

Visión por computador. Práctica 2.

José Carlos Martínez Velázquez

15 de Noviembre de 2016

Contents

Detección de puntos Harris multiescala. (Opción 10 puntos)	2
Uso de los detectores KAZE/AKAZE. (Opción 10 puntos)	17
Construcción de mosaicos o panoramas (Opción 10 puntos)	21
Cuestiones técnicas	30
Referencias	31

Detección de puntos Harris multiescala. (Opción 10 puntos)

1.- Detección de puntos Harris multiescala (MOPS, Brown, Szeliski, Winder, 2004). En este apartado se trata de implementar el detector de regiones descrito en el artículo mencionado. En este algoritmo se suponen que todas las imágenes están tomadas en una escala semejante (zoom y distancia iguales) por lo que el objetivo es extraer todas las regiones relevantes que haya a distintas escalas enteras de las imágenes. Por cada región detectada necesitaremos guardar la siguiente información: (las coordenadas x ,y de su centro, su orientación, su escala)). Usar para ello la estructura que resulte más fácil. Presentar los resultados con las imágenes Yosemite.rar.

1.a) Escribir una función que extraiga la lista potencial de puntos Harris en una imagen a distintas escalas. Para ello construiremos una pirámide Gaussiana de la imagen con 3 escalas usando sigma=1. Sobre cada nivel de la pirámide usar la función OpenCV cornerEigenValsAndVecs para extraer los mapas de auto-valores de la matriz Harris en cada píxel (fijar los valores de blockSize en el rango [3-13] y ksize en el rango [3-9]), usar la versión de nivel de gris de las imágenes). Crear una matriz con el valor del criterio selección Harris asociado a cada píxel usando ($k=0.04$). Implementar la fase de supresión de valores non-máximos sobre dicha matriz (ver descripción al final). Ordenar de mayor-a-menor(ver sortIdx()) en cada escala los puntos resultantes de acuerdo a su valor Harris. Seleccionar al menos 1500 puntos de entre los de mayor valor distribuidos entre las distintas escalas (las escalas más bajas tendrán más puntos: 70-20-10). Mostrar el resultado dibujando un círculo sobre la imagen original de radio proporcional a la escala y centro las coordenadas de los puntos (ver circle()).

Vamos a hacer este apartado por partes. En primer lugar, vamos a realizar la fase de detección de puntos Harris. Nuestro objetivo es usar los valores Harris para detectar esquinas. Pero ¿Cómo detectamos una esquina en una imagen? Un valor alto del criterio de Harris en un entorno, nos dice que es muy probable que en dicho entorno en la imagen haya una esquina.

El detector de esquinas de Harris consiste en pasear una ventana de un determinado tamaño (`blocksize`) por una imagen, buscando cambios en la dirección de x y en la dirección de y . Para comparar los cambios de una posición de la ventana en la imagen usamos error cuadrático (SSD). Entonces, para cada píxel, tenemos a la expresión:

$$E(u, v) = \sum_{(x,y) \in W} [I(x + u, y + v) - I(x, y)]^2$$

Donde u, v representan el movimiento realizado por la ventana desde la posición anterior. Mediante desarrollo matemático [1] llegamos a la expresión:

$$E(u, v) \approx \sum_{(x,y) \in W} I_x^2 u^2 + 2 \sum_{(x,y) \in W} I_x I_y u v + \sum_{(x,y) \in W} I_y^2 v^2$$

Que puede ser expresada como:

$$E(u, v) = \begin{pmatrix} u & v \end{pmatrix} \underbrace{\begin{pmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{pmatrix}}_H \begin{pmatrix} u \\ v \end{pmatrix}$$

Donde I_x^2 es la derivada de la imagen con respecto a x al cuadrado, I_y^2 la derivada con respecto a y al cuadrado e $I_x I_y$, el producto de ambas derivadas. La matriz H es llamada Matriz de Harris o matriz de covarianzas de

las derivadas y es justo lo que calcula la función `cornerEigenValsAndVecs` de OpenCV. Para cada píxel, obtendremos una sextupla $(\lambda_1, \lambda_2, x_1, y_1, x_2, y_2)$, donde:

- λ_1, λ_2 son los valores propios de la matriz H.
- x_1, y_1 son los vectores propios de λ_1
- x_2, y_2 son los vectores propios de λ_2

Sean ahora Λ_1 y Λ_2 las matrices formadas por todos los valores λ_1 y λ_2 respectivamente, obtenemos una nueva matriz I_H del mismo tamaño que la matriz imagen de tal forma que dado un píxel $I(i, j)$ en la imagen original, el valor Harris asociado dicho píxel es $I_H(i, j)$. El criterio Harris o la forma de calcular la matriz I_H es:

$$I_H = \Lambda_1 \Lambda_2 - k(\Lambda_1 + \Lambda_2)^2$$

Donde $k \in [0.04, 0.06]$

El código asociado a esta parte en Python sería el siguiente:

```
"""
Detector de esquinas de Harris. Dada una imagen, se calculan sus eigenvalores
a partir de su imagen en escala de grises. Para ello necesitamos definir
el tamaño del vecindario o entorno y la apertura del operador Sobel k (tamaño
máscara derivadas). El valor harris sera calculado para cada pixel i por
(lambda_i_1)*(lambda_i_2)-alpha*((lambda_i_1)+(lambda_i_2)) ^2.
"""

def harrisCornerDetector(image,blocksize,k_size,alpha=0.04,borderType=0):
    r = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    if(borderType==1):
        eigenvvr=cv2.cornerEigenValsAndVecs(r,
                                              blockSize=blocksize,
                                              ksize=k_size,
                                              borderType=cv2.BORDER_REFLECT)
    elif(borderType==2):
        eigenvvr=cv2.cornerEigenValsAndVecs(r,
                                              blockSize=blocksize,
                                              ksize=k_size,
                                              borderType=cv2.BORDER_REPLICATE)
    else:
        eigenvvr=cv2.cornerEigenValsAndVecs(r,
                                              blockSize=blocksize,
                                              ksize=k_size)

    l1,l2,x1,y1,x2,y2=cv2.split(eigenvvr)
    rnew=(l1*l2)-alpha*((l1+l2)*(l1+l2))
    image_corner=cv2.merge([rnew,rnew,rnew])
    return image_corner
```

Podemos comparar esta función con lo que nos devuelve la misma función en OpenCV. En este ejemplo, `blocksize=5` y `ksize=7` (apertura del operador Sobel para el cálculo de las derivadas).

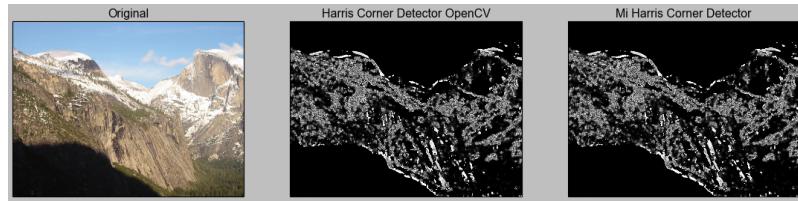


Figura 1. Comparación del detector de esquinas de Harris de OpenCV con el implementado.

Como se ve en la figura, apenas hay diferencias y vamos bien encaminados.

Ahora que ya tenemos un algoritmo para obtener el valor del criterio Harris en cada píxel, vamos a implementar la fase de supresión de no máximos. Necesitaremos una nueva matriz NMS , de valores a 0 (negra), una ventana de un determinado tamaño, muy importante que sea impar, que nos marcará la separación mínima entre los máximos (`windowSize`), un umbral de calidad (`threshold`) y la matriz de valores con el criterio Harris. El proceso es simple: pasearemos la ventana por la matriz de valores Harris. Dado que el tamaño de la ventana es impar, siempre tendremos un elemento central. Si dicho elemento es máximo local en la ventana, todos los valores correspondientes en la matriz NMS serán 0, excepto el central, que tomará el valor 255. Si dicho valor no es máximo local, sólo él toma el valor 0 en la matriz NMS .

Cabe destacar que si inicializamos la matriz NMS a 255, es decir, a blanco, todos los valores que estén a los bordes se detectarán como máximos a no ser que un punto muy cercano sea detectado como máximo local, tal y como muestra la siguiente imagen, para una ventana de 21 píxeles y un umbral de 0.05:

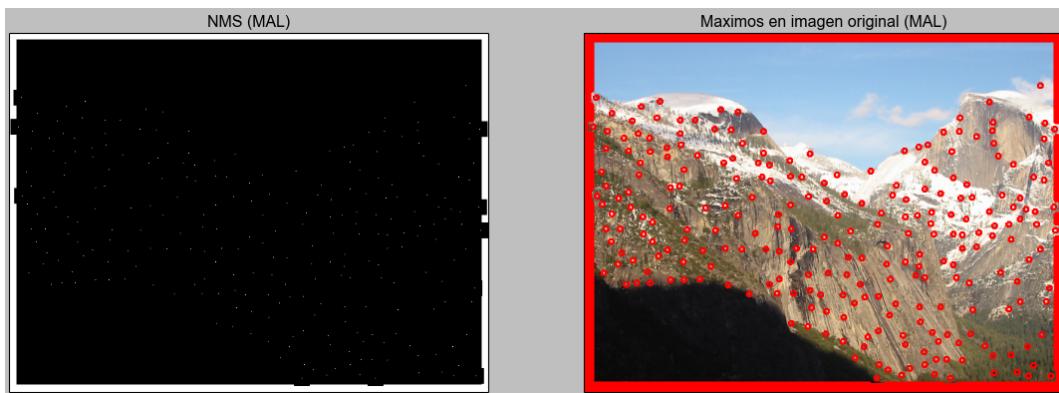


Figura 2. Supresión de no máximos **mal** implementada, Matriz NMS inicializada a 255.

Para corregir este error, basta con inicializar como decíamos, la matriz NMS al valor 0, así aunque podríamos perder algún máximo si la ventana es muy grande, nos quitaremos un montón de valores basura, lo que es preferible. El resultado es el siguiente, para los mismos valores de tamaño de ventana y umbral anteriores:

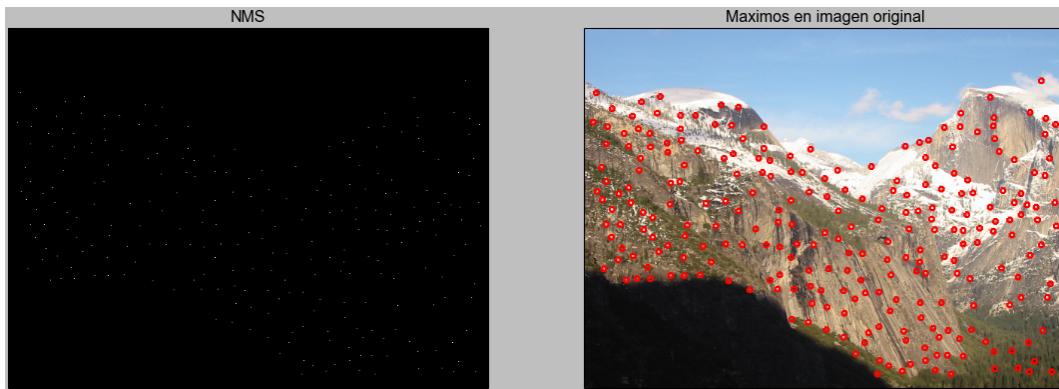


Figura 3. Supresión de no máximos **bien** implementada, Matriz NMS inicializada a 0.

Aunque aquí no se puede apreciar bien que se detectan esquinas, vamos a ver un ejemplo donde las esquinas son mucho más evidentes:

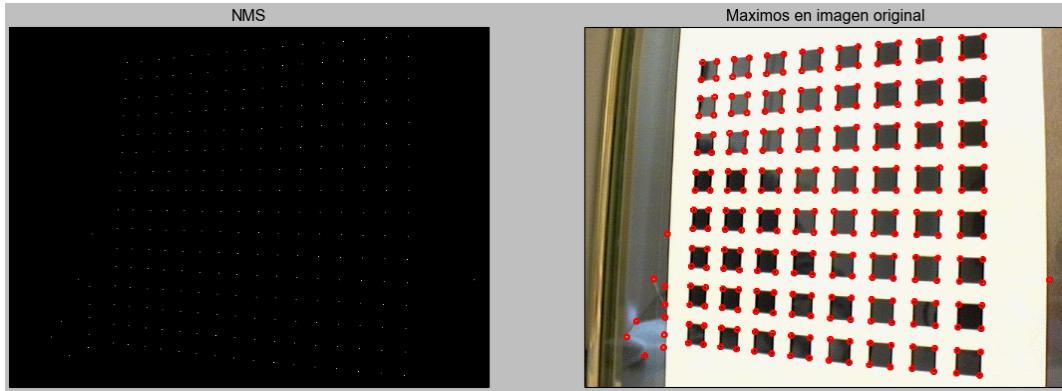


Figura 4. Detección evidente de esquinas.

El código de la fase de supresión de no máximos **correcto** es el siguiente:

```
"""
Fase de supresion de no maximos. Dado el tamaño de la ventana se calcula una posicion de 7
comienzo asi como una posicion de final para fila y columna. A partir de aqui trabajamos
con una imagen en escala de grises representante de la imagen real y una matriz del mismo
tamaño que la imagen toda a negro. Se recorre en orden de izquierda a derecha y de arriba
a abajo de forma que si el pixel del centro de la ventana es maximo, se queda todo negro
menos el pixel central, si no, el pixel central queda negro.
"""

def nonMaximumSuppression(image,windowSize,thresold):
    startPosition=int(windowSize/2)
    finishpositionrow=image.shape[0]-startPosition
    finishpositioncol=image.shape[1]-startPosition
    r = cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)

    newr=np.zeros( (r.shape[0],r.shape[1]) )
    for i in xrange(startPosition,finishpositionrow):
        for j in xrange(startPosition,finishpositioncol):
            window=r[i-startPosition:(i+1)+startPosition,
                      j-startPosition:(j+1)+startPosition]
            if(isLocalMax(window) and np.amax(window)>thresold):
                modifyZeros(newr,i,j,windowSize)
                newr[i][j]=255

    #print("Puntos detectados "+str(np.count_nonzero(newr)))
    newr=np.uint8(newr)
    return cv2.merge([newr,newr,newr])
```

Donde llamamos a las funciones auxiliares `isLocalMax` que nos permite saber si el elemento central de una matriz es máximo local y la función `modifyZeros` que nos permite poner una región de una matriz a 0, indicando el centro y el tamaño de la ventana:

```
"""
La funcion isLocalMax permite saber si el valor del centro de una matriz es
maximo local. El numero de filas y columnas debe ser siempre impar, de lo
```

```

contrario, siempre devuelve False, pues no existe un elemento central.
"""

def isLocalMax(neighborhood):
    nrow=len(neighborhood)
    ncol=len(neighborhood[0])
    if (nrow%2!=0 and ncol%2!=0):
        max_v=neighborhood[nrow/2][ncol/2]
        return (np.amax(neighborhood)==max_v)
    else:
        return false

"""

La función modifyZeros pone a 0 una región cuadrada de una matriz. Indicando
el pixel central, calcula los pixeles que debe poner a 0 sumando y restando
la mitad en el eje X y el eje Y.
"""

def modifyZeros(matrix,row,col,windowsize):
    matrix[row-windowsize/2:(row+1)+windowsize/2,col-windowsize/2:(col+1)+windowsize/2]=0

```

Ya tenemos la forma de calcular los valores de criterio Harris y la supresión de no máximos. El siguiente paso es calcular los puntos en cada escala, ordenarlos en base a su valor de criterio Harris y devolverlos. El procedimiento es simple: para cada escala de la imagen, aplicar el cálculo de valores de criterio Harris y posteriormente la fase de supresión de no máximos. Obtendríamos algo así:

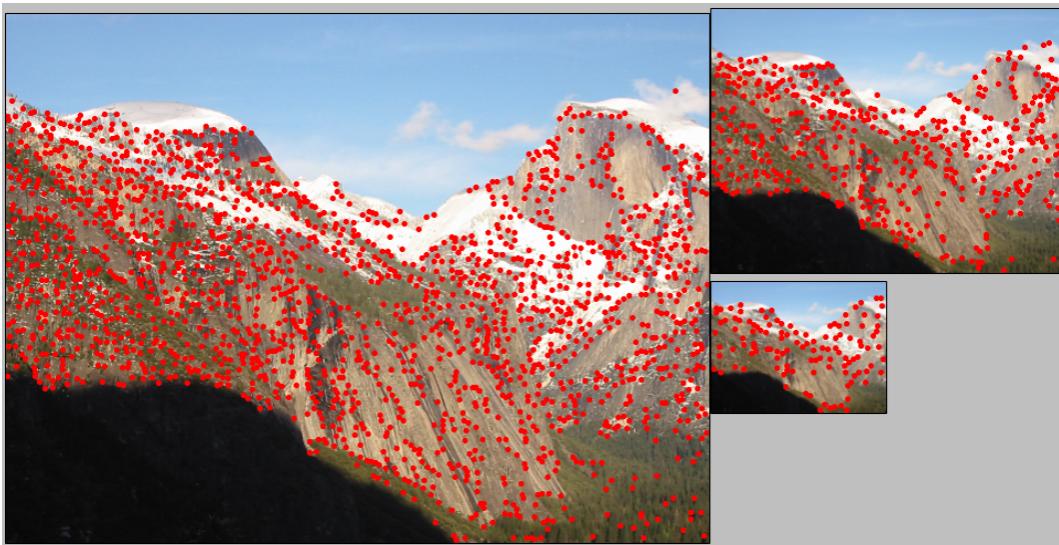


Figura 5. Obtención de puntos en cada escala.

Para ordenar los puntos tenemos dos opciones: juntarlos todos y devolver los más fuertes independientemente de la escala a la que pertenezcan (orden absoluto) o bien ponderar los puntos que queremos de cada imagen. En este segundo caso, podríamos decir que queremos un 70% de puntos de la imagen de mayor tamaño, un 20% de la segunda y un 30% de la tercera. A efectos prácticos y con muchos puntos, usar estos porcentajes es similar (que no igual) a usar orden absoluto, pero podrían querer usarse otras proporciones. La función que implemento, `getStrongestPoints` permite las dos opciones, con el parámetro `weights`, que si es vacío estamos pidiendo un orden absoluto. Cada punto será devuelto en una tupla `(x,y,escala)`. Su código es el siguiente:

```

def getStrongestPoints(pir,blockSize,kSize,suppressionsize,
                      thresold,maxPoints,k=0.04,bordertype=0,
                      weights=[]):
    harrispir=[]
    nmspir=[]
    npointsPir=[]
    nScales=len(pir)

    if(len(weights)==0):
        for i in xrange(nScales):
            harrispir=[harrisCornerDetector(pir[i],
                                              blockSize,
                                              kSize,
                                              k,
                                              bordertype) for i in xrange(nScales)]
            nmspir=[nonMaximumSuppression(harrispir[i],
                                           suppressionsize,
                                           thresold) for i in xrange(nScales)]
            npointsPir.extend(sortPoints(harrispir[i],
                                         nmspir[i],
                                         maxPoints,i))

    elif (len(weights)==len(pir) and sum(weights)<1):
        for i in xrange(nScales):
            harrispir=[harrisCornerDetector(pir[i],
                                              blockSize,
                                              kSize,
                                              k,
                                              bordertype) for i in xrange(nScales)]
            nmspir=[nonMaximumSuppression(harrispir[i],
                                           suppressionsize,
                                           thresold) for i in xrange(nScales)]
            npointsPir.extend(sortPoints(harrispir[i],
                                         nmspir[i],
                                         int(maxPoints*weights[i]),i))
    else:
        raise UnboundLocalError, "Length of weights must be equal
                                to lenght of pyramid and must amount 1."
    npointsPir=sorted(npointsPir, key=lambda x: -x[3])
    return [(x,y,escala) for (x,y,escala,valor) in npointsPir[0:maxPoints]]

```

Un ejemplo de llamada a esta función para obtener un orden absoluto sería:

```
getStrongestPoints(piramide,blockSize,kSize,supressionSize,thresold,numPuntos)
```

Mientras que para un orden ponderado, haríamos:

```
getStrongestPoints(piramide,blockSize,kSize,supressionSize,thresold,numPuntos,weights=[0.7,0.2,0.1])
```

Como vemos, internamente la función `getStrongestPoints` llama a una función auxiliar `sortPoints` que ordena los puntos con la función `sortIdx` que provee OpenCV. Dado que `sortIdx` ordena los puntos de una escala, si elegimos un tipo de orden y otro habrá que reordenar, pero no será una operación costosa pues `sortIdx` proporciona un orden parcial. Para que un punto sea tenido en cuenta, debe haber sido seleccionado como máximo en la fase de supresión de no máximos, por ello es necesario pasar la matriz de valores harris y la matriz de valores máximos. El código de `sortPoints` es el siguiente:

```

"""
Dada una matriz de valores harris y una matriz con supresion de no maximos,
sortPoints devuelve una lista de puntos detectados como maximos ordenados en
funcion de su valor harris. Para ello uso la funcion sortIdx que provee
OpenCV.
"""
def sortPoints(imageharris,imagenms,nPoints,scale):
    nfilas=imageharris.shape[0]
    ncol=imageharris.shape[1]
    r = cv2.cvtColor(imageharris,cv2.COLOR_RGB2GRAY)
    rn = cv2.cvtColor(imagenms,cv2.COLOR_RGB2GRAY)
    lista=r.reshape((1,r.shape[0]*r.shape[1]))[0]
    ordered=cv2.sortIdx(lista,cv2.SORT_EVERY_COLUMN+cv2.SORT_DESCENDING)
    points=[]
    for posicion in xrange(ordered.shape[0]):
        fila=ordered[posicion][0]/ncol
        columna=ordered[posicion][0]-(fila*ncol)
        if(rn[fila][columna]==255):
            points.append((fila,columna,scale,r[fila][columna]))
    return points[0:nPoints+1]

```

Necesitamos también una función que pinte los puntos recibidos en las imágenes, esta función `paintPointsInImage`, dibujará círculos en los puntos con un radio inversamente proporcional a su escala, esto es, los puntos más pequeños pertenecen a la imagen más grande. El código de la función para dibujar en una imagen los puntos es el siguiente:

```

"""
La funcion paintPointsInImage pinta en una imagen una lista de
puntos, con diametro inversamente proporcional a su tamaño.
"""
def paintPointsInImage(image,points,r=255,g=0,b=0,thickness=2):
    newimg=np.copy(image)
    for point in points:
        cv2.circle(newimg,
                   (int(point[1])*2**int(point[2]),
                    int(point[0])*2**int(point[2]) ),
                   5+2**(2*(int(point[2]))),
                   (r,g,b),
                   thickness)
    #print("Se estan pintando "+str(len(points))+" puntos.")
    return newimg

```

Para comparar los dos criterios de selección de puntos, fijémonos en la siguiente figura, donde se han ordenado 100 puntos:

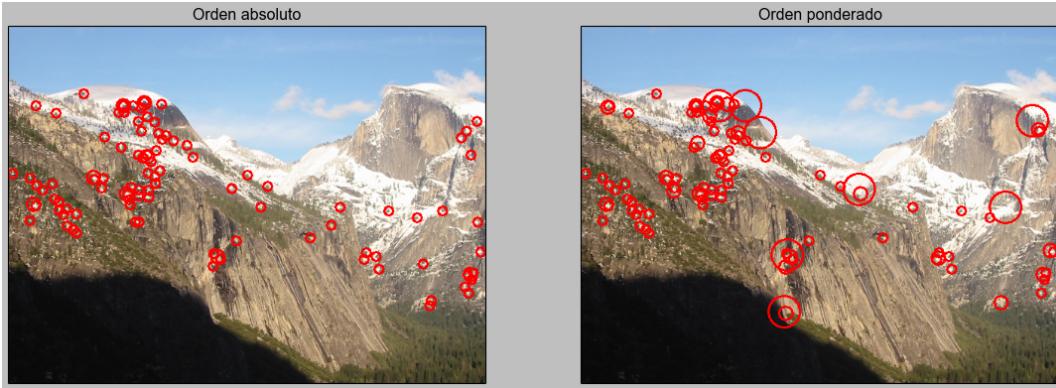


Figura 6. Resultados de la selección absoluta de puntos y la ponderada.

En el orden absoluto, ningún punto de la escala más pequeña fue seleccionado, mientras que con el orden ponderado, se han tenido que elegir 10 de la escala pequeña (algunos se solapan pues estarán muy próximos), 20 de la mediana y 70 de la grande. Recordemos que se están pintando con un radio **inversamente proporcional** a la imagen de la que proceden.

Ya tenemos todo lo necesario. Vamos a elegir 1500 puntos de entre los detectados. Los parámetros para el ejemplo serán:

- Tamaño de bloque/vecindario (blocksize) = 5
- Apertura del operador Sobel (ksize) = 7
- Tamaño de la máscara de supresión = 8
- Umbral de calidad = 0.05
- Número de puntos = 1500
- Pesos de las ponderaciones = 70% de puntos de la imagen grande, 20% de la mediana, 10% de la pequeña

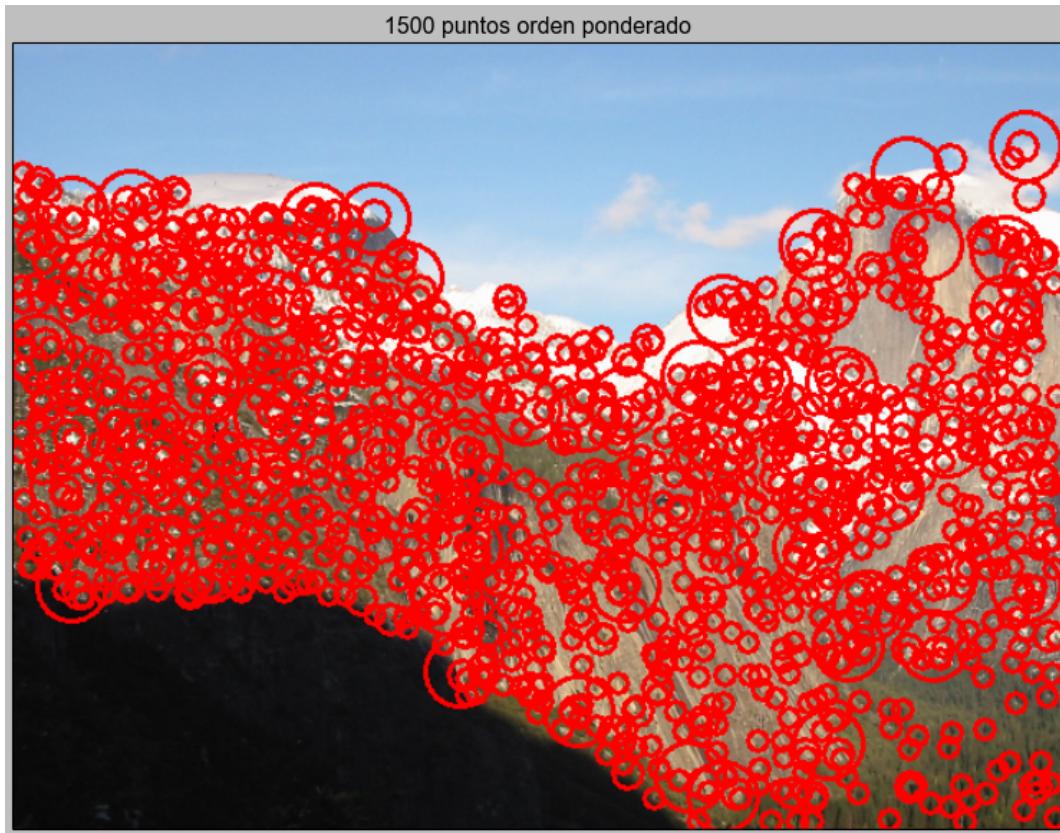


Figura 7. Resultados de la selección ponderada de 1500 puntos.

El umbral de calidad juega un papel importante pues vemos cómo gracias a ello, se reduce enormemente la probabilidad de que se detecten puntos en zonas en las que realmente no hay esquinas, tales como el cielo o la sombra, donde no parece haber nada relevante.

1.b) Extraer los valores (x,y, escala) de los puntos resultantes en el apartado anterior en un vector y refinar su posición espacial a nivel sub-píxel usando la función OpenCV usando `cornerSubPix()` con la imagen del nivel de pirámide correspondiente. Actualizar los datos (x,y ,escala) de cada uno de los puntos encontrados.

Como vimos en el apartado anterior, la función `getStrongestPoints` ya nos devolvía todos los puntos en este formato. Lo que necesitamos es una función que nos refine los puntos. Usaremos la función `cornerSubPix` que proporciona OpenCV en el proceso. El código para refinar los puntos encontrados es:

```
"""
La funcion refinePoints ajusta las coordenadas de los puntos fuertes encontrados. Deadpoint es una tupla, normalmente (-1,1). Hay que refinar cada punto en la escala a la que pertenece.
"""

def refinePoints(pir,criteria,previouspoints,maskSize,zerozone):
    pir_gray=[]
    coords_scale=[]
    result_scale=[]
    nImages=len(pir)

    for scale in xrange(nImages):
        pir_gray.append(cv2.cvtColor(pir[scale]
                                     ,cv2.COLOR_RGB2GRAY))
        coords_scale.append([])
        for point in previouspoints:
            if(point[2]==scale):
                coords_scale[scale].append(np.array(
                    np.float32(
                        [point[0],point[1]])))
    result_scale=[(cv2.cornerSubPix(
        pir_gray[i],
        np.array(coords_scale[i]),
        (int(maskSize/2),int(maskSize/2)),
        zerozone,
        criteria )) for i in xrange(len(pir))]

    newpoints=[]
    for i in xrange(len(result_scale)):
        for j in xrange(len(result_scale[i])):
            newpoints.append( (result_scale[i][j][0],
                              result_scale[i][j][1],i) )

    return newpoints
```

La función `refinePoints` recibe la pirámide de la que provienen los puntos, un criterio de parada (numero máximo de iteraciones y umbral de diferencia que se considera relevante), los puntos que queremos refinar y el tamaño del vecindario donde buscar una mejora. Este último no debe ser muy grande, pues si el refinamiento de un punto i estuviera muy lejos del mismo, nuestro trabajo previo no hubiera servido de nada. El refinamiento de un punto es importante hacerlo en la imagen (escala) de la que proviene, pues de no ser así, el refinamiento podría resultar en cualquier parte, pues podría no coincidir con una esquina.

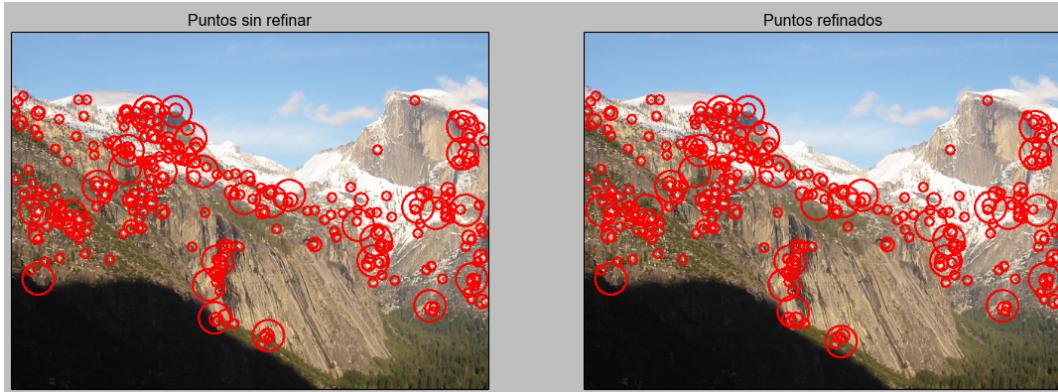


Figura 8. Resultado del refinamiento de puntos.

El refinamiento de la figura anterior ha sido realizado con un tamaño de vecindario 7, un criterio de parada basado en 30 iteraciones máximo con un umbral $\varepsilon = 0.001$. Como se puede apreciar, no habría grandes diferencias con lo obtenido sin refinamiento. Vemos cómo por ejemplo, en la zona inferior, alrededor de la sombra sí que hay algunos cambios que podrían ser significativos en etapas de cálculo posteriores.

1.c) Calcular la orientación relevante de cada punto Harris, usando un alisamiento fuerte de las derivadas en x e y de las imágenes, en la escala correspondiente, como propone el paper MOPS de Brown&Szeliski&Winder. (Apartado 2.5) y añadir la información del ángulo al vector de información del punto. Pintar sobre la imagen de círculos anterior un radio en cada círculo indicando la orientación estimada en ese punto.

Aquí tenemos tres tareas principales: Calcular la orientación, añadirla a los puntos en formato $(x, y, escala, orientación)$ y representarla gráficamente sobre la imagen. Comencemos por la primera. El paper MOPS [2] nos dice que el vector de orientación en cada píxel proviene de la expresión:

$$u_l(x, y) = \nabla_{\sigma_0} P_l(x, y)$$

Donde $\nabla_{\sigma_0} P_l(x, y)$ representa la gradiente (derivada) suavizada con $\sigma_0 = 4.5$ de cada píxel. Esta expresión nos lleva al vector de orientación $[cos\theta \ sen\theta] = u/|u|$. No debemos olvidar que lo que intentamos hallar, la orientación, es un ángulo, concretamente θ . De esta expresión, podemos saber que:

$$\begin{aligned} u_{l_x} &= \nabla_{\sigma_0} \frac{\partial I}{\partial x} = cos\theta \\ u_{l_y} &= \nabla_{\sigma_0} \frac{\partial I}{\partial y} = sen\theta \end{aligned}$$

Es decir, la primera componente del vector u_l es la derivada con respecto a x de la imagen, suavizada con $\sigma_0 = 4.5$, que se corresponde con el coseno del ángulo que buscamos. De igual modo, la segunda componente del vector u_l es la derivada con respecto a y identicamente suavizada, que se corresponde con el seno del ángulo que buscamos.

Por las propiedades de los ángulos, podemos conor un ángulo conociendo su seno y su coseno mediante arcotangente. Entonces, hallar la orientación (ángulo) en un píxel consiste en aplicar la siguiente operación:

$$\theta = tan^{-1}(tan(\theta)) = tan^{-1}\left(\frac{sen\theta}{cos\theta}\right) = arctan\left(\frac{sen\theta}{cos\theta}\right) = arctan\left(\frac{\frac{\partial I}{\partial y}}{\frac{\partial I}{\partial x}}\right)$$

Sabemos calcular la gradiente de una imagen con respecto a x e y (con Sobel) y suavizarla convenientemente con $\sigma_0 = 4.5$. Esto nos proporcionaría dos matrices que contendrían el coseno y seno del ángulo de cada píxel respectivamente. Ahora debemos preguntarnos qué pasaría si algún valor de la derivada con respecto a x contuviese 0. Esto sería una situación de error, pues estaríamos dividiendo por cero. Entonces, tal y como explican en [3], los valores a 0 de la derivada en x (cosenos), deberían convertirse en 1, donde quedaría la ecuación $sen\theta = \theta \Rightarrow \theta = 0$. Con esta expresión aplicada a cada píxel, podemos obtener una nueva matriz O, donde O(i,j) nos indica la orientación (θ) del píxel (i,j). El código que calcula la orientación es el siguiente:

```
"""
La funcion getOrientation calcula la orientacion del vector gradiente
en cada pixel en radianes. Para ello alisa fuertemente las derivadas
con un sigma 4.5. La derivada en X, al ser divisor, tenemos que no puede
ser 0, entonces le asignamos 1 como nos explican en el video siguiente:
https://www.youtube.com/watch?v=j7r3C-otk-U (sobre el minuto 10 - 11)
"""

def getOrientation(image,sigma=4.5,kSize=3,bordertype=cv2.BORDER_REFLECT):
    r = cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)
    derx=cv2.Sobel(r,cv2.CV_32F,1,0,ksize=kSize)
    dery=cv2.Sobel(r,cv2.CV_32F,0,1,ksize=kSize)
    derx=my_imGaussConvolveSingleShape(derx,sigma,bordertype)
    dery=my_imGaussConvolveSingleShape(dery,sigma,bordertype)
```

```

derx[derx==0.] = 1.
orientation=np.arctan(dery/derx)
return orientation

```

Para añadir la orientación a los puntos, vamos a calcularla primero con la función que acabamos de explicar y luego añadiremos este dato a la lista de puntos que ya teníamos. Para hacer esto, nos creamos una nueva función `addOrientationToPoints` que recibe la imagen y una lista de puntos hallados con anterioridad. Dicha función calcula la orientación de cada píxel en la imagen y, a los que están en la lista de puntos pasada por parámetro, les añade la orientación. El código es el siguiente:

```

"""
La funcion addOrientationToPoints añade la orientacion a cada punto
de forma que cada punto tiene formato (row,col,escala,orientacion).
"""

def addOrientationToPoints(image,points):
    orientation=getOrientation(image)
    newpoints=[(row,col,scale,orientation[row][col]) for (row,col,scale) in points]
    return newpoints

```

Para recopilar todas las tareas vistas hasta ahora, creamos una única función que nos permitirá obtener una lista ordenada por criterio Harris de puntos refinados que incluyen orientación. Esta función es `getStrongestPointsComplete` y su código es el siguiente:

```

"""
La funcion getStrongestPointsComplete es una compilacion de
getStrongestPoints (obtener los puntos fuertes en base a su
valor harris), la ordenacion, la fase de refinamiento y el
calculo de la orientacion. Asi, a partir de una piramide, y
un numero maximo de puntos asi como otros parametros como
tamaños de mascara permite obtener una lista de puntos fuertes
ordenados en base a su valor Harris de manera que cada punto
es una cuadrupla (fila, columna, escala, orientacion)
"""

def getStrongestPointsComplete(pir,blockSize,kSize,suppresionsize,thresold,maxPoints,
                               criteria,maskSize,deadzone,sigma=4.5,
                               k=0.04,bordertype=0,weights=[]):
    #FASE DE OBTENCION DE PUNTOS FUERTES Y ORDENACION
    points=getStrongestPoints(pir,blockSize,kSize,
                               suppressionsize,thresold,
                               maxPoints,k,bordertype,weights)

    #FASE DE REFINAMIENTO DE PUNTOS FUERTES
    points=refinePoints(piry1,criteria,points,maskSize,deadzone)

    #FASE DE CALCULO DE LA ORIENTACION
    points=addOrientationToPoints(pir[0],points)

    return points

```

Ahora ya podemos obtener una lista de puntos en formato (*fila, columna, escala, orientacion*) refinados llamando a una sola función. Nuestra siguiente tarea consistirá en tratar de representar gráficamente la orientación. Dado un punto de interés, lo representábamos con un círculo de radio inversamente proporcional a la escala de la que provenía en la imagen de mayor tamaño de la pirámide Gaussiana. Ahora, vamos a representar la orientación del siguiente modo:

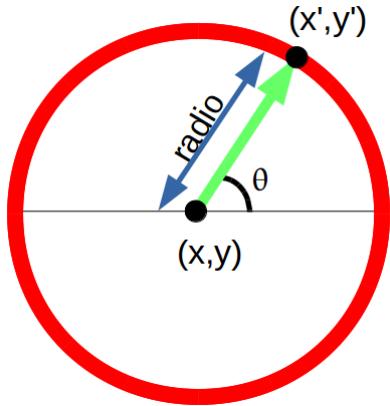


Figura 9. Representación gráfica de la orientación.

La flecha que define la orientación se dibujará con la función `arrowedLine` que proporciona OpenCV, pero dicha función necesita un punto de origen (x, y) y un punto de destino (x', y') . No hay problema, el punto de origen lo conocemos y el punto de destino puede calcularse mediante las siguientes expresiones:

$$x' = x + \cos\theta \cdot \text{radio}$$

$$y' = y + \sin\theta \cdot \text{radio}$$

Fijémonos que también podríamos calcular el seno y el coseno con el valor correspondiente de las matrices de derivadas. La función que añadirá los puntos a la imagen se codifica del siguiente modo.

```
"""
La funcion paintPointsAndOrientationInImage
pinta los puntos y la orientacion de los mismos.
"""

def paintPointsAndOrientationInImage(image,points_with_orientation,
                                      rp=255,gp=0,bp=0,
                                      ro=0,go=255,bo=0,
                                      thickness=2):
    newimg=np.copy(image)
    for pnt in points_with_orientation:
        radius=5+2**((2*(int(pnt[2]))))
        pointorigin=(int(pnt[1])*2**int(pnt[2]),int(pnt[0])*2**int(pnt[2]))
        pointdstx=int(pointorigin[0] +(math.cos(pnt[3])*radius))
        pointdsty=int(pointorigin[1] -(math.sin(pnt[3])*radius))
        pointdst=(pointdstx,pointdsty)
        cv2.circle(newimg,pointorigin, radius, (rp,gp,bp), thickness)
        cv2.arrowedLine(newimg,
                        pointorigin,
                        pointdst,
                        (ro,go,bo),
                        thickness)
    return newimg
```

Así aprovechamos lo que ya teníamos de forma que el radio sigue siendo inversamente proporcional a la escala, pero añadimos una característica más a cada punto que se dibuja. El resultado de representar los puntos con orientación en la imagen es el siguiente:

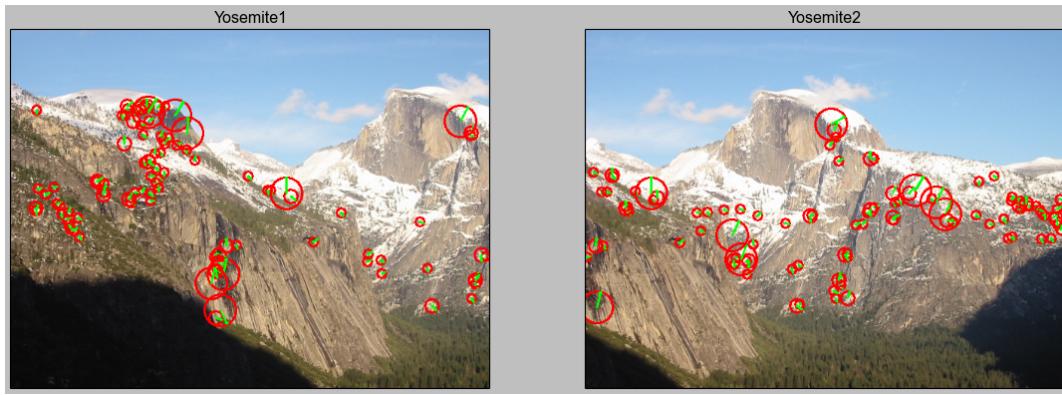


Figura 10. Representación de la orientación de 100 puntos en cada imagen.

Uso de los detectores KAZE/AKAZE. (Opción 10 puntos)

2.- Usar los detectores OpenCV (KAZE/AKAZE) sobre las imágenes de Yosemite.rar. Extraer extraer sus listas de keyPoints y establecer las correspondencias existentes entre ellas (Ayuda: usar la clase DescriptorMatcher y BFMatcher de OpenCV). Valorar la calidad de los resultados obtenidos bajo el criterio de correspondencias OpenCV “BruteForce+crossCheck”.

Usaremos primero KAZE con todas las opciones disponibles y posteriormente AKAZE del mismo modo. Estableceremos una comparativa con una muestra de 500 puntos en correspondencia. Implementaré, por comodidad las siguientes funciones:

```
"""
La funcion getKeyPointsAndDescriptorsKAZE permite obtener
los keypoints y descriptores usando el detector KAZE
"""

def getKeyPointsAndDescriptorsKAZE(image):
    kaze = cv2.KAZE_create()
    kp = kaze.detect(image, None)
    kp, des = kaze.compute(image, kp)
    return kp,des

"""
La funcion getKeyPointsAndDescriptorsAKAZE permite obtener
los keypoints y descriptores usando el detector AKAZE
"""

def getKeyPointsAndDescriptorsAKAZE(image):
    akaze = cv2.AKAZE_create()
    kp = akaze.detect(image, None)
    kp, des = akaze.compute(image, kp)
    return kp,des

"""

La funcion getMatchesBF permite obtener matches a partir
de los descriptores usando BFMatcher
"""

def getMatchesBF(des1,des2,crosscheck=True):
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=crosscheck)
    matches = bf.match(des1,des2)
    matches = sorted(matches, key = lambda x:x.distance)
    return matches

"""

La funcion getMatchesDescriptor permite obtener matches
a partir de los descriptores usando DescriptorMatcher
"""

def getMatchesDescriptor(des1,des2):
    dm = cv2.DescriptorMatcher_create("BruteForce")
    matches = dm.match(des1,des2)
    matches = sorted(matches, key = lambda x:x.distance)
    return matches
```

```

"""
La funcion drawMatches permite dibujar las correspondencias
entre dos imagenes. Devuelve una imagen que une las dos
pasadas por parametro y dibuja lineas rectas entre los puntos
correspondidos
"""
def drawMatches(img1,kp1,img2,kp2,matches):
    return cv2.drawMatches(img1,kp1,img2,kp2,matches,None,flags=2)

```

Detector KAZE con DescriptorMatcher opción BruteForce

La clase DescriptorMatcher permite dos opciones: “BruteForce” y “BruteForce-Hamming”. La segunda opción no tiene sentido, pues no utilizamos distancias de Hamming, sino distancias Euclídeas (norma L2 o distancia L2). De hecho, si intentamos utilizarla, obtenemos un error. Analicemos los resultados con esta combinación:

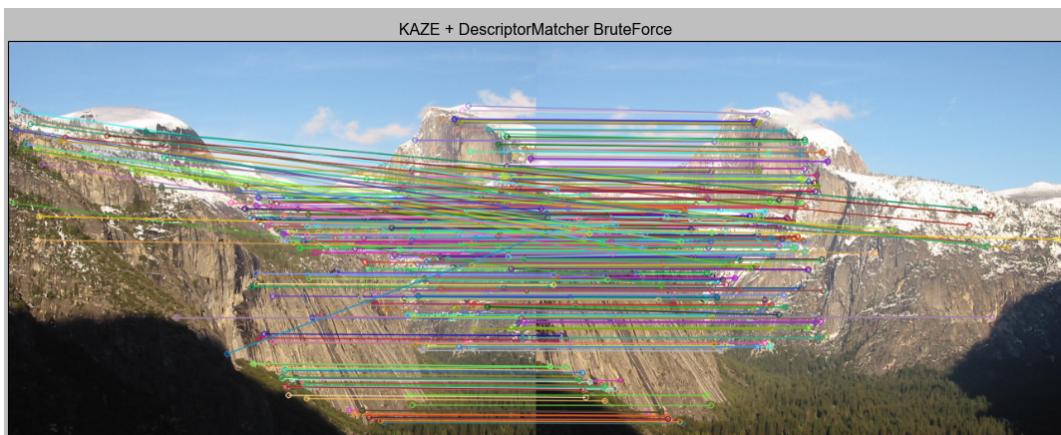


Figura 11. 500 puntos detectados con KAZE + DescriptorMatcher modo BruteForce.

Esta combinación encontró 1223 puntos en total.

Detector KAZE con BFMatcher opción BruteForce sin Crosscheck

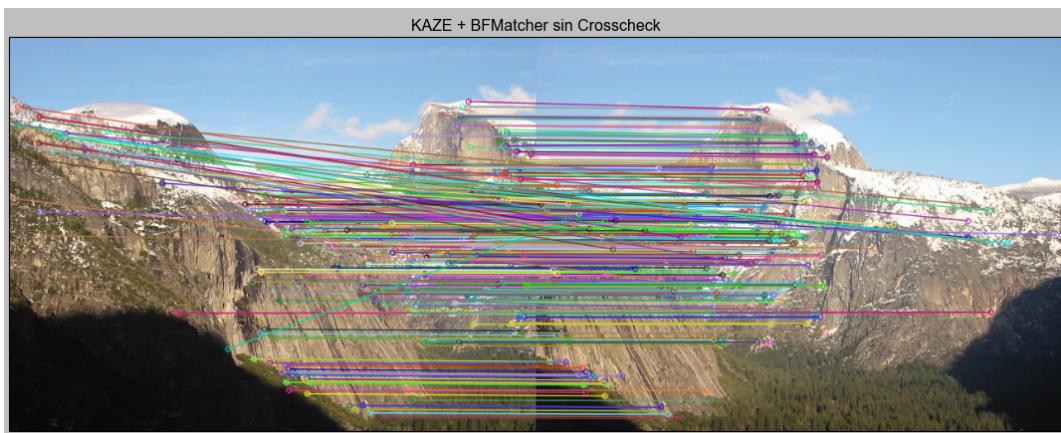


Figura 12. 500 puntos detectados con KAZE + BFMatcher modo BruteForce.

Esta combinación, al igual que la anterior, encontró 1223 puntos en total.

Detector KAZE con BFMatcher opción BruteForce con Crosscheck

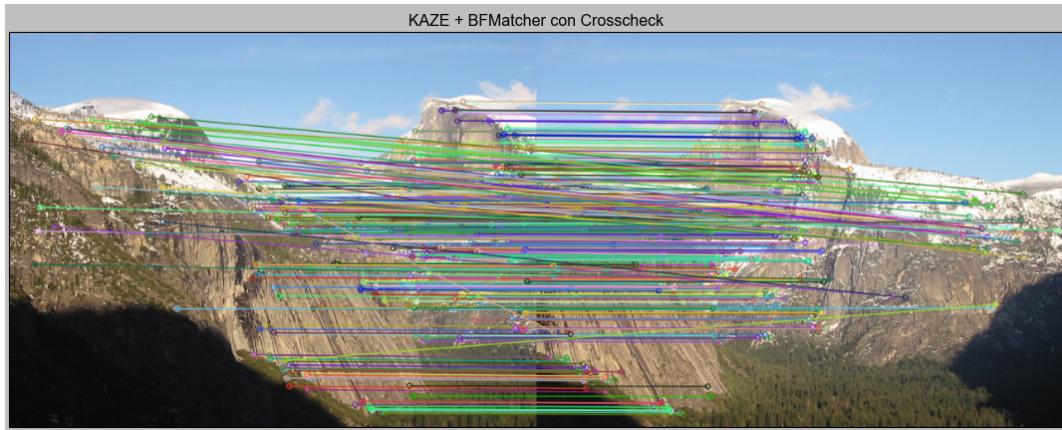


Figura 13. 500 puntos detectados con KAZE + BFMatcher modo BruteForce+Crosscheck.

Esta combinación encontró 691 puntos en total.

A pesar de que parecía la opción más razonable y la mejor, el detector KAZE con BFMatcher BruteForce+Crosscheck queda en evidencia ante las dos anteriores. Aunque elige menos puntos, la muestra (que por otra parte son gran parte de los puntos encontrados por la combinación) comete más errores que sus dos competidores. Entre las otras dos combinaciones, son calcadas, daría igual una que otra.

Vamos a continuación a analizar el detector AKAZE.

Detector AKAZE con DescriptorMatcher opción BruteForce



Figura 14. 500 puntos detectados con AKAZE + BFMatcher modo BruteForce+Crosscheck.

Aparentemente desastroso en comparación con las mismas opciones y tamaño de muestra que KAZE. El detector KAZE comete menos errores, aunque AKAZE en esta combinación encuentra 1367 puntos en total.

Detector AKAZE con BFMatcher opción BruteForce sin Crosscheck

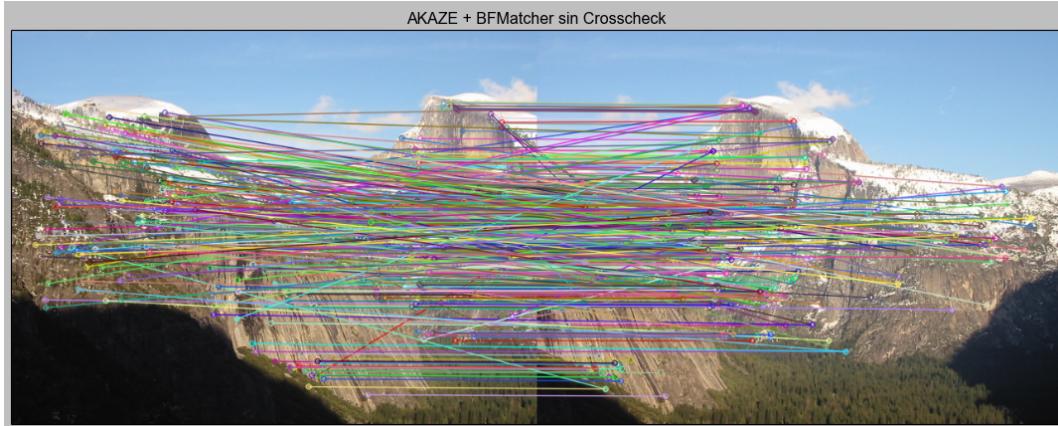


Figura 15. 500 puntos detectados con AKAZE + BFMatcher modo BruteForce+Crosscheck.

En este caso, igual que en el anterior KAZE comete menos errores. También encuentra 1367 en total.

Detector AKAZE con BFMatcher opción BruteForce con Crosscheck

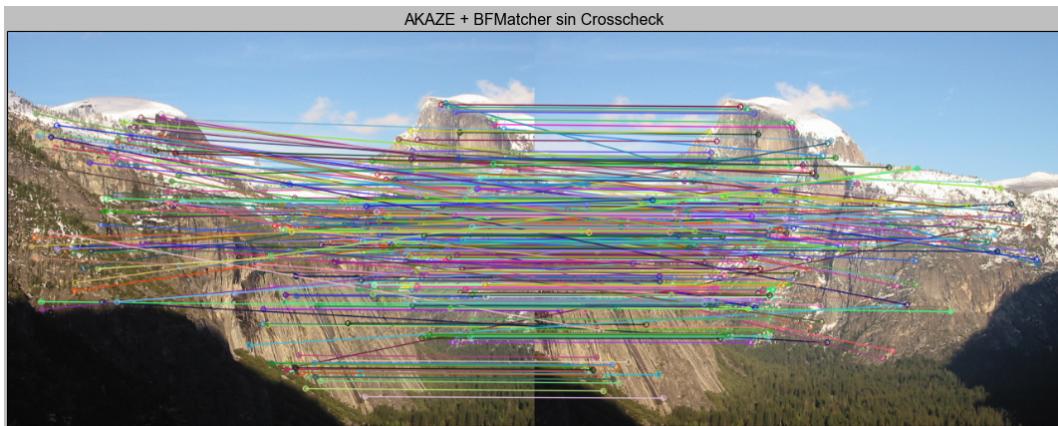


Figura 16. 500 puntos detectados con AKAZE + BFMatcher modo BruteForce+Crosscheck.

Al igual que en el caso anterior, AKAZE es superado por KAZE, aunque en total AKAZE encuentra 747 puntos por los 691 de KAZE en este modo.

En teoría AKAZE es una versión acelerada de KAZE y debería hallar las mismas correspondencias, aunque más rápido. En este caso, ya sea por las imágenes en concreto o por otras razones, aunque AKAZE encuentra más puntos, la proporción de errores que comete KAZE, comparando siempre con las mismas opciones y parámetros, es menor. Entre las dos opciones y ante la evidencia elegiría KAZE, pero habría que analizar más casos para poder decidir con más rotundidad. No podemos asegurar que este caso no es una excepción y tampoco que KAZE en general es mejor, aunque en este caso así sea. En un experimento hacia el final del documento enfrentaré ambos detectores en la creación de panoramas, viendo que los fallos cometidos no parecen derivar en consecuencias catastróficas.

Construcción de mosaicos o panoramas (Opción 10 puntos)

3.Escribir una función que forme un Mosaico de calidad a partir de $N > 3$ imágenes relacionadas por homografías, sus listas de keyPoints calculados de acuerdo al punto anterior y las correspondencias encontradas entre dichas listas. Estimar las homografías entre ellas usando la función `findHomography(p1,p2, CV_RANSAC,1)`. Para el mosaico será necesario. a) definir una imagen en la que pintaremos el mosaico; b) definir la homografía que lleva cada una de las imágenes a la imagen del mosaico; c) usar la función `warpPerspective()` para trasladar cada imagen al mosaico.

Primera aproximación: Construcción de un panorama con dos imágenes.

Básicamente una homografía es una matriz que relaciona dos planos y debe cumplir dos condiciones: debe ser una matriz 3×3 y su determinante debe ser distinto de cero. En este ámbito una homografía va a relacionar dos imágenes, entonces, una homografía de una imagen I_1 a una imagen I_2 , es decir H_{12} , hace corresponder puntos de I_1 con puntos de I_2 en un espacio de coordenadas homogéneas. Cuando multiplicamos las coordenadas de los puntos de I_1 por la homografía, algunos puntos de I_1 tomarán las mismas coordenadas que algunos puntos de I_2 y el resto de puntos quedarán colocados proporcionalmente. La siguiente figura mostraría la imagen I_1 multiplicada por una homografía H_{12} que hace corresponder los puntos verdes de I_1 con los puntos verdes de I_2 .

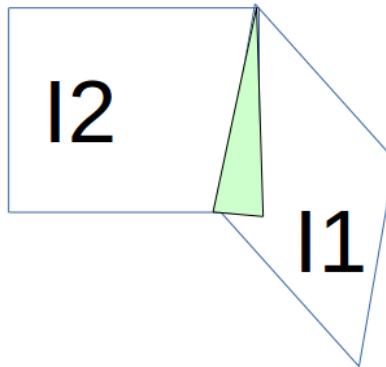


Figura 17. Puntos relacionados por una homografía.

Un ejemplo de homografía sería una translación. Imaginemos por ejemplo que tenemos una imagen cuya esquina superior izquierda tiene, lógicamente, coordenadas $P = (0, 0)$. Queremos llevar esta esquina a la posición central de un lienzo que tiene a filas y b columnas, es decir, queremos convertir las coordenadas $P = (0, 0)$ en $P' = (a/2, b/2)$. Entonces, definimos la homografía (de translación):

$$H_0 = \begin{pmatrix} 1 & 0 & a/2 \\ 0 & 1 & b/2 \\ 0 & 0 & 1 \end{pmatrix}$$

Acto seguido convertimos las coordenadas de nuestra imagen a coordenadas homogéneas, en este caso, la esquina que queremos trasladar se convertiría en $(0 \ 0 \ w)^T$, donde $w = 1$. Multiplicamos la homografía por cada punto (para ilustrar, sólo mostramos la esquina en cuestión):

$$P' