

Benemérita Universidad Autónoma de Puebla  
Facultad de Ciencias de la Computación

# Procesamiento Digital de Imágenes

1er Parcial PDI



# BUAP

Docente:

- Ivan Olmos Pineda

---

**Alumno**

**Matrícula**

---

Jonatan Enrique Badillo Tejeda

202028319

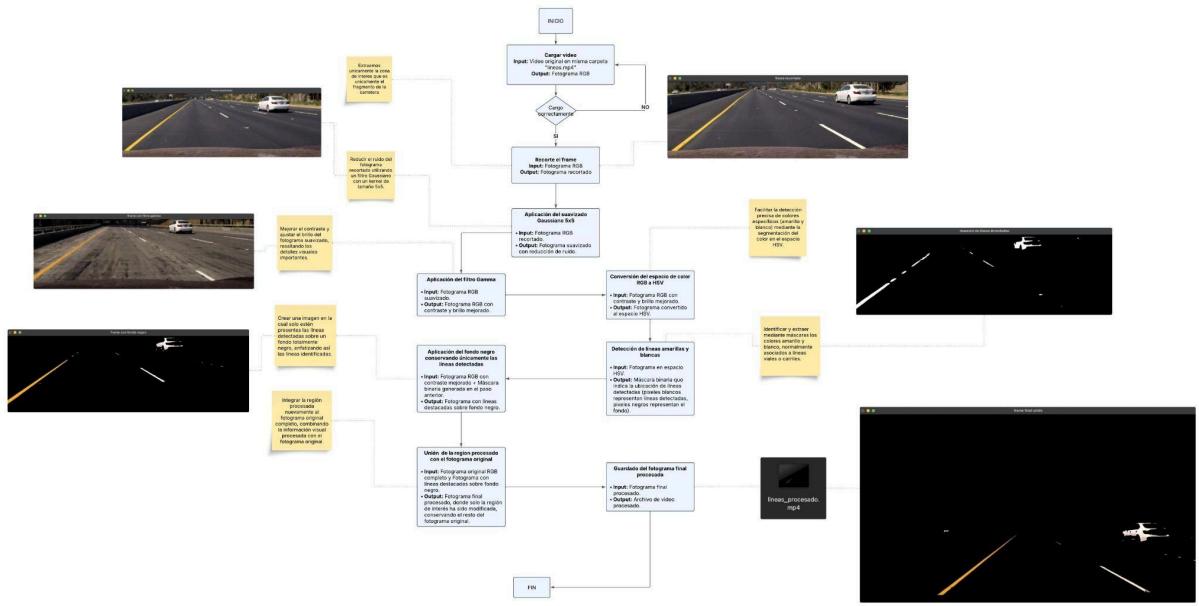
**Instrucciones:** Analiza cada una de las preguntas que se exponen a continuación. Realiza las implementaciones correspondientes, las cuales se deberán entregar en la fecha indicada por el profesor.

Diseñar e implementar una serie de operadores punto y regionales para el procesamiento digital de imágenes que permitan mejorar un video para resaltar las líneas de la carretera que aparecen en el mismo, con el objetivo de apoyar a la conducción autónoma. Para este proyecto, se podrán emplear estrategias como transformaciones en espacios de color, operadores de aclarado y obscurecimiento, operadores basados en histograma, operadores de suavizado (analizados en clase). La implementación se deberá de realizar en Python con OpenCV, así como otras bibliotecas de visualización como Matplotlib. No se podrán usar operadores de alto nivel ya implementados en OpenCV, ya que todos los operadores usados deberán ser implementados por el estudiante.

**Video:**  [Video\\_explicacion.mov](#) (también está en el archivo zip)

## Diagrama de flujo para el procesamiento del video

- Para ver en mejor calidad en drive:  [Diagrama de Flujo.jpeg](#) (tambien esta en el archivo zip)



Link de github del proyecto trabajado: [github](#)

## Importación de bibliotecas

- Se importan las bibliotecas necesarias.

- numpy se utiliza para la manipulación de matrices (imágenes).
- cv2 es la biblioteca principal (OpenCV) para cargar, procesar y guardar el video.

### Función principal:

- Aplica todo el procesamiento descrito (recorte, detección de líneas, etc.) y guarda el video procesado.

```
# funcion principal
if __name__ == "__main__":
    ruta_video = 'lineas.mp4' # ruta del video de entrada
    ruta_salida = 'lineas_procesado.mp4' # ruta para exportar video procesado

    # llamar a la funcion para procesar el video
    cargar_video(ruta_video, ruta_salida)
```

A continuación se describe cada bloque con su respectivo input y output:

- **Carga del video:**
  - **Objetivo:** Cargar el archivo de vídeo que esté localizado en la misma carpeta del proyecto para poder extraer los frames.
  - **Input:** Video original.
  - **Output:** Frames extraídos del video.



- **Recorte del Frame:**

- **Objetivo:** Extraer una porción específica de la imagen que contiene la carretera, reduciendo así el área que será procesada, para enfocarse únicamente en la parte relevante del video.
- **Input:** Frame original RGB.
- **Output:** Frame RGB recortado en la Región de Interés especificada.

```
def recorte(frame):
    x1, y1 = 280, 400 # coordenadas de la esquina superior izquierda
    x2, y2 = 1280, 720 # coordenadas de la esquina inferior derecha
    if frame.shape[0] < y2 or frame.shape[1] < x2:
        return frame # devuelve el frame original si las coordenadas no son validas
    return frame[y1:y2, x1:x2] # devuelve solo la region de interes
```

La función **recorte(frame)** extrae una región específica de una imagen (frame) utilizando coordenadas predefinidas.

- Defino la región de interés con las coordenadas **(x1, y1)** y **(x2, y2)**.
- Verifica si el frame es lo suficientemente grande; si no, devuelve el frame original.
- Recorta y devuelve solo la parte de la imagen dentro de esas coordenadas.



- **Suavizado Gaussiano (5x5):**

- **Objetivo:** Reducir ruido para mejorar la calidad visual.
- **Input:** Frame RGB recortado.
- **Output:** Frame suavizado con ruido reducido.

```

# funcion para aplicar suavizado gaussiano 5x5
# ayuda a reducir el ruido en la imagen

def suavizar(frame):
    # definir kernel gaussiano 5x5 para el filtro
    kernel_gaussian_5x5 = np.array([[1, 4, 7, 4, 1],
                                    [4, 16, 26, 16, 4],
                                    [7, 26, 41, 26, 7],
                                    [4, 16, 26, 16, 4],
                                    [1, 4, 7, 4, 1]]) * (1/273)

    # aplicar filtro gaussiano 5x5 para suavizar la imagen
    return cv2.filter2D(frame, -1, kernel_gaussian_5x5)

```

La función **suavizar(frame)** aplica un **suavizado gaussiano 5x5** para reducir el ruido en la imagen.

- Defino un kernel gaussiano 5x5, una matriz de pesos que atenúa las variaciones bruscas de color.
- Aplico el filtro con **cv2.filter2D()**, convolucionando la imagen con el kernel para suavizarla.
- Devuelve la imagen filtrada.



- **Aplicación de Filtro Gamma:**
  - **Objetivo:** Ajustar el brillo y contraste para mejorar la visibilidad de las líneas en la carretera.
  - **Input:** Frame RGB suavizado.
  - **Output:** Frame RGB con contraste y brillo ajustado.

```

# funcion para aplicar filtro gamma
# sirve para ajustar el brillo de la imagen

def filtro_gamma(frame):
    # Filtro Gamma
    gamma = 2
    image_RGB_gamma = np.array(255*(frame / 255)**gamma, dtype="uint8")
    return image_RGB_gamma

```

La función **filtro\_gamma(frame)** ajusta el **brillo y contraste** de la imagen mediante una **corrección gamma**.

- Defino un valor de gamma (**gamma = 2**), lo que hace que los píxeles más oscuros se aclaren más.
- Aplica la transformación gamma con la fórmula **255 \* (frame / 255) \*\* gamma**, ajustando la intensidad de los píxeles.
- Devuelve la imagen con el brillo modificado.



- **Conversión a HSV:**
  - **Objetivo:** Facilitar la detección precisa de colores específicos (líneas amarillas y blancas).
  - **Input:** Frame RGB mejorado.
  - **Output:** Frame convertido al espacio HSV.

```

hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV) # convertimos la imagen a espacio de color hsv

```

Esto facilita la segmentación de colores, ya que el espacio HSV separa la intensidad del color.

- **Detección de Líneas Amarillas y Blancas:**
  - **Objetivo:** Crear máscara binaria que resalte exclusivamente líneas amarillas y blancas.
  - **Input:** Frame HSV.
  - **Output:** Máscara binaria con líneas destacadas.

```
def detectar_lineas(frame):
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV) # convertimos la imagen a espacio de color hsv

    # definir rangos de color para amarillo y blanco
    lower_yellow = np.array([15, 150, 150]) # definimos el rango de color amarillo bajo en hsv
    upper_yellow = np.array([30, 255, 255]) # definimos el rango de color amarillo alto en hsv
    lower_white = np.array([0, 0, 200]) # definimos el rango de color blanco bajo en hsv
    upper_white = np.array([180, 50, 255]) # definimos el rango de color blanco alto en hsv

    # crear mascaras para detectar los colores
    mask_yellow = cv2.inRange(hsv, lower_yellow, upper_yellow) # crear mascara para el color amarillo
    mask_white = cv2.inRange(hsv, lower_white, upper_white) # crear mascara para el color blanco

    combined_mask = cv2.bitwise_or(mask_yellow, mask_white) # combinar ambas mascaras
    return combined_mask
```

La función **detectar\_lineas(frame)** identifica líneas amarillas y blancas en una imagen.

1. Convierte la imagen a HSV para facilitar la segmentación de color.
2. Define los rangos de amarillo y blanco en HSV.
3. Crea máscaras binarias para cada color.
4. Combina las máscaras y devuelve la imagen con las líneas detectadas.



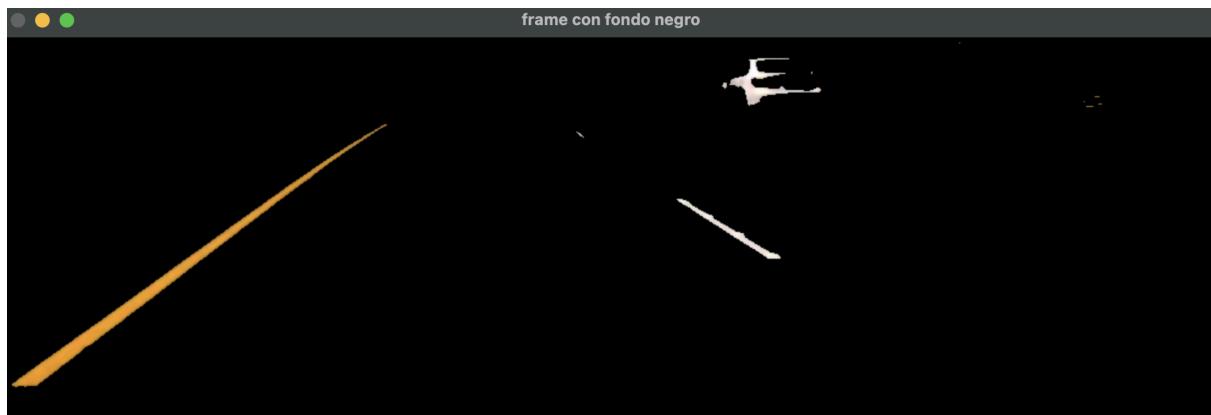
- **Aplicación del fondo negro:**
  - **Objetivo:** Destacar visualmente las líneas detectadas eliminando información irrelevante.
  - **Input:** Frame RGB mejorado + Máscara binaria de líneas.
  - **Output:** Frame con fondo negro y líneas resaltadas.

```
# funcion para aplicar fondo negro a todo menos las lineas detectadas

def aplicar_fondo_negro(frame, mask_lineas):
    resultado = cv2.bitwise_and(frame, frame, mask=mask_lineas) # aplicar la mascara al frame original
    fondo_negro = np.zeros_like(frame) # crear un frame negro del mismo tamaño
    frame_con_lineas = cv2.add(fondo_negro, resultado) # agregar las lineas al fondo negro
    return frame_con_lineas
```

La función **aplicar\_fondo\_negro(frame, mask\_lineas)** mantiene solo las líneas detectadas sobre un fondo negro.

1. Aplica la máscara al frame original para conservar sólo las líneas.
2. Crea un fondo negro del mismo tamaño que la imagen.
3. Combina ambos para mostrar solo las líneas sobre el fondo negro.
4. Devuelve la imagen procesada.



- **Unión de la región procesada con frame original:**
  - **Objetivo:** Integrar visualmente la región procesada en el frame original.
  - **Input:** Frame original RGB completo y región procesada.
  - **Output:** Frame combinado final.

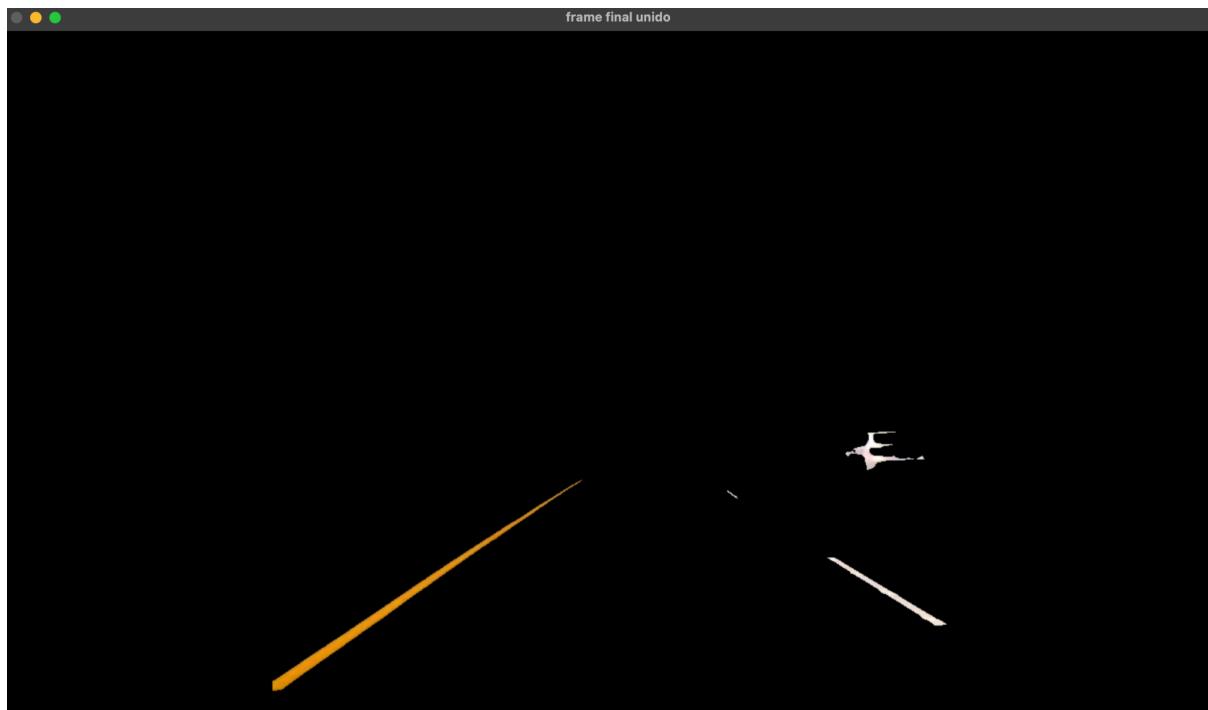
```
# funcion para unir el frame procesado con el fondo negro
# se usa para conservar solo la parte procesada del frame

def union(frame, frame_recortado):
    x1, y1 = 280, 400 # coordenadas de la esquina superior izquierda
    x2, y2 = 1280, 720 # coordenadas de la esquina inferior derecha
    frame_negro = np.zeros_like(frame) # creamos un frame negro del mismo tamaño
    frame_negro[y1:y2, x1:x2] = frame_recortado # colocamos el frame procesado en la region correspondiente
    return frame_negro
```

La función **union(frame, frame\_recortado)** inserta la región procesada en un fondo negro.

1. Crea un frame negro del mismo tamaño que la imagen original.

2. Coloca la región procesada en su posición original dentro del fondo negro.
3. Devuelve el frame resultante, conservando solo la parte procesada.



- **Guardar video procesado:**
  - **Objetivo:** Almacenar el resultado del procesamiento.
  - **Input:** Frames finales procesados.
  - **Output:** Archivo de video procesado en formato MP4.

