

VISIÓN DINÁMICA

Práctica I

Implementación, experimentación y análisis de los algoritmos de Lucas Kanade y Horn&Schunk para cómputo del flujo óptico.

Autor:

Jose Miguel Zamora Batista

17 de abril de 2021

Índice general

1. Lucas Kanade	1
1.1. Método de mínimos cuadrados: Descripción y análisis de la complejidad.	1
1.2. Método directo: Descripción y análisis de la complejidad.	3
1.3. Análisis de resultados y comparación de los métodos.	3
1.3.1. Comparación de tiempos	4
1.3.2. Comparación de resultados	5
2. Horn&Schunk	8
2.1. Método iterativo	8
2.2. Análisis de resultados	9
3. Conclusiones	11

Capítulo 1

Lucas Kanade

El algoritmo de flujo óptico de Lucas-Kanade es una técnica sencilla que puede estimar el movimiento en unas imágenes de una escena. Queremos asociar un vector de movimiento $\vec{V} = (u, v)$ a cada par de píxeles de la imagen en el instante t y en $t + 1$.

El algoritmo Lucas-Kanade hace algunas suposiciones implícitas:

- La iluminación es uniforme.
- Superficies con reflexión no anómala.
- Objeto debe moverse en plano paralelo al plano imagen
- Movimientos pequeños en los intervalos t y $t + 1$.
- Constancia de brillo y color

El algoritmo funciona tratando de adivinar en qué dirección se ha movido un objeto para poder detectar los cambios locales de intensidad. Es un método local y disperso, donde los píxeles de una región se mueven en bloque. Estas regiones se denominan vecindades.

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

1.1. Método de mínimos cuadrados: Descripción y análisis de la complejidad.

El método de mínimos cuadrados surgió por la necesidad de añadir alguna restricción para obtener un sistema compatible determinado y solucionar el problema de la apertura, proveniente de la ecuación ¹.

$$I_x u + I_y v + I_t = 0$$

Donde:

- I_x es la media de las derivadas en el eje horizontal de la imagen t y $t + 1$.
- I_y es la media de las derivadas en el eje vertical de la imagen t y $t + 1$.

¹Sólo se puede estimar la componente del flujo en la dirección del gradiente.

- I_t es la imagen diferencia entre $t + 1$ y t .

Este método asume que la velocidad en una vecindad es pequeña y constante. Esta vecindad es llamada ventana de integración. Por ejemplo, con una ventana de integración de tamaño 3×3 se tomarían 9 píxeles (*el central y los 8 de alrededor*), de esta forma se transforma la ecuación anterior en la siguiente.

$$\begin{bmatrix} I_{x1} & I_{y1} \\ I_{x2} & I_{y2} \\ \dots & \dots \\ I_{xN} & I_{yN} \end{bmatrix} \cdot \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -I_{t1} \\ -I_{t2} \\ \dots \\ -I_{tN} \end{bmatrix}$$

De esta forma, se tienen N ecuaciones y 2 incógnitas. Por lo que tenemos un sistema sobre-determinado que cumple la forma $Ax = b$.

Desarrollando, tenemos $x = (A^T A)^{-1} A^T b$, quedando:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i I_{xi}^2 & \sum_i I_{xi} I_{yi} \\ \sum_i I_{xi} I_{yi} & \sum_i I_{yi}^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i I_{xi} I_{ti} \\ -\sum_i I_{yi} I_{ti} \end{bmatrix}$$

Por un lado, se tiene la matriz 2×2 con los gradientes espaciales y el vector 2×1 de desajuste temporal.

La idea, principalmente, es recorrer todos los píxeles de la imagen, extraer su vecindad correspondiente y calcular el vector $[u \ v]^T$ aplicando la ecuación anterior.

Dicha ecuación se puede implementada en *Python* se puede codificar de la siguiente forma:

```

1  def lk_pinv(ix, iy, it):
2
3      _A = np.zeros((2, 2))
4
5      _A[0, 0] = np.sum(ix * ix)
6      _A[1, 0] = _A[0, 1] = np.sum(ix * iy)
7      _A[1, 1] = np.sum(iy * iy)
8
9      _b = np.zeros((2, 1))
10
11     _b[0, 0] = -np.sum(ix * it)
12     _b[1, 0] = -np.sum(iy * it)
13
14     _V = np.linalg.pinv(_A) @ _b
15
16     return _V

```

Se puede comprobar que la complejidad en tiempo de la ecuación depende del tamaño de la vecindad, concretamente, para un kernel de tamaño N la complejidad será $O(5N^2 + N^3)$ (por un lado las 10 operaciones suma y multiplicaciones y N^3 por el algoritmo del producto matricial.)². Por lo que la complejidad en notación “Big O” es $O(N^3)$ en cada vecindad. Es decir, este método depende enormemente del tamaño de la ventana de integración. En cuanto a la complejidad en memoria es despreciable en los estándares actuales, pero básicamente consiste en el alojamiento de 6 matrices con N^2 elementos cada 1. Por lo que en términos de complejidad sería $O(N^2)$ en cada vecindad.

²el algoritmo `pinv` se considera constante en este estudio por simplicidad

1.2. Método directo: Descripción y análisis de la complejidad.

El método directo surge al aplicar el desarrollo de la inversa del método anterior. Básicamente, se trata de un método que calcula, como su nombre indica, directamente los valores del vector $[u \ v]^T$ para cada píxel. Este método, como el anterior se aplica en la vecindad alrededor de cada píxel. El resultado de este desarrollo consiste:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -\sum_i I_{yi}^2 \sum_i I_{xi} I_{ti} + \sum_i I_{xi} I_{yi} \sum_i I_{yi} I_{ti} \\ \sum_i I_{xi}^2 \sum_i I_{yi}^2 - \sum_i I_{xi} I_{yi} \sum_i I_{xi} I_{yi} \\ \sum_i I_{xi} I_{yi} \sum_i I_{xi} I_{ti} - \sum_i I_{xi}^2 \sum_i I_{yi} I_{ti} \\ \sum_i I_{xi}^2 \sum_i I_{yi}^2 - \sum_i I_{xi} I_{yi} \sum_i I_{xi} I_{yi} \end{bmatrix}$$

Este resultado, codificado en *Python* se traduce en el siguiente código:

```
1  def lk_direct(ix, iy, it):
2
3      ix2  = np.sum(ix * ix)
4      iy2  = np.sum(iy * iy)
5      ixiy = np.sum(ix * iy)
6
7      ixit = np.sum(ix * it)
8      iyit = np.sum(iy * it)
9
10     d = ix2 * iy2 - ixiy ** 2 + 1e-9
11
12     u = (-iy2 * ixit + ixiy * iyit) / d
13     v = (ixiy * ixit - ix2 * iyit) / d
14
15     return [u, v]
```

En este código se encuentran 10 operaciones con matrices $N \times N$, siendo N el tamaño de la ventana de integración, y el resto de operaciones son con escalares, concretamente 12. Por lo que la complejidad en tiempo sería $O(10N^2) + 12 \sim O(N^2)$. En la parte de espacio que ocupa dicha operación, se trata de 3 matrices $N \times N$ y 8 valores flotantes, por lo que sería $O(3N^2 + 8) \sim O(N^2)$.

1.3. Análisis de resultados y comparación de los métodos.

Antes de comenzar con el análisis completo y la comparación, es importante conocer la raíz común de los algoritmos y estudiar la complejidad total. Como se ha comentado antes, tanto `lk_pinv` como `lk_direct` reciben una vecindad para obtener el valor del vector velocidad para el pixel central, sin embargo, antes se han tenido que calcular I_x , I_y y I_t desde las imágenes t y $t + 1$.

En *Python*, este procedimiento sería:

```

1  def optical_flow_lk(im_prev, im_next, window_size=3, funct=lk_pinv):
2      pad = window_size // 2
3
4      # Normalize images between (0, 1)
5      im0, im1 = im_prev.copy(), im_next.copy()
6
7      # Pad images
8      im0, im1 = np.pad(im0, pad), np.pad(im1, pad)
9
10     # Compute Ix, Iy, It
11     ix = gradient_x(im0, im1)
12     iy = gradient_y(im0, im1)
13     it = gradient_t(im0, im1)
14
15     # Allocate space for V = (u, v)
16     u, v = [np.zeros_like(ix) for _ in range(2)]
17     h, w = ix.shape
18
19     for row in range(pad, h - pad):
20         for col in range(pad, w - pad):
21             # Calculate slice indexes
22             ri, rj = row - pad, row + pad + 1
23             ci, cj = col - pad, col + pad + 1
24
25             # Slice the matrixes
26             _ix = ix[ri:rj, ci:cj]
27             _iy = iy[ri:rj, ci:cj]
28             _it = it[ri:rj, ci:cj]
29
30             # Compute optical flow
31             u[row, col], v[row, col] = funct(_ix, _iy, _it)
32
33     # Return non-padded (u, v) images.
34     return u[pad:h - pad], v[pad:w - pad]

```

Simplemente teniendo en cuenta el procesado por índice, es decir, los bucles `for` anidados se tiene una complejidad en tiempo $O(N * M * O_{funct})$ es decir tenemos que ejecutar la función `funct` por cada par de índice (i, j) de una matriz de N filas y M columnas. Teniendo esto en mente, las complejidades finales serían:

$$O(lk_pinv) = O(N * M * K^3)$$

$$O(lk_direct) = O(N * M * K^2)$$

Donde K sería el tamaño de la ventana de integración y determinando que ambos algoritmos dependen principalmente del tamaño de la imagen y del tamaño de la vecindad.

