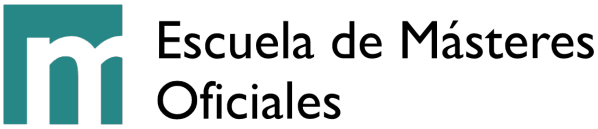
UNIVERSIDAD REY JUAN CARLOS

ESCUELA DE MÁSTERES OFICIALES

Máster Universitario en Visión Artificial



“FILTRO DE PARTICULAS”

Autor:
Vicente Gilabert Maño

MADRID, 2022

Contents

1 Introducción

En este trabajo se va a estudiar e implementar el filtro de partículas. Es muy utilizado en visión artificial y el robótica para la estimación de la posición de los objetos para poder realizar un seguimiento. Fue propuesto por primera vez por Gordon et al. (1993), aunque Isard y Blake (1996) realizaron una adaptación para la resolución de problemas de seguimiento.

Se puede implementar para seguir un objeto o múltiples, por lo que lo hace muy potente a diferentes escenarios.

La base de este algoritmo es aproximar una función de densidad de probabilidad (pdf) que describe el estado de un sistema. Esta pdf se aproxima mediante un conjunto de muestras discretas llamadas partículas. Cada partícula representa un posible estado del sistema, x_i, junto con su peso asociado, w_i, como medida de la verosimilitud de dicho estado $p_i = (x_i, w_i)$.

Consideraciones iniciales: El estado de la partícula vendrá determinado por la posición del objeto de interés en la imagen, es decir, [x, y]. Sin embargo, también se pueden añadir más estados a la partícula como por ejemplo el tamaño del objeto, en cuyo caso tendría la forma: [x, y, lx, ly].

Los pasos principales para la implementación del algoritmo son:

1. **Inicialización:** Esta etapa lleva a cabo una inicialización del estado de las partículas de tipo aleatorio sobre toda la imagen.

2. **Evaluación:** La etapa de evaluación debe estimar el peso de cada partícula, dada la medida. Se utiliza un filtrado de color para obtener los pesos. También se pueden utilizar otras técnicas como la sustracción de fondo.

3. **Estimación:** Puede estar basada en un promedio ponderado de los estados de las partículas o elegir el estado de aquélla con mayor peso. El resultado se representará online gráficamente como una caja que contenga al objeto seguido en cada fotograma.

4. **Selección:** Se utiliza el método de la ruleta para seleccionar las partículas de mayor peso.

5. **Difusión:** A las partículas seleccionadas en el instante anterior, se les aplicará una perturbación aleatoria con una distribución gaussiana.

6. **Modelo de movimiento:** El modelo de movimiento será, como mínimo, una perturbación gaussiana sobre los estados de las partículas.

2 Implementación

En los siguientes puntos se va a explicar los pasos realizados para la implementación del filtro de partículas.

2.1 Inicialización

En el primer paso del algoritmo, el usuario debe seleccionar la cantidad de partículas (N) y el tamaño de la partícula (M). En el siguiente ejemplo hemos utilizado $N = 100$ y $M = 50$.

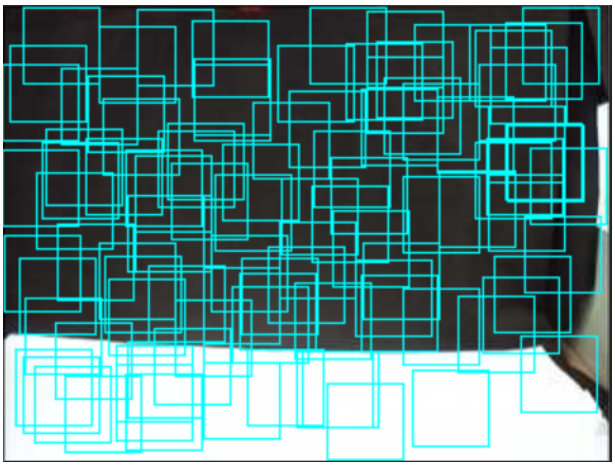


Figure 1: ds

Estas partículas son guardadas en un np.array((N,4)), donde cada partícula es representada por (x, y, vx, vy). Los valores de (x, y) representan la posición de la partícula y (vx, vy) es la estimación de movimiento de la partícula.

2.2 Evaluación

Una vez que tenemos todas las partícula aleatoriamente obtenidas, vamos a calcular cuales de estas partículas han caído dentro de la región del objeto que estamos buscando (pelota).

Utilizando un filtrado de color en el espacio hsv, se obtiene una máscara en cada *frame* donde obtenemos los píxeles que cumplen este filtrado. La forma en la que evaluamos a las partículas es simplemente comprobar cuantos píxeles blancos (del objeto) tenemos en cada partícula.

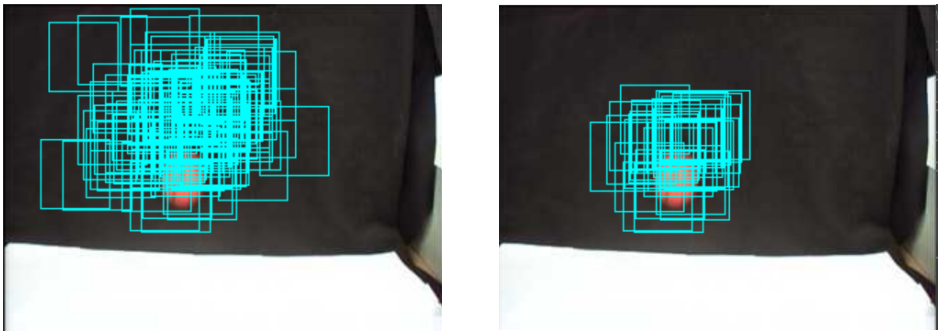
Figure 2: Imagen original a la izquierda. Imagen procesada con el filtrado de color a la derecha.

Para cada partícula se ha obtenido un peso, que se calcula con contar los píxeles blancos (del objeto) tenemos en cada una de las partículas y dividir esto por el área total de la partícula. Con esto obtenemos un valor que nos es independiente al tamaño de las partículas, aunque para nuestro problema no tendrá un efecto porque el tamaño de todas las partículas esta fijado con M.

2.3 Estimación y selección

Estas dos etapas, se ha decidido juntarlas en una, ya que la etapa de estimación es simplemente obtener las partícula con un peso diferente a 0. Con lo que estamos descartando aquellas que no solapan con el objeto de interés.

Una vez tenemos las partículas con cierto peso, lo que hacemos es un re-muestreo de las partículas utilizando el método de la ruleta. Lo que se consigue con esto es multiplicar aquellas partículas con peso.



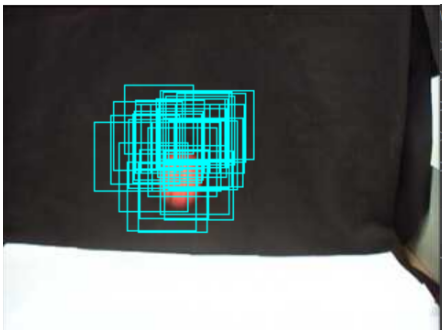
(a) Todas las partículas generadas del frame anterior.

(b) Solo partículas con peso y remuestreo posterior.

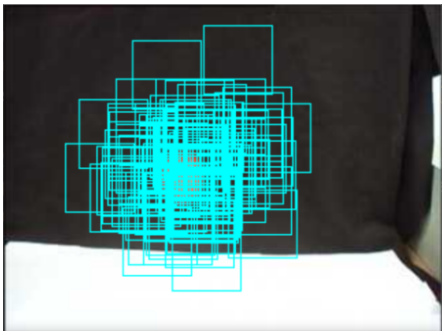
2.4 Difusión

Una vez tenemos a las partículas con peso re-muestredas para un total de N utilizando el método de la ruleta, lo que se aplica es una difusión sobre todas estas partículas.

Con este paso lo que conseguimos es darle robustez al algoritmo frente a cambios de dirección del objeto a seguir, porque si disponemos de suficientes partículas lo más seguro es que alguna llegue a solapar con el objeto.



(a) Etapa selección: solo partículas con peso y remuestreadas.

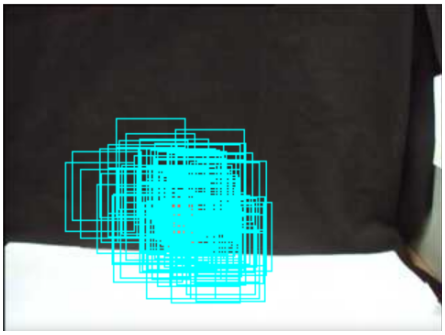


(b) Partículas con difusión.

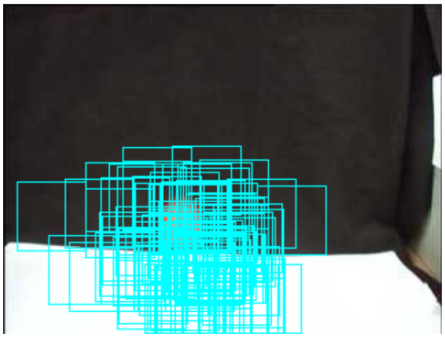
Para aplicar la difusión, lo que hacemos es a cada partícula aplicarle un movimiento aleatorio con una distribución guassina. También hemos utilizado elitismo, es decir la mejor partícula no es modificada mediante el proceso de difusión. Utilizando esta distribución estamos añadiendo un conocimiento a priori, donde el objeto en el siguiente frame es más probable que este cerca del estado actual. Uno de los parámetros que debemos ajustar de manera experimental es la desviación típica de esta distribución. En nuestro caso hemos elegido un valor de $std = 15$.

2.5 Predicción

En la etapa actual lo que se pretende estimar es la dirección de movimiento, por lo que podemos anticiparnos a un movimiento del objeto. Si el objeto sigue una trayectoria podremos estimar bien su posición y movimiento, mientras que si tiene un cambio de dirección, fallaremos en el movimiento pero alguna partícula solapara gracias a difusión.



(a) Partículas con difusión.



(b) Partículas con predicción de movimiento.

Como se puede ver en la figura anterior, la imagen de la derecha se puede ver como las partículas están más desplazadas hacia abajo. Esto es debido a la estimación de movimiento, ya que la pelota en este estado esta bajando.

Para la estimación de movimiento, lo que hacemos es ir aprendiendo la dirección de movimiento del objeto. Se obtiene una estimación de velocidad y con este velocidad se mueve la partícula.

$$v_x(t+1) = v_x(t) + N(0, std)$$
$$v_y(t+1) = v_y(t) + N(0, std)$$

$$x(t+1) = x(t) + v_x(t+1)$$
$$y(t+1) = y(t) + v_y(t+1)$$

Uno de los parámetros que debemos ajustar de manera experimental es la desviación típica de esta distribución gaussiana. En nuestro caso hemos elegido un valor de $std = 10$.

3 Resultados

Se adjunta algunas imágenes de todas las etapas en un $frame$ determinado.

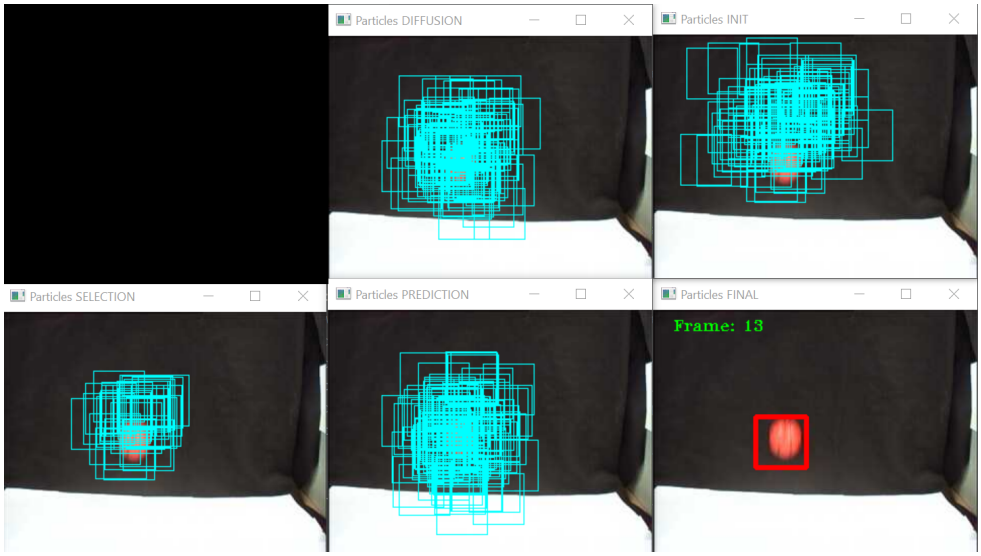


Figure 6

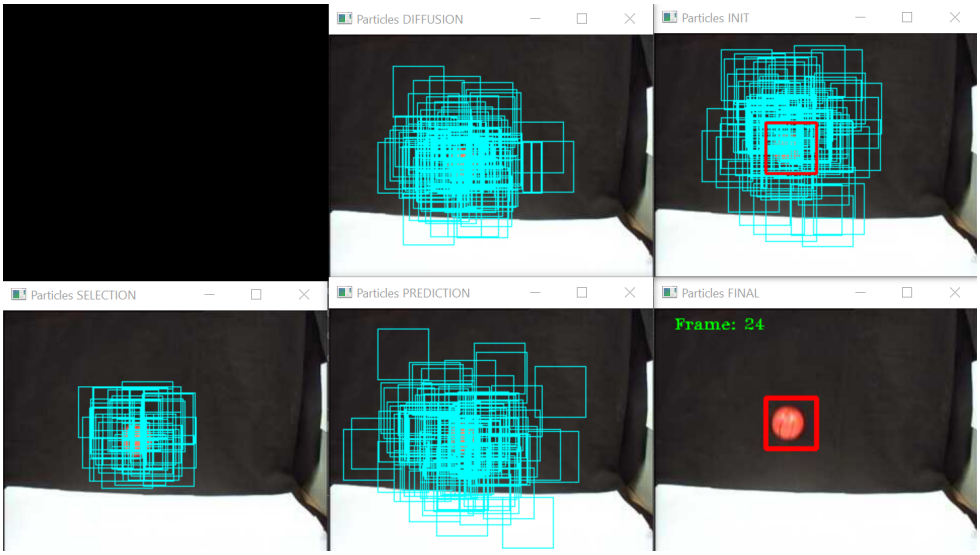


Figure 7

Se ha probado el algoritmo en un video de tenis para seguir a un jugador. Se utiliza un filtrado de color (rosa) para encontrar al jugador. Los parametros configurables son: $M = 50, N = 800$

Figure 8: Seguir al jugador de tenis.

Se han grabado algunos vídeos de diferentes ejecuciones para el mismo problema con diferentes parámetros. Se adjuntan los enlaces a continuación:

1. $difusion_{std} = 15$, $prediccion_{std} = 5$: Video 1 Youtube
2. $difusion_{std} = 15$, $prediccion_{std} = 10$: Video 2 Youtube

3. $difusion_{std} = 20$, $prediccion_{std} = 10$: Video 3 Youtube
4. FOLLOW ROGER FEDERER. $difusion_{std} = 15$, $prediccion_{std} = 10$: Video 4 Youtube

4 Conclusiones

Se ha implementado un algoritmo del filtrado de partícula utilizando un tamaño fijo de partícula (M) y estimando la velocidad de las partículas (v_x, v_y). En el caso de que durante la ejecución se pierda el objeto, se ha añadido un caso para empezar de nuevo y lanzar la partículas aleatoria miente en toda la imagen para tratar de recuperarla. Se ha observado que cuanto mayor es la desviación típica para la distribuciones de difusión y predicción, un resultado más fino se obtiene y se tiende menos a perder la pelota.
Se ha probado ha añadir dos estados más para controlar el tamaño de las partículas, pero no se ha resuelto de manera positiva. En algunos casos las partículas tendían a crecer de una manera extraña sin adaptarse a la forma del objeto.
Por lo tanto como concusión final, se ha implementado el filtrado de partículas para un objeto con cuatro grados de libertad (x, y, v_x, v_y). Se pueden ver resultados positivos en los vídeos incluidos en la memoria.

5 Anexos

En la siguiente sección se adjuntan los códigos de cada etapa del algoritmo o funciones utilizadas para alguna función en especial.

5.1 Repositorio

Se ha dejado el código completo de la práctica, en el repositorio de la asignatura. -> GITHUB

5.2 Filtrado de color (hsv)

Función desarrollada para el filtrado de color en el espacio HSV. Esta preparada para este problema, donde estamos detectando el color rojo de la pelota.

```
def hsv_filter(img):
    lower1 = np.array([0, 80, 80])
    upper1 = np.array([5, 255, 255])
    lower2 = np.array([170, 80, 80])
    upper2 = np.array([180, 255, 255])

    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    lower_mask = cv2.inRange(hsv, lower1, upper1)
    upper_mask = cv2.inRange(hsv, lower2, upper2)
    mask = lower_mask + upper_mask
    _, mask = cv2.threshold(mask, 20, 255, cv2.THRESH_BINARY)
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))
    mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel, iterations=2)

    return mask
```

5.3 Inicialización de partículas

Se inicializan las partículas aleatoriamente con un total de N y un tamaño de M.

```
def particles_init(img, N, M):
    dim = img.shape
    M_half = int(M / 2)
    new_particles = []
    vx = vy = 0

    for i in range(0, N):
        y = np.random.randint(M_half, dim[0] - M_half)
        x = np.random.randint(M_half, dim[1] - M_half)
        new_particles.append([x, y, vx, vy])
    return np.array(new_particles)
```

5.4 Obtener pesos

Se obtiene el peso de cada partícula con la imagen filtrada de color (máscara).

```
def particles_weight(img, particles, M):
    weights = []
    M_half = int(M / 2)

    for x, y, _, _ in particles:
        if np.sum(img) != 0:
            y1 = y - M_half
```

```
            y2 = y + M_half
            x1 = x - M_half
            x2 = x + M_half
            weights.append(np.sum(img[y1:y2, x1:x2]) / (4*M_half*M_half))
        else:
            weights.append(0)

    if np.sum(weights) != 0:
        # normalizar los pesos.
        weights = weights / np.sum(weights)

    # Obtener pesos acumulados.
    weights_cum = np.cumsum(weights)

    return np.nan_to_num(weights), np.nan_to_num(weights_cum)
```

5.5 Selección

Aplicando el método de la ruleta, se remuestrean las partículas con peso.

```
def selection_stage(particles, weights_cum):
    new_particles = []

    for _ in range(len(particles)):
        rdm = np.random.uniform()
        idx = np.argmin(rdm > weights_cum)
        new_particles.append(particles[idx])

    return np.array(new_particles)
```

5.6 Difusión

Se aplica una difusión a la posición de las partículas de manera aleatoria con un tamaño de la gaussiana (std).

```
def diffusion(img, particles, w, M, std=10):
    new_particles = []
    # Elitismo. Nos quedamos con la mejor particula antes de aplicar la difusion sobre el resto
    .
    new_particles.append(particles[np.argmax(w)])

    for i in range(len(particles)-1):
        x, y, vx, vy = particles[i]
        x_offset = int(np.random.normal(0, std))
        y_offset = int(np.random.normal(0, std))
        new_particles.append([x+x_offset, y+y_offset, vx, vy])

    # Funcion para evitar las que las partículas se salgan fuera de la imagen despues de la
    difusión.
    new_particles = particles_checkPosition(img, new_particles, M)
    return np.array(new_particles)
```

5.7 Predicción

Se obtiene la velocidad (estimación de dirección del movimiento) de las partículas.

```python
def prediction(img, particles, M, std=5):
    new_particles = []

    for i in range(len(particles)):
        x, y, vx, vy = particles[i]
        vx_offset = int(np.random.normal(0, std))
        vy_offset = int(np.random.normal(0, std))
        # Estimacion de movimiento.
        # vx(t+1) = vx(t) + N(0,std) // vy(t+1) = vy(t) + N(0,std)
        vx_new = vx + vx_offset
        vy_new = vy + vy_offset
        # Actualizamos la posicion de la particula con el movimiento anterior.
        # x(t+1) = x(t) + vx(t+1) // y(t+1) = y(t) + vy(t+1)
        x_new = x + vx_new
        y_new = y + vy_new
        new_particles.append([x_new, y_new, vx_new, vy_new])

    # Funcion para evitar las que las partículas se salgan fuera de la imagen despues de la prediccion.
    new_particles = particles_checkPosition(img, new_particles, M)
    return np.array(new_particles)
```

5.8 Comprobación de partículas

Esta función es utilizada para evitar que las partículas se salgan de la imagen, ya que al aplicar un movimiento aleatorio puede darse el caso de salirse fuera.

```python
def particles_checkPosition(img, particles, M):
    dim = img.shape
    M_half = int(M / 2)
    new_particles = []

    for x, y, vx, vy in particles:
        x_new = x
        y_new = y
        x1 = x - M_half
        y1 = y - M_half
        x2 = x + M_half
        y2 = y + M_half
        if x1 < 0: x_new = abs(x) + M_half + 1
        if y1 < 0: y_new = abs(y) + M_half + 1
        if x2 > dim[1]: x_new = dim[1] - M_half - 1
        if y2 > dim[0]: y_new = dim[0] - M_half - 1
        #print([x, y], [x1, y1, x2, y2], [x_new, y_new])
        new_particles.append([x_new, y_new, vx, vy])

    return new_particles
```

5.9 Dibujar mejor partícula

Dibujamos el resultado final, con la mejor partícula en color rojo.

```python
def draw_best_particle(img_in, particles, M, weight):

    if np.count_nonzero(weight) != 0:
        M_half = int(M / 2)
        best_particle = particles[np.argmax(weight)]
        drawed = cv2.rectangle(img_in, (best_particle[0]-M_half, best_particle[1]-M_half),
                               (best_particle[0]+M_half, best_particle[1]+M_half),
```

```
                                (0, 0, 255), 3)
        return drawed
    else:
        drawed = img_in
        return drawed
```