VC

Visión por computador

Trabajo 3: Detección de puntos relevantes y construcción de panoramas

Autor: Jesus Medina Taboada

Correo: jemeta@correo.ugr.es

Profesor: Pablo Mesejo

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Curso 2020 - 2021

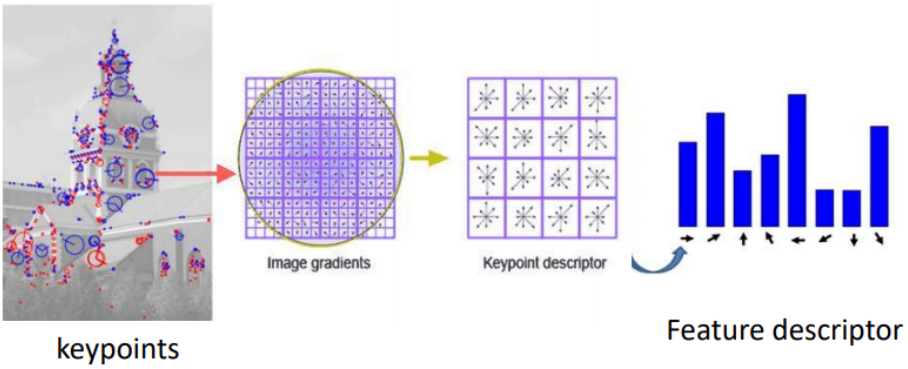
Índice general

0.1. Introducción

En esta práctica vamos a trabajar con puntos relevantes, también llamados keypoints y con la construcción de panoramas o mosaicos de imágenes alineadas.

En prácticas anteriores resolvimos problemas como el filtrado de bajas y altas frecuencias y la clasificación de imágenes. Sin embargo en este nuevo trabajo nos alejaremos de estas técnicas basadas en aprendizaje y nos centraremos en otro de los problemas clásicos de la visión por computador, moviendonos más hacia un enfoque de geometría. Pasamos pues, de problemas donde se aprende a realizar una tarea de reconocimiento visual a través de ejemplos, a problemas en donde las propiedades geométricas juegan un papel fundamental.

A continuación hablare brevemente sobre aspectos teoricos basicos para comprender mejor los puntos que trabajaremos.
Deteccion puntos de interés.
A partir de una imagen recuperamos una serie de puntos o localizaciones que contienen información relevante para llevar a cabo una cierta tarea. Se pretende resumir una imagen en una serie de puntos clave que permitan por ejemplo compararla o alinearla con otra. Existen muchos algoritmos en visión por computador que extraen keypoints de una imagen y que calculan un descriptor de características a partir de la región de píxeles que rodea a dicho puntos de interés. Es importante conocer la diferencia entre el punto de interés y el descriptor de características o 'feature descriptor' asociado al mismo. Un algoritmo que ya hemos estudiado y conocemos es SIFT, un ejemplo del proceso de su ejecución sería el mostrado a continuación. Se parte de una imagen sobre la que se detectan una serie de keypoints, se realizan una serie de operaciones sobre ellos obteniendo un vector de características de cada punto que nos permitirá hacer una comparación entre ellos para saber ver por ejemplo si son similares.



Resumiendo tendríamos 3 elementos clave, puntos de interés, detector(algoritmo que detecta los puntos) y descriptor(algoritmo que describe las características asociadas a dicho punto).
Hemos trabajado con detectores como bordes(Canny,Sobel), regiones(LoG,DoG) y ahora conoceremos los detectores de esquinas Harris. En cuanto a los descriptores estamos familiarizados con SIFT pero en este caso usaremos uno llamado KAZE el cual proporciona OpenCV.
OBJETIVO FINAL
Nuestro trabajo se desarrollará en 3 etapas detección-descripcción de puntos de interés, emparejamiento o matching y por último la creación de panoramas o mosaicos (image stiching). Como meta final buscaremos crear una composición dadas varias imágenes desde diferentes perspectivas de una misma escena.

Capítulo 1

Detección de puntos Harris

Lo primero será realizar la detección de puntos Harris, un punto Harris o esquina es una región de la imagen con gran variación en intensidad en todas direcciones.

Los puntos de Harris de una imagen se obtienen a partir de la Matriz de Harris de esa imagen I:

$$H(x,y) = \nabla_{\sigma_d} I(x,y) \nabla_{\sigma_d} I(x,y)^T * g_{\sigma_i}(x,y)$$

Podemos aproximar la matriz de Harris si localmente tomamos un vecindario de pixeles o ventana(W):

$$H = \begin{pmatrix} \sum_{(x,y)\in W} I_x^2 & \sum_{(x,y)\in W} I_x I_y \\ \sum_{(x,y)\in W} I_x I_y & \sum_{(x,y)\in W} I_y^2 \end{pmatrix}$$

Lo que nos interesa en cada punto de la imágen es conocer los valores propios de la matriz H. Con ellos podemos aplicar un criterio para escoger si son relevantes para nuestro problema o no.

$$f_{HM}(x,y) = \frac{det(H(x,y))}{tr(H(x,y))} = \frac{\lambda_1 \lambda_2}{\lambda_1 + \lambda_2}$$

Este criterio conocido como Criterio Harris nos permite calcular un valor(f) en cada punto(x,y) a partir de los valores propios de su matriz H. Si este valor supera un cierto umbral podremos considerarlo relevante, de lo contrario lo descartamos.
Podemos saber que si ambos valores singulares son grandes, el valor f lo será también. Si alguno de los valores es pequeño, f puede ser pequeño ya que el producto decrece rápidamente. Y si ambos son pequeños, f tiende a 0. Debemos tener cuidado con el caso en que la suma de ambos valores sea igual a 0 ya que f no se podría calcular al ser un cociente con denominador cero. Si se da este caso fijamos el valor f a cero.

```
def criterio_harris(l1,l2):
    if (l1+l2 == 0):
        return 0
    return (l1*l2)/(l1+l2)
```

1.1. Pirámide Gaussiana

Para calcular los valores propios de la matriz H nos ayudaremos de la función de OpenCV **cornerEigenValsAndVecs** que nos devuelve directamente los valores propios en cada punto de la imagen.Esta tiene 2 argumentos: blocksize que es el tamaño de la ventana/vecindario y ksize que es el tamaño del filtro para calcular las derivadas.
Para ser más exactos la función calcula para cada pixel(p) y cosiderando para cada vecindario de un tamaño dado(nxn el cual discutiremos más adelante) la matriz de covariación de derivadas de la siguiente forma y donde las derivadas son computadas a través del operador Sobel. Para el calculo de la derivada como las imágenes de ejemplo proporcionadas no son muy ruidosas se ha escogido un tamaño 3x3.

$$M = \begin{bmatrix} \sum_{S(p)} (dI/dx)^2 & \sum_{S(p)} dI/dx dI/dy \\ \sum_{S(p)} dI/dx dI/dy & \sum_{S(p)} (dI/dy)^2 \end{bmatrix}$$

Devolviendo un array de 6 valores. Los 2 valores propios de M(l1,l2), los vectores propios(x1,y1) correspondiente al primer valor propio y los vectores propios(x2,y2) correspondiente al segundo valor propio.

Una vez calculamos el valor f a todos los puntos de la imagen, los cribamos por un umbral que hemos determinado 10.
Ya sabemos como calcular puntos Harris, ahora vamos a aplicarlo sobre una pirámide Gaussiana de 3 niveles y de sigma 1, es decir calcular los puntos en 3 escalas distintas. Esto nos libra de la decisión del tamaño de ventana que usaremos debido a que una vez que definamos el tamaño de la ventana a usar en el nivel más bajo de la pirámide(imagen original) no habrá que cambiarlo más.
Al pasar el nivel siguiente de la pirámide Gaussiana la ventana que usaríamos sería como multiplicar por 2 su tamaño, si empezamos con un 3x3 en el nivel 1, al pasar al nivel 2 sería como usar en esencia un 7x7 y al pasar al nivel 3 un 14x14.
De esta forma conseguimos extraer puntos con distintas escalas.
Luego solo queda calcular los puntos Harris en cada uno de los niveles de la

pirámide y para cada imagen. Pero antes realizaremos una supresión de no máximos y calcularemos la orientación del punto para luego poder dibujarlo.

```
def calculate_harris(src,level,block_size = 3,ksize = 3,threshold = 10):
    # Obtenemos los valores y vectores propios(l1,l2, eiv11,eiv12, eiv21,eiv22)
    e_v = cv.cornerEigenValsAndVecs(src,blockSize = block_size,ksize = ksize)

    # Calculamos matriz con valor segun criterio harris
    first_m = np.asarray([[ criterio_harris(e_v[i,j,0],e_v[i,j,1])
                for j in range(src.shape[1])]
                for i in range(src.shape[0])])

    # Nos quedamos con los valores mayores que un umbral
    threshold_m = np.asarray([[ first_m[i,j] if first_m[i,j] > threshold else 0
                    for j in range(first_m.shape[1])]
                    for i in range(first_m.shape[0])])


    # Suprimimos los no máximos en una distancia X por X del vecindario
    sup_no_max_m  = supresionNoMax(threshold_m,5)

    # Return keypoints
    return get_keypoints(sup_no_max_m,block_size,level)
```

1.2. Supresión de No-Máximos

Una vez tenemos los puntos Harris de cada nivel de la pirámide para cada imagen, vamos a someterlos a una supresión de no máximos. Principalmente lo hacemos porque en este punto tenemos puntos Harris con valores altos y muy cercanos entre si, debido a que tenemos esquinas(máximo valor) y sus pixeles cercanos(valor alto también pero inferior a la esquina). La explicación es que los gradientes del area cercana son similares. Queremos quedarnos solo con las esquinas, es decir los valores mas altos de entre una zona cercana en la que podríamos observar esta confusión. Con esto conseguiremos que los puntos estén bien distribuidos por toda la imagen.
La función que usaremos toma como argumentos una imagen de entrada y un valor que será la distancia entre máximos locales. Esta comprueba para cada pixel de la imagen si los pixeles adyacentes tienen un valor más alto, en cuyo caso ponemos a 0 el pixel en una copia de la imagen. Finalmente devolvemos la imagen con los puntos finalmente seleccionados.

```python
def supresionNoMax(im,dist=3):

    # Pixeles de las imágenes
    filas=im.shape[0]
    columnas=im.shape[1]

    # Realizamos una copia de la imagen
    supre=np.copy(im)

    # Para cada pixel de la imagen, comprobamos si los pixeles adyacentes
    # tienen un valor más alto, en cuyo caso ponemos a 0 el pixel en la copia
    for i in range(0,filas):
        for j in range(0,columnas):
            pixel=im[i][j]
            fil_inf=max(i-int((dist-1)/2),0)
            fil_sup=min(i+int((dist-1)/2)+1,filas)
            col_inf=max(j-int((dist-1)/2),0)
            col_sup=min(j+int((dist-1)/2)+1,columnas)
            for k in range(fil_inf,fil_sup):
                for l in range(col_inf,col_sup):
                    if im[k][l]>pixel:
                        supre[i][j]=0.0

    return supre
```

Para la práctica se desean obtener unos 2000 puntos Harris. Con unas proporciones aproximadas del 70 % (1400 puntos) para el primer nivel, 25 %(500 puntos) para el segundo nivel, y 5 % (100 puntos) para el tercee nivel de la pirámide gaussiana.
Los parámetros con los que podemos jugar para llegar a ese refinamiento son: El tamaño filtro para el cáculo de las derivadas a la hora de calcular los valores propios, el cual podríamos fijar en 3 o 5, pero como se nos dijo que las imágenes de ejemplo son poco ruidosas fijaremos este valor en 3.
El tamaño de la ventana para detectar puntos Harris, que fijaremos al principio en el nivel 1 de nuestra pirámide. Cuanto más grande sea este valor, mayor será la escala a la que cogeremos los puntos Harris por lo que si le damos un valor muy grande a este parámetro podríamos estar calculando puntos Harris con un vecindario muy grande, lo que es difícil de regular ya que más tarde al querer establecer un umbral comun para las escalas necesitaremos uno muy bajo para obtener puntos en los últimos niveles de la pirámide lo que conlleva obtener un gran número de puntos en el primer nivel y no obtenemos la proporción deseada. Tenemos que tener en cuenta que es muy importante obtener un número adecuado de puntos Harris en cada escala y que sean representativos de la misma.
Tamaño de la ventana de supresion de no máximos, cuanto mayor es, más puntos estamos suprimiendo, este hecho puede repercutir en un decremento grande de los puntos totales.
El umbral del criterio para seleccionar o no un punto Harris, se tiene que calcular empíricamente segun vayamos ajustando el resto de parámetros.

Finalmente tras muchas pruebas y distintas combinaciones de estas 3

últimas variables, he llegado a la conclusión de que la combinación que más se aproxima más es:

- Supresión Máximos = 5
- Tamaño ventana escala = 7
- Umbral= 30

He considerado que es importante un tamaño pequeño de ventana para que no se escapen detalles, pero aun probando muchas combinaciones los puntos del primer nivel siempre son mayores a los porcentajes que se piden, por ello he considerado que aunque haya muchos más keypoints del primer nivel, es preferible asegurarse de que haya suficientes de los otros 2 niveles. Finalmente probé por cambiar el tamaño de la máscara de la derivada de 3 a 5 y me dió una mejor proporción de los datos.

1.3. Estructura KeyPoint

A continuación queremos pintar los keypoints en las imágenes sobre las que estamos trabajando, para ello debemos crear una estructura KeyPoint que consta de el punto en cuestión(coordenadas x,y) la escala del keypoint(circunferencia) y por último la orientación del gradiente en su punto central tras un alisamiento de la imagen con un sigma 4'5 que se dibujara como un radio.
Con la ayuda del paper "Multi-Image Matching using Multi-Scale Oriented Patches"he desarrollado la función get_keypoint que aplica el alisamiento a la imagen y calcula para cada pixel mayor de 0 (keypoint que pasaron la supresión de no máximos) el objeto KeyPoint de OpenCV que consta de las coordenadas, escala y ángulo del gradiente:

```python
def get_keypoints(img,block_size,level):
    dx = maskDerivKernels(img,1,0)
    dy = maskDerivKernels(img,0,1)

    dx = gaussian2D(dx,4.5)
    dy = gaussian2D(dy,4.5)
    keypoints = []
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            if(img[i,j] > 0):
                ox = dx[i,j]
                oy = dy[i,j]
                cos,sen = ox/(math.sqrt(ox**2 + oy**2)),oy/(math.sqrt(ox**2 + oy**2))
                ori = math.atan2(sen,cos)*180/math.pi
                keypoints.append(cv.KeyPoint(j*(2**level),
                                             i*(2**level),
                                             _size = block_size*(level+1),
                                             _angle = ori))
    return keypoints
```

1.4. Dibujar KeyPoints

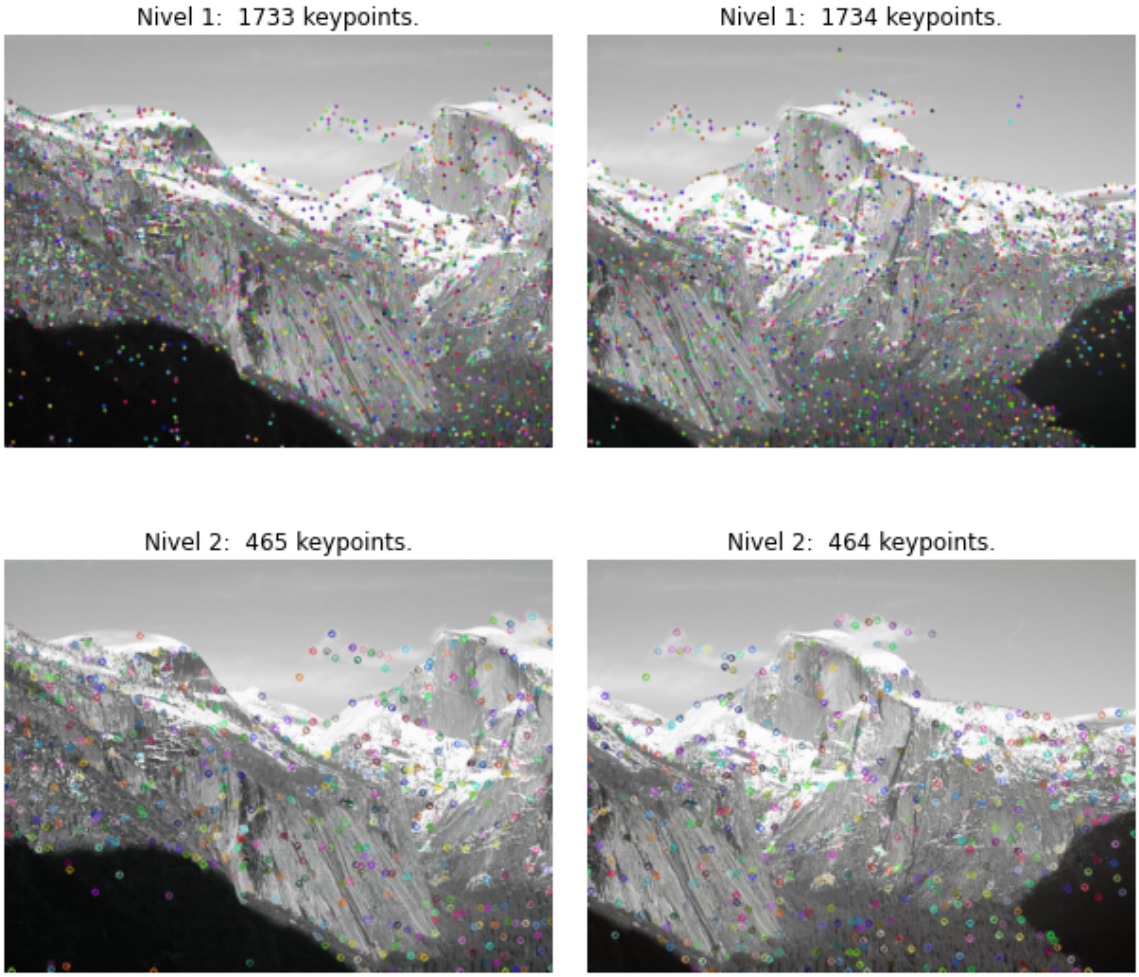
En este apartado mostraremos los resultados de todo lo que hemos hecho hasta ahora.

Resultados de los puntos Harris encontrados en cada octava

–	Yosemir 1	Yosemir 2
Nivel 1	1733	1734
Nivel 2	465	464
Nivel 3	135	130
Total	2333	2328

Los puntos Harris dibujados sobre las imágenes y a diferentes escalas, con su representativa circunferencia y un radio que indica la orientación del gradiente.
Para calcular la escala, es decir la circunferencia que luego se dibuja se proponia la fórmula block_size * 2$\hat{}$(level-1).
Se podría haber decidio aumentar este radio para una mejor visualización cuando se dibujan los puntos pero al final he usado la fórmula propuesta.
Eso solo serviría para visualizar, engañamos un poco con la medida del circulo por mero esteticismo ya que no influye nada porque la circunferencia no muestra realmente la región que se ha usado para detectar el punto.
Para pintar los puntos usaremos la función de OpenCV drawKeyPoints().

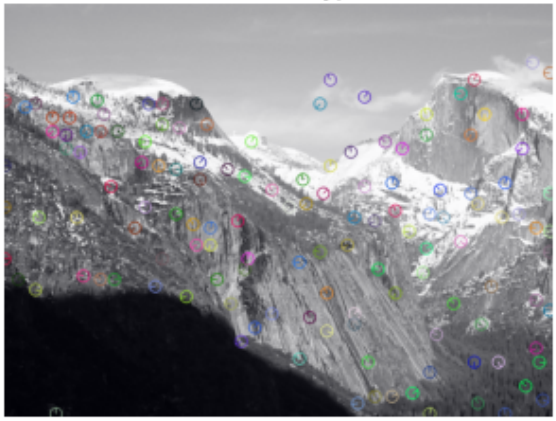
En la columna de imágenes de la izquierda veremos los puntos encontrados en la imagen Yosemite 1 y a la derecha los encontrados en la imagen Yosemite 2.



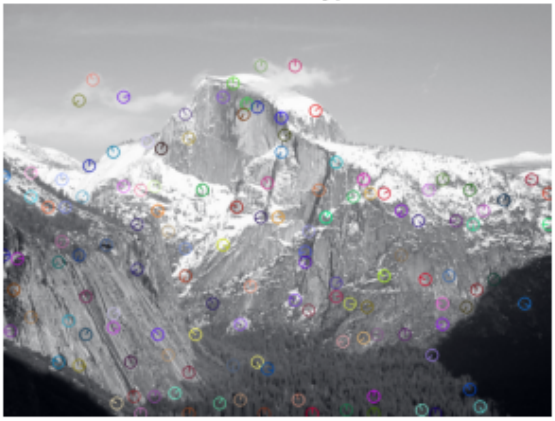
Algunas observaciones sobre los resultados:

- En los puntos de Harris que obtenemos en la imagen original vemos como la mayoría de ellos están sobre la zona montañosa, que es donde observamos cambios de colores y formas más grandes, por lo que los gradientes serán más grandes en los entornos de esos puntos y por tanto tenemos más puntos de Harris.
- Podemos notar una notable reducción en los puntos entre nivele. El alisamiento que se ha producido en la imagen hace que los gradientes en la misma desciendan en los vecindarios. Además, el borrado y filas de columnas implica que la imagen es más pequeña y la supresión de no máximos afecta algo más a los vecindarios.

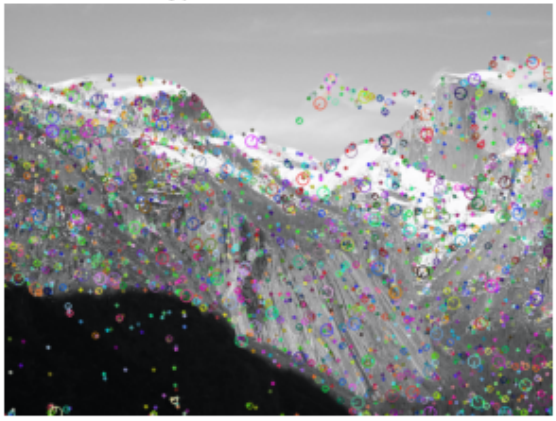
Nivel 3: 135 keypoints.



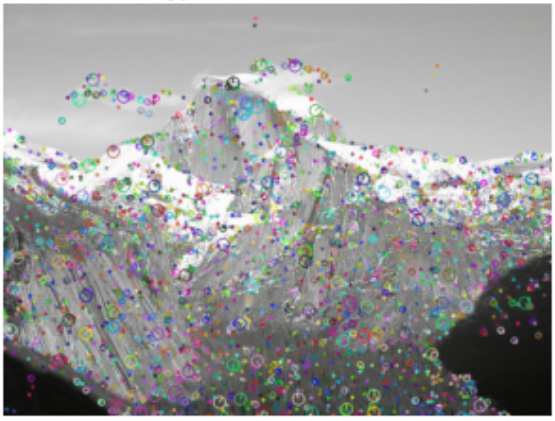
Nivel 3: 130 keypoints.



Keypoints totales: 2333



Keypoints totales: 2328



1.5. Corrección subpixel

Para acabar queremos hacer un refinamiento del punto Harris encontrado. El punto que calculamos puede no ser del todo correcto por varios motivos. Para empezar estamos pasando un valor de un número real a unas cordenadas enteras(pixeles) y por otra parte la elección del tamaño la ventana mediante la cual calculamos el punto Harris puede hacer que la esquina cambie porque cojamos mas o menos gradientes.Por lo que finalmente podríamos tener una variación no solo de subpixeles, sino incluso de varios pixeles en la imagen.
Por lo tanto queremos refinarlo y para ello calcularemos las coordenadas subpixel(x,y) de cada KeyPoint usando la función cornerSubPix de OpenCV.

Hay que resaltar la diferenciación de que hasta ahora habíamos calculado la esquina a nivel de pixel y nosotros queremos hacer una correccion a nivel subpixel,es decir pixeles reales.

Buscaremos un refinamiento en una ventana dada, si no encontramos dentro de la ventana no pasa nada pero si encontramos una mejor esquina dentro de esa ventana lo cogeríamos.

Tras hacer la correción o refinamiento, mostraremos 3 puntos aleatorios de los Keypoints que hemos obtenido en la imagen 1. Y lo mostraremos como imágenes interpoladas con zoom 10x10.
En concreto mostraremos una región 9x9 del entorno de los puntos escogidos.
En estas imágenes señalaremos el punto calculado originalmente y el punto refinado para comparar qué tanto se ha desplazado y si ha mejorado.

Como hemos dicho,este algoritmo se utiliza para refinar las coordenadas de las esquinas con una precisión de subpíxeles y tiene 5 argumentos:

- La imagen de entrada
- Las coordenadas de la esquina que se quiere refinar
- La mitad del tamaño de ventana de búsqueda(winsize), si ponemos por ejemplo una ventana 5x5 realmente estamos búscando en una ventana 11x11
- zeroZone con la que se puede configurar la exclusión de píxeles del cálculo que hace que una parte central de la ventana de búsqueda sea ignorada.

- Criterio de parada, ya que es un algoritmo iterativo, en nuestro caso hemos pensado que 50 iteraciones o 1 céntesima es un criterio razonable de parada.

Vamos a establecer la venta de refinamiento en 11x11 es decir con un máximo de 5 píxeles, en la memoria del trabajo se decía de usar una 5x5 pero comentandolo con el profesor nos dijo que en muchos casos no se realizaría un refinamiento ya que un 5x5 significa solo un margen de 2 píxeles.

Una vez tenemos los nuevos puntos refinados, cogemos la región 7x7 alrededor del píxel orginal que buscabamos refinar y le aplicamos una interpolación lineal con la función de OpenCV resize(), aumentando la región x10 veces.

Finalmente pintamos en esta región el punto original y el refinado para comprobar que funciona nuestro estudio.

Lamentablemente no he podido lograr que mi método funcione, me daba unos errores que no he podido solucionar a tiempo pero la estructura del código es correcta y la mayoría de pasos funcionan, el único error es que estoy pasando los keypoints originales de una forma errónea a la hora de calcular el refinamiento.

Capítulo 2

Emparejamiento keypoints

KAZE y AKAZE son detectores y descriptores de características open-source que se puede usar con libertad en aplicaciones comerciales a diferencia de otros que hemos visto como SIFT que están patentados y no se pueden usar libremente.
KAZE nació como una alternativa y superó a otros métodos ya implementados en OpenCV, sin embargo calculaba características más lentamente.
Asi surgió AKAZE(Aceletaed KAZE) como una mejora de KAZE y que contaba con una aceleración a la hora de obtener las características.
Gracias a estos detectores-descriptores podremos detectar y emparejar keypoints en dos imagenes donde se muestra la misma escena desde dos perspectivas diferentes.

En el anterior punto de este trabajo aprendimos a detectar los keypoints. El siguiente paso para poder llegar al nuestro objetivo de realizar el mosaico es el emparejamiento o matching.
Este tipo de correspondencias o emparejamientos nos permite responder a la pregunta,¿dónde se encuentra el punto X de la primera imagen en la seguna imagen?

Usaremos el método de AKAZE detectAndCompute() que dada una imagen devuelve los keypoints y descriptores. Lo haremos con cada una de las imagenes de Yosemite.
Lo siguiente será establecer las correspondencias existentes entre ambas imágenes. Usaremos el objeto BFMatcher de OpenCV y emparejaremos los puntos de 2 formas diferentes según estos criterios de correspondencia:

BruteForce+crossCheck

Según este criterio, por cada descriptor se encontrará el más cercano en el segundo conjunto de descriptores probando uno por uno. Además, utilizamos crosscheck que aumentaremos la consistencia produciendo mejores resultados.

```
def matchBruteForce(im1,im2):
    # Creamos AKAZE
    akaze = cv.AKAZE_create()
    # Obtenemos los keypoints y los descriptores
    kp1,d1 = akaze.detectAndCompute(im1,None)
    kp2,d2 = akaze.detectAndCompute(im2,None)
    # Creamos objeto BF MAtcher
    bfmatcher = cv.BFMatcher.create(crossCheck = True)
    # Emparejamiento de fuerza bruta
    matches = bfmatcher.match(d1,d2)

    return kp1,kp2,d1,d2,matches
```

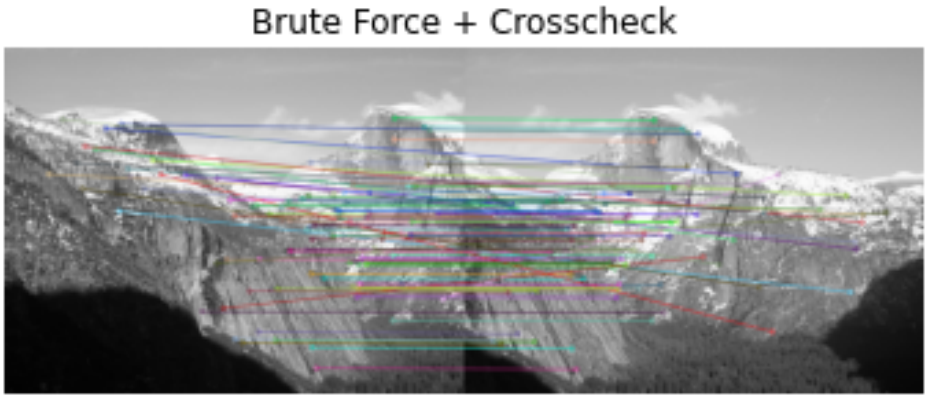
Lowe-Average-2NN

Tambien usa un emparejamiento de fuerza bruta, pero tomará ahora los 2 vecinos más cercanos. Luego, dado un ratio y tomando pares de matches, se quedará con un número de matches válidos. En concreto, dado un par de matches m, n, si la distancia de m es menor que la distancia de n por el ratio, entonces tomará m como válido.

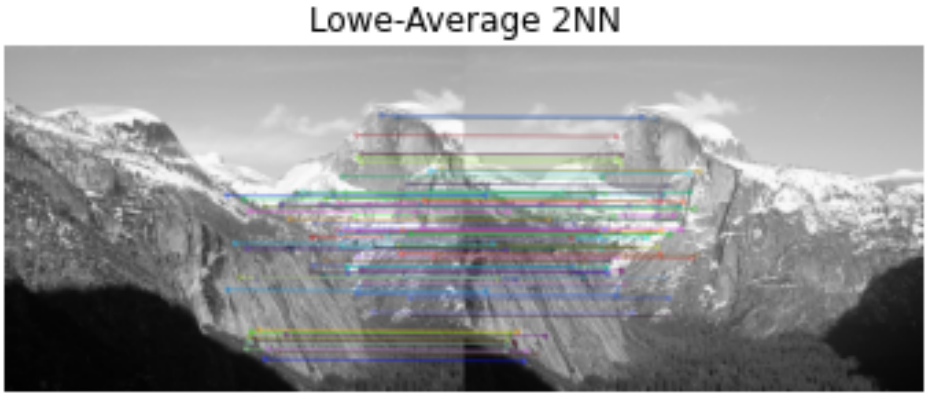
```
def matchLoweAvg2NN(im1,im2,ratio = 0.7):
    # Creamos AKAZE
    akaze = cv.AKAZE_create()
    # Obtenemos los keypoints y los descriptores
    kp1,d1 = akaze.detectAndCompute(im1,None)
    kp2,d2 = akaze.detectAndCompute(im2,None)
    # Creamos objeto BF MAtcher
    bfmatcher = cv.BFMatcher.create()
    # Usamos emparejamiento 2NN
    matches = bfmatcher.knnMatch(d1,d2,k=2)
    # Nos quedamos con los emparejamiento no ambiguos
    valid = []
    for m,n in matches:
        if m.distance < n.distance*ratio:
            valid.append([m])

    return kp1,kp2,d1,d2,valid
```

2.1. Valoración de resultados

Para valorar los emparejamientos que estamos haciendo mostraremos ambas imágenes en un mismo canvas pintando las lineas que unen puntos de sendas imagenes con difetentes colores.
Usaremos una muestra de 100 puntos elegidos aleatoriamente de entre los que hemos encontrado. Mostraremos los resultados para cada uno de los criterios de correspondencia empleados y se valoraran a juicio visual y estudiando las tendencias de las lineas dibujadas.



En la imagen podemos apreciar que, a pesar de que tenemos muchas parejas de puntos que efectivamente representan el mismo punto en la imagen real, existen algunas parejas que no se identifican con los matches reales, si vemos por ejemplo la línea roja que comienza en la parte derecha baja de la segunda imagen y que tiene una trayectoria rectilinea ascendente, nos damos cuenta que no señalan el mismo punto de la escena.
Detectamos por tanto que este algoritmo de Brute Force puede dar errores, debido posiblemente a quedarse directamente con el descriptor más cercano encontrado.
Este ejemplo al tener dos imagenes poco desplazada podemos decir que si encontramos una linea con una inclinacion notable sería un error.



En este caso, podemos ver que todas las líneas van de forma totalmente paralela al eje de abcisas y conectan posiblemente con el punto que representa al mismo en ambas imágenes. Es por ello que podemos afirmar que este criterio nos da mejores resultados en cuanto a emparejamiento de KeyPoints entre las imágenes.

Capítulo 3

Mosaico de 3 imágenes

Es este apartado finalmente uniremos todos los conceptos que hemos ido trabajando hasta ahora para lograr componer 3 imágenes en una sola.
Primero deberemos conocer el término más importante para el desarrollo de este apartado, la homografía.
Una homografía es una funcion o matriz h 3x3 con determinante distinto de cero, que transforma una imagen en otra. Es decir para 2 imágenes que muestran el mismo plano pero desde dos perspectivas diferentes, podremos mediante su homografía describir una relación entre ambas. Técnicamente podemos definirla como un isomorfismo entre dos espacios proyectivos de la misma dimensión

Para cada 2 imagenes tenemos una homografia unívoca.

Todo el peso de nuestro calculo se basará en la correcta obtención de esas homografías, una vez las tengamos, tenemos todo lo necesario para contruir el panorama, no necesitamos nada más.
Como queremos realizar la composición de 3 imágenes necesitaremos 2 homografías: homografía(imagen2-imagen1) y la homografía(imagen2-imagen2).

Implementaremos una técnica que a partir de las homografias construye el panorama. El panorama con el que vamos a trabajar es un panorama lineal, es decir hay una primera y una última imagen en sentido direccional, de derecha a izquierda o de izquierda a derecha. La primera se solapa con la segunda y la segunda se solapa con la tercera etc.. pero solo se solapan con su siguiente y anterior y no con ninguna más.
Nosotros simplemente usaremos una función que nos proporciona OpenCV pare encontrar la homografía,findHomography(), previamente necesitamos calcular la lista de keypoints y sus emparejamientos como hicimos en el

apartado anterior, con la función matchLoweAvg2NN().
Usaremos el parámetro RANSAC, que sirve para que la función aplique el algoritmo de mismo nombre.

```python
def homography(img1,img2):
    # Obtenemos puntos descriptores y emparejamientos
    k1,k2,d1,d2,matches = matchLoweAvg2NN(img1,img2)
    # Ordenamos los puntos de emparejamiento
    orig = np.float32([k1[p[0].queryIdx].pt for p in matches]).reshape(-1,1,2)
    dest = np.float32([k2[p[0].trainIdx].pt for p in matches]).reshape(-1,1,2)
    # Obtenemos la homografía usando RANSAC
    h = cv.findHomography(orig,dest,cv.RANSAC,1)[0]

    return h
```

Una vez las tenemos, la composición se hace también con una función de OpenCV warpPerspective(), la cual recibe el canvas,qué imagen se añade y con que homografía. Para introducir nuevas imágenes necesitamos calcular la expresión de la homografía composición, la cual se obtiene con el producto matricial de la nueva homografía con la actual.
Si las homografias están bien calculadas y son exactas, el mosaico aparece bien alineado y sin problemas geométricos, sin embargo se pueden notar problemas de fotometría, es decir, partes a distinta iluminación.

Un parámetro importante de la función warpPerspective() es el Border_trasnparnt porque si no lo ponemos cada vez que la llamamos pone a 0 el canvas y no se va realizando la composición.

```python
def mosaic(img1,img2,img3):
    canvas = getCanvas([img1,img2,img3])
    h = homography(img2,img1)
    id = identity_h(img1,canvas)
    # Introduce img1 en el canvas
    canvas = cv.warpPerspective(img1,id,(canvas.shape[1],canvas.shape[0]),
                                dst = canvas,borderMode = cv.BORDER_TRANSPARENT)
    comp = np.dot(id,h)
    # Introduce img2 en el canvas
    canvas = cv.warpPerspective(img2,comp,(canvas.shape[1],canvas.shape[0]),
                                dst = canvas,borderMode = cv.BORDER_TRANSPARENT)

    h2 = homography(img3,img2)
    comp = np.dot(comp,h2)
    # Introduce img3 en el canvas
    canvas = cv.warpPerspective(img3,comp,(canvas.shape[1],canvas.shape[0]),
                                dst = canvas,borderMode = cv.BORDER_TRANSPARENT)
    return canvas
```

Un punto importante es fijar la primera imagen en nuestro canvas que inicialmente no es mas que una gran imagen en negro suficientemente grande, pero como en este apartado solo trabajaremos con 3 imágenes no es tan relevante y empezaremos copiando la imagen 1 en el canvas, que será nuestra homografía identidad, y a partir de esta, añadimos la seguna y tercera imagen en este orden. Este es un problema que si abordaremos en el siguiente

apartado cuando manejemos un número de imágenes mayor.
Por último eliminamos las partes negras sobrantes de nuestro canvas para poder mostrar los resultados adecuadamente.

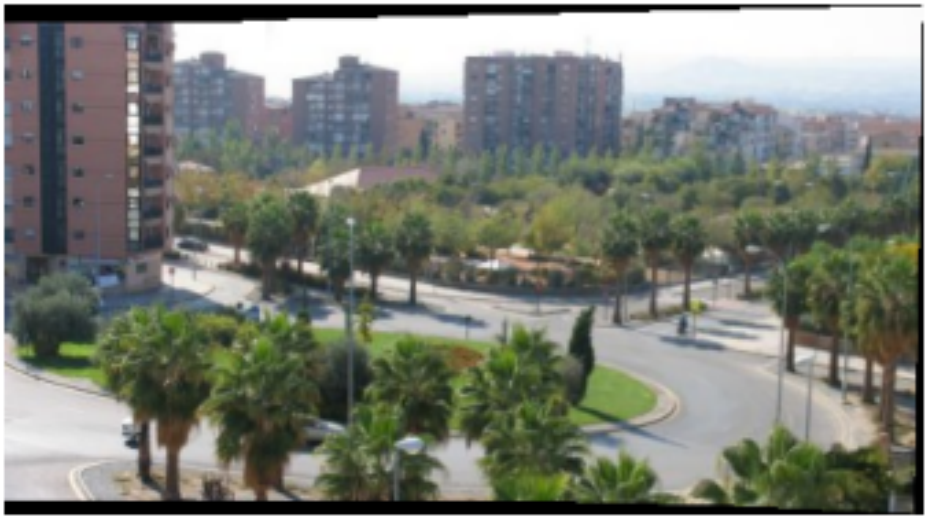
```
def remove_extra(img):
    indexR =[]
    indexC = []
    for i in range(img.shape[0]):
        if np.count_nonzero(img[i]) == 0:
            indexR.append(i)
    for i in range(img.shape[1]):
        if np.count_nonzero(img[:,i]) == 0:
            indexC.append(i)
    img = np.delete(img,indexR,axis = 0)
    img = np.delete(img,indexC,axis = 1)
    return img
```

Para este ejercicio cambiaremos de conjunto de imágenes al conjunto mosaico.rar. A continuación se muestran las 3 imágenes con las que vamos a trabajar y la composición final de todas ellas, se han realizado en escala de grises y a color para poder apreciar mejor los resultados obtenidos. El resultado es bastante bueno y a simple vista nos da la sensación de ser una única imagen por la continuidad que muestra.

Imágenes de la composición

Composición escala de grises de 3 imágenes

Composición a color de 3 imágenes



Capítulo 4

Mosaico 10 imágenes

En esencia realizaremos la misma tarea que el apartado anterior pero con un número mayor de imágenes, en vez de 3 lo haremos con 10 imágenes del conjunto mosaico.rar.
Como ya sabemos estamos trabajando con un panorama lineal por lo que las imágenes tienen un orden de izquieda a derecha.
Crearemos una función para generalizar el problema para n imágenes, seguirá las bases ya sentadas en el apartado anterior.
El único y muy importante punto diferentes es la forma en la que iremos componiendo las imágenes. Al tener como hemos dicho un panorama lineal, podríamos empezar añadiendo al canvas la primera imagen de la izquierda y luego ir añadiendo una a una por la derecha.
Pero de esta forma estaríamos calculando la homografía de la nueva imagen a añadir con la imagen actual total, la cual se va acumulando conforme añadimos más imágenes.
Cuando realicemos la composición con las últimas imágenes nos encontramos que se han ido acumulando las deformaciones y el resultado de añadir estas últimas imagenes puede ser desastroso.
Por ello la solución que hemos tomado para que la calidad de nuestro mosaico sea buena ha sido fijar la primera imagen del canvas como la central de entre todas las imágenes con las que trabajaremos. El proceso de composición se divide ahora en 2 fases.
Primero añadimos las imágenes que están a la izquierda de la imágen central y en segundo lugar iremos añadiendo las imágenes que están a la derecha.
De esta forma se disminuye el error que se propaga cuando las homografías se aplican sobre las imágenes y las unen.

Resultados

Finalmente obtenemos un resultado bastante satisfactorio donde veremos que todas las imágenes se han integrado correctamente. Como seguimos trabajando con el conjunto de imagenes mosaico.rar si comparamos con el apartado anterior vemos que es la misma escena pero con un campo de visión mucho más amplio.
Para finalizar veremos las imágenes de las que partíamos y el mosaico final en escala de grises y a color para poder apreciar mejor la homogeneidad de la imagen final.

Imágenes del problema

Conclusión

Así concluye está práctica, hemos recorrido un camino paso a paso hasta conseguir nuestro mosaico. Hemos aprendido como calcular los keypoints, como emparejarlos con el mejor criterio y finalmente como unirlos haciendo uso de nuestros conocimentos sobre homografías. Los resultados han sido bastante buenos con una calidad nada despreciable.

Bibliografía

[1] *Documentation opencv cornereigenvalsandvecs()*. https://docs.opencv.org/master/dd/d1a/group__imgproc__feature.html.

[2] *Image matching*. https://ai.stanford.edu/~syyeung/cvweb/tutorial2.html.

[3] *Interest point detector and feature descriptor survey*. https://core.ac.uk/download/pdf/81870989.pdf.