

**UNIVERSIDAD REY JUAN CARLOS**

**ESCUELA DE MÁSTERES OFICIALES**

**Máster Universitario en Visión Artificial**

---



Universidad  
Rey Juan Carlos



Escuela de Másteres  
Oficiales

## **“OPTICAL FLOW”**

Autor:

**Vicente Gilabert Maño**

***MADRID, 2022***

# Contents

<b>1</b>	<b>Introducción</b>	<b>2</b>
<b>2</b>	<b>Lukas-Kanade</b>	<b>3</b>
2.1	Método 1: Mínimos cuadrados.	4
2.2	Método 2: Cálculo directo.	5
2.3	Resultados y comparación de tiempos	6
<b>3</b>	<b>Horn-Schunck</b>	<b>9</b>
3.1	Análisis de resultados	10
3.2	Filtrado de ruido	14
<b>4</b>	<b>Conclusiones</b>	<b>15</b>
<b>5</b>	<b>Bibliografía</b>	<b>16</b>
<b>6</b>	<b>Anexos</b>	<b>17</b>
6.1	Obtener derivadas espacio-temporales	17
6.2	Quiver Plot	17
6.3	Color Plot	18
6.4	Combinación de quiver y color plot	18

# 1 Introducción

La estimación de movimiento en la visión artificial puede ayudar a la detección y seguimiento de objetos (en movimiento) de una secuencia de *frames*, estimación de mapas de profundidad, estimación de estructuras 3D, etc. En nuestro caso vamos a centrar en la aplicación del análisis del campo de movimiento mediante el cálculo del flujo óptico.

El flujo óptico (*OF:Optical Flow*) son unos métodos utilizados para describir el movimiento de objetos, suponiendo que la intensidad se mantiene constante y los movimientos son pequeños entre *frames* consecutivos.

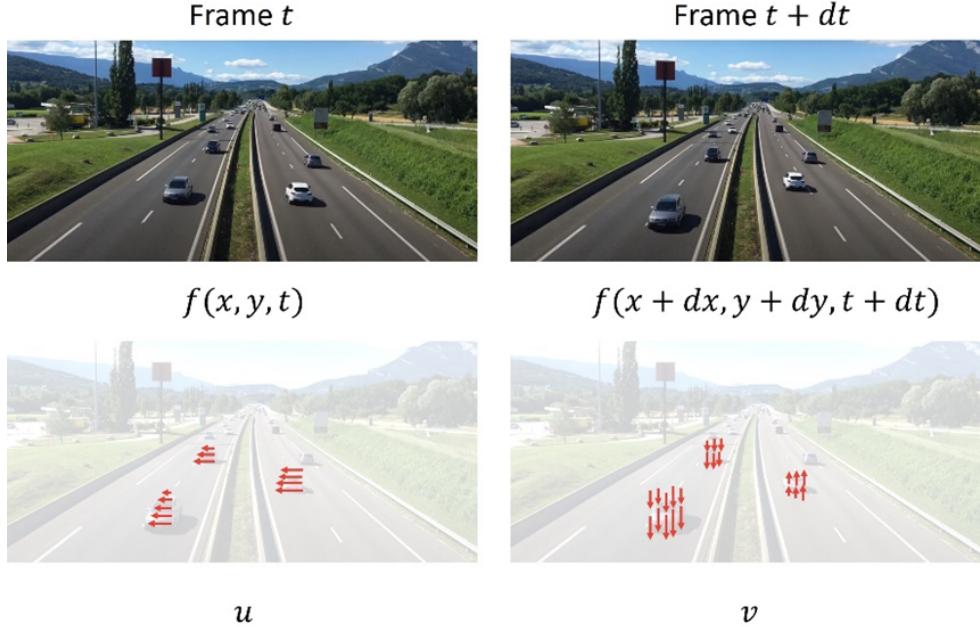


Figure 1: Ejemplo flujo óptico en t y t+dt.

Una definición de flujo óptico podría ser un patrón de movimiento de píxeles entre dos imágenes consecutivas. Se trata de obtener la variación de la intensidad de los píxeles y encontrar la dirección en la que varían. Existen diferentes estrategias para abordar el problema:

- **Dense OF:** El flujo óptico denso se basa en obtener los vectores de flujo de todos los píxeles de una imagen, por lo que esta técnica es muy lenta, pero más precisa que la dispersa.
- **Sparse OF:** El flujo óptico disperso se basa en obtener puntos de interés de una imagen utilizando técnicas como SIFT, SURF, etc. y seguir estos puntos dentro de una ventana. Es un algoritmo mucho más rápido, pero menos preciso que el anterior.

Existen implementaciones donde utilizan las imágenes piramidales para tratar de ser invariantes a escala, ya que al depender de una ventana no siempre se consigue un resultado óptimo cuando el objeto cambia de tamaño.

En nuestro caso se va a trabajar con el caso denso, es decir analizando todos los píxeles de la imagen con las técnicas Lukas-Kanade y Horn-Schunck. Son técnicas que aplican métodos diferentes, el primero con cálculo diferencial y el segundo con una adaptación para el cálculo variacional. En los siguientes puntos se van a analizar estos dos algoritmos.



Figure 2: Izquierda: flujo óptico disperso. Derecha: flujo óptico denso.

## 2 Lukas-Kanade

El método de flujo óptico de *Lucas-Kanade* es una técnica utilizada para estimar el movimiento en imágenes de una escena. Se asocia un vector de movimiento ( $u, v$ ) de cada píxel de la escena comparando el instante  $t$  y  $t + 1$ . Está basado en encontrar los cambios de intensidad de los píxeles para así detectar la dirección del movimiento. Es un método local donde para cada píxel se analiza una región que se denominada vecindad. En nuestro caso vamos a utilizar el método para todos los píxeles de la imagen, por lo que es un método denso, aunque puede utilizarse de forma dispersa solo utilizando los puntos característicos.

Para utilizar el algoritmo de *Lucas-Kanade* también es necesario conocer la limitaciones y las suposiciones del mismo:

- La iluminación de la escena debe ser uniforme.
- Superficies con reflexión no anómala que pueda confundir al algoritmo.
- El objeto debe moverse en plano paralelo al plano imagen.
- Los movimientos entre los intervalos  $t$  y  $t + 1$  deben ser pequeños dentro de un margen (la vecindad).
- La escena debe tener un brillo y color constante.

En esta práctica vamos a ver dos implementaciones del algoritmo, con calculo de mínimos cuadrados y con un método directo. Ambos métodos utilizan las etapas comunes siguientes:

- Pre-filtrado
- Extracción de medidas (derivadas espacio-temporales, puntos característicos de SIFT o HoG)
- Integración de las medidas para producir campo de movimiento 2D
  - Método 1: Mínimos cuadrados.
  - Método 2: Cálculo directo.

A continuación, se explican los fundamentos y la implementación de los diferentes métodos de *Lucas-Kanade* para el cálculo del flujo óptico, así como una comparación de dos implementaciones.

## 2.1 Método 1: Mínimos cuadrados.

Esta resolución por una aproximación de mínimos cuadrados trata de minimizar la ecuación en una vecindad ( $M \times M$ ). Desarrollando la ecuación de mínimos cuadrados se llega a la siguiente expresión:

$$A\vec{v} = b$$

$$A^T A \vec{v} = A^T b$$

$$\vec{v} = \frac{A^T b}{A^T A} = (A^T A)^{-1} A^T b$$

$$\vec{v} = \frac{A^T b}{A^T A} = (A^T A)^{-1} A^T b$$

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum I_{xi}^2 & \sum I_{xi} I_{yi} \\ \sum I_{xi} I_{yi} & \sum I_{xi}^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum I_{xi} I_{ti} \\ -\sum I_{yi} I_{ti} \end{bmatrix} \quad (1)$$

La primera matriz a la derecha de la igualdad, se llama matriz de gradientes espaciales ( $2 \times 2$ ) y la segunda matriz es la contribución temporal ( $2 \times 1$ ). Donde  $I_{xi} = \frac{I_x(t) + I_x(t+1)}{2}$ ,  $I_{yi} = \frac{I_y(t) + I_y(t+1)}{2}$ ,  $I_t = I'(t+1) - I'(t)$ .

$I_x(t)$  y  $I_y(t)$  es la derivada en  $x$  e  $y$  en  $t$ ,  $I_x(t+1)$  y  $I_y(t+1)$  es la derivada en  $x$  e  $y$  en  $t+1$ ,  $I'(t+1)$  y  $I'(t)$  son las imágenes originales con un filtrado de suavizado. Estas operaciones anteriores se calcula para píxel de la imagen con una vecindad de tamaño  $M \times M$ .

Este método de calculo, tiene el problema de obtener la la pseudo-inversa ya que es un proceso lento de calcular.

A continuación se muestra la función implementada para este método:

```

1 def LK_optical_flow_v1(img1, img2, M=3, stride=1):
2     if len(img1.shape) < 3 and len(img1.shape) < 3:
3         dim = img1.shape
4         flow = np.zeros((dim[0], dim[1], 2))
5
6         Ix, Iy, It = get_gradients_optical_flow(img1, img2)
7
8         for u in range(M, dim[0] - M + 1, stride):
9             for v in range(M, dim[1] - M + 1, stride):
10                 sumIx2 = np.sum(np.power(Ix[u:u+M, v:v+M], 2))
11                 sumIxIy = np.sum(np.multiply(Ix[u:u+M, v:v+M], Iy[u:u+M, v:v+M]))
12                 sumIy2 = np.sum(np.power(Iy[u:u+M, v:v+M], 2))
13                 sumIxIt = np.sum(np.multiply(Ix[u:u+M, v:v+M], It[u:u+M, v:v+M]))
14                 sumIyIt = np.sum(np.multiply(Iy[u:u+M, v:v+M], It[u:u+M, v:v+M]))
15
16                 A = np.array([[sumIx2, sumIxIy], [sumIxIy, sumIy2]])
17                 b = np.array([[-sumIxIt], [-sumIyIt]])
18                 A_inv = np.linalg.pinv(A)
19                 result = np.dot(A_inv, b)
20                 flow[u, v, 0] = result[0]
21                 flow[u, v, 1] = result[1]
22

```

```

23     return flow
24
25 else:
26     print("Error input data. Images must be in grayscale.")
27     return

```

El resultado de velocidad del algoritmo esta directamente relacionado con el tamaño de la vecindad ( $M$ ) y el salto para el cálculo ( $stride$ ). Cuanto mayor sea la vecindad mayor será el numero de operaciones para cada píxel y cuanto menor sea  $stride$  recorrerá más píxeles para realizar el calculo.

## 2.2 Método 2: Cálculo directo.

Una vez obtenida la ecuación 1, se desarrolla matemáticamente para simplificar el proceso de calculo, obteniendo la siguiente ecuación que implementaremos en *Python* para nuestro problema.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -\sum I_{yi}^2 \sum I_{xi} I_{ti} + \sum I_{xi} I_{yi} \sum I_{yi} I_{ti} \\ \sum I_{xi}^2 \sum I_{yi}^2 - \sum I_{xi} I_{yi} \sum I_{xi} I_{yi} \\ \sum I_{xi} I_{yi} \sum I_{xi} I_{ti} - \sum I_{xi}^2 \sum I_{yi} I_{ti} \\ \sum I_{xi}^2 \sum I_{yi}^2 - \sum I_{xi} I_{yi} \sum I_{xi} I_{yi} \end{bmatrix} \quad (2)$$

Donde  $I_{xi} = \frac{I_x(t) + I_x(t+1)}{2}$ ,  $I_{yi} = \frac{I_y(t) + I_y(t+1)}{2}$ ,  $I_t = I'(t+1) - I'(t)$ .  $I_x(t)$  y  $I_y(t)$  es la derivada en  $x$  e  $y$  en  $t$ ,  $I_x(t+1)$  y  $I_y(t+1)$  es la derivada en  $x$  e  $y$  en  $t+1$ ,  $I'(t+1)$  y  $I'(t)$  son las imágenes originales con un filtrado de suavizado. Estas operaciones anteriores se calcula para píxel de la imagen con una vecindad de tamaño  $M \times M$ .

A continuación se muestra la función implementada para este método:

```

1 def LK_optical_flow_v2(img1, img2, M=3, stride=1):
2     if len(img1.shape) < 3 and len(img1.shape) < 3:
3         dim = img1.shape
4         flow = np.zeros((dim[0], dim[1], 2))
5
6     Ix, Iy, It = get_gradients_optical_flow(img1, img2)
7
8     for u in range(M, dim[0] - M + 1, stride):
9         for v in range(M, dim[1] - M + 1, stride):
10            sumIx2 = np.sum(np.power(Ix[u:u+M, v:v+M], 2))
11            sumIxIy = np.sum(np.multiply(Ix[u:u+M, v:v+M], Iy[u:u+M, v:v+M]))
12            sumIy2 = np.sum(np.power(Iy[u:u+M, v:v+M], 2))
13            sumIxIt = np.sum(np.multiply(Ix[u:u+M, v:v+M], It[u:u+M, v:v+M]))
14            sumIyIt = np.sum(np.multiply(Iy[u:u+M, v:v+M], It[u:u+M, v:v+M]))
15
16            denom = ((sumIx2 * sumIy2) - (sumIxIy * sumIxIy))
17            if denom != 0.0:
18                flow[u, v, 0] = ((-sumIy2 * sumIxIt) + (sumIxIy * sumIyIt)) / denom
19                flow[u, v, 1] = ((sumIxIy * sumIxIt) - (sumIx2 * sumIyIt)) / denom
20            else:
21                flow[u, v, 0] = 0
22                flow[u, v, 1] = 0
23
24    return flow
25
26 else:
27     print("Error input data. Images must be in grayscale.")
     return

```

### 2.3 Resultados y comparación de tiempos

En esta sección se va a realizar una comparación de tiempos de ejecución de los dos métodos implementados. En la siguiente gráfica se observan los resultados obtenidos.

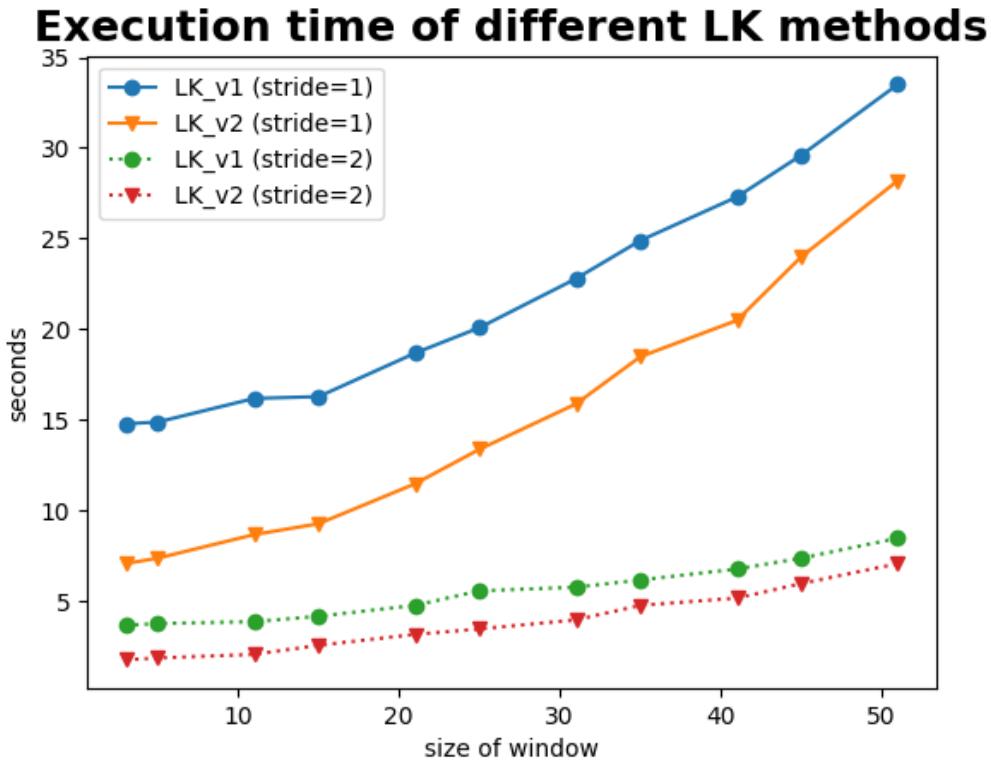


Figure 3: Representación de los tiempos de ejecución en segundos (eje y) con el tamaño de la vecindad (eje x)

Donde la linea azul corresponde con  $LK\_v1$  que hace referencia al método 1 (mínimos cuadrados) y la naranja  $LK\_v2$  hace referencia al método 2 (calculo directo). Despues variando el  $stride$  a 2 tenemos para los dos métodos otros resultados con los colores verde ( $LK\_v1$ ) y rojo ( $LK\_v2$ ).

Se han hecho diferentes pruebas pero viendo la gráfica para un  $stride=1$ , se puede observar que a mayor tamaño de ventana mayor tiempo de computo. Esto es debido a que hay más operaciones a realizar por píxel. Una forma de reducir las operaciones totales, es el uso del  $stride=2$ , con esto conseguimos tiempos mucho menores pero no se calcula el vector de movimiento para todos los píxeles, sino que para la mitad de ellos. También existe la opción de utilizar un  $stride$  mayor.

Las siguientes imágenes son algunos resultados obtenidos con diferentes valores de vecindad y *stride*:

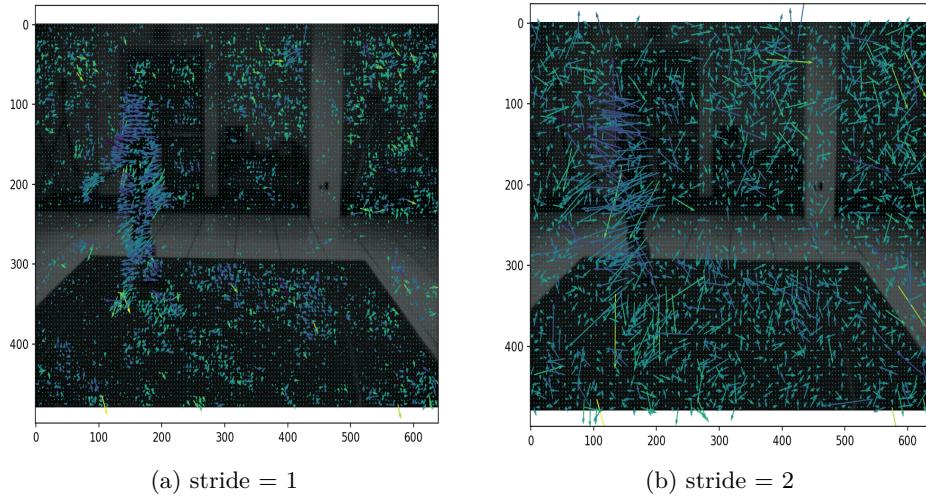


Figure 4: Resultados para el flujo óptico con una vecindad  $M = 5$

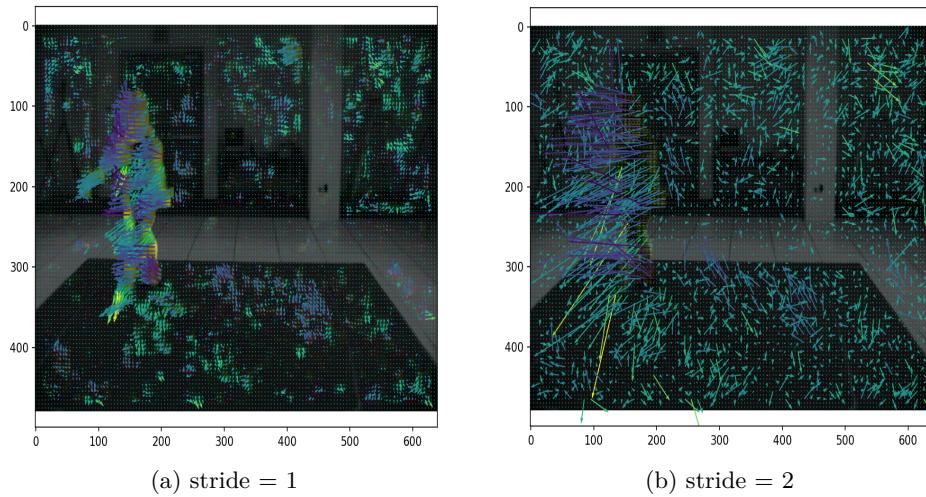


Figure 5: Resultados para el flujo óptico con una vecindad  $M = 11$

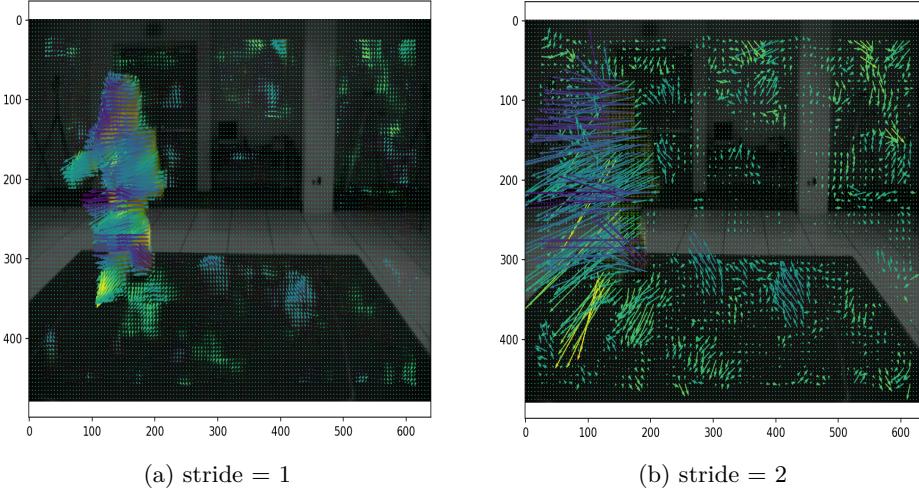


Figure 6: Resultados para el flujo óptico con una vecindad  $M = 25$

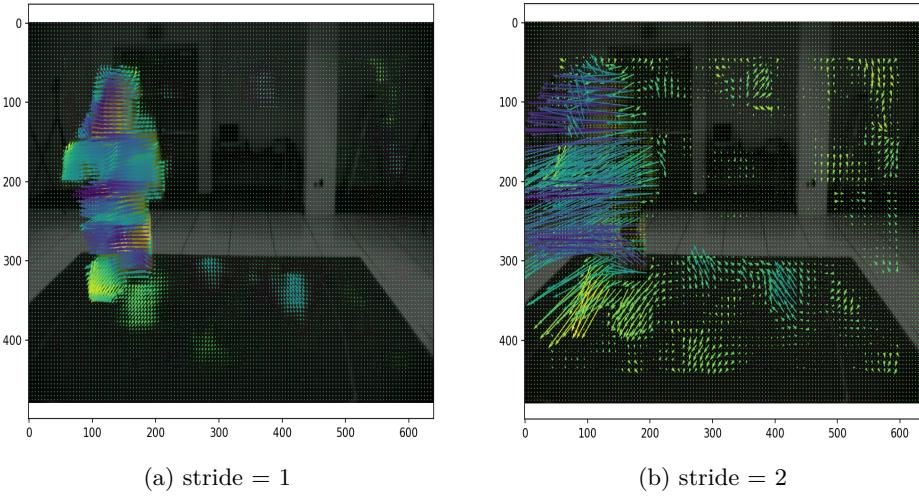


Figure 7: Resultados para el flujo óptico con una vecindad  $M = 41$

Como conclusión del método de *Lukas-Kanade*, tendremos que dependiendo de la aplicación de nuestro problema, la configuración será diferente. Si necesitamos menor ruido, necesitaremos una mayor vecindad o un filtrado extra por lo que esto aumentara el tiempo de ejecución. Por otra parte, si lo que prima es la velocidad deberíamos reducir la vecindad y aumentar el *stride*.

### 3 Horn-Schunck

El algoritmo de *Horn-Schunck* es un método global, denso y de cálculo variacional que propone una condición de suavidad a la *ERFO* (Ecuación de Restricción de Flujo Óptico). Continuando con la idea principal de estos autores en trabajos posteriores se han propuesto mejoras o variaciones del mismo.

Existen algunos problemas con este algoritmo que es necesario conocer para trabajar correctamente:

- Desplazamientos grandes del objeto entre *frames*.
- Estructuras no rígidas o deformables entre *frames*.
- Discontinuidades
- Oclusiones del objeto.

La suavidad en el flujo óptico se refiere a las transiciones en zonas de las imágenes: una transición de una zona clara a una oscura debe ocurrir suavemente, pasando por las intensidades intermedias. En este método, se combina el error de la *ERFO* (3) con un error para la suavidad (4).

$$I_x u + I_y v + I_t = 0 \quad (3)$$

$$\varepsilon_S^2 = u_x^2 + u_y^2 + v_x^2 v_y^2 \quad (4)$$

Combinando las dos ecuaciones anteriores, se puede llegar a un error global que podemos usar para minimizarlo.

$$E^2 = \sum_i \sum_j (\varepsilon_S^2(i, j) + \lambda \varepsilon_S^2(i, j)) \quad (5)$$

Esto se convierte en un problema de optimización donde buscamos el error mínimo. Desarrollando a través del cálculo numérico (*Gauss-Seidel*) se obtiene un método iterativo con dos hiper-parámetros, lambda y el número de iteraciones. La siguientes dos ecuaciones son la que vamos a implementar en *Python*:

$$u^k = u^{-k-1} - I_x \left( \frac{I_x u^{-k-1} + I_y v^{-k-1} + I_t}{\lambda^2 + I_x^2 + I_y^2} \right) \quad (6)$$

$$v^k = v^{-k-1} - I_y \left( \frac{I_x u^{-k-1} + I_y v^{-k-1} + I_t}{\lambda^2 + I_x^2 + I_y^2} \right) \quad (7)$$

En cada iteración se van actualizando las matrices de movimiento u y v. Donde  $u^{-k-1}$  y  $v^{-k-1}$  es la media de una vecindad,  $I_x = \frac{I_x(t) + I_x(t+1)}{2}$ ,  $I_y = \frac{I_y(t) + I_y(t+1)}{2}$ ,  $I_t = I'(t+1) - I'(t)$ . Para el caso de lambda es un parámetro que actúa como ponderación entre ambas magnitudes de error.

Al tratarse de un problema iterativo, debemos de asignar un numero fijo de iteraciones a ejecutar por el algoritmo o establece un criterio de parada automático basado como por ejemplo en el modulo de  $U_{actual} - U_{anterior}$ .

```

1 def HS_optical_flow(img1, img2, its=300, alpha=2, delta=10**-1, convergence=True):
2     if len(img1.shape) < 3 and len(img1.shape) < 3:
3         dim = img1.shape
4         kernel = np.array([[1/12, 1/6, 1/12], [1/6, 0, 1/6], [1/12, 1/6, 1/12]])
5         U = np.zeros_like(img1)
6         V = np.zeros_like(img1)
7         flow = np.zeros((dim[0], dim[1], 2))
8
9     Ix, Iy, It = get_gradients_optical_flow(img1, img2)
10
11    iter_counter = 0
12
13    while True:
14        iter_counter += 1
15        # Check if our iteration is on range.
16        if iter_counter > its:
17            break
18
19        # Compute local averages of the flow vectors
20        uAvg = cv2.filter2D(U, -1, kernel)
21        vAvg = cv2.filter2D(V, -1, kernel)
22        uNumer = np.multiply((np.multiply(Ix, uAvg) + np.multiply(Iy, vAvg) + It), Ix)
23        uDenom = alpha ** 2 + np.power(Ix, 2) + np.power(Iy, 2)
24        # U -> flow[:, :, 0]
25        U_prev = U
26        U = uAvg - np.divide(uNumer, uDenom)
27
28        vNumer = np.multiply((np.multiply(Ix, uAvg) + np.multiply(Iy, vAvg) + It), Iy)
29        vDenom = alpha ** 2 + np.power(Ix, 2) + np.power(Iy, 2)
30        # V -> flow[:, :, 1]
31        V = vAvg - np.divide(vNumer, vDenom)
32        print("Iteration number: ", iter_counter)
33
34        #If Check if our iteration is on range.
35        if convergence:
36            diff = np.linalg.norm(U - U_prev, 2)
37            print("Iteration number: ", iter_counter, " / Error: ", diff, diff < delta)
38            if diff < delta:
39                break
40
41        flow[:, :, 0] = U
42        flow[:, :, 1] = V
43        return flow
44    else:
45        print("Error input data. Images must be in grayscale.")
46        return

```

### 3.1 Análisis de resultados

Después de la implementación y el testeo del algoritmo se comentan los resultados en este apartado. El método *Horn-Schonck* con pocas iteraciones obtiene resultados muy buenos con un tiempo menor al método de *Lukas-Kanade*. Este tiempo de ejecución es dependiente del tamaño de la imagen de entrada y de el número de iteraciones.

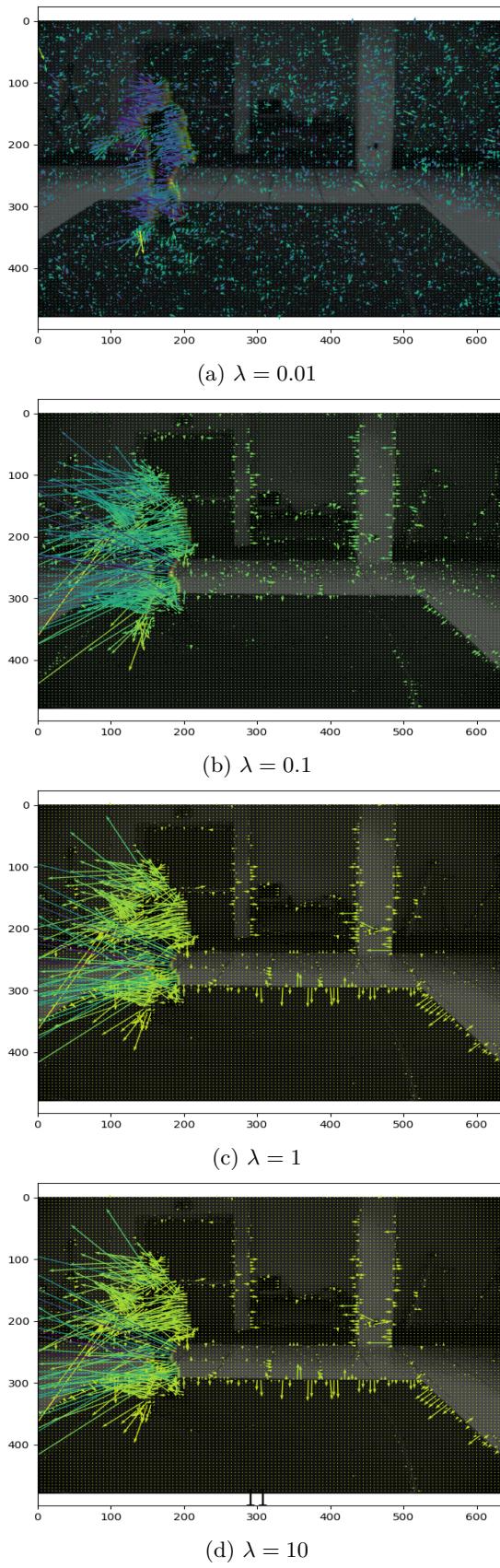


Figure 8: Resultados para 10 iteraciones. Tiempo ejecución: 0.4 segundos

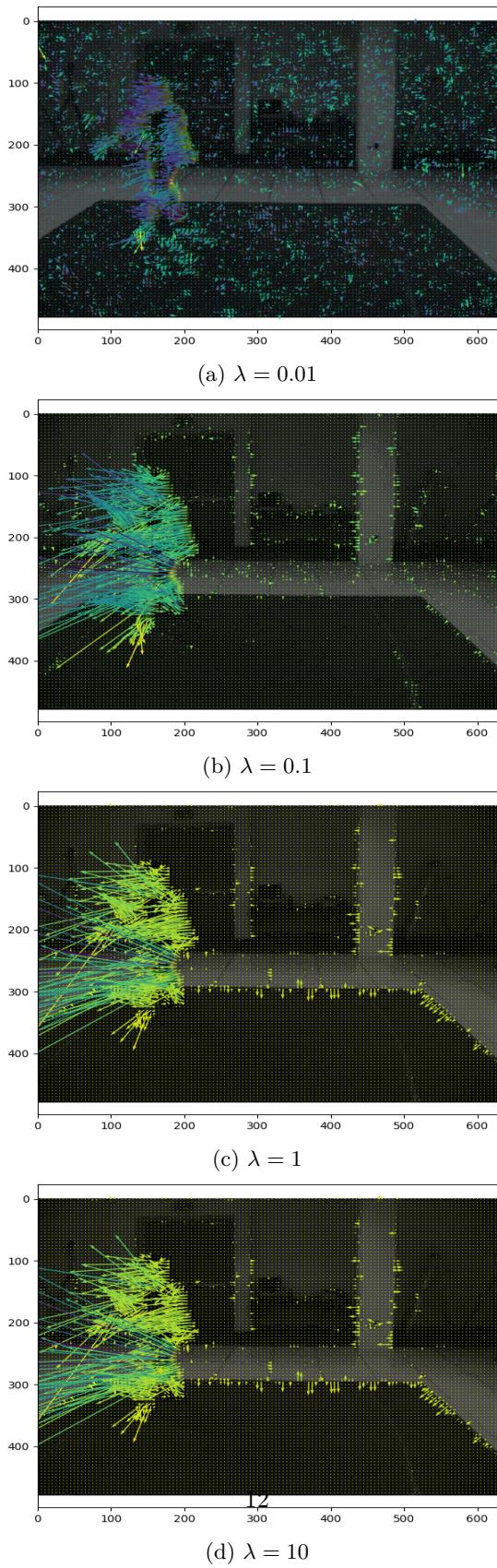


Figure 9: Resultados para 25 iteraciones. Tiempo ejecución: 1.02 segundos

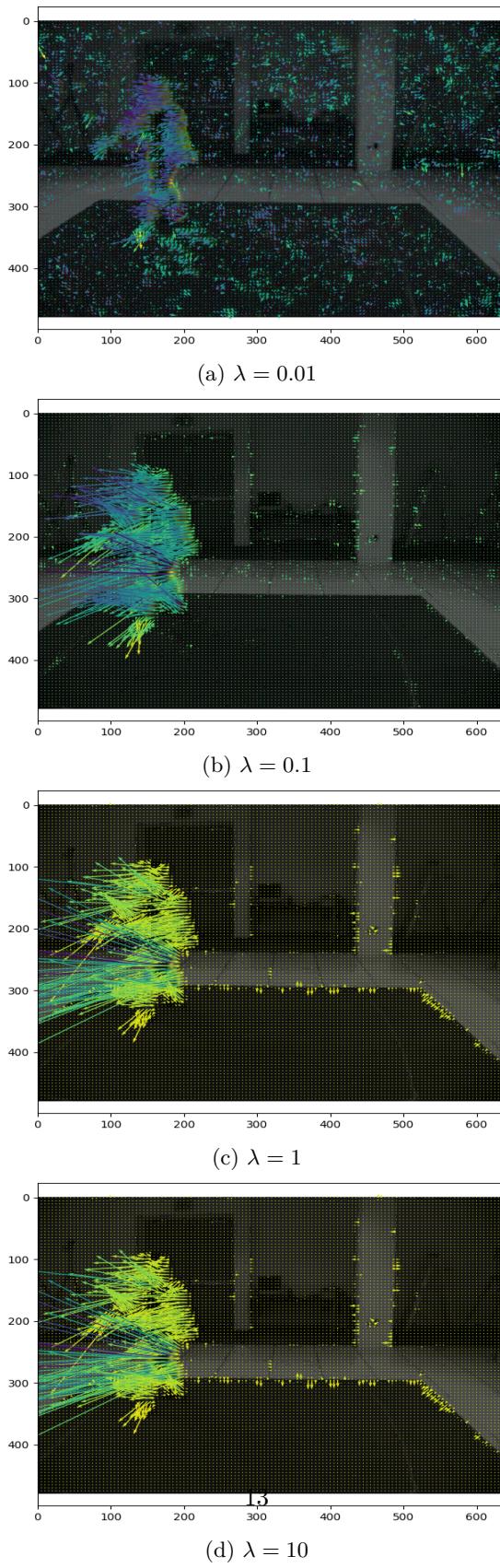
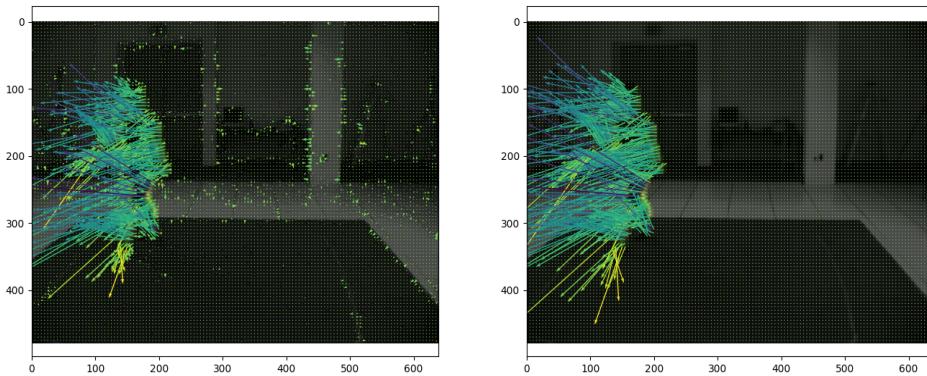


Figure 10: Resultados para 50 iteraciones. Tiempo ejecución: 2.03 segundos

Las figuras anteriores muestran varios ejemplos con diferentes iteraciones y valores de lambda. Revisando las imágenes podemos extraer que para valores más pequeños de lambda obtenemos más ruido en el resultado, es decir, es capaz de detectar flujo óptico en pequeñas áreas mientras que con un valor más elevado resalta más el objeto con más movimiento de la escena, reduciendo el ruido.

### 3.2 Filtrado de ruido

Este algoritmo se convierte en una muy buena solución cuando necesitamos resultados más rápidos, ya que es el más rápido de los tres probados. Incluso en menos de 1 segundo se pueden obtener buenos resultados aunque con algo de ruido. Para eliminar este ruido se ha programando una función de filtrado (post-procesado). Lo que pretendemos es eliminar el ruido pequeño que existe en la imagen. A continuación se puede ver un ejemplo:



(a) Imagen sin filtrado posterior. Tiempo de ejecución: 1.04 segundos.  
(b) Imagen con el filtrado posterior. Tiempo de ejecución: 1.09 segundos.

Figure 11: Resultados para 25 iteraciones y  $\lambda = 0.1$ .

A continuación se muestra la función que ha programado para la eliminación del ruido. Donde lo que se pretende es obtener el modulo de U y V, y eliminar un porcentaje ( $th$ ) de los valores menores del modulo.

```

1 def remove_noise(flow_in, th = 0.1):
2     dim = flow_in.shape
3     flow_out = np.zeros_like(flow_in)
4     magnitude = np.linalg.norm(flow_in, axis=2)
5     magnitude[magnitude < (th * (np.abs(np.max(magnitude) - np.min(magnitude)))] = 0
6
7     for y in range(0, dim[0]):
8         for x in range(0, dim[1]):
9             if magnitude[y, x] != 0:
10                 flow_out[y, x, :] = flow_in[y, x, :]
11
12     return flow_out

```

Esta función también se ha utilizado para el método de Lukas-Kanade.

## 4 Conclusiones

Se han implementado en *Python* los algoritmos de flujo óptico de *Lukas-Kanade* y *Horn-Schunck*. El primero esta basado el calculo diferencia y el segundo en calculo variacional.

Se han analizado los resultados de ambos métodos, comparando tiempos y diferentes parámetros para la ejecución. Los resultados para ambos métodos son muy buenos, obteniendo un correcto flujo óptico para las dos secuencias probadas.

El algoritmo de *Lucas-Kanade* parece obtener un resultado más fino en cuanto a las flechas obtenidas, aunque con un tiempo bastante superior con mi implementación (*bluces for*).

En cambio *Horn-Schunk* obtiene buenos resultado con bastante menor tiempo, por lo que va a depender de la aplicación final cual seria el método o parámetros óptimos para nosotros. Utilizando el método de convergencia se puede ahorrar tiempo de iteraciones, aunque hay que fijar un umbral experimentalmente.

Utilizando la función desarrollada para la eliminación de ruido, se consiguen mejorar ambos resultados con un muy poco más de tiempo de procesado.

## 5 Bibliografía

- Transparencias de la asignatura Visión Dinámica del Máster Universitario de Visión Artificial. Antonio Sanz Montemayor, Juan José Pantrigo y David Concha.
- Lucas, Bruce and Kanade, Takeo. (1981). An Iterative Image Registration Technique with an Application to Stereo Vision.
- Horn, Bertholdand Schunck, Brian. (1981). Determining Optical Flow. Artificial Intelligence.
- Meinhardt-Llopis, Enric and Sánchez Pérez, Javier and Kondermann, Daniel. (2013). Horn-Schunck Optical Flow with a Multi-Scale Strategy. Image Processing On Line.
- [Flujo Óptico Lukas-Kanade y Gunner Farneback. Created by: UniPython](#)
- [ntroduction to Motion Estimation with Optical Flow. Created by: NanoNets](#)
- [Advanced Computer Vision – Motion Estimation With Optical Flow. Created by: datahacker.rs](#)
- [Optical Flow Using Horn and Schunck Method. Created by: datahacker.rs](#)

## 6 Anexos

### 6.1 Obtener derivadas espacio-temporales

```
1 def get_gradients_optical_flow(img1, img2, Show=False):
2     kernelX = np.array([[-1, 1], [-1, 1]]) * 0.25
3     kernelY = np.array([[1, -1], [1, 1]]) * 0.25
4     kernelT = np.ones([2, 2]) * 0.25
5
6     # fx -> Ix = (Ix1 + Ix2) / 2
7     Ix = (cv2.filter2D(img1, -1, kernelX) + cv2.filter2D(img2, -1, kernelX)) / 2
8     # fy -> Iy = (Iy1 + Iy2) / 2
9     Iy = (cv2.filter2D(img1, -1, kernelY) + cv2.filter2D(img2, -1, kernelY)) / 2
10    # ft -> It = It2 - It1
11    It = cv2.filter2D(img2, -1, kernelT) + cv2.filter2D(img1, -1, -kernelT)
12
13    return Ix, Iy, It
```

### 6.2 Quiver Plot

```
1 def plot_quiver(ax, flow, spacing=5, normalize=False, **kwargs):
2     dim = flow.shape
3     x = np.arange(0, dim[1], spacing)
4     y = np.arange(0, dim[0], spacing)
5     flow = flow[np.ix_(y, x)]
6     X, Y = np.meshgrid(x, y)
7     n = -5
8
9     #Normalize the arrows:
10    if normalize:
11        eps = 0
12        #eps = np.finfo(np.float32).eps
13        U = np.divide(flow[:, :, 0], np.sqrt(flow[:, :, 0]**2 + flow[:, :, 1]**2) + eps)
14        V = np.divide(flow[:, :, 1], np.sqrt(flow[:, :, 0]**2 + flow[:, :, 1]**2) + eps)
15        U = np.nan_to_num(U)
16        V = np.nan_to_num(V)
17    else:
18        U = flow[:, :, 0]
19        V = flow[:, :, 1]
20
21    color_array = np.sqrt(((V - n) / 2)**2 + ((U - n) / 2)**2)
22    kwargs = {**dict(angles="xy", scale_units="xy"), **kwargs}
23    ax.quiver(X, Y, U, V, color_array, **kwargs)
24    ax.set_ylim(sorted(ax.get_ylimits(), reverse=True))
25    ax.set_aspect("equal")
26    return ax
```

### 6.3 Color Plot

```
1 def plot_rgb(flow, img):
2     dim = img.shape
3     mask = np.zeros((dim[0], dim[1], 3))
4     mask = mask.astype(np.uint8)
5     mask[:, :, 1] = 255
6
7     # Computes the magnitude and angle of the 2D vectors
8     magnitude, angle = cv2.cartToPolar(flow[:, :, 0], flow[:, :, 1])
9     # Sets image hue according to the optical flow direction
10    mask[:, :, 0] = angle * 180 / np.pi / 2
11    # Sets image value according to the optical flow magnitude (normalized)
12    mask[:, :, 2] = cv2.normalize(magnitude, None, 0, 255, cv2.NORM_MINMAX)
13    # Converts HSV to RGB (BGR) color representation
14    rgb = cv2.cvtColor(mask, cv2.COLOR_HSV2BGR)
15
16    return rgb
```

### 6.4 Combinación de quiver y color plot

```
1 def draw_optical_flow(flow, img2, quiver_plot=True, scale=None, color_plot=True):
2     fig2, ax2 = plt.subplots(figsize=(10, 6), nrows=1, ncols=1)
3     transp = 1
4     if quiver_plot:
5         plot_quiver(ax2, flow, 5, normalize=False)
6     if color_plot:
7         transp = 0.3
8         rgb1 = plot_rgb(flow, img2)
9         plt.imshow(rgb1, interpolation='none')
10    plt.imshow(img2, cmap='gray', interpolation='none', alpha=transp)
11
12    plt.show()
```