UNIVERSIDAD DE GRANADA

Visión por Computador
grado en ingeniería informática

Práctica 3

Detección de puntos relevantes y Construcción de panoramas

Autor
Vladislav Nikolov Vasilev

Rama
Computación y Sistemas Inteligentes

ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2019-2020

Índice

1. Detección de puntos Harris

En esta primera sección vamos a estudiar el comportamiento del detector de puntos Harris, y cómo podemos utilizarlo para extraer *keypoints* de la imagen. Para ello, vamos a implementar algunas funciones que simulen el comportamiento de las de `OpenCV`. Además, vamos a experimentar con los parámetros que reciben dichas funciones para ver cómo se ve afectada la salida. Finalmente, vamos a refinar un poco los puntos que hemos obtenido, para ver cómo de buenas son las estimaciones que hemos hecho.

1.1. Estimación de los *keypoints* utilizando Harris

Nuestro principal objetivo es, dada una imagen en escala de grises, obtener los *keypoints*, los cuáles pueden ser utilizados para otras muchas cosas, como por ejemplo para el *matching* entre imágenes. Existen muchos operadores que podemos utilizar para extraer los *keypoints*, pero nosotros aquí vamos a utilizar el operador de Harris, y lo combinaremos con algunas técnicas más. Es importante recalcar la parte de **imagen en escala de grises**, ya que Harris solo funciona con imágenes de este tipo.

Harris se utiliza principalmente para extraer las esquinas de una imagen. Aprovechando dicha información, podemos sacar información relevante que sea representativa de la imagen que tenemos a diversas escalas, de forma que en un principio nos fijamos en qué se encuentra en las escalas más bajas, y por tanto en detalles más destacables a simple vista, y luego vamos subiendo, hasta ver detalles algo más ocultos o pequeños.

Para extraer información interesante utilizando Harris, podemos seguir el siguiente esquema:

1. Extraer puntos de Harris de la imagen.
2. Aplicar un umbral a los puntos obtenidos anteriormente, eliminando aquellos que no lo superen.
3. Aplicar supresión de no máximos, quedándonos solo con el máximo local.
4. Utilizar los puntos restantes como *keypoints*, obteniendo información sobre sus posiciones, la escala y su orientación.

Este proceso se ha extraído del paper de Matthew Brown [1], y si se quiere tener información más en detalle, se recomienda consultarlo. Nosotros iremos mucho más al grano, y explicaremos lo esencial.

Lo primero que necesitamos es tener la pirámide Gaussiana de la imagen, ya que así tenemos distintas escalas de la imagen. También necesitamos las pirámides de las derivadas, ya que las utilizaremos para calcular las orientaciones de los píxels en cada una de las escalas. De esta forma, tenemos todas las escalas de la imagen y de las derivadas asociadas a dichas escalas calculadas desde un principio, con lo cuál no necesitamos ningún cálculo adicional en el proceso.

Para obtener la pirámide la imagen original, nos hemos ayudado de la siguiente función:

```
def compute_gaussian_pyramid(img, n_octaves):
    """
    Funcion que permite calcular una piramide Gaussiana de n_octaves escalas

    Args:
        img: Imagen de la que extraer la piramide
        n_octaves: Numero de octavas que tiene que tener la piramide
    Return:
        Devuleve la piramide Gaussiana de la imagen de entrada
    """
    # Crear lista que contendra la piramide Gaussiana
    # Inicialmente contiene la imagen de entrada (el nivel mas bajo)
    gauss_pyr = [img]

    # Obtener piramide
    for i in range(1, n_octaves):
        gauss_pyr.append(cv2.pyrDown(gauss_pyr[i-1]))

    return gauss_pyr
```

Para obtener la pirámide, nos hemos ayudado de la función `pyrDown` de `OpenCV`. Esta función lo que hace es devolver el siguiente nivel de la pirámide, encargándose de aplicar el alisamiento Gaussiano correspondiente y de reducir el tamaño de la imagen en el proceso a un cuarto de la original (se reduce la mitad en el eje de las X y la mitad en el eje de las Y.

Para extraer las pirámides de las derivadas, hemos utilizado la siguiente función:

```
def compute_derivative_pyramids(img, ksize_der, n_octaves, sigma=4.5):
    """
    Funcion que calcula las piramides Gaussianas de las derivadas en los
    ejes X e Y dada una imagen de entrada. La imagen de entrada ese alisada
    inicialmente con un filtro Gaussiano de sigma 4.5

    Args:
        img: Imagen de la que extraer las piramides de las derivadas
        ksize_der: Tamaño del kernel de la derivada
```

```
        n_octaves: Numero de imagenes que compondran las piramides
        sigma: Sigma del alisamiento Gaussiano (default: 4.5)
    Return:
        Devuelve dos listas, una para la piramide de las derivadas
    en el eje
        X y otra para la piramide de las derivadas en el eje Y
    """
    # Aplicar alisamiento Gaussiano
    smooth = gaussian_kernel(img, int(3*sigma) * 2 + 1, sigma)

    # Calcular derivadas
    dx = derivative_kernel(smooth, ksize_der, 1, 0)
    dy = derivative_kernel(smooth, ksize_der, 0, 1)

    # Añadir derivadas a las correspondiendtes listas
    dx_pyr = [dx]
    dy_pyr = [dy]

    # Crear piramide
    for i in range(1, n_octaves):
        dx_pyr.append(cv2.pyrDown(dx_pyr[i-1]))
        dy_pyr.append(cv2.pyrDown(dy_pyr[i-1]))

    return dx_pyr, dy_pyr
```

Se puede ver que en general no hay mucho misterio en el proceso, ya que es bastante parecido a lo que se pudo ver en la función anterior. Al principio se hace un alisamiento Gaussiano con $\sigma = 4.5$, tal y como se indica en el *paper*. El tamaño del *kernel* es proporcional a σ, de forma que se está discretizando en el rango $[-3\sigma, 3\sigma]$. Se puede ver como dicho tamaño se incrementa en uno en la función, con el objetivo de que sea impar. Al aplicar este alisamiento lo que conseguimos es eliminar las frecuencias altas de la imagen, eliminando ruido. Después calculamos las derivadas con las funciones de la práctica anterior, aplicando por debajo el operador de `Sobel` con una apertura de `ksize` (un *kernel* de dicho tamaño, dicho de otra forma). Finalmente, sacamos las pirámides utilizando de nuevo `pyrDown`.

Vamos a ver ahora cómo se calculan ahora los puntos de Harris. Recordemos que los puntos se calculan para cada píxel de la imagen original, valiéndose para ello de la descomposición en valores singulares de una matriz determinada. Esta descomposición nos permite obtener dos valores singulares: λ_1 y λ_2. Lo importante de aquí es saber que estos valores ofrecen información sobre cómo de rápido cambian los valores de la intensidad de la imagen en una ventana de un tamaño determinado en los ejes X e Y respectivamente. Además, nos permiten deducir una serie de cosas:

- Si λ_1 y λ_2 son pequeños, entonces no se produce un cambio de intensidad en la región. Por tanto, se podría ignorar dicha información, ya que no aporta información relevante.

- Si uno es más grande que el otro, significa que hay un borde en el eje del mayor valor singular, ya que la variación de intensidad es mayor en un eje que en el otro.
- Si los dos son grandes, significa que hay una variación de la intensidad grande en ambos ejes, y por tanto, que estamos en una esquina.

Lo dicho anteriormente se puede ver también en la siguiente figura, donde además se muestra qué tipo de elipse formarían los valores singulares (ya que también se puede hacer una interpretación geométrica de lo dicho anteriormente):

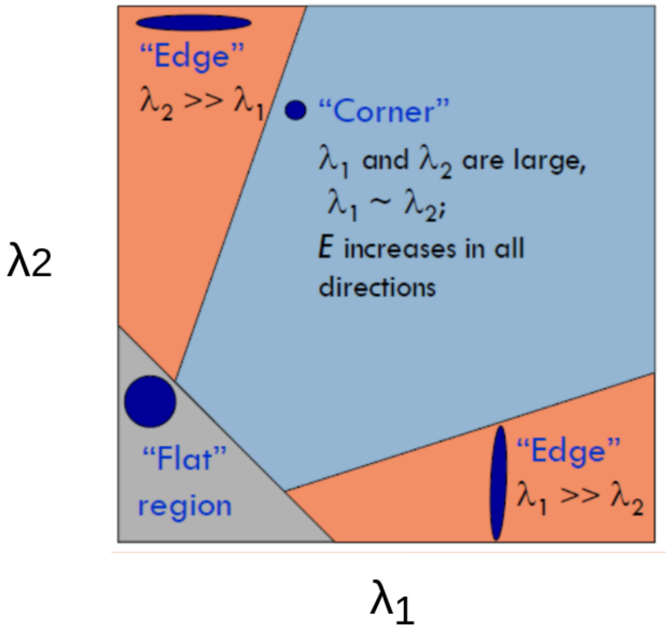

Figura 1: Ilustración de cómo interpretar los valores de λ_1 y λ_2.

Teniendo los valores anteriores para cada píxel, podemos calcular la media armónica de un píxel, la cuál llamaremos p, de la siguiente forma:

$$p = \frac{\lambda_1 \lambda_2}{\lambda_1 + \lambda_2} \tag{1}$$

Una vez dicho todo esto, vamos a ver cómo se podría implementar:

```python
def compute_points_of_interest(img, block_size, ksize):
    """
    Funcion que calucla los puntos de interes dada una imagen de
    entrada.
```

```
    Dichos puntos de interes son calculados mediante el operador de Harris.

    Args:
        img: Imagen de la que sacar los puntos de interes
        block_size: Tamaño del bloque que se va a tener en cuenta a la hora de
                        calcular los valores singulares.
        ksize: Tamaño del operador de Sobel
    Return:
        Devuelve una imagen del mismo tamaño que la entrada que contiene los
        puntos de interes calculados con el operador de Harris
    """
    # Obtener valores singulares y vectores asociados
    sv_vectors = cv2.cornerEigenValsAndVecs(img, block_size, ksize)

    # Quedarse solo con los valores singulares
    # Los valores singulares son los dos primeros valores de la matriz
    sv = sv_vectors[:, :, :2]

    # Calcular valor de cada pixel como \frac{lamb1 * lamb2}{lamb1 + lamb2}
    # Ahi donde el denominador sea 0, se pone un 0, para evitar que se calcule
    # un infinito
    prod_vals = np.prod(sv, axis=2)
    sum_vals = np.sum(sv, axis=2)
    points_interest = np.divide(prod_vals,
        sum_vals,
        out=np.zeros_like(img),
        where=sum_vals!=0.0
    )

    return points_interest
```

Lo primero que hacemos es utilizar la función `cornerEigenValsAndVecs()` de `OpenCV` para obtener los valores λ_1, λ_2 y los vectores propios asociados a cada uno de los valores singulares de cada píxel de la imagen. Esto nos dará de salida una matriz de las mismas dimensiones que la de entrada, pero cada posición contendrá los 6 valores anteriormente dichos. Los parámetros que se le pasan son `img`, que es la imagen de donde extraer la información, `block_size`, que indica cuál es el tamaño de la región alrededor del píxel que se debe consultar para obtener los valores singulares, y `ksize`, que indica el tamaño del *kernel* de Sobel que va a utilizar la función.

A continuación nos quedamos solo con los valores singulares, los cuáles están situados en las dos primeras posiciones. Calculamos la suma y el producto para cada par de λ_1 y λ_2, y después calculamos la media armónica. Para evitar problemas donde por ejemplo el denominador es 0, la operación solo se realiza para aquellos valores en los que el denominador sea distinto de 0. De esta forma, en las posiciones en las que no se dé, se pondrá un 0, ya que se considerará que no ofrecen información relevante. Dicha operación se puede ver en el código anterior en las líneas 27-31, donde en la línea 29 se declara que la salida será una matriz inicialmente con ceros y en la 30 se especifica que la división solo se haga en aquellas posiciones donde el denominador sea distinto de 0.

Una vez que hemos obtenido los puntos de Hrris, el siguiente paso es aplicar un umbral, de forma que los píxels de la imagen resultante que están por debajo del valor umbral serán eliminados, poniéndolos a 0. De esta forma, podemos eliminar aquellos puntos con valores bajos, ya que la mayoría de ellos estarán asociados a regiones planas, es decir, regiones donde la intensidad varíe muy poco, y por tanto, donde los valores λ_1 y λ_2 sean bajos. También es posible que en el proceso se elimine algún punto asociado a un borde que no sea muy importante, aunque eso depende bastante del valor umbral que se utilice. En el *paper* utilizaron un valor de 10, aunque nosotros probaremos luego con otros valores.

Para aplicar el umbral a la imagen, hemos hecho una función, la cuál se muestra a continuación:

```
def threshold_points_of_interest(points, threshold):
    """
    Funcion que aplica un umbral sobre una imagen, poniendo los pixels por
    debajo del umbral a 0

    Args:
        points: Puntos/Imagen sobre la que aplicar la umbralizacion
        threshold: Valor umbral
    Return:
        Devuelve una imagen en la que los valores por debajo del umbral han
        sido puestos a 0
    """
    points[points < threshold] = 0.0
```

Lo único que se hace es encontrar las posiciones en las que el píxel tenga un valor inferior al umbral y se pone dicho píxel a 0.

Posteriormente, tenemos que aplicar la supresión de no máximos, de forma que eliminamos los valores que no sean máximos locales. De esta forma, eliminamos valores que puedan estar asociados a ruido. El código es casi el mismo que el utilizado en la práctica 1. La única diferencia es que ahora el tamaño de la ventana está parametrizado, pero el funcionamiento sigue siendo el mismo que teníamos anteriormente.

Una vez que hemos aplicado los pasos anteriores, los píxels que queden "vivos" en la imagen son los que ofrecen información relevante sobre esta, ya que son aquellos que podríamos decir que, en general, ofrecen información sobre las esquinas que se puedan encontrar en la imagen a una escala determinada. Por tanto, podríamos tratar dichos puntos como descriptores o *keypoints*, aunque nos falta algo más de información. Tenemos que conocer, aparte de la posición del punto, la escala en la que se ha detectado y su orientación.

Calcular la escala es algo trivial. Siguiendo las indicaciones proporcionadas, podemos calcular dicho valor como $blockSize \times nivel\_piramide$, donde $blockSize$ es el tamaño del bloque que se ha utilizado para calcular los valores de λ_1 y λ_2 para cada píxel, y $nivel\_piramide$ es, como su propio nombre indica, el nivel actual de la pirámide.

No obstante, el cálculo de la orientación no es tan directo como en el caso anterior, ya que necesitamos información sobre los gradientes de un punto determinado en una escala concreta. Afortunadamente, aquí es donde entran en juego las pirámides Gaussianas que calculamos al principio para las derivadas de la imagen, las cuales nos facilitan mucho la vida. Vamos a ver primero la implementación y luego comentaremos lo que se hace:

```python
def compute_orientation(dx_grad, dy_grad):
    """
    Funcion que calcula la orientacion del gradiente de una serie de puntos

    Args:
        dx_grad: Derivadas en el eje X
        dy_grad: Derivadas en el eje Y
    Return:
        Devuelve un array en el que estan las orientaciones de todos los
        pares de gradientes de dx_grad y dy_grad. Las orientaciones estan
        en grados, y se encuentran en el rango [0, 360)
    """
    # Obtener vectores u y sus normas
    u = np.concatenate([dx_grad.reshape(-1,1), dy_grad.reshape(-1,1)], axis=1)
    u_norm = np.linalg.norm(u, axis=1)

    # Calcular vectores [cos \theta, sen \theta]
    vec_cos_sen = u / u_norm.reshape(-1, 1)
    cos_vals = vec_cos_sen[:, 0]
    sen_vals = vec_cos_sen[:, 1]

    # Calcular sen/cos arreglando posibles errores como 0/0 y x/0
    # Se arreglan los errores poniendolos a 0.0
    orientations = np.divide(sen_vals,
        cos_vals,
```

```
26          out=np.zeros_like(sen_vals),
27          where=cos_vals!=0.0
28      )
29
30      # Obtener \theta usando arcotangente (resultado en radianes
31      # entre [-pi/2, pi/2])
32      orientations_rad = np.arctan(orientations)
33
34      # Obtener angulos y arreglarlos (sumar 180 grados en caso de que
         cos < 0
35      # y pasarlos al rango [0, 360], eliminando negativos)
36      orientations_degrees = np.degrees(orientations_rad)
37      orientations_degrees[cos_vals < 0.0] += 180.0
38      orientations_degrees[orientations_degrees < 0.0] += 360.0
39
40      return orientations_degrees
```

La función recibe como parámetro las derivadas en el eje X y en el eje Y de los puntos "vivos" de una escala determinada. Lo primero que hace es calcular el vector de gradientes $\mathbf{u}$ asociado a cada par de de derivadas, el cuál viene dado por $\mathbf{u} = [\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}]$. Se calcula dicho vector para todas las parejas a la vez, ya que solo consiste en juntar las derivadas, poniéndolas como vectores columna. También se calcula $|\mathbf{u}|$, que es la norma euclídea de $\mathbf{u}$.

Después, se divide cada vector $\mathbf{u}_i$ entre su norma $|\mathbf{u}|_i$, resultando en un vector donde tenemos que los valores son $[cos(\theta), sin(\theta)]$. Estos valores son el coseno y el seno del ángulo θ que queremos calcular. De aquí sacamos los valores de forma separada, para poder acceder a ellos de forma más sencilla. Los cosenos están en la primera columna, y los senos en la segunda.

Ahora, para obtener la orientación, lo primero que tenemos que hacer es dividir el seno entre el coseno para cada vector. De esta forma obtenemos $tan(\theta)$. Esta operación puede verse en las líneas 24-28. De nuevo, tal y como pasaba en el caso de los puntos de Harris, es posible que alguno de los senos sea 0. Para evitar que el resultado no sea válido, se aplica la corrección vista anteriormente, donde solo se realiza la operación allí donde el seno sea distinto a 0. En caso contrario, la salida generada es 0.

A partir del resultado anterior ya podemos sacar el ángulo. Como el resultado anterior es la tangente de θ, podemos aplicar la función arco tangente, la cuál es la inversa, para sacar el ángulo. El principal problema es que el valor devuelto está en radianes, y está en el rango $[-\frac{\pi}{2}, \frac{\pi}{2}]$. Por tanto, tenemos que pasar los ángulos de radianes a grados. Dicha transformación se realiza con una función de `numpy`, la cuál es `degrees()`. Esta función pasará los valores a grados, aunque estarán en el rango $[-90, 90]$. Por tanto, hay una serie de operaciones extra que tenemos que hacer:

1. Tenemos que identificar los puntos cuyo coseno sea negativo, ya que los valores obtenidos anteriormente se sitúan en el primer y en el cuarto cuadrante. Esto viene a raíz de que la tangente es positiva tanto en el primer como en el tercer cuadrante, y es negativa tanto en el segundo como en el cuarto cuadrante. Por tanto, tenemos que ajustar los ángulos obtenidos. Para hacer esto, podemos guiarnos por el coseno, ya que este es positivo en el primer y cuarto cuadrante y negativo en los otros dos. Por tanto, simplemente tenemos que buscar los índices de los cosenos que sean negativos, y sumar a los ángulos en las mismas posiciones 180, de forma que se ajusten al ángulo en el cuadrante que les corresponda.

2. Puede que aun queden valores negativos porque están en el cuarto cuadrante. En un principio, esto no debería ser un problema, pero `OpenCV` exige que los grados estén en el rango $[0, 360)$. Por tanto, a todos los ángulos menores que 0, hay que sumarles 360.

Después de todo este proceso, ya tenemos las orientaciones calculadas, y podríamos proceder a la creación de *keypoints* con la información que tenemos.

```python
def harris_corner_detection(img, block_size, window_size, ksize_der,
                            n_octaves, threshold=10.0):
    """
    Funcion que detecta los puntos de Harris de una imagen a distintas
    escaslas.

    Args:
        img: Imagen de la que se quieren extraer los puntos de Harris
        block_size: Tamaño del bloque que se va a tener en cuenta a la hora de
                    calcular los valores singulares.
        window_size: Tamaño de la ventana al realizar la supresion de no
                     maximos
        ksize_der: Tamaño del operador de Sobel (utilizado en el calculo
                   de los valores singulares)
        n_octaves: Numero de octavas/escalas de la imagen de la que sacar
                   puntos
        threshold: Umbral utilizado para eliminar todos los valores inferiores
                   a este.
    Return:
        Devuelve dos listas: una que contiene los keypoints extraidos y otra
        que contiene los keypoints corregidos
    """
    # Obtener piramide gaussiana de la imagen
```

```python
    img_pyr = compute_gaussian_pyramid(img, n_octaves)

    # Obtener piramides de las derivadas
    dx_pyr, dy_pyr = compute_derivative_pyramids(img, ksize_der, n_octaves)

    # Lista de keypoints y keypoints corregidos
    keypoints = []
    corrected_keypoints = []

    for i in range(n_octaves):
        # Obtener puntos de interes de la escala
        points_interest = compute_points_of_interest(img_pyr[i],
            block_size,
            ksize_der
        )

        # Aplicar umbralizacion
        threshold_points_of_interest(points_interest, threshold)

        # Aplicar supresion de no maximos
        points_interest = non_max_supression(points_interest, window_size)

        # Obtener valores mayores que 0.0 (aquellos que no han sido eliminados)
        points_idx = np.where(points_interest > 0.0)

        # Calcular escala del KeyPoint
        # Hace falta incrementar el valor de i en 1 porque se empieza en 0
        scale = (i+1) * block_size

        # Obtener las derivadas correspondientes a los puntos no eliminados
        dx_grad = dx_pyr[i][points_idx]
        dy_grad = dy_pyr[i][points_idx]

        # Calcular orientaciones de los puntos no eliminados
        orientations = compute_orientation(dx_grad, dy_grad)

        # Lista que contiene los keypoints de la octava/escala
        # Se corrigen las coordenadas segun la escala
        keypoints_octave = [cv2.KeyPoint(x*2**i, y*2**i, scale, o)
                            for y, x, o in zip(*points_idx, orientations)]

        # Unir las coordenadas de forma que sean n vectores [x,y] formando una
        # matriz
        points_x = points_idx[0].reshape(-1,1)
        points_y = points_idx[1].reshape(-1,1)
        points = np.concatenate([points_x, points_y], axis=1)
```

```python

        # Establecer criterio de parada
        # Se parara o bien a las 15 iteraciones o cuando epsilon sea menor a 0.01
        criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 15, 0.01)

        # Corregir keypoints
        points = cv2.cornerSubPix(img_pyr[i],
            points.astype(np.float32),
            (3,3),
            (-1,-1),
            criteria
        )

        # Redondear, cambiar x por y y viceversa (OpenCV carga las imagenes
        # invirtiendo los ejes) y transformar coordenada a la de la imagen original
        points = np.round(points)
        points = np.flip(points, axis=1)
        points *= 2**i

        # Guardar keypoints y keypoints corregidos
        keypoints.append(keypoints_octave)
        corrected_keypoints.append(points)

    return keypoints, corrected_keypoints
```

2. Extracción de descriptores AKAZE

3. Mosaico de dos imágenes

4. Mosaico de N imágenes

Referencias

[1] M. Brown, R. Szeliski, and S. Winder. Multi-image matching using multi-scale oriented patches. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 510–517 vol. 1, June 2005.