CENTRO DE ENSEÑANZA TÉCNICA INDUSTRIAL

INGENIERÍA EN MECATRÓNICA

Visión artificial

Práctica 8:
Detección de bordes

Alumna:
Vanessa Aguirre Diaz

Registro:
22310274

Fecha:
01 de junio del 2025

Objetivo:

Dejar en la imagen solamente los bordes que deseamos y saber cuál es el mejor método.

Desarrollo teórico:

En visión artificial, los bordes son las zonas de una imagen donde hay cambios bruscos en la intensidad (también llamada luminancia o brillo). Estos cambios suelen indicar límites entre objetos, formas, texturas o regiones distintas dentro de la escena.

¿Por qué son importantes los bordes?

Los bordes representan la estructura esencial de una imagen. A partir de ellos, un sistema de visión artificial puede:

- Detectar contornos de objetos.
- Separar figuras del fondo.
- Reconocer formas y patrones.
- Medir, contar o rastrear objetos en movimiento.
- Reducir información sin perder lo esencial de la escena.

Los detectores de bordes en esta práctica son:

Laplaciano

¿Qué es?

Es un operador de segundo orden, lo que significa que analiza la variación del gradiente en una imagen. Detecta bordes sin importar su dirección (horizontal, vertical o diagonal).

¿Cómo funciona?

- Calcula la segunda derivada de la imagen.
- Busca los lugares donde la intensidad cambia bruscamente en cualquier dirección.
- Es sensible al ruido, porque amplifica pequeñas variaciones.

¿Para qué se usa?

- Para detectar bordes generales y contornos suaves.
- Muy útil cuando no necesitas distinguir dirección.

¿Cómo mejorarlo?

- Aplicar un filtro Gaussiano antes de usarlo para reducir el ruido.

Sobel X

¿Qué es?

Es un operador de primer orden que detecta bordes en dirección horizontal (es decir, cambios verticales en la imagen).

¿Cómo funciona?

- Calcula el gradiente horizontal (derivada en X).
- Detecta los bordes verticales (como columnas, bordes de objetos de pie).

```
sobelx = cv2.Sobel(frame, cv2.CV_64F, 1, 0, ksize=5)
```

- 1, 0: indica que detecta en X (horizontal).
- ksize: tamaño del filtro (más grande → más suavizado, pero menos detalle fino).

Sobel Y

¿Qué es?

Otro operador Sobel que detecta bordes en dirección vertical (cambios horizontales en la imagen).

¿Cómo funciona?

- Calcula el gradiente vertical (derivada en Y).
- Detecta los bordes horizontales (como líneas de piso, techos, horizontes).

```
sobely = cv2.Sobel(frame, cv2.CV_64F, 0, 1, ksize=5)
```

- 0, 1: indica que detecta en Y (vertical).

Canny

¿Qué es?

El detector de bordes más preciso y completo de OpenCV. Utiliza una combinación de técnicas para obtener bordes nítidos, continuos y sin ruido.

¿Cómo funciona?

1. Aplica un filtro Gaussiano (reduce el ruido).
2. Calcula gradientes de Sobel.
3. No máxima suppression: afina los bordes.

4. Hysteresis: conserva solo los bordes entre dos umbrales (suaves y fuertes).

edges = cv2.Canny(frame, 100, 200)

- 100: umbral mínimo (bordes débiles).
- 200: umbral máximo (bordes fuertes).
- Solo se muestran los bordes fuertes o los conectados a ellos.

¿Cómo obtener una mejor detección de bordes?

1. Reducir el ruido

Aplica un filtro Gaussiano:

blurred = cv2.GaussianBlur(frame, (5, 5), 0)

2. Escoger los umbrales adecuados (especialmente en Canny)

- Experimenta con distintos valores de umbral:

edges = cv2.Canny(blurred, 50, 150) # Mejores resultados en muchas imágenes reales

3. Usar imágenes en escala de grises

Muchos algoritmos funcionan mejor con una sola canal de color:

gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

4. Combinar detectores

Puedes combinar Sobel X e Y para obtener bordes más completos:

sobel_combined = cv2.magnitude(sobelx, sobely)

Desarrollo práctico

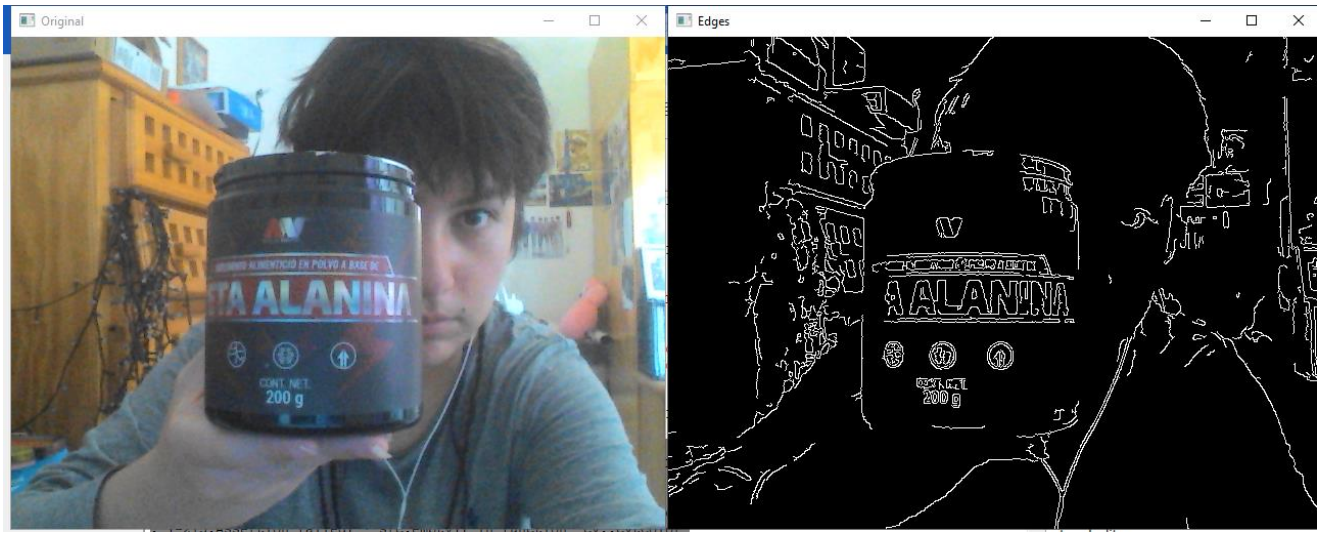
Partiremos de 2 códigos proporcionados por el tutorial:

Detección de bordes con Canny:

```
# Practica 8: Deteccion de bordes Pt.2: Canny

import cv2                  # Importa la librería OpenCV, que se usa para procesar imágenes y video.
import numpy as np          # Importa NumPy para manipular matrices (arrays), útil para definir rangos de color.

cap = cv2.VideoCapture(0)   # Inicializa la captura de video desde la cámara con índice 0 (cámara principal).

while(1):                   # Inicia un bucle infinito para procesar video en tiempo real.

    # Captura un frame de la cámara
    _, frame = cap.read()   # 'frame' contiene la imagen capturada. El primer valor se ignora con '_'.
```

```
    # Convierte la imagen de formato BGR (por defecto en OpenCV) a HSV (tono, saturación, valor).
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    # Define un rango inferior del color (valores de tono, saturación y brillo).
    lower_red = np.array([30,150,50])     # Tono bajo para detectar un color rojizo/naranja.
    upper_red = np.array([255,255,180])   # Tono alto para limitar el rango.

    # Crea una máscara en blanco y negro: blanco para pixeles dentro del rango, negro para los demás.
    mask = cv2.inRange(hsv, lower_red, upper_red)

    # Aplica la máscara a la imagen original. Solo se mantienen los píxeles en el rango de color definido.
    res = cv2.bitwise_and(frame, frame, mask=mask)

    # Muestra la imagen original capturada de la cámara.
    cv2.imshow('Original', frame)

    # Aplica el detector de bordes de Canny. Detecta contornos en la imagen original.
    edges = cv2.Canny(frame, 100, 200)
    # - 100: umbral inferior para detectar bordes.
    # - 200: umbral superior.

    # Muestra la imagen con los bordes detectados.
    cv2.imshow('Edges', edges)

    # Espera 5 milisegundos a que se presione una tecla.
    # Si se presiona la tecla 'Esc' (código ASCII 27), se sale del bucle.
    k = cv2.waitKey(5) & 0xFF
    if k == 27:
        break

# Cierra todas las ventanas que fueron abiertas con imshow()
cv2.destroyAllWindows()

# Libera la cámara para que pueda ser usada por otros programas.
cap.release()
```

Obteniendo el siguiente resultado:

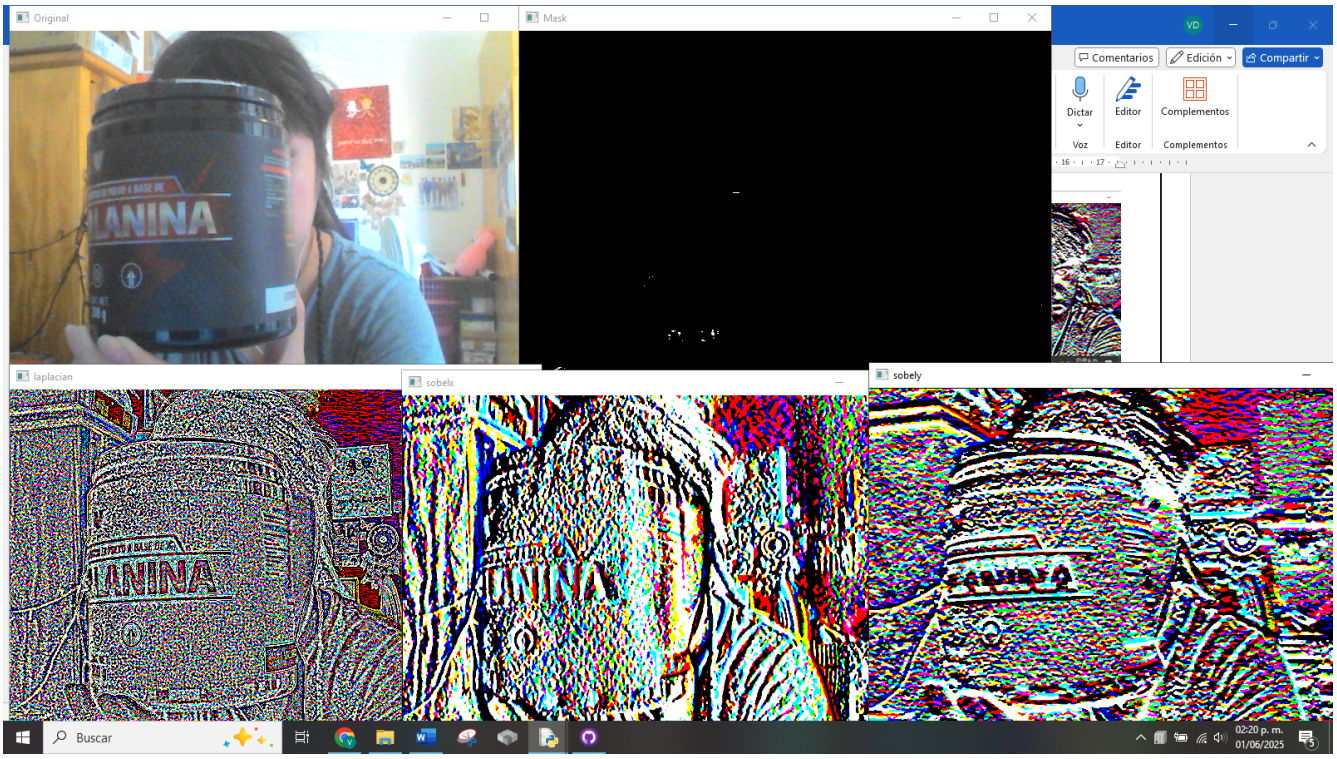


Y tenemos sobel x, sobel y y laplaciano en este otro código:

```
# Practica 8: Deteccion de bordes Pt.1
# Laplaciano, Sobelx, Sobely

import cv2                  # Importa la librería OpenCV para procesamiento de imágenes y video.
import numpy as np          # Importa NumPy para operaciones numéricas con arrays, útil para manejar matrices de imágenes.

cap = cv2.VideoCapture(0)    # Crea un objeto que captura video desde la cámara con índice 1.
                             # (Puede que necesites usar 0 en algunas computadoras si la cámara principal es la que deseas.)

while(1):                    # Bucle infinito que se ejecuta continuamente para capturar y procesar cada frame.

    # Captura un frame del video
    _, frame = cap.read()    # Lee un frame de la cámara. `_` ignora el valor de retorno de éxito; `frame` es la imagen capturada.

    # Convierte la imagen capturada de BGR (azul, verde, rojo) a HSV (tono, saturación, valor)
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    # Define un rango inferior y superior para un color en el espacio HSV (aquí parece ser un color rojo anaranjado)
    lower_red = np.array([30,150,50])     # Límite inferior del color que se quiere detectar (tono claro).
    upper_red = np.array([255,255,180])   # Límite superior del color.

    # Crea una máscara que deja pasar solo los pixeles dentro del rango de color definido
    mask = cv2.inRange(hsv, lower_red, upper_red)

    # Aplica la máscara sobre la imagen original para resaltar solo las áreas con el color seleccionado
    res = cv2.bitwise_and(frame, frame, mask=mask)

    # Aplica el filtro de Laplace para detectar bordes (derivadas de segundo orden)
    laplacian = cv2.Laplacian(frame, cv2.CV_64F)  # Usa precisión de 64 bits para evitar pérdida de información.

    # Aplica el filtro Sobel en dirección X (horizontal)
    sobelx = cv2.Sobel(frame, cv2.CV_64F, 1, 0, ksize=5)
    # Parámetros:
    # - 1 y 0 indican que se toma la derivada en x, no en y.
    # - ksize=5 es el tamaño del kernel (más grande = más suavizado)

    # Aplica el filtro Sobel en dirección Y (vertical)
    sobely = cv2.Sobel(frame, cv2.CV_64F, 0, 1, ksize=5)

    # Muestra la imagen original
    cv2.imshow('Original', frame)

    # Muestra la máscara en blanco y negro (zonas que cumplen con el color definido)
    cv2.imshow('Mask', mask)

    # Muestra el resultado del filtro de Laplace (bordes detectados)
    cv2.imshow('laplacian', laplacian)

    # Muestra los bordes detectados en dirección horizontal
    cv2.imshow('sobelx', sobelx)
```

```
    # Muestra los bordes detectados en dirección vertical
    cv2.imshow('sobely', sobely)

    # Espera 5 milisegundos por una tecla; si es la tecla Esc (código ASCII 27), se sale del bucle
    k = cv2.waitKey(5) & 0xFF
    if k == 27:
        break

# Cierra todas las ventanas creadas por OpenCV
cv2.destroyAllWindows()

# Libera la cámara para que otros programas puedan usarla
cap.release()
```

Con el que obtenemos este resultado.



Como podemos ver, por el momento, la detección de bordes a través de Canny es el que nos da el mejor resultado, ya que los otros tres tiene mucho ruido debido a que el filtro es muy sensible y detecta bordes hasta en las texturas de los objetos, así que haremos algunas modificaciones a este segundo código para intentar obtener un resultado parecido al Canny.

1. Teniendo en cuenta que el sobel se basa en identificar el cambio de color en la imagen, decidí ecualizar la imagen e incrementarle el contraste a través de la variable contraste_manual, en el que se hicieron pruebas con diferentes valores de contraste.
2. Para obtener ambos ejes (bordes en x y bordes en y), se realizó una operación nand entre el resultado obtenido con la función sobel x y sobel y, con la intención de que solo en los puntos donde ambos coinciden se mantengan en negro y lo demás se haga blanco.

3. Como el resultado final aun presentaba un poco de ruido, se usaron las operaciones morfológicas de la practica pasada de apertura y cierre para intentar eliminarlo.

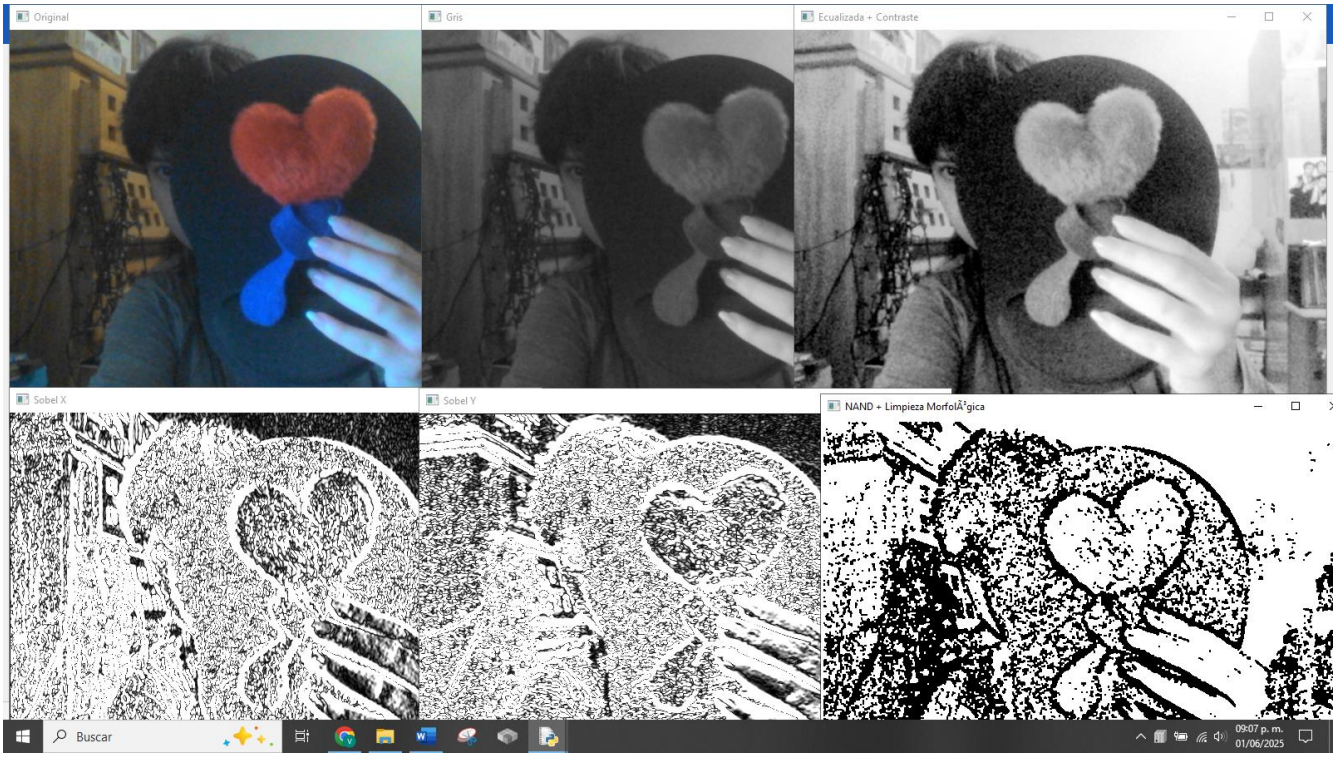
Con esto obtenemos el siguiente código final:

```
# Detección de bordes con ajuste manual de contraste, umbral de Sobel y limpieza morfológica
# Por Vanessa Aguirre

import cv2
import numpy as np

cap = cv2.VideoCapture(0)

# Parámetros ajustables
contraste_manual = 1       # Aumenta o disminuye el contraste (1 = sin cambio)
umbral_sobel = 250         # Umbral para binarizar Sobel X y Y
kernel_morfologico = 6     # Tamaño del kernel para apertura y cierre (debe ser impar y >1)

while True:
    ret, frame = cap.read()
    if not ret:
        continue

    # 1. Mostrar imagen original
    cv2.imshow('Original', frame)

    # 2. Convertir a escala de grises
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    cv2.imshow('Gris', gray)

    # 3. Ecualizar histograma para aumentar contraste general
    equalized = cv2.equalizeHist(gray)

    # 4. Aplicar aumento de contraste manual
    contrasted = cv2.convertScaleAbs(equalized, alpha=contraste_manual, beta=0)
    cv2.imshow('Ecualizada + Contraste', contrasted)

    # 5. Calcular gradientes Sobel X y Y
    sobelx = cv2.Sobel(contrasted, cv2.CV_64F, 1, 0, ksize=5)
    sobely = cv2.Sobel(contrasted, cv2.CV_64F, 0, 1, ksize=5)

    # 6. Convertir a 8 bits
    sobelx_abs = cv2.convertScaleAbs(sobelx)
    sobely_abs = cv2.convertScaleAbs(sobely)

    # 7. Mostrar Sobel X y Y
    cv2.imshow('Sobel X', sobelx_abs)
    cv2.imshow('Sobel Y', sobely_abs)

    # 8. Umbralización
    _, sobelx_bin = cv2.threshold(sobelx_abs, umbral_sobel, 255, cv2.THRESH_BINARY)
    _, sobely_bin = cv2.threshold(sobely_abs, umbral_sobel, 255, cv2.THRESH_BINARY)

    # 9. NAND lógico: NOT(AND)
```

```
    and_result = cv2.bitwise_and(sobelx_bin, sobely_bin)
    nand_result = cv2.bitwise_not(and_result)

    # 10. Crear kernel para operaciones morfológicas
    kernel = np.ones((kernel_morfologico, kernel_morfologico), np.uint8)

    # 11. Aplicar apertura (quita ruido blanco pequeño) y luego cierre (quita puntos negros)
    apertura = cv2.morphologyEx(nand_result, cv2.MORPH_OPEN, kernel)
    cierre = cv2.morphologyEx(apertura, cv2.MORPH_CLOSE, kernel)

    # 12. Mostrar resultado final sin ruido
    cv2.imshow('NAND + Limpieza Morfológica', cierre)

    # 13. Salir con ESC
    if cv2.waitKey(5) & 0xFF == 27:
        break

# Liberar cámara y cerrar ventanas
cap.release()
cv2.destroyAllWindows()
```

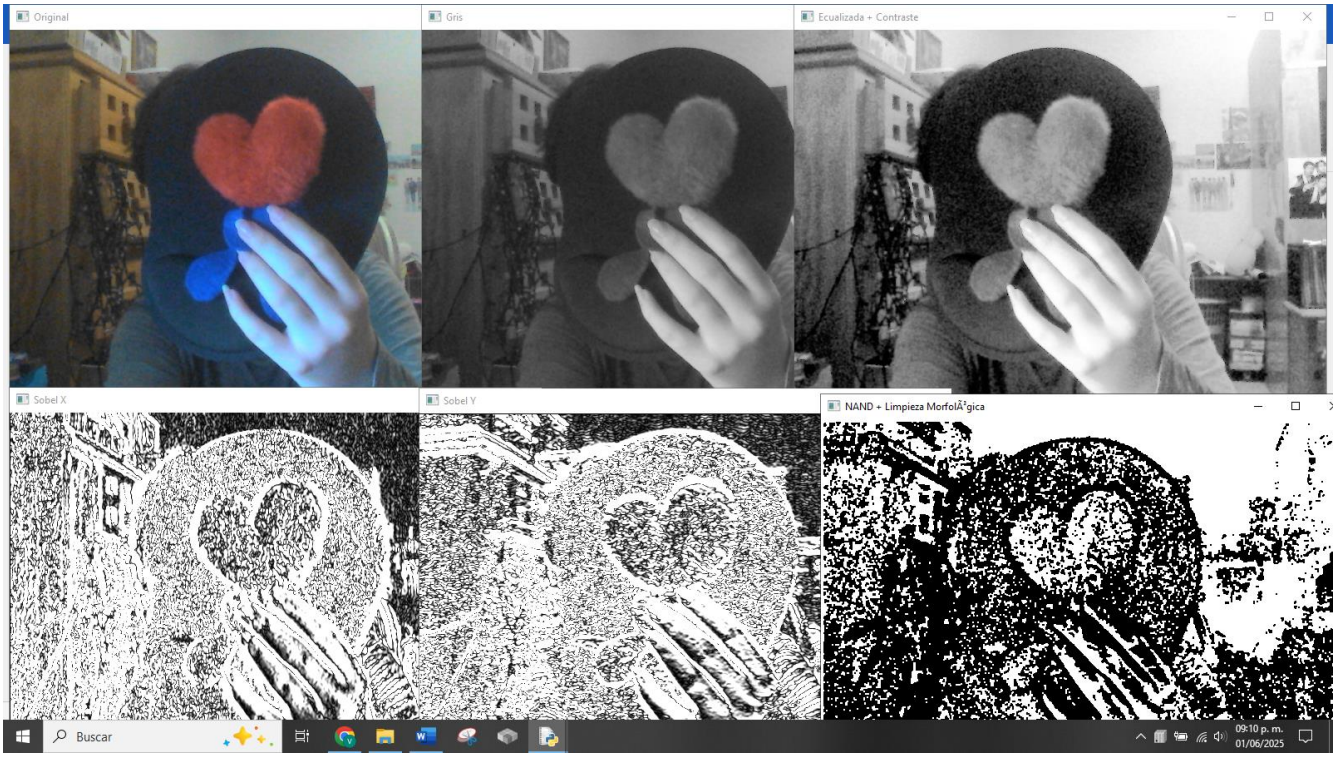
Este es el resultado con el ajuste teniendo en cuanta la iluminación del lugar:

- Contraste = 1
- Umbral del sobel = 250
- Kernel = 2



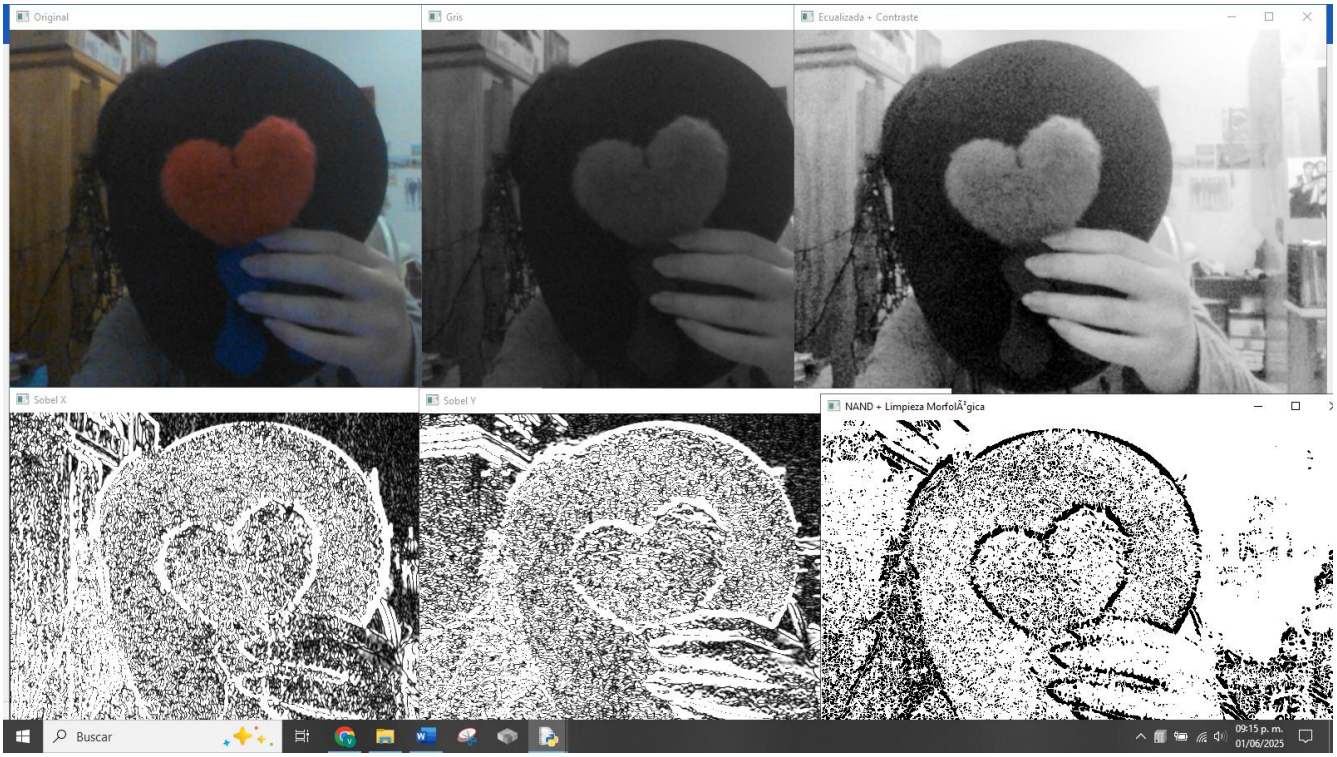
Estos valores, sobre todo el de contraste, deben ajustarse según la iluminación del lugar y la que le esté llegando al objeto, ya que si hay mucha iluminación y se añade mas contraste, algunos bordes pueden perderse.

Ahora veamos qué pasa si movemos el valor de umbral del sobel de 250 a 150. Este es el resultado:

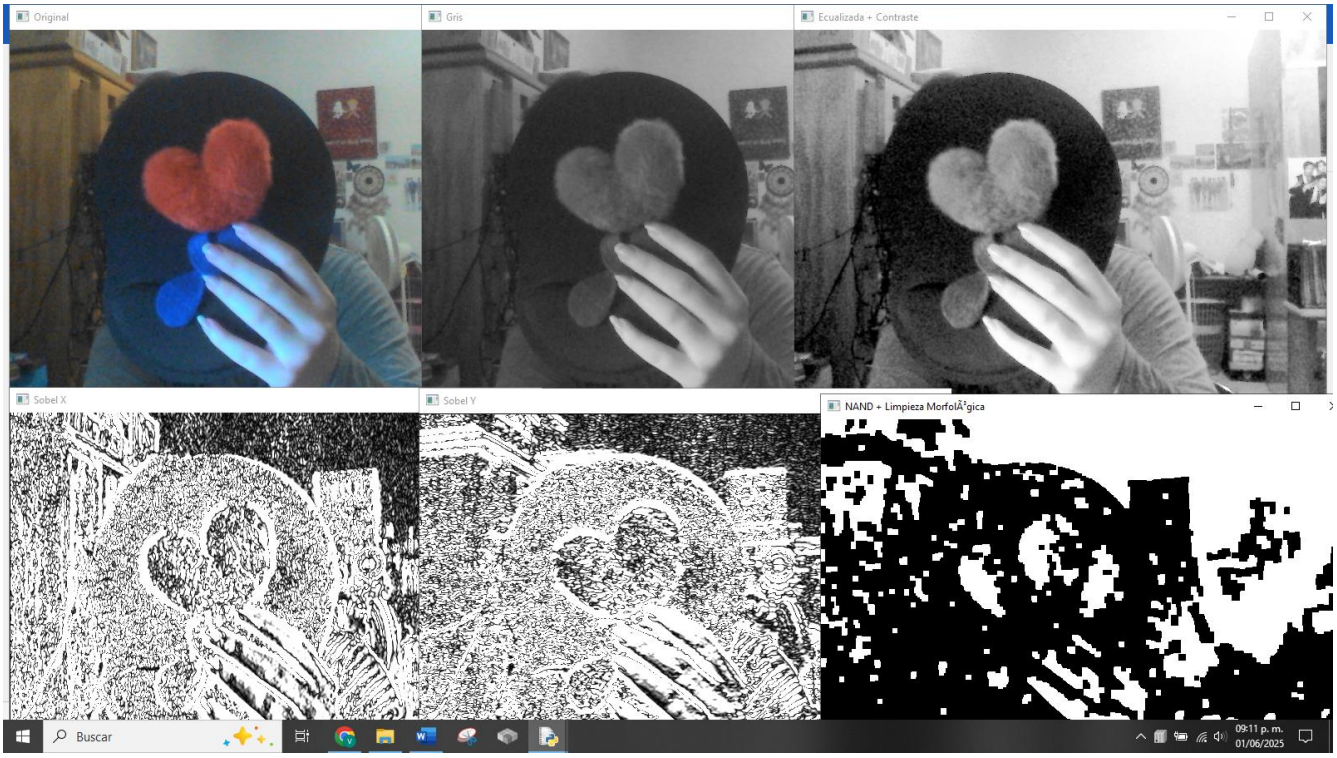


Si bien ya podíamos distinguir los bordes del objeto, aun se podía ver un poco de ruido tanto dentro del objeto como en l ambiente, por lo que haciendo uso de lo visto la practica pasada, se aplicaron operaciones morfológicas de apertura y cierre para eliminar ese ruido y poder apreciar mejor los bordes. Para ello, la variable kernel es la que define el tamaño de la matriz que se utilizara en estas operaciones.

Kernel = 1:



Kernel = 6:



Hacer un kernel demasiado grande hace que se puedan llegar a perder los bordes.

Conclusión:

Los bordes en el procesamiento de imágenes representan las zonas donde hay cambios abruptos en la intensidad de una imagen, y son fundamentales para identificar la estructura, forma y límites de los objetos dentro de un frame, y esto se lleva a cabo usando operadores como Sobel, Laplaciano o Canny. Se pudo ver que el método mas efectivo es el Canny, ya que los otros por si solos son más sensibles incluso a texturas, por lo que en esta practica se realizaron ajustes y se combinaron para intentar crear un filtro menos sensible con una respuesta mas parecida a la obtenida en el Canny.

En esta práctica se desarrolló un detector de bordes utilizando el operador Sobel, permitiendo identificar los contornos de los objetos capturados por la cámara en tiempo real. Primero, se convirtió la imagen a escala de grises para reducir la complejidad del procesamiento, y luego se aplicó el operador Sobel en las direcciones horizontal (X) y vertical (Y) para calcular los gradientes de intensidad. Estos gradientes fueron umbralizados para resaltar los bordes más marcados, y se combinó la información de ambos ejes mediante una operación lógica NAND, que permitió obtener un resultado más selectivo. Finalmente, se incorporaron operaciones morfológicas (apertura y cierre) ajustables mediante una variable, con el fin de eliminar el ruido (puntos aislados) y mejorar la limpieza del resultado final.

Bibliografía:

https://www.cs.buap.mx/~daniel.valdes/docs/Deteccion_de_bordes_mediante_el_algoritmo_de_Canny.pdf

https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm