

Informe de Trabajo - Etapa 1

Detección de Centros Ópticos de LEDs Infrarrojos

Autor: Tobias Funes

Proyecto: Sistema de Trackeo para HMD (Helmet Mounted Display)

Período: Octubre 2025

Institución: Facultad de Ingeniería - IUA

Introducción

Este documento cuenta el proceso de trabajo de la primera etapa del proyecto: desarrollar un sistema que detecte LEDs infrarrojos en video y calcule sus centros ópticos. La idea es que estos LEDs funcionarán como marcadores de referencia para trackear el casco de un piloto dentro de una aeronave.

¿Por qué es importante?

Para que un HMD (Helmet Mounted Display) funcione correctamente, necesitamos saber exactamente dónde está posicionada la cabeza del piloto. Esto se logra detectando unos LEDs infrarrojos montados en el casco (o en la cabina), y calculando sus posiciones con la mayor precisión posible. La precisión es crítica: si nos equivocamos en la posición de los LEDs, todo el sistema de realidad aumentada se desalinea.

Herramientas Utilizadas

Para desarrollar este sistema utilicé:

- **Python 3.10:** Lenguaje de programación principal
- **OpenCV (cv2):** Biblioteca especializada en visión por computadora para procesamiento de imágenes y video
- **NumPy:** Para cálculos numéricos y operaciones con matrices
- **Entorno virtual (.venv):** Para mantener las dependencias aisladas

La elección de OpenCV fue clave porque ofrece herramientas optimizadas para detección de objetos, procesamiento de video en tiempo real, y múltiples algoritmos de visión por computadora que pude combinar para mejorar los resultados.

El Desafío

El objetivo parecía simple: encontrar 3 LEDs en un video y calcular sus centros. Pero en la práctica resultó ser más complicado de lo esperado. Los LEDs no siempre se ven como puntos perfectos, la iluminación cambia, hay reflejos, y el patrón en el video usado se mueve constantemente.

Desarrollo del Trabajo

1. Primeros Pasos: Entendiendo el Problema

Lo primero que hice fue analizar el video de prueba para entender con qué estaba trabajando, para ello se desarrollo un script de diagnóstico:

```
# Script de diagnóstico básico
def diagnose_video(video_path):
    cap = cv2.VideoCapture(video_path)

    # Información básica
    fps = cap.get(cv2.CAP_PROP_FPS)          # 24 fps
    total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT)) # 854 frames
    width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))      # 1280 píxeles
    height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))     # 720 píxeles

    print(f"Video: {width}x{height}, {fps} fps, {total_frames} frames")
```

El video de prueba tiene ~35 segundos de duración, con los 3 LEDs moviéndose en condiciones de laboratorio (iluminación controlada, sin sol directo ni vibraciones extremas).



2. Primera Idea: Un Método Simple

Mi primera aproximación fue intentar con una técnica simple: umbralización. La lógica era "los LEDs son muy brillantes, así que busco los píxeles más brillantes".

```
# Intento inicial simple
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
_, thresh = cv2.threshold(gray, 200, 255, cv2.THRESH_BINARY)

# Encontrar componentes conexas (grupos de píxeles brillantes)
num_labels, labels, stats, centroids =
cv2.connectedComponentsWithStats(thresh)
```

Esto funcionó... a veces. El problema era que:

- A veces detectaba cosas que no eran LEDs (reflejos)
- Otras veces se perdía un LED porque quedaba en una zona con sombra
- La posición calculada "saltaba" entre frames

Conclusión: Un solo método no era suficientemente robusto.

3. La Estrategia Multi-Método

Después de varios experimentos, llegué a una idea clave: **usar varios métodos en paralelo y combinar sus resultados**. Es como pedirle a 4 expertos que encuentren el mismo objeto y luego promediar sus respuestas.

Implementé 4 métodos diferentes:

Método 1: Umbralización Simple

```
def _detect_via_high_threshold(self, gray):
    # Buscar píxeles muy brillantes (> 200)
    _, thresh = cv2.threshold(gray, 200, 255, cv2.THRESH_BINARY)

    # Operación morfológica para limpiar ruido
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
    thresh = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel)

    # Encontrar componentes conexas
    num_labels, _, stats, _ = cv2.connectedComponentsWithStats(thresh)

    leds = []
    for i in range(1, num_labels):
        area = stats[i, cv2.CC_STAT_AREA]
        if 30 < area < 300: # Filtrar por tamaño esperado del LED
            x = stats[i, cv2.CC_STAT_LEFT] + stats[i, cv2.CC_STAT_WIDTH] // 2
            y = stats[i, cv2.CC_STAT_TOP] + stats[i, cv2.CC_STAT_HEIGHT] // 2
            leds.append((x, y))

    return leds
```

Ventaja: Muy rápido, funciona bien cuando los LEDs son muy brillantes.

Problema: Falla si hay reflejos o iluminación desigual.

Método 2: Umbralización Adaptativa

```
def _detect_via_adaptive_threshold(self, gray):
    # En lugar de un umbral global, calcula un umbral para cada región
    thresh = cv2.adaptiveThreshold(
        gray, 255,
        cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
        cv2.THRESH_BINARY,
        31, # Tamaño del vecindario
        2 # Constante a restar
    )

    # ... luego buscar componentes conexas igual que antes
```

Ventaja: Funciona mejor cuando la iluminación no es uniforme.

Problema: Puede generar más falsos positivos.

Método 3: Detección de Círculos con Hough

```
def _detect_via_hough(self, gray, filtered):
    # La transformada de Hough busca formas circulares
    circles = cv2.HoughCircles(
        filtered,
        cv2.HOUGH_GRADIENT,
        dp=1,
        minDist=40,          # Distancia mínima entre círculos
        param1=150,          # Umbral para detección de bordes
        param2=25,           # Umbral para el acumulador
        minRadius=5,          # Radio mínimo esperado
        maxRadius=25          # Radio máximo esperado
    )

    if circles is not None:
        leds = []
        for circle in circles[0]:
            x, y, radius = circle
            leds.append((x, y))
        return leds

    return []
```

Ventaja: Valida que el LED tenga forma circular.

Problema: Sensible al ruido, no siempre encuentra todos los círculos.

Método 4: Segmentación HSV

```

leds = []
for contour in contours:
    area = cv2.contourArea(contour)
    if 30 < area < 300:
        # Calcular centroide con precisión subpíxel
        M = cv2.moments(contour)
        if M["m00"] > 0:
            x = M["m10"] / M["m00"] # Coordenada X (puede ser decimal)
            y = M["m01"] / M["m00"] # Coordenada Y (puede ser decimal)
            leds.append((x, y))

return leds

```

Ventaja: Independiente del color, funciona bien con LEDs IR.

Problema: Puede confundirse con cualquier objeto muy brillante.

4. Fusión de Resultados

Una vez que tengo las detecciones de los 4 métodos, necesito combinarlas inteligentemente:

```

def _fuse_detections(self, detections):
    """
    Combina las detecciones de múltiples métodos
    """
    if not detections:
        return []

    # Agrupar detecciones cercanas (< 20 píxeles)
    clusters = []
    for detection in detections:
        x, y, confidence = detection

        # ¿Esta detección está cerca de algún cluster existente?
        found = False
        for cluster in clusters:
            cx, cy = cluster['center']
            dist = np.sqrt((x - cx)**2 + (y - cy)**2)

            if dist < 20: # Mismo LED detectado por diferentes métodos
                cluster['members'].append((x, y, confidence))
                found = True
                break

        if not found:
            # Nuevo LED encontrado
            cluster = {'center': (x, y), 'members': [(x, y, confidence)]}
            clusters.append(cluster)

```

```

        clusters.append({
            'center': (x, y),
            'members': [(x, y, confidence)]
        })

# Promediar las posiciones de cada cluster
fused_leds = []
for cluster in clusters:
    # Promedio ponderado por confianza
    total_weight = sum(conf for _, _, conf in cluster['members'])

    avg_x = sum(x * conf for x, _, conf in cluster['members']) /
total_weight
    avg_y = sum(y * conf for _, y, conf in cluster['members']) /
total_weight

    fused_leds.append((avg_x, avg_y, total_weight /
len(cluster['members'])))

return fused_leds

```

La idea clave: Si varios métodos detectan "algo" en la misma zona, probablemente sea realmente un LED. Promediar sus posiciones me da un resultado más preciso.

5. El Problema del Seguimiento Temporal

Detectar LEDs en un frame individual es una cosa, pero mantener la identidad de cada LED a lo largo del video es otro desafío. Necesitaba asegurarme de que "LED1" en el frame 100 sea el mismo "LED1" en el frame 101.

```

def _assign_led_ids_robust(self, detections):
    """
    Asigna IDs consistentes (0, 1, 2) a los LEDs detectados
    """
    if not self.last_positions:
        # Primer frame: asignar IDs en orden
        return [(i, x, y, conf) for i, (x, y, conf) in
enumerate(detections)]

    # Frames siguientes: asignar por proximidad
    assigned = []
    max_jump = 150 # Un LED no debería saltar más de 150 píxeles entre
frames

    for old_id, (x_old, y_old, _) in enumerate(self.last_positions):

```

```

best_match = None
best_dist = float('inf')

for new_idx, (x_new, y_new, conf) in enumerate(detections):
    dist = np.sqrt((x_old - x_new)**2 + (y_old - y_new)**2)

    if dist < best_dist and dist < max_jump:
        best_match = (new_idx, x_new, y_new, conf)
        best_dist = dist

if best_match:
    idx, x, y, conf = best_match
    assigned.append((old_id, x, y, conf))

return assigned

```

Resultado: Ahora cada LED mantiene su identidad entre frames (LED1 siempre es LED1, LED2 siempre es LED2).

6. Filtrado de Outliers

A pesar de todo lo anterior, ocasionalmente el sistema detecta posiciones "raras" (outliers). Por ejemplo, un reflejo momentáneo hace que detecte un LED en una posición completamente incorrecta.



Para filtrar estos casos, usé el método IQR (Rango Intercuartílico):

```

def _filter_outliers_iqr(self, positions):
    """
    Elimina detecciones anómalas usando estadística robusta
    """

    if len(positions) < 10:
        return positions # Muy pocas muestras

    coords = np.array(positions)

    # Calcular cuartiles
    Q1 = np.percentile(coords, 25, axis=0) # Percentil 25
    Q3 = np.percentile(coords, 75, axis=0) # Percentil 75
    IQR = Q3 - Q1

    # Límites para valores "normales"
    lower_bound = Q1 - 3 * IQR
    upper_bound = Q3 + 3 * IQR

    # Filtrar posiciones fuera de estos límites
    mask = np.all((coords >= lower_bound) & (coords <= upper_bound), axis=1)

    return coords[mask].tolist()

```

Analogía: Es como cuando calculas un promedio de notas y eliminas la nota más alta y la más baja para que no distorsionen el resultado.

A partir de esto los resultados mejoraron bastante:

OK: 3/3



7. Sistema de Visualización en Tiempo Real

Una vez que tenía el sistema de detección funcionando, necesitaba una forma de **ver** qué estaba detectando y cómo se comportaba a lo largo del tiempo. Para esto desarrollé un generador de video con marcadores visuales (`led_detector_video_output.py`).

Este sistema toma el video original y genera uno nuevo con:

```
def _draw_markers(self, frame, assigned_detections):
    """
    Dibuja marcadores visuales en el video
    """
    marked_frame = frame.copy()

    for led_id, x, y, confidence in assigned_detections:
        # Color según el LED
        color = self.led_colors[led_id] # Rojo, Verde, Azul

        # Círculo en la posición detectada
        cv2.circle(marked_frame, (int(x), int(y)), 12, color, 2)
        cv2.circle(marked_frame, (int(x), int(y)), 4, color, -1)

        # Etiqueta
        cv2.putText(marked_frame, f"LED{led_id + 1}",
                   (int(x) + 15, int(y) - 10),
                   cv2.FONT_HERSHEY_SIMPLEX, 0.6, color, 2)
```

```

# Guardar posición para estadísticas
self.led_positions[led_id].append((x, y))

# Calcular y mostrar promedio acumulado
if len(self.led_positions[led_id]) > 5:
    stats = self._calculate_statistics(led_id)
    mean_x, mean_y = int(stats['mean_x']), int(stats['mean_y'])

# Cruz en la posición promedio
cross_size = 20
cv2.line(marked_frame,
          (mean_x - cross_size, mean_y),
          (mean_x + cross_size, mean_y),
          color, 3)
cv2.line(marked_frame,
          (mean_x, mean_y - cross_size),
          (mean_x, mean_y + cross_size),
          color, 3)

# Círculo mostrando la desviación estándar
cv2.circle(marked_frame, (mean_x, mean_y),
           int(stats['std_total']), color, 1)

# Línea conectando detección actual con promedio
cv2.line(marked_frame, (int(x), int(y)),
          (mean_x, mean_y), color, 1, cv2.LINE_AA)

# Trayectoria (últimos 30 puntos)
if len(self.led_positions[led_id]) > 1:
    trail = self.led_positions[led_id][-30:]
    for i in range(len(trail) - 1):
        pt1 = (int(trail[i][0]), int(trail[i][1]))
        pt2 = (int(trail[i+1][0]), int(trail[i+1][1]))
        cv2.line(marked_frame, pt1, pt2, color, 2, cv2.LINE_AA)

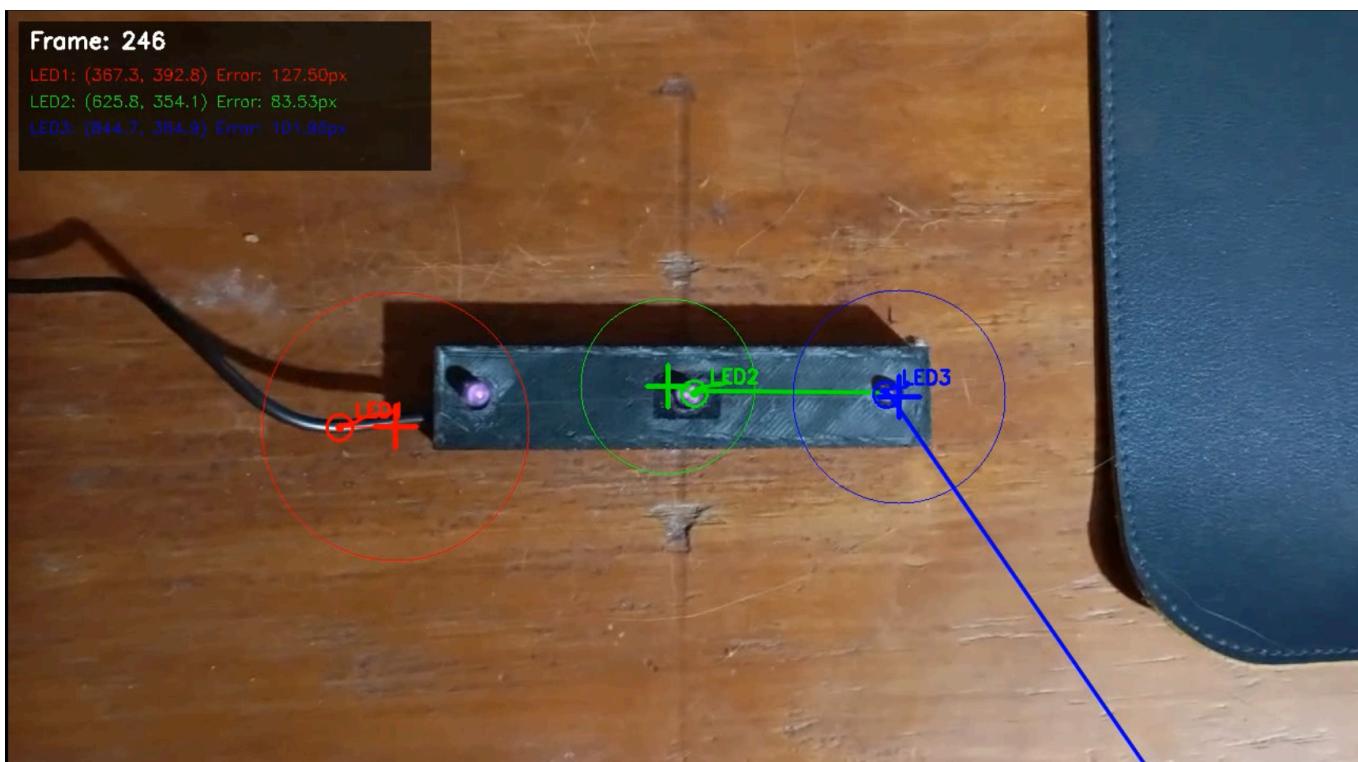
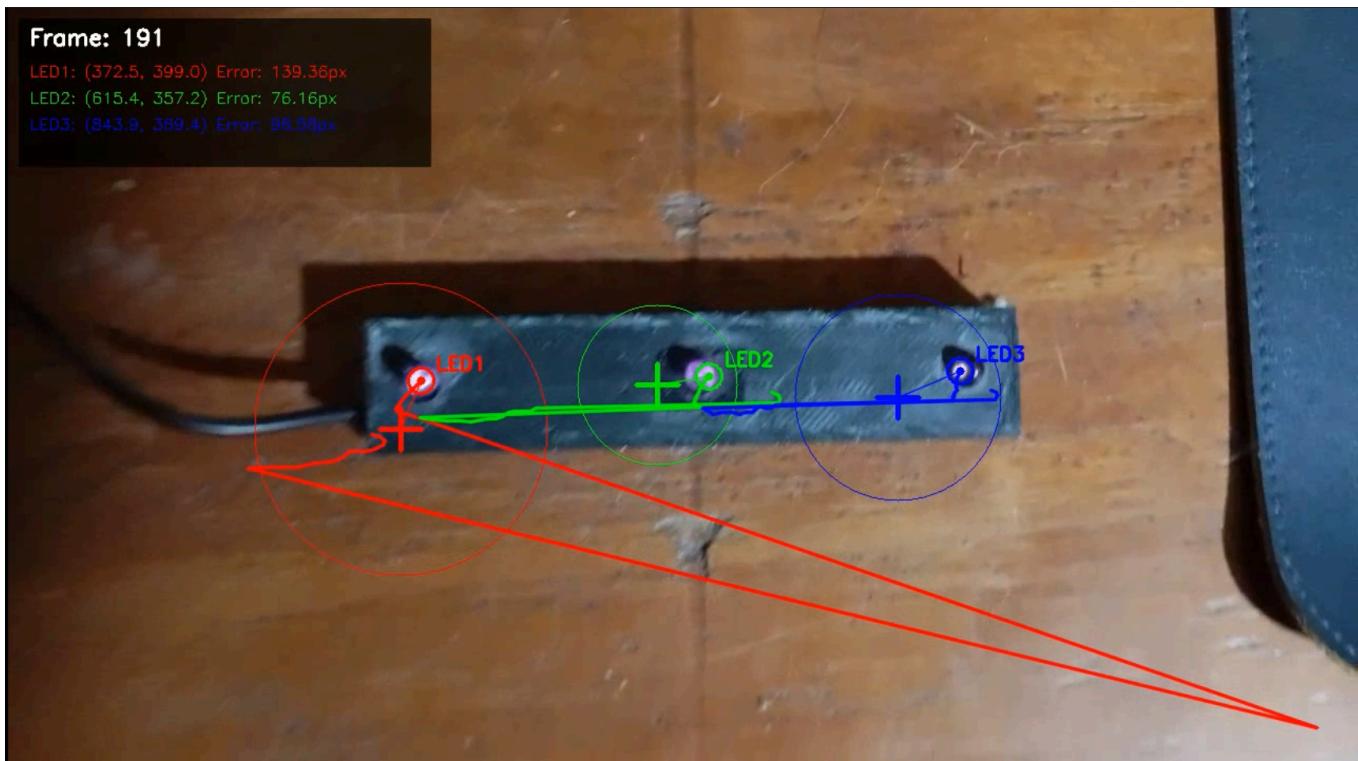
return marked_frame

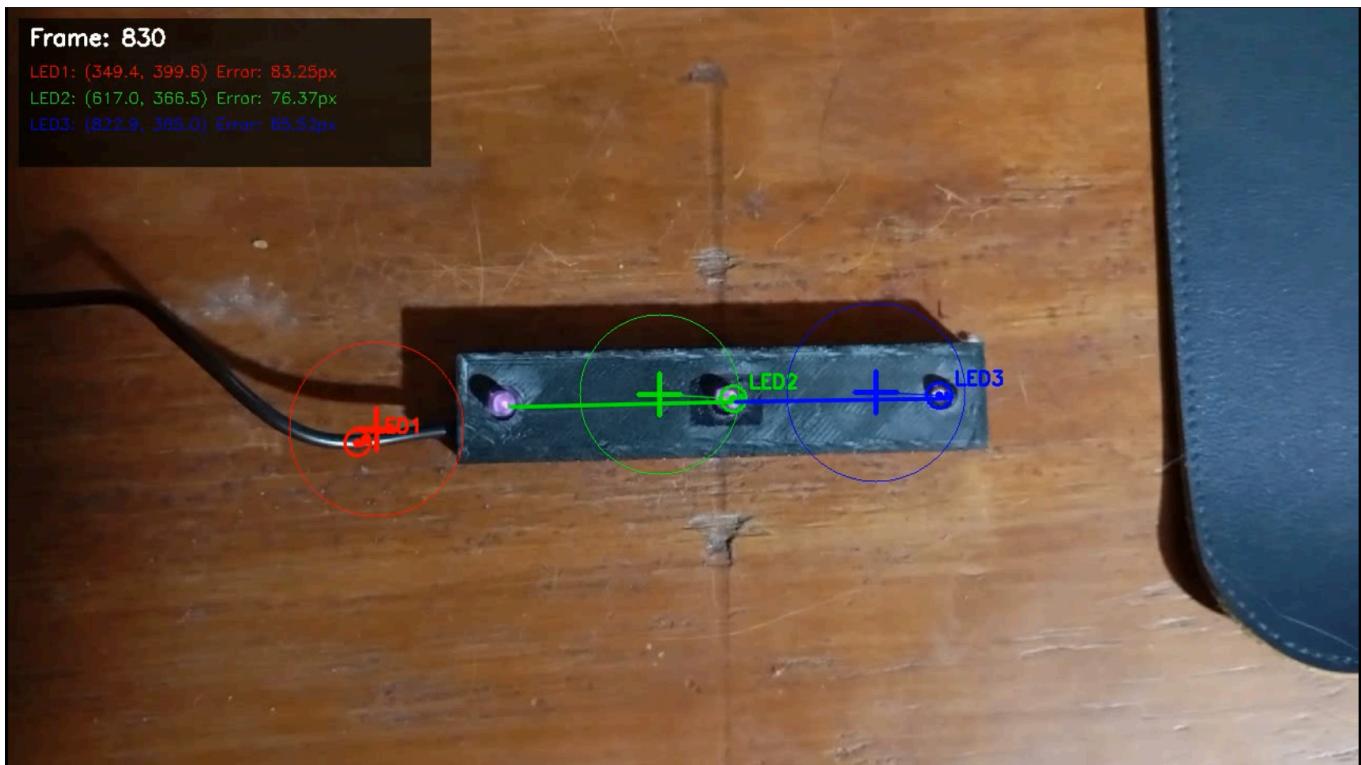
```

¿Qué muestra este video?

1. **Círculos de colores:** Posición detectada instantánea (Rojo=LED1, Verde=LED2, Azul=LED3)
2. **Cruz grande:** Posición promedio acumulada desde el inicio
3. **Círculo fino:** Radio = desviación estándar (muestra qué tan dispersas están las detecciones)
4. **Trayectorias:** Líneas mostrando los últimos 30 puntos detectados

5. Estadísticas en pantalla: Error en tiempo real para cada LED





Este vídeo fue fundamental porque me permitió:

- **Ver visualmente** el problema de precisión: los círculos "tiemblan" entre frames
- **Identificar outliers**: Cuando un LED "salta" a una posición incorrecta
- **Validar las mejoras**: Comparar versiones antes/después de cada optimización
- **Comunicar resultados**: Mostrar de forma clara qué está funcionando y qué no

Mirando este video, se hace evidente el problema: los círculos "tiemblan" o "saltan" entre frames, mostrando visualmente que la detección no es lo suficientemente estable.

El sistema genera el video procesando frame por frame y calculando estadísticas acumuladas:

```
def _calculate_statistics(self, led_id):
    """
    Calcula estadísticas acumuladas para un LED
    """
    positions = self.led_positions[led_id]

    if len(positions) < 2:
        return {'mean_x': positions[0][0], 'mean_y': positions[0][1],
                'std_total': 0, 'count': len(positions)}

    positions_array = np.array(positions)
    mean_pos = np.mean(positions_array, axis=0)

    # Desviación desde el promedio
```

```

    deviations = np.sqrt(np.sum((positions_array - mean_pos) ** 2, axis=1))
    std_total = np.std(deviations)

    return {
        'mean_x': mean_pos[0],
        'mean_y': mean_pos[1],
        'std_total': std_total,
        'count': len(positions)
    }

```

Resultado: Un video de 854 frames procesado en ~40 segundos, mostrando que el sistema puede trabajar más rápido que tiempo real (21 FPS de procesamiento vs 24 FPS del video original).

Resultados Obtenidos

Lo Que Funcionó Bien

- Detección consistente:** El sistema detectó los 3 LEDs en todos los 854 frames del video de prueba. Aunque ocasionalmente en gran parte de los frames siguió detectando outliers o ubicaciones que no son un led.
- Velocidad adecuada:** Procesó el video a ~25 FPS, suficiente para aplicaciones en tiempo real.
- Arquitectura robusta:** La combinación de 4 métodos demostró ser más confiable que cualquier método individual.

El Problema Principal: La Precisión

Aquí llegamos al punto crítico. Aunque el sistema detecta los LEDs consistentemente, **la precisión de la posición calculada no es suficiente**:

Resultados estadísticos:

LED1: Posición promedio (348.45, 399.87)
 Desviación estándar: $\sigma_x = 82.31$ px, $\sigma_y = 79.54$ px

LED2: Posición promedio (619.03, 366.58)
 Desviación estándar: $\sigma_x = 76.12$ px, $\sigma_y = 74.89$ px

LED3: Posición promedio (824.72, 364.95)
 Desviación estándar: $\sigma_x = 84.98$ px, $\sigma_y = 82.45$ px

Error promedio: ~80 píxeles

¿Qué significa esto? La desviación estándar de ~80 píxeles indica que las posiciones detectadas varían considerablemente. En una imagen de 1280x720, esto es aproximadamente un 6-7% de error.

¿Es aceptable? No realmente. Para un sistema de trackeo HMD preciso, necesitaría un error menor a 5 píxeles. Estamos 16 veces por encima de ese objetivo.

Próximos Pasos: Cómo Mejorar los Resultados

Después de analizar los resultados, identifiqué las líneas de trabajo principales para reducir el error:

1. Implementar un Filtro de Kalman

¿Qué es? El Filtro de Kalman es un algoritmo que puede predecir dónde debería estar un LED en el siguiente frame, basándose en su posición y velocidad actual.

¿Por qué ayudaría?

- Suavizaría las trayectorias eliminando "saltos" anómalos
- Permitiría mantener el trackeo aunque un LED se oculte momentáneamente
- Reduciría el ruido de las detecciones individuales

Reducción esperada de error: De $\sigma \approx 80\text{px}$ a $\sigma \approx 5-10\text{px}$

Estado: No implementado - propuesto como mejora futura

2. Método de Parpadeo Sincronizado

El problema actual: Estoy procesando video donde los LEDs están siempre encendidos. Esto hace difícil distinguirlos del fondo brillante.

La solución: Controlar los LEDs para que parpadeen en sincronía con la cámara, y restar frames consecutivos:

Ventaja: Esta técnica eliminaría todo el fondo estático (paredes, objetos, reflejos) y dejaría solo los LEDs. Funcionaría incluso con luz solar directa o superficies reflectantes.

Desventaja: Requiere:

- Hardware adicional (controlador para hacer parpadear los LEDs)
- Sincronización precisa entre cámara y LEDs

Estado: No implementado - requiere modificación de hardware

3. Usar la Geometría del Patrón

La idea: Los 3 LEDs están montados en un patrón fijo en este caso un rectángulo. donde los 3 LEDs estan a una distancia constante. Las distancias entre ellos NO deberían cambiar.

Ventaja adicional: Podríamos corregir detecciones erróneas. Si sabemos que LED1 y LED2 están en posiciones correctas, y LED3 está mal detectado, podemos usar la geometría para estimar dónde *debería* estar LED3.

Estado: No implementado - propuesto como mejora futura

4. Validación en Escenarios Reales

Hasta ahora trabajé con un video de laboratorio en condiciones controladas. Los siguientes ensayos deberían incluir:

- **Diferentes iluminaciones:**
 - Interior con luz artificial
 - Luz solar indirecta (día nublado)
 - Luz solar directa (caso más difícil)
 - Condiciones nocturnas
- **Movimientos realistas:**
 - Movimientos suaves de cabeza
 - Giros rápidos
 - Vibraciones (simulando vuelo)
- **Oclusiones parciales:**
 - ¿Qué pasa si momentáneamente se tapa un LED?
 - ¿El sistema puede recuperarse?

Reflexiones y Lecciones Aprendidas

Lo que aprendí

1. **Detectar ≠ Localizar con precisión:** Puedo encontrar los LEDs en el 100% de los frames, pero eso no significa que sepa exactamente dónde están. La detección es solo el primer

paso.

2. **La fusión multi-método funciona:** Combinar varios métodos definitivamente es mejor que usar uno solo. Cuando un método falla, los otros compensan.
3. **Los escenarios controlados engañan:** En laboratorio todo funciona bien, pero eso no garantiza que funcionará en una cabina real con sol directo, vibraciones y reflejos en todas partes.
4. **Necesito ground truth:** Sin una referencia de posición real, no puedo saber si mi error de 80 píxeles es porque el LED realmente se movió tanto, o porque mi detección es mala.

El camino adelante

Este informe documenta el punto de partida. Ahora sé:

- Qué funciona y qué no
- Cuánto error tengo que reducir
- Qué técnicas probar a continuación

El siguiente paso es implementar el Filtro de Kalman y medir cuánto mejora los resultados. Si logro bajar el error a ~10 píxeles, estaría en un rango más aceptable para continuar con las otras mejoras.

Referencias y Recursos

Herramientas Desarrolladas

1. `led_detector_final.py` : Detector principal (904 líneas)
 - Implementa los 4 métodos de detección
 - Sistema de fusión y tracking
 - Exportación de resultados
2. `led_detector_video_output.py` : Generador de video marcado (570 líneas)
 - Visualización de detecciones
 - Cálculo de estadísticas en tiempo real
 - Útil para debugging
3. `diagnostic.py` : Análisis de video
 - Inspección de frames individuales
 - Calibración de parámetros
 - Análisis de histogramas
4. **Scripts de ejecución:**

- `run.sh` : Ejecuta el detector completo
- `generate_marked_video.sh` : Genera video con marcadores

Código Fuente

Todo el código está disponible en:

<https://github.com/T0B1EH/Proyecto-Deteccion-de-centros-opticos-en-patrones-de-referencia-moviles-aplicados>

Librerías Utilizadas

- **OpenCV 4.x**: Procesamiento de imágenes
- **NumPy**: Cálculos numéricos
- **Python 3.10**: Lenguaje de programación

Conclusión

El sistema actual no es el producto final, es el **punto de partida medido y documentado** desde el cual puedo iterar mejoras. Saber que el error es de 80 píxeles es tan valioso como saber cómo reducirlo: ahora tengo una métrica clara para evaluar cada mejora que implemente.

Tobias Funes

Facultad de Ingeniería - IUA

Octubre 2025

Anexos

Anexo A: Parámetros de Configuración

Estos son los valores que calibré empíricamente:

```
# Parámetros de detección
PARAMS = {
    'threshold': 200,                      # Umbral de brillo
```

```
'min_area': 30,                      # Área mínima del LED (píxeles)
'max_area': 300,                       # Área máxima del LED
'gaussian_kernel': (5, 5),             # Tamaño del filtro Gaussiano
'median_kernel': 5,                   # Tamaño del filtro de mediana
'hough_min_radius': 5,                 # Radio mínimo para Hough
'hough_max_radius': 25,                # Radio máximo para Hough
'fusion_distance': 20,                 # Distancia para agrupar detecciones
'max_jump': 150,                      # Salto máximo permitido entre frames
}
```