j, c] = 255
10                else:
11                    imagen_cruces[i, j, c] = 0
12
13 imagen_cruces = imagen_cruces.astype(np.uint8)
14 return imagen_cruces
```

Mascara Laplace



Figura: Lena Laplace



Figura: Girl2 Laplace



Figura: Barco Laplace

Mascara Laplace



Figura: Flor Laplace

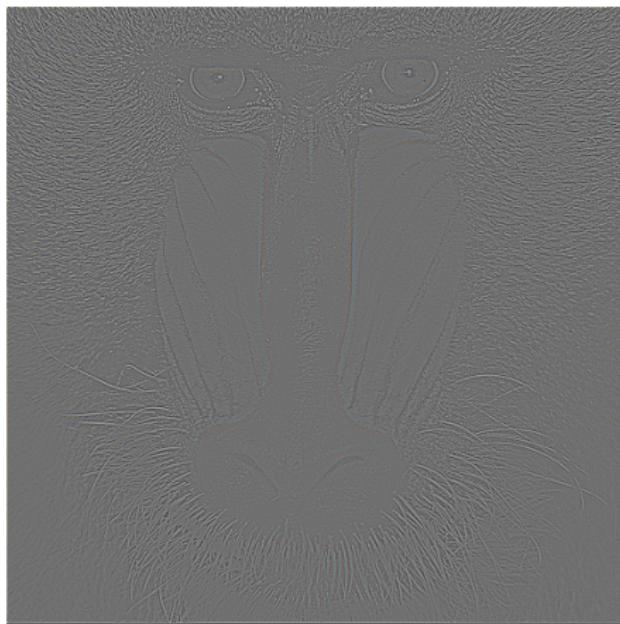


Figura: Baboo Laplace

Mascara Laplace Gaussiana



Figura: Lena Laplace



Figura: Girl2 Laplace



Figura: Barco Laplace

Mascara Laplace Gaussiana



Figura: Flor Laplace



Figura: Baboo Laplace

Cruce por cero VS.

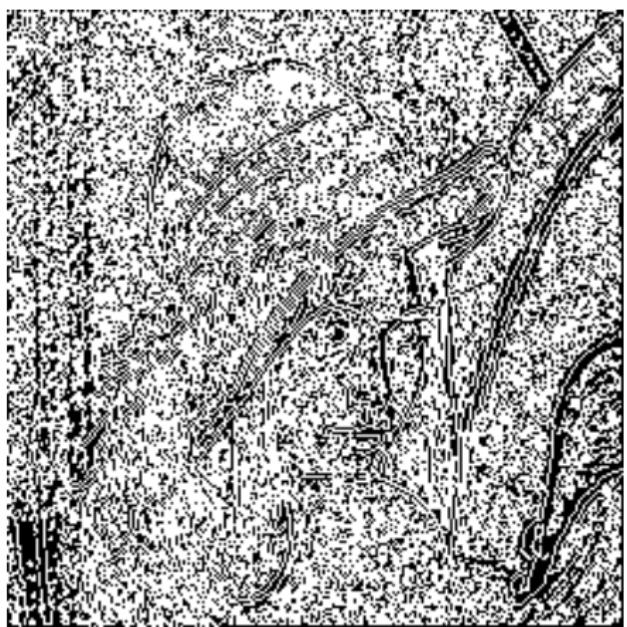


Figura: Lena Laplace



Figura: Lena Laplace Gauss

Cruce por cero VS.

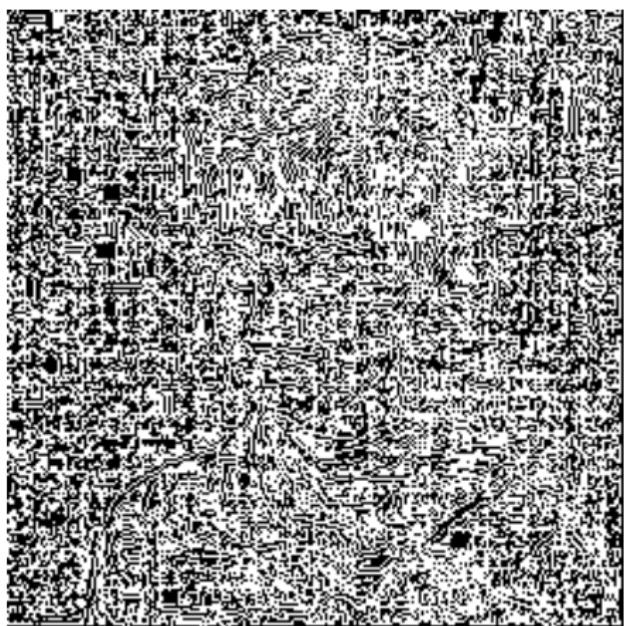


Figura: Girl2 Laplace



Figura: Girl2 Laplace Gauss

Cruce por cero VS.

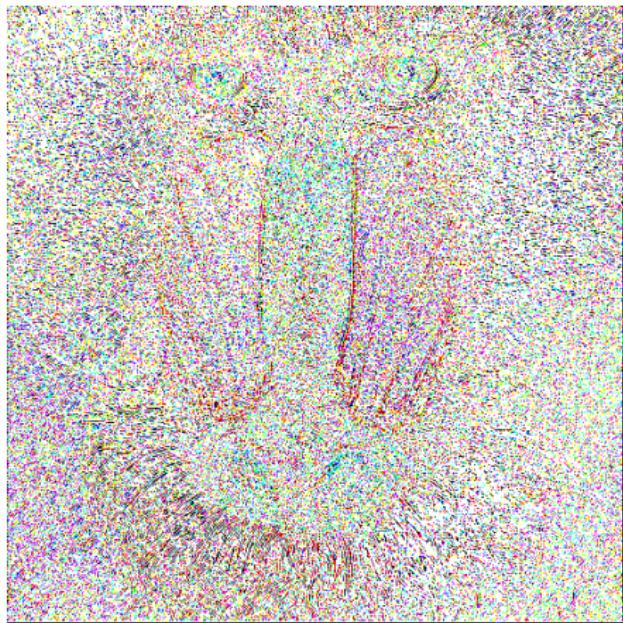


Figura: Baboo Laplace



Figura: Baboo Laplace Gauss

Cruce por cero VS.

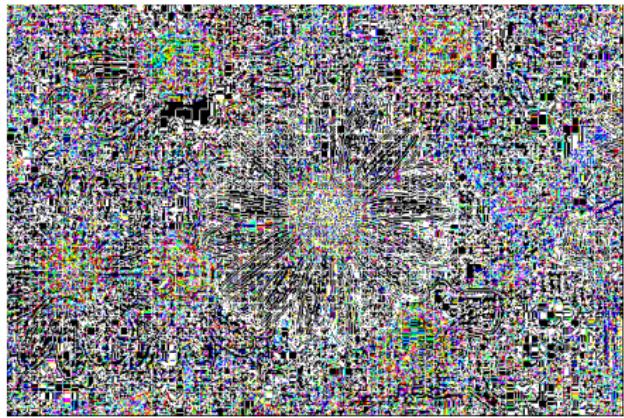


Figura: Flor Laplace

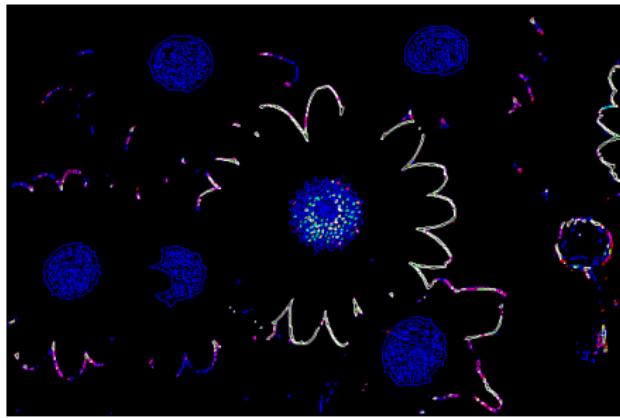


Figura: Flor Laplace gauss

Evaluación pendiente



Figura: Lena umbral 50



Figura: Lena umbral 128



Figura: Lena umbral 200

Evaluación pendiente Gaussiana

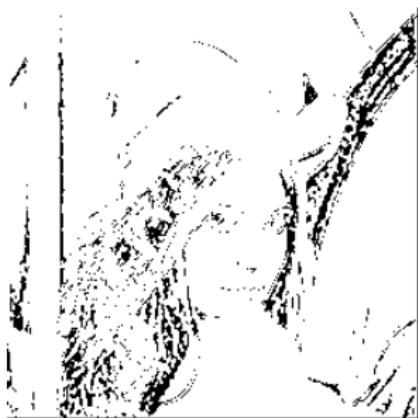


Figura: Lena umbral 50

Figura: Lena umbral 128

Figura: Lena umbral 200

Evaluación pendiente Gaussiana Negativa



Figura: Lena umbral 50



Figura: Lena umbral 128



Figura: Lena umbral 200

Evaluación pendiente

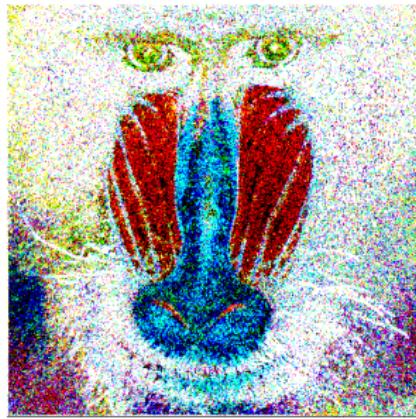


Figura: Baboon umbral 50

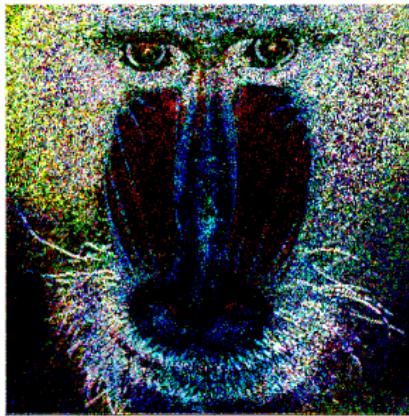


Figura: Baboon umbral
128

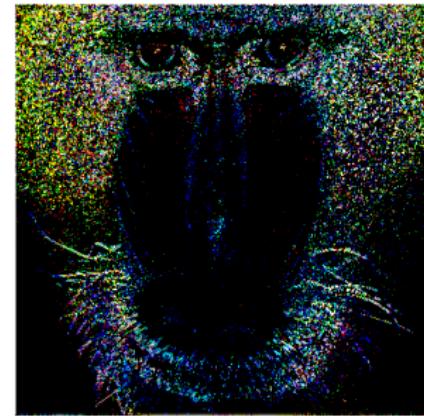


Figura: Baboon umbral
200

Evaluación pendiente Gaussiana



Figura: Baboon umbral 50



Figura: Baboon umbral
128

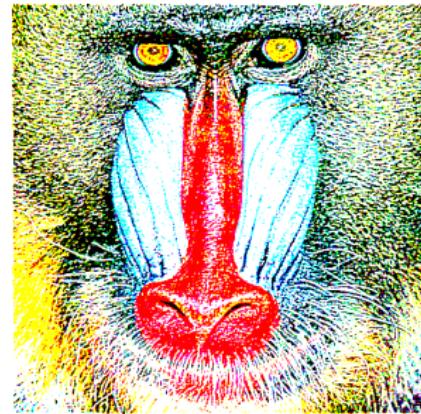


Figura: Baboon umbral
200

Evaluación pendiente Gaussiana Negativa

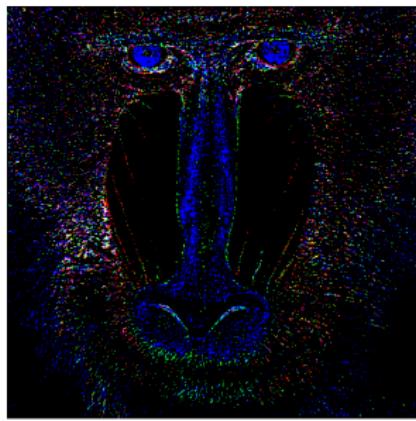


Figura: Baboon umbral 50



Figura: Baboon umbral
128



Figura: Baboon umbral
200

Difusión: Conceptos

Difusión Isotrópica

- Resuelve la ecuación del calor de conducción isotrópica:
 $I_t(x, y, t) = \Delta I(x, y).$
- Es equivalente a la convolución con un filtro Gaussiano:
 $I(x, y, t) = G_t(x, y) * I_0(x, y).$

Difusión Anisotrópica

- **Objetivo:** Mejorar la imagen sin destruir los bordes; suavizar *dentro* de las regiones, pero no *a través* de ellas.
- **Ecuación:** Utiliza un coeficiente de conducción variable $c(x, y, t)$.

$$I_t(x, y, t) = \nabla \cdot c(x, y, t) \nabla I(x, y))$$

- $c \approx 1$ en regiones uniformes y $c \approx 0$ en los bordes. Se estima usando el módulo del gradiente: $c(x, y, t) = g(|\nabla I|)$.

Implementación de Perona-Malik

- Se sigue la forma iterativa:

$$I^{t+1}(x, y) = I^t(x, y) + \lambda (D_N c_N + D_E c_E + D_O c_O + D_S c_S)$$

- D son las diferencias de intensidad (derivadas) en las cuatro direcciones:

$$D_N = I(x - 1, y) - I(x, y)$$

$$D_S = I(x + 1, y) - I(x, y)$$

$$D_O = I(x, y - 1) - I(x, y)$$

$$D_E = I(x, y + 1) - I(x, y)$$

- Los coeficientes c se calculan usando funciones de difusión $g(D)$, que dependen de la magnitud de la derivada
- λ es un parámetro que da importancia a las variaciones; un valor típico es $\lambda = 0,25$.

Funciones de difusión alternativas (Perona-Malik)

Funciones de conducción $g(|\nabla I|)$

- **Detector de Leclerc:**

$$g(|\nabla I|) = \exp\left(-\frac{|\nabla I|^2}{\sigma^2}\right)$$

- **Detector de Lorentz:**

$$g(|\nabla I|) = \frac{1}{1 + \frac{|\nabla I|^2}{\sigma^2}}$$

- En ambos casos, $|\nabla I|$ es la magnitud del gradiente en cada píxel.

Isotropica

```
1 def difusion_isotropica():
2     global t_difusion, imagen_actual, imagen_operativa,
3         lambda_anistropica
4     if len(imagen_original.shape) == 3:
5         imagen_operativa = fc.anistropica(imagen_actual,t_anistropica=
6             t_difusion,lambda_anistropica=lambda_anistropica,grises=False,
7             isotropica=True)
8     else:
9         imagen_operativa = fc.anistropica(  imagen_actual,
10             t_anistropica=t_difusion,lambda_anistropica=lambda_anistropica,
11             isotropica=True)
12
13     mostrar_imagen(imagen_operativa, lblOutputImage)
14     set_status("Difusion isotropica aplicado a la imagen de trabajo.")
```

Anisotrópica

```
1 def diffusion_anistropica():
2     global t_anistropica, imagen_actual, imagen_operativa,
3         lambda_anistropica, sigma_sensibilidad
4
5     if len(imagen_original.shape) == 3:
6         imagen_operativa = fc.anistropica(  imagen_actual,
7             t_anistropica=t_anistropica,lambda_anistropica=lambda_anistropica,
8             sigma_sensibilidad=sigma_sensibilidad,grises=False)
9     else:
10        imagen_operativa = fc.anistropica(imagen_actual,
11            t_anistropica=t_anistropica,lambda_anistropica=lambda_anistropica,
12            sigma_sensibilidad=sigma_sensibilidad)
13
14    mostrar_imagen(imagen_operativa, lblOutputImage)
15    set_status("Difusion anistropica aplicado a la imagen de trabajo."
16 )
17
```

Función general (Isotrópica)

```
1  for t in range(int(t_anistropica)):  
2      for i in range(H):  
3          for j in range(W):  
4              imagen_asintropica[i, j] += derivadas(  
5                  imagen_asintropica, i, j,  
6                  sigma_sensibilidad=sigma_sensibilidad,  
7                  lamba_anistropica=lamba_anistropica,  
8                  isotropica=True  
9              )  
10             # Color  
11            for t in range(int(t_anistropica)):  
12                for i in range(H):  
13                    for j in range(W):  
14                        for c in range(C):  
15                            imagen_asintropica[i, j, c] += derivadas(  
16                                imagen_asintropica[:, :, c],  
17                                i, j,  
18                                sigma_sensibilidad=sigma_sensibilidad,  
19                                lamba_anistropica=lamba_anistropica,  
20                                isotropica=True)
```



Función general (Anisotrópica)

```
1 for t in range(int(t_anistropica)):  
2     for i in range(H):  
3         for j in range(W):  
4             imagen_asintropica[i, j] += derivadas(  
5                 imagen_asintropica, i, j,  
6                 sigma_sensibilidad=sigma_sensibilidad,  
7                 lamba_anistropica=lamba_anistropica  
8             )  
9  
# Color  
10 for t in range(int(t_anistropica)):  
11     for i in range(H):  
12         for j in range(W):  
13             for c in range(C):  
14                 imagen_asintropica[i, j, c] += derivadas(  
15                     imagen_asintropica[:, :, c],  
16                     i, j,  
17                     sigma_sensibilidad=sigma_sensibilidad,  
18                     lamba_anistropica=lamba_anistropica  
19                 )
```

Función Derivadas (Isotrópica)

```
1     H, W = imagen_actual.shape[:2]
2     suma = 0.0
3     centro = imagen_actual[i, j]
4
5     if j + 1 < W: # Norte
6         DN = imagen_actual[i, j + 1] - centro
7         suma += DN
8
9     if i - 1 >= 0: # Este
10        DE = imagen_actual[i - 1, j] - centro
11        suma += DE
12
13    if i + 1 < H: # Oeste
14        DO = imagen_actual[i + 1, j] - centro
15        suma += DO
16
17    if j - 1 >= 0: # Sur
18        DS = imagen_actual[i, j - 1] - centro
19        suma += DS
```

Función Derivadas (Anisotrópica)

```

1     H, W = imagen_actual.shape[:2]
2     suma = 0.0
3     centro = imagen_actual[i, j]
4
5     if j + 1 < W: # Norte
6         DN = imagen_actual[i, j + 1] - centro
7         suma += DN * funcion_g(DN, sigma_sensibilidad, funcion)
8
9     if i - 1 >= 0: # Este
10        DE = imagen_actual[i - 1, j] - centro
11        suma += DE * funcion_g(DE, sigma_sensibilidad, funcion)
12
13    if i + 1 < H: # Oeste
14        DO = imagen_actual[i + 1, j] - centro
15        suma += DO * funcion_g(DO, sigma_sensibilidad, funcion)
16
17    if j - 1 >= 0: # Sur
18        DS = imagen_actual[i, j - 1] - centro
19        suma += DS * funcion_g(DS, sigma_sensibilidad, funcion)
20

```



Función g(D)

```
1 def funcion_g(valor,sigma_sensibilidad, funcion="Leclerc"):  
2     if funcion == "Leclerc":  
3         return Leclerc(valor,sigma_sensibilidad)  
4     elif funcion == "Lorentz":  
5         return Lorentz(valor,sigma_sensibilidad)  
6     else:  
7         raise ValueError("Funcion no valida. Usa Leclerc o Lorentz.")  
8 def Leclerc(valor,sigma_sensibilidad=1):  
9     division = -(valor**2) / (sigma_sensibilidad**2)  
10    resultado = np.exp(division)  
11    return resultado  
12  
13 def Lorentz(valor,sigma_sensibilidad=1):  
14    division = -(valor**2) / (sigma_sensibilidad**2)  
15    resultado = 1 / (division + 1)  
16    return resultado  
17
```

Evaluación Ruido Sal y Pimienta (Isotrópica)



Figura: Ruido 0.01, t = 5



Figura: Ruido 0.01, t = 15



Figura: Ruido 0.01, t = 30

Evaluación Ruido Sal y Pimienta (Anisotrópica)



Figura: Ruido 0.01, $t = 5$



Figura: Ruido 0.01, $t = 15$



Figura: Ruido 0.01, $t = 30$

Evaluación Ruido Sal y Pimienta (Isotrópica)



Figura: Ruido 0.05, t = 5



Figura: Ruido 0.05, t = 15



Figura: Ruido 0.05, t = 30

Evaluación Ruido Sal y Pimienta (Anisotrópica)



Figura: Ruido 0.05, $t = 5$



Figura: Ruido 0.05, $t = 15$



Figura: Ruido 0.05, $t = 30$

Evaluación Ruido Sal y Pimienta (Mediana)



Figura: Ruido 0.01



Figura: Ruido 0.05

Evaluación Ruido Gaussiano (Isotrópica)

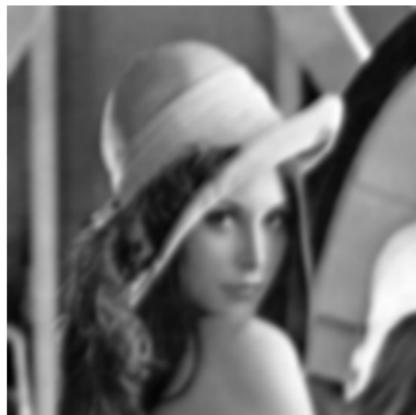


Figura: Ruido= 30 %,
desvío=10, t = 5

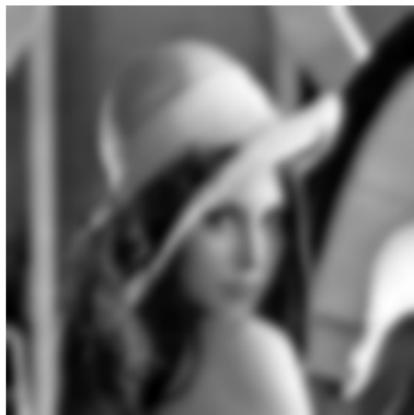


Figura: Ruido= 30 %,
desvío=10, t = 15

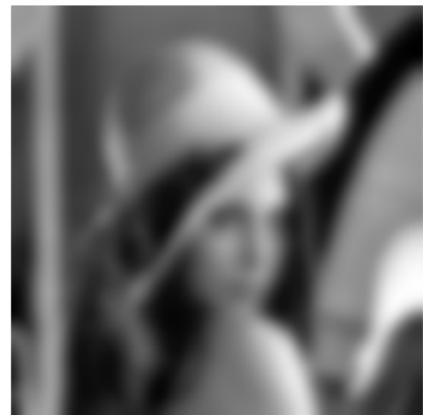


Figura: Ruido= 30 %,
desvío=10, t = 30

Evaluación Gaussiano (Anisotrópica)



Figura: Ruido= 30 %,
desvío=10, t = 5



Figura: Ruido= 30 %,
desvío=10, t = 15

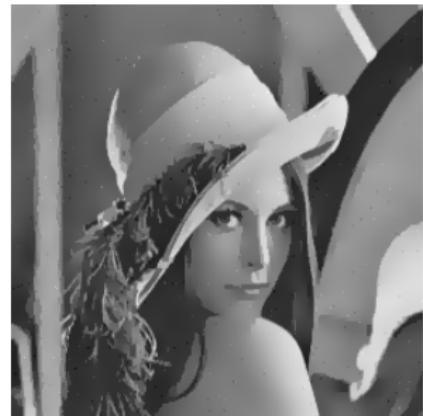


Figura: Ruido= 30 %,
desvío=10, t = 30

Evaluación Ruido Gaussiano (Isotrópica)



Figura: Ruido= 50 %,
desvío=20, t = 5

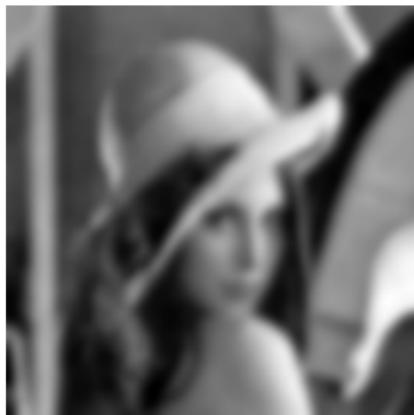


Figura: Ruido= 50 %,
desvío=20, t = 15

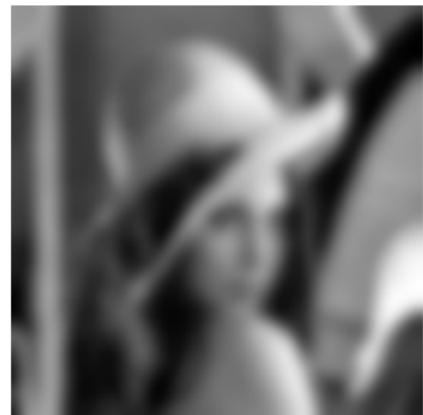


Figura: Ruido= 50 %,
desvío=20, t = 30

Evaluación Ruido Gaussiano (Anisotrópica)



Figura: Ruido= 50 %,
desvío=20, t = 5



Figura: Ruido= 50 %,
desvío=20, t = 15



Figura: Ruido= 50 %,
desvío=20, t = 30

Evaluación Gaussiano (Mediana)

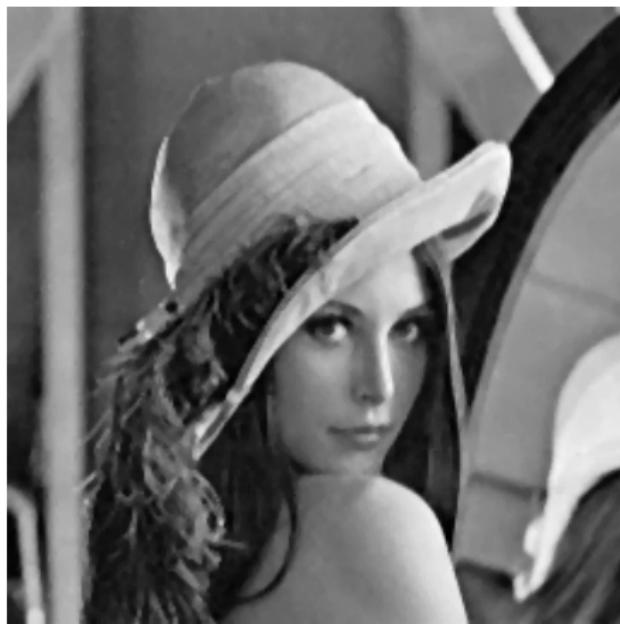


Figura: Ruido= 30 %, desvío=10



Figura: Ruido= 50 %, desvío=20

Descripción y Ecuación

Descripción

- Reemplaza cada píxel por un **promedio pesado** de sus vecinos.
- **Ventajas:** Reduce el ruido y **preserva los bordes**.
- Los pesos dependen de dos factores: la distancia espacial y las diferencias en la intensidad (color).
- Un píxel debe estar cerca y tener un valor de color similar para influenciar.

Ecuación

$$I_{\text{filt}}(x) = \frac{1}{W_x} \sum_{x_i \in \Omega} I(x_i) \underbrace{G_{\sigma_s}(\|x_i - x\|)}_{\text{Componente espacial}} \underbrace{G_{\sigma_r}(\|I(x_i) - I(x)\|)}_{\text{Componente de intensidad}}$$

Observación: La máscara **no es fija**, ya que depende del valor de intensidad del píxel en cuestión, $I(x)$.

Parámetros y Aplicación

$$W_z = \sum_{z_i \in \Omega} G_{\sigma_s}(\|z_i - z\|) G_{\sigma_r}(\|I(z_i) - I(z)\|)$$

- Se utiliza para normalizar los pesos, asegurando que la suma total sea 1.
- Permite que el resultado del filtro conserve la escala original de la imagen.

Parámetros y Consideraciones

- Depende de σ_s (suavizado espacial) y σ_r (suavizado en intensidad).
- Si se incrementa σ_s , el filtro se asemeja al Gaussiano.
- Si se incrementa σ_r , se produce demasiado suavizado, borroneando características.
- **Desventaja:** Tiene un **alto costo computacional**.

Función Bilateral

```
1 def filtro_bilateral():
2     global imagen_actual, imagen_operativa,sigma_color,sigma_espacial
3
4     t = int((2*sigma_espacial )+1)
5     ax = np.linspace(-(t - 1) / 2, (t - 1) / 2, t)
6     xx, yy = np.meshgrid(ax, ax)
7     mascara_gauss_espacial = np.exp(-(xx**2 + yy**2) / (2 *
8         sigma_espacial**2))
9
9     imagen_original = imagen_actual.copy().astype(np.float32)
10
11    if len(imagen_original.shape) == 3:
12        gauss_imagen = fc.mascara(imagen_original,
13            mascara_gauss_espacial, "Gaussiano Color", grises=False,
14            estandarizar=True,sigma_color = sigma_color)
15    else:
16        gauss_imagen = fc.mascara(imagen_original,
17            mascara_gauss_espacial, "Gaussiano Color", grises=True,
18            estandarizar=True,sigma_color = sigma_color)
```



Aplicación Máscara Bilateral

```
1 elif tipo == "Gaussiano Color":  
2     suma_pesos = 0  
3     for i in range(-ancho, ancho + 1):  
4         for j in range(-ancho, ancho + 1):  
5             x, y = pixel[0] + i, pixel[1] + j  
6             if 0 <= x < imagen.shape[0] and 0 <= y < imagen.shape[1]:  
7                 valor_central = imagen[pixel[0], pixel[1]]  
8                 valor_vecino = imagen[x, y]  
9  
10                diff = valor_vecino - valor_central  
11                peso_color = np.exp(-(diff**2) / (2 * sigma_color**2))  
12  
13                peso_total = mascara[i + ancho, j + ancho] *  
peso_color  
14  
15                nuevo_pixel += valor_vecino * peso_total  
16                suma_pesos += peso_total  
17            if suma_pesos != 0:  
18                nuevo_pixel /= suma_pesos
```



Evaluación Gaussiano (Bilateral)

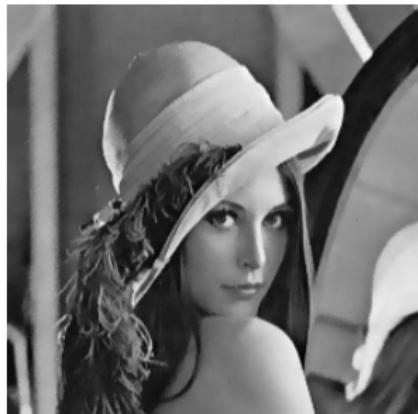


Figura: Ruido= 30 %,
desvío=10, $\sigma_s= 3$ $\sigma_r=15$



Figura: Ruido= 30 %,
desvío=10, $\sigma_s= 5$ $\sigma_r=30$

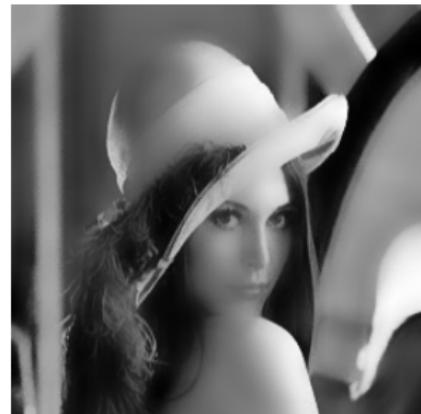


Figura: Ruido= 30 %,
desvío=10, $\sigma_s= 9$ $\sigma_r=60$

Evaluación Gaussiano (Anisotrópica)



Figura: Ruido= 30 %,
desvío=10, t = 5



Figura: Ruido= 30 %,
desvío=10, t = 15



Figura: Ruido= 30 %,
desvío=10, t = 30

Evaluación Gaussiano (Bilateral)



Figura: Ruido= 50 %,
desvío=20, $\sigma_s= 3$ $\sigma_r=15$



Figura: Ruido= 50 %,
desvío=20, $\sigma_s= 5$ $\sigma_r=30$

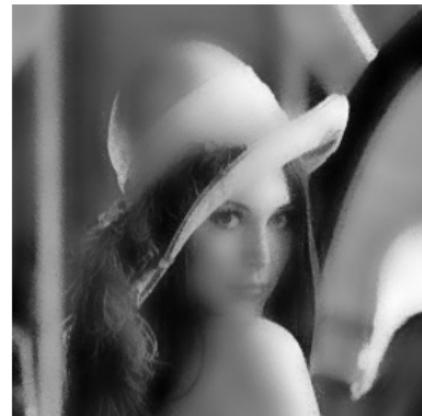


Figura: Ruido= 50 %,
desvío=20, $\sigma_s= 9$ $\sigma_r=60$

Evaluación Ruido Gaussiano (Anisotrópica)



Figura: Ruido= 50 %,
desvío=20, t = 5



Figura: Ruido= 50 %,
desvío=20, t = 15



Figura: Ruido= 50 %,
desvío=20, t = 30

Evaluación Ruido Sal y Pimienta (Bilateral)



Figura: Ruido= 0.01, $\sigma_s= 3 \sigma_r=15$



Figura: Ruido= 0.01, $\sigma_s= 5 \sigma_r=30$



Figura: Ruido= 0.01, $\sigma_s= 9 \sigma_r=60$

Evaluación Ruido Sal y Pimienta (Anisotrópica)



Figura: Ruido 0.01, t = 5



Figura: Ruido 0.01, t = 15



Figura: Ruido 0.01, t = 30

Evaluación Ruido Sal y Pimienta (Bilateral)



Figura: Ruido= 0.05, $\sigma_s= 3 \sigma_r=15$



Figura: Ruido= 0.05, $\sigma_s= 5 \sigma_r=30$



Figura: Ruido= 0.05, $\sigma_s= 9 \sigma_r=60$

Evaluación Ruido Sal y Pimienta (Anisotrópica)



Figura: Ruido 0.05, t = 5



Figura: Ruido 0.05, t = 15



Figura: Ruido 0.05, t = 30

Umbralización Óptima Iterativa

- La umbralización busca un valor T para binarizar la imagen, separando el fondo del objeto de interés.

Algoritmo de Búsqueda Iterativa

- ① Seleccionar un umbral inicial T .
- ② Umbralizar para obtener dos grupos de píxeles: $G_1(\geq T)$ y $G_2(< T)$.
- ③ Calcular promedios de intensidad m_1 (para G_1) y m_2 (para G_2).
- ④ Tomar un nuevo umbral: $T = \frac{1}{2}(m_1 + m_2)$.
- ⑤ Repetir hasta que la diferencia entre T sucesivos sea menor que un ΔT predefinido.

Método de Umbralización de Otsu

- **Objetivo:** Calcular el umbral t que logre la **mínima dispersión** de niveles de gris dentro de cada clase (C_1 y C_2) y la **máxima dispersión** entre clases.
- El método maximiza la varianza entre clases ($\sigma_B^2(t)$).
- **Umbral óptimo** (t^*):

$$t^* = \arg \max_{0 \leq t \leq L-1} \sigma_B^2(t)$$

- **Ventajas:** Es totalmente automático y resistente al ruido.

Pasos del Algoritmo: Probabilidades y Clases

Definiciones iniciales: L niveles de gris; f_i es la cantidad de píxeles con nivel i ; N es la cantidad total de píxeles.

- ① **Paso 1: Histograma Normalizado (p_i) :** La probabilidad de ocurrencia del nivel de gris i :

$$p_i = \frac{f_i}{N}, \quad i = 0, \dots, L - 1$$

- ② **Clases definidas por el umbral t :**

- C_1 : Píxeles con intensidad en $[0, \dots, t]$.
- C_2 : Píxeles con intensidad en $[t + 1, \dots, L - 1]$.

- ③ **Paso 2: Probabilidad Acumulada de la Clase C_1 ($P_1(t)$):** Probabilidad de que un píxel pertenezca a la clase C_1 :

$$P_1(t) = \sum_{i=0}^t p_i, \quad t = 0, 1, \dots, L - 1$$

Pasos del Algoritmo: Promedios y Varianza

- ④ **Paso 3: Promedio Ponderado Acumulado ($m(t)$):**

$$m(t) = \sum_{i=0}^t ip_i, \quad t = 0, \dots, L-1$$

- ⑤ **Paso 4: Promedio Ponderado Global (m_G):**

$$m_G = \sum_{i=0}^{L-1} ip_i$$

- ⑥ **Paso 5: Varianza Entre Clases ($\sigma_B^2(t)$):** Mide la dispersión entre C_1 y C_2 .

$$\sigma_B^2(t) = \frac{[m_G P_1(t) - m(t)]^2}{P_1(t)(1 - P_1(t))} \quad t = 0, \dots, L-1$$

Una vez hallado t^* , la imagen se binariza

Segmentación en Color por Bandas

- Utiliza técnicas de umbralización para segmentar una imagen RGB en hasta 8 regiones (como máximo).

Pasos del Método

- ① Buscar el umbral óptimo (ej., Otsu) para cada banda (R, G, B), obteniendo r_{opt} , g_{opt} , b_{opt} .
- ② Umbralizar las tres bandas por separado.
- ③ Concatenar las bandas umbralizadas, resultando en una imagen donde cada píxel puede tomar uno de los 8 posibles colores primarios o secundarios (Negro, Rojo, Amarillo, Magenta, Verde, Cyan, Azul, Blanco).

Función Umbralización iterativa

```
1 def umbralizacion_iterativa():
2     global imagen_actual, imagen_operativa,t_inicial,t_predefinido
3     imagen_original = imagen_actual.copy().astype(np.float32)
4
5     if len(imagen_original.shape) == 3:
6         umbral_it, umbral,iteraciones = fc.umbralizacion_iterativa(
7             imagen_original,t_inicial,t_predefinido,grises=False)
8     else:
9         umbral_it, umbral,iteraciones = fc.umbralizacion_iterativa(
10            imagen_original,t_inicial,t_predefinido,grises=True)
11
12     imagen_operativa = umbral_it
13     mostrar_imagen(imagen_operativa, lblOutputImage)
14     set_status(f"Umbral iterativo aplicado a la imagen de trabajo.
Umbral/es aplicados {umbral}. Con {iteraciones} iteraciones")
```

Función Umbralización Otsu

```
1 def umbralizacion_Otsu():
2     global imagen_actual, imagen_operativa
3
4     imagen_original = imagen_actual.copy().astype(np.float32)
5
6     if len(imagen_original.shape) == 3:
7         umbral_it, umbral = fc.umbralizacion_Otsu(imagen_original,
8             grises=False)
9     else:
10        umbral_it, umbral = fc.umbralizacion_Otsu(imagen_original,
11            grises=True)
12
13     imagen_operativa = umbral_it
14     mostrar_imagen(imagen_operativa, lblOutputImage)
15     set_status(f"Umbral iterativo aplicado a la imagen de trabajo.
16     Umbral/es aplicados {umbral}")
```

Función umbral iterativo

```
1 t = t_inicial
2 iteracion = 0
3 while True:
4     iteracion += 1
5     G1 = imagen_original[imagen_original > t]
6     G2 = imagen_original[imagen_original <= t]
7
8     if G1.size == 0 or G2.size == 0:
9         break
10
11    M1, M2 = G1.mean(), G2.mean()
12    t_nuevo = (M1 + M2) / 2
13
14    if abs(t_nuevo - t) < t_predefinido:
15        t = int(round(t_nuevo))
16        break
17    t = t_nuevo
18    imagen_binaria = funcion_umbral_preview(imagen_original, t, grises=
19    True, estandarizar=True)
20    return imagen_binaria + iteracion
```



Función umbral iterativo (Color)

```
1 canales = cv2.split(imagen_original)
2 umbrales = []
3 iteraciones = []
4 for canal in canales:
5     t = t_inicial
6     iteracion = 0
7     while True:
8         iteracion += 1
9         G1 = canal[canal > t]
10        G2 = canal[canal <= t]
11        M1, M2 = G1.mean(), G2.mean()
12        t_nuevo = (M1 + M2) / 2
13        if abs(t_nuevo - t) < t_predefinido:
14            t = int(round(t_nuevo))
15            break
16        t = t_nuevo
17        umbrales.append(t)
18        iteraciones.append(iteracion)
19
20
```

Umbral Iterativo (Grises)



Figura: Iterativo. Umbral: 86, iteración: 8



Figura: Otsu. Umbral: 86

Umbral Iterativo (Grises)



Figura: Iterativo. Umbral: 86, iteración: 6



Figura: Otsu. Umbral: 87

Umbral Iterativo (Color)



Figura: Iterativo. Umbral: 130;129;144,
iteración: 4;3;7

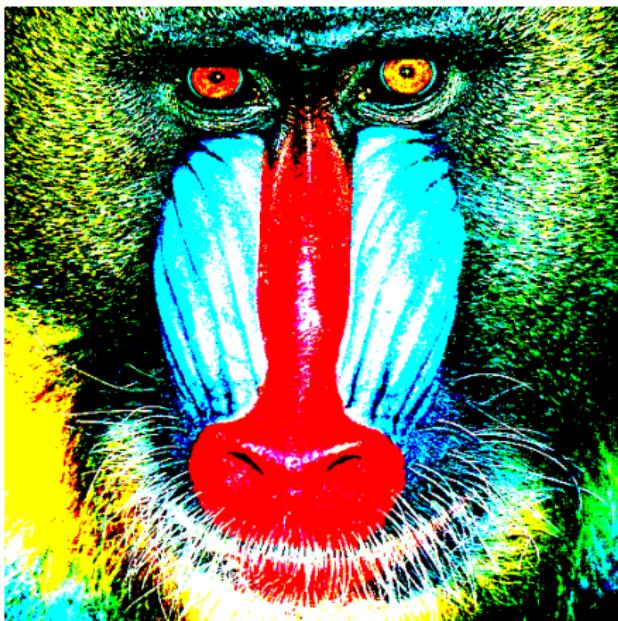


Figura: Otsu. Umbral: 131;129;145

Umbral Iterativo (Color)

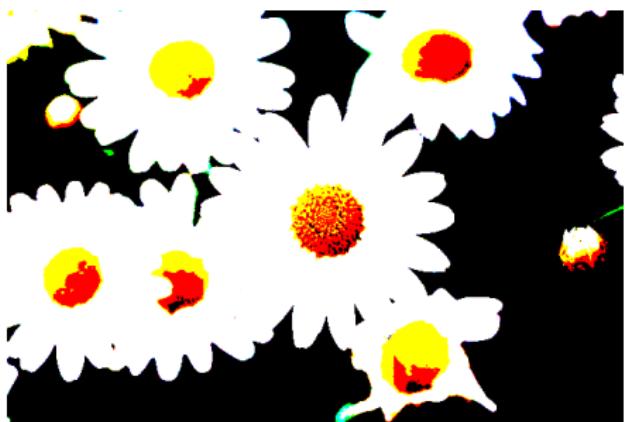


Figura: Iterativo. Umbral: 120;135;122,
iteración: 4;4;4



Figura: Otsu. Umbral: 120;134,120

Umbral Iterativo (Grises) Gauss



Figura: Iterativo. Ruido: 30 %, desvío: 10. Umbral: 86, iteración: 8



Figura: Otsu. Ruido: 30 %, desvío: 10. Umbral: 86

Umbral Iterativo (Grises) Gauss



Figura: Iterativo. Ruido: 50 %, desvío: 20. Umbral: 111, iteración: 8



Figura: Otsu. Ruido: 50 %, desvío: 20. Umbral: 111

Umbral Iterativo (Grises) Sal y Pimienta



Figura: Iterativo. Ruido: 0.01. Umbral: 88, iteración: 6



Figura: Otsu. Ruido: 0.01. Umbral: 84

Umbral Iterativo (Grises) Sal y Pimienta



Figura: Iterativo. Ruido: 0.05. Umbral: 95, iteración: 8



Figura: Otsu. Ruido: 0.05. Umbral: 73

Umbral Iterativo (Color) Gauss

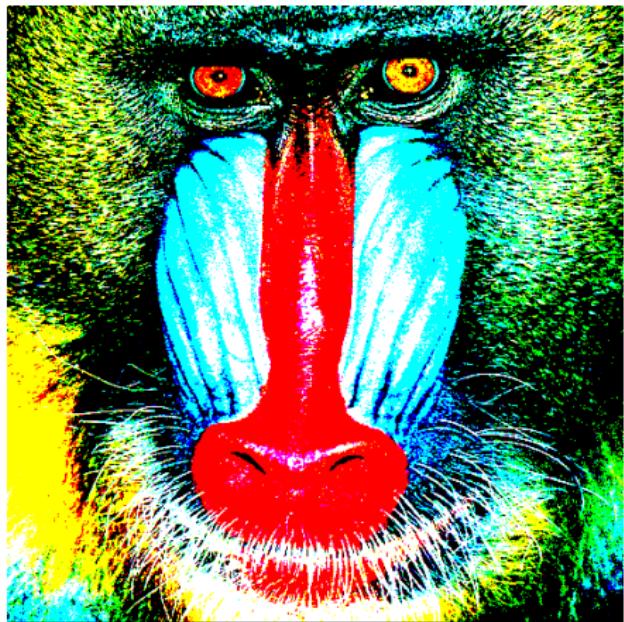


Figura: Iterativo. Ruido: 30 %, desvío: 10. Umbral: 126;125;138, iteración: 3;4;6

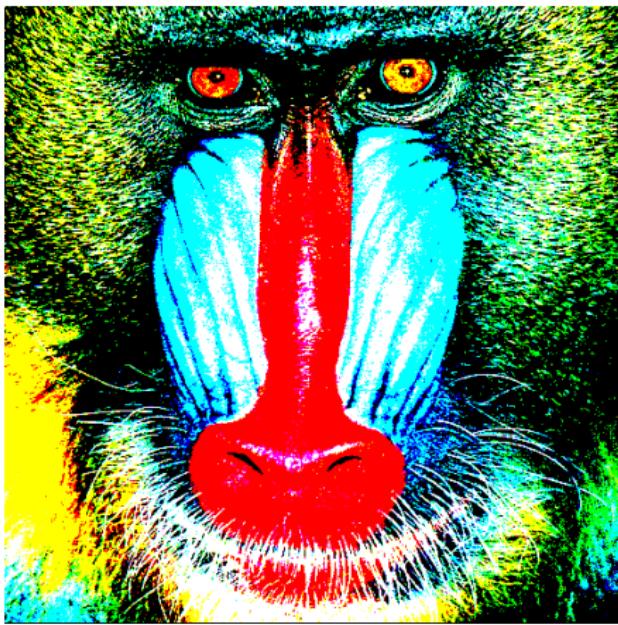


Figura: Otsu. Ruido: 30 %, desvío: 10. Umbral: 127;126;139

Umbral Iterativo (Color) Gauss

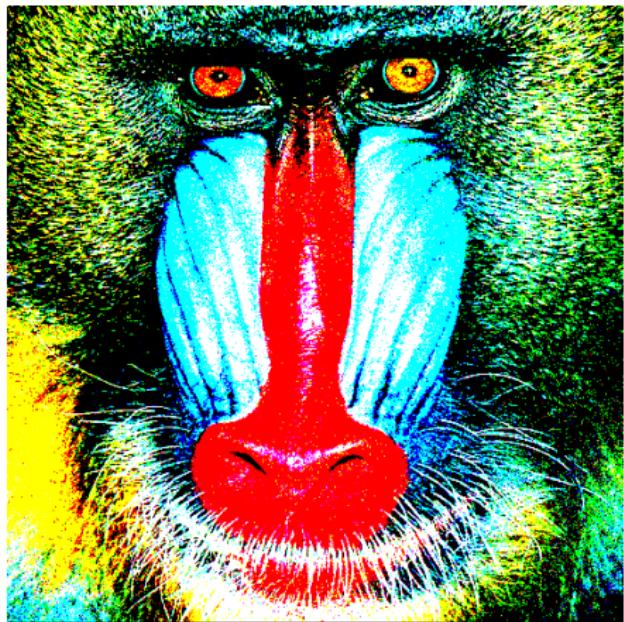


Figura: Iterativo. Ruido: 50 %, desvío: 20. Umbral: 126;125;135, iteración: 3;4;6

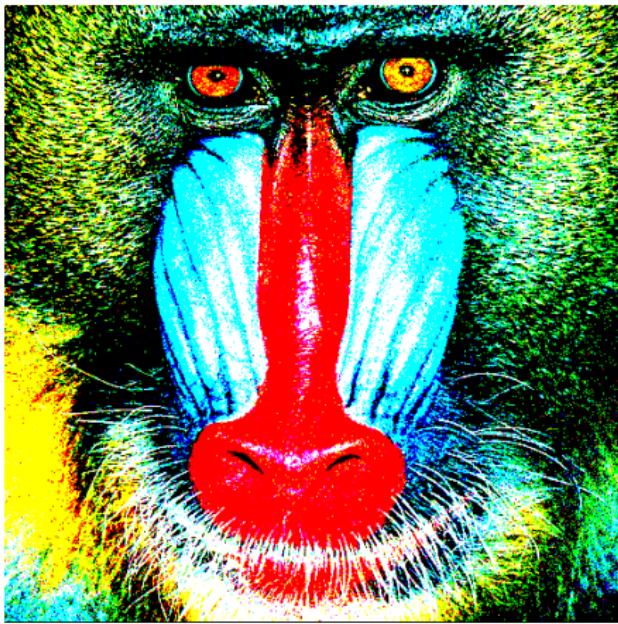


Figura: Otsu. Ruido: 50 %, desvío: 20. Umbral: 126;125;136

Umbral Iterativo (Color) Sal y Pimienta



Figura: Iterativo. Ruido: 0.01. Umbral: 131;129;144, iteración: 4;3;7



Figura: Otsu. Ruido: 0.01. Umbral: 124;124;138

Umbral Iterativo (Color) Sal y Pimienta



Figura: Iterativo. Ruido: 0.05. Umbral: 134;129;143, iteración: 5;3;8

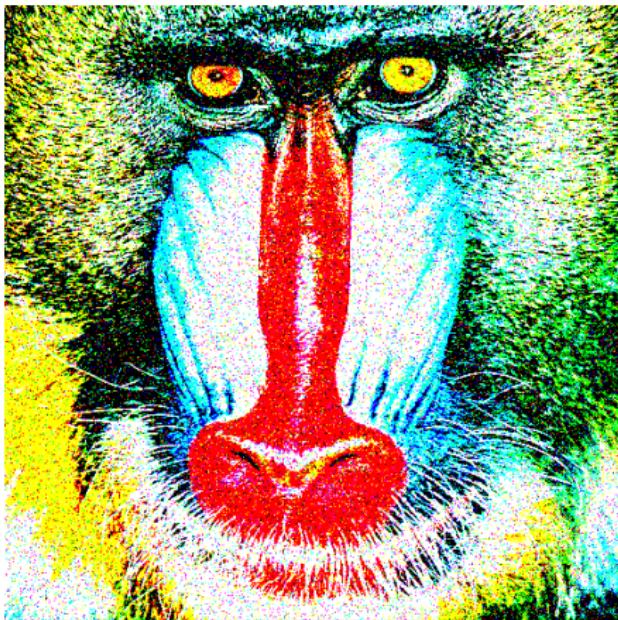


Figura: Otsu. Ruido: 0.05. Umbral: 104;106;116