1.3.1. Comparación de tiempos

Para comenzar la comparación es interesante ver como se comportan ambos métodos en cuanto al tiempo de ejecución, la figura 1.1 muestra la ejecución de las implementaciones de Lucas Kanade antes descritas con diferentes tamaños de ventana de integración. Para esta evaluación la imagen de ejemplo contaba con una resolución de 640x480 píxeles.

Como se puede ver, las gráficas no siguen la complejidad teórica planteada anteriormente. Esto es debido a las optimizaciones implementadas en las operaciones matriciales por la

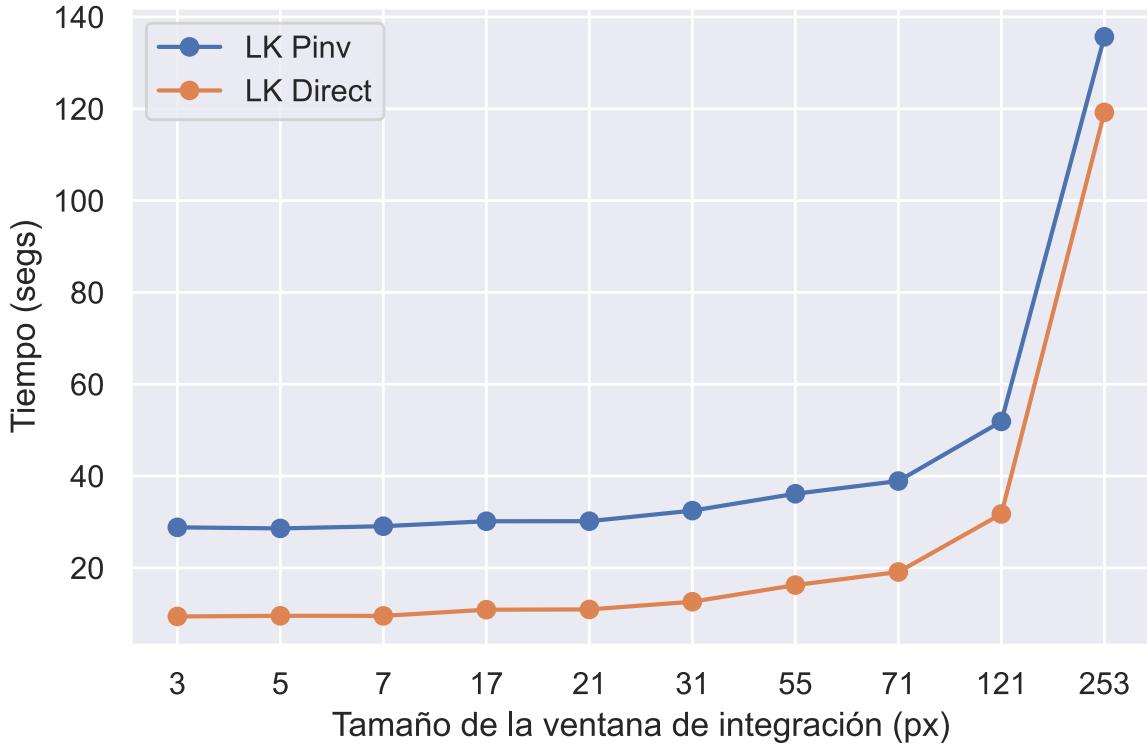


Figura 1.1: Tiempos de ejecución de Lucas Kanade para diferentes tamaños de ventana de integración

librería *Numpy*³ y el desenrollamiento de bucles de *Python*. Se puede apreciar, además la diferencia de tiempos entre los algoritmos se hace cada vez menor, esto podría ser debido al número de operaciones que consideramos atómicas (suma, producto de Hadamard, división) para el cálculo de la complejidad, pero que para tamaños de matrices más altos empiezan a tomar importancia. El número de este tipo de operaciones es de 10 para el caso de mínimos cuadrados contra 22 para el método directo. No obstante, veremos más adelante que no tienen sentido ciertos tamaños de ventana de integración. Por lo que, apriori, para resolver problemas de flujo óptico es mejor el método directo, en cuestión de tiempo de ejecución.

1.3.2. Comparación de resultados

Antes se vistió como influyen el número de operaciones en el tiempo final del algoritmo. Sin embargo, los resultados no se distancian entre sí, lo cual tiene sentido dado que el método directo es el desarrollo del método de mínimos cuadrados. A continuación se presentan algunos de dichos resultados.

En las figuras 1.2 se muestran las diferencias entre los métodos de Lucas Kanade. En la parte superior de cada una de las imágenes se puede ver la componente u y v del vector velocidad, así como su módulo de dicho vector y ángulo que forma con el eje horizontal (Imágenes simplemente de muestra para ver el funcionamiento del algoritmo).

³Principales optimizaciones SIMD: “Single Instruction Multiple Data” que aplican una instrucción a varios datos.

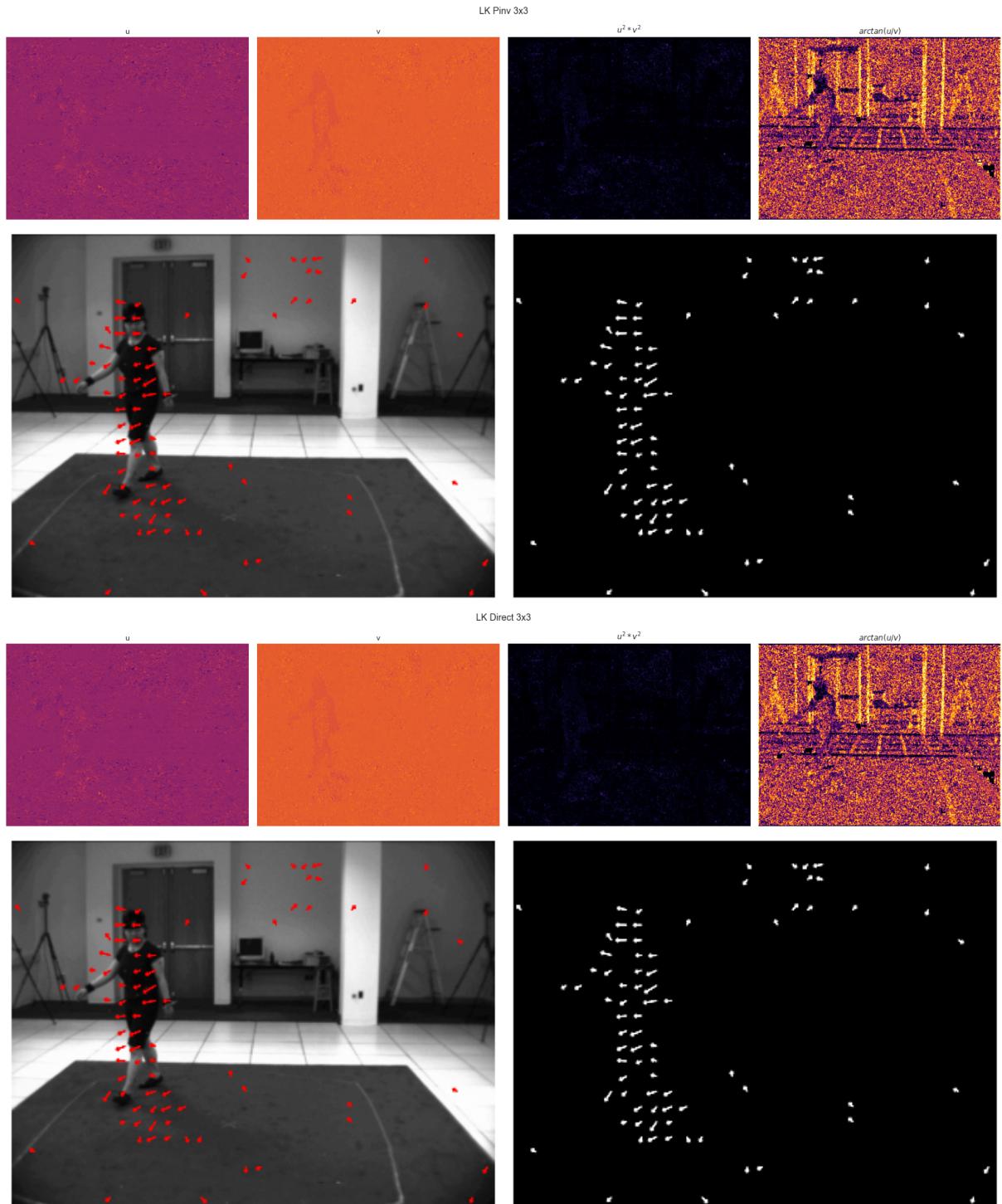


Figura 1.2: Lucas Kanade con ventana de integración 3x3

Como se puede ver, en comparación con las figuras 1.3 a mayor ventana de integración menor ruido, pero mayor dispersión del vector de velocidad en la zona del movimiento. Esta dispersión llega a ser tal que llega a aparecer flujo óptico donde realmente no lo hay.

Sin embargo, a mayor ventana de integración mayor posibilidad en la variación del movimiento de los objetos, a costa de disipar los movimientos pequeños.

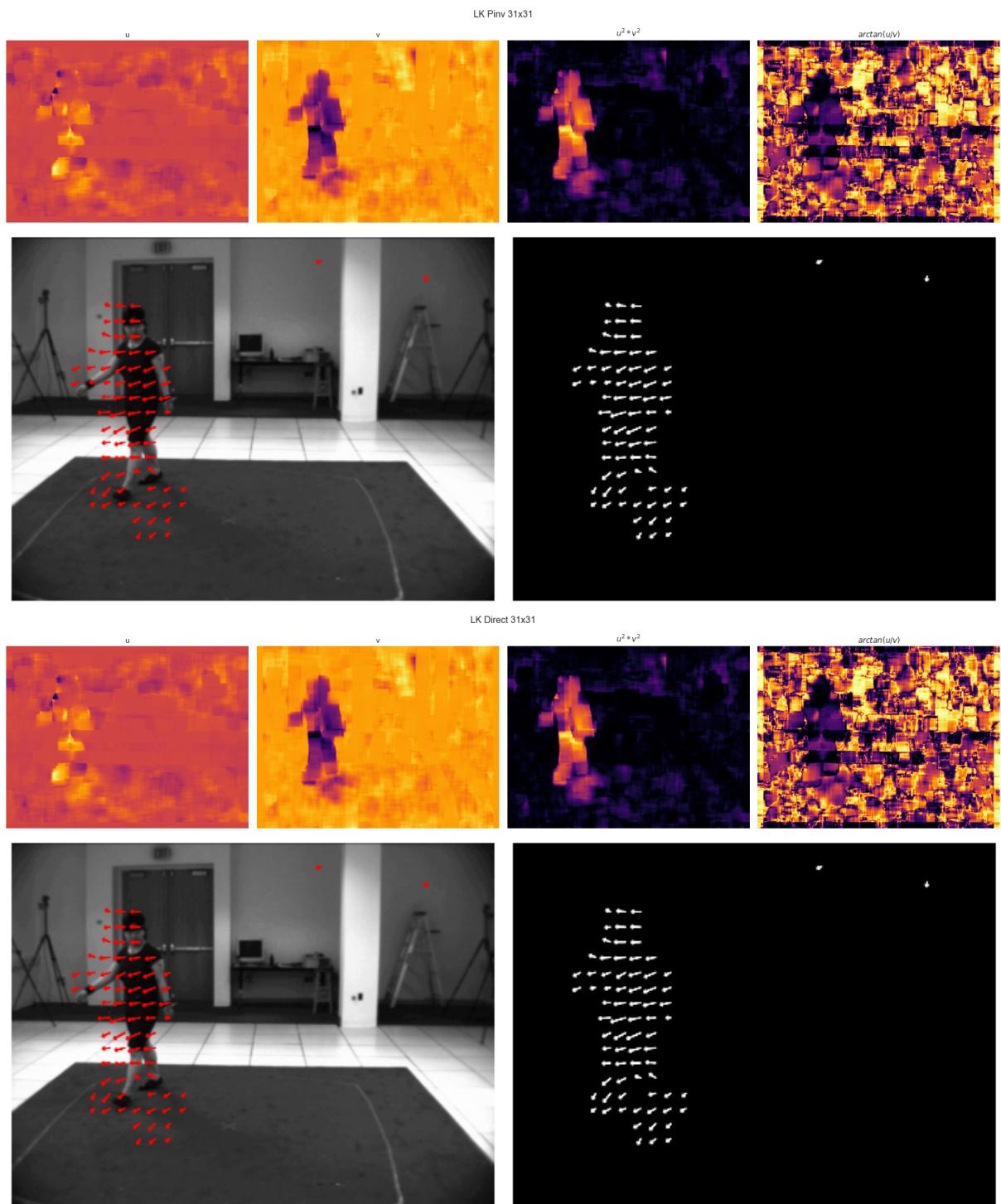


Figura 1.3: Lucas Kanade con ventana de integración 31x31

Capítulo 2

Horn&Schunk

Para estimar flujo óptico, el método HornSchunk introduce una condición de suavidad para resolver el problema de apertura. El método se califica como global porque determina el vector de movimiento del flujo óptico en cada píxel de la imagen, en contraposición de los métodos de Lucas Kanade descritos anteriormente.

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 ERFO⁴ con un error para la suavidad,

$$\varepsilon_S^2 = u_x^2 + u_y^2 + v_x^2 + v_y^2$$

Si denotamos como ε a la ecuación de la ERFO tenemos:

$$\varepsilon^2 = (I_x u + I_y v + I_t)^2$$

El error total que se busca es minimizar es:

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

Donde λ es un parámetro que actúa como ponderación de ambas magnitudes de error. Por lo que, λ es grande si la imagen no tiene mucho error y bajo si contiene mucho ruido.

2.1. Método iterativo

El error E^2 de antes se puede minimizar iterativamente a través de cálculo numérico (Gauss-Seidel).

$$u^k = \bar{u}^{k-1} - I_x \frac{I_x \bar{u}^{k-1} + I_y \bar{v}^{k-1} + I_t}{\lambda^2 + I_x^2 + I_y^2} \quad v^k = \bar{v}^{k-1} - I_y \frac{I_x \bar{u}^{k-1} + I_y \bar{v}^{k-1} + I_t}{\lambda^2 + I_x^2 + I_y^2}$$

⁴Ecuación de Restricción del Flujo Óptico

El código en *Python* quedaría:

```

1  def compute(ix, iy, it, lamb, n_iter, min_error=0.001):
2
3      u, v = np.zeros_like(ix), np.zeros_like(iy)
4      errors = []
5
6      for _ in range(n_iter):
7          # Compute means
8          u_avg = cv.GaussianBlur(u, (3, 3), 0)
9          v_avg = cv.GaussianBlur(v, (3, 3), 0)
10
11         # Common part
12         n = ix * u_avg + iy * v_avg + it
13         d = lamb ** 2 + ix ** 2 + iy ** 2
14         r = n / d
15
16         # Update new (u, v) values
17         u = u_avg - ix * r
18         v = v_avg - iy * r
19
20         # Compute error
21         errors.append(
22             compute_error(u, v, ix, iy, it, lamb)
23         )
24
25     return u, v, errors
26
27
28 def optical_flow_hs(im_prev, im_next,
29                     lamb=0.001, n_iter=8):
30
31     # Normalize images between (0, 1)
32     im0, im1 = im_prev.copy() / 255., im_next.copy() / 255.
33
34     # Compute Ix, Iy, It
35     ix = (np.gradient(im0, axis=0) + np.gradient(im1, axis=0)) / 2
36     iy = (np.gradient(im0, axis=1) + np.gradient(im1, axis=1)) / 2
37     it = im1 - im0
38
39     return compute(ix, iy, it, lamb, n_iter)

```

Como se puede ver en el código, la complejidad de este algoritmo recae en el número de iteraciones principalmente (ya que hemos considerado el resto de operaciones matriciales como atómicas). Es decir, si I es el número de iteraciones y N, M el tamaño de las imágenes, tendríamos una complejidad $O(I * N * M) \sim O(I)$.

2.2. Análisis de resultados

Este algoritmo es el que menos tarda de los tres, y dependiendo del tamaño de la imagen y del número de iteraciones puede estar por debajo del segundo de ejecución.

La figura muestra la diferencia de ejecuciones con el parámetro $\lambda = 0,005$ y $\lambda = 0,01$ como se puede ver, con un valor de λ pequeño se ve una imagen de flujo más ruidosa, es decir, que es capaz de detectar flujo óptico de forma global en pequeñas áreas, mientras que con un λ más elevado se realza la zona donde está el movimiento, disminuyendo el ruido.

En ambas ejecuciones se dejó un total de 20 iteraciones aunque a partir de la iteración 8 y 10 respectivamente el error estaba por debajo de $1e - 3$.

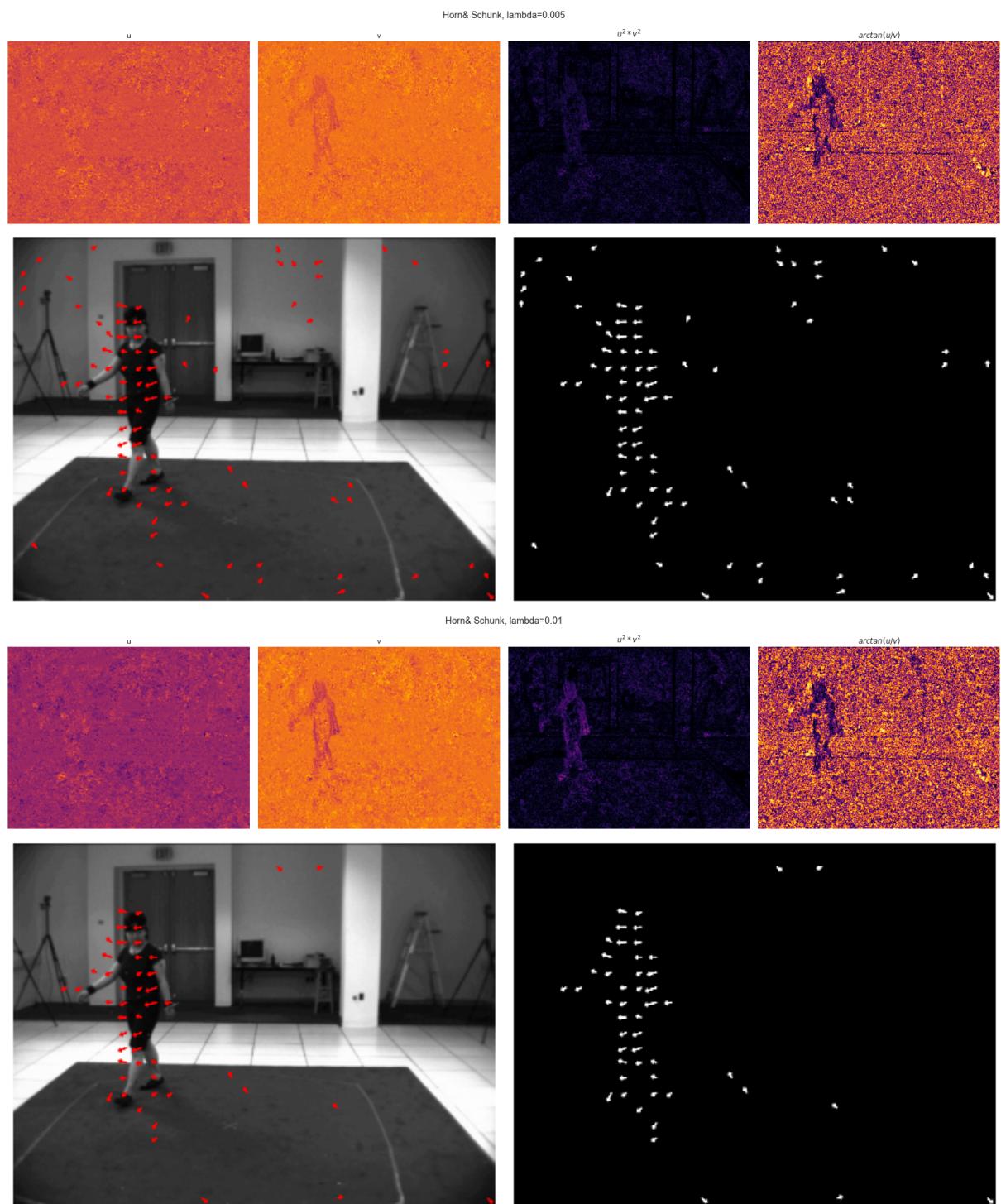


Figura 2.1: Horn&Schunk con lambdas: 0.005 y 0.01

Capítulo 3

Conclusiones

Para comprobar que los algoritmos fueron implementados correctamente, se han probado sobre imágenes de prueba para poder comparar resultados. En primer lugar, figura 3.1 muestra una esfera girando hacia la derecha.

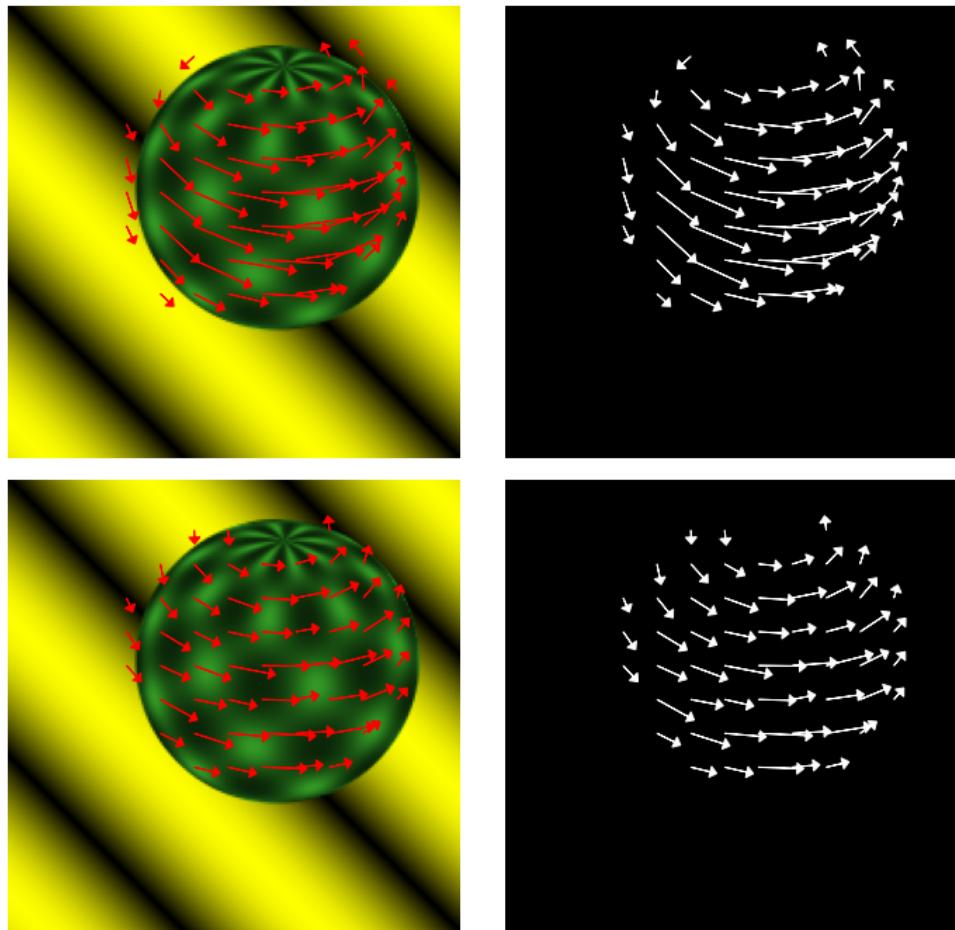


Figura 3.1: Lucas Kanade (15×15) contra Horn&Schunk $\lambda = 0,001$

Con ambos algoritmos, tanto con Lucas Kanade como con Horn&Schunk se muestra un correcto flujo óptico en ambas. Por lo que para esta implementación preferiría Horn&Schunk debido al tiempo de ejecución (por debajo del segundo).

La siguiente imagen a comprobar es una oficina en la que la cámara se aleja, figura 3.2,

como se puede ver el flujo óptico se representa bien en ambas. Aunque con Lucas Kanade hay más consistencia en las flechas.

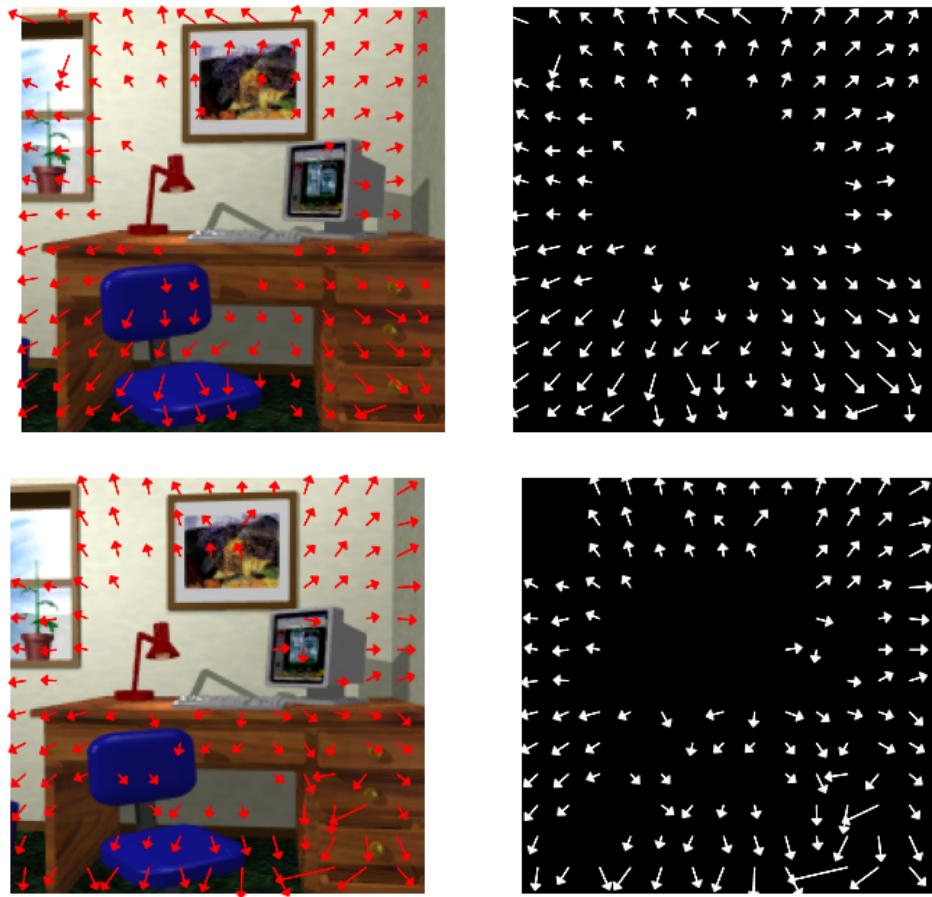


Figura 3.2: Lucas Kanade (15×15) contra Horn&Schunk $\lambda = 0,001$

Tanto el método de Lucas Kanade como Horn&Schunk son algoritmos para calcular el flujo óptico muy efectivos. Aunque, por el tiempo de ejecución el método de Lucas Kanade se vuelve completamente inútil para aplicaciones en tiempo real, (al menos con la implementación más básica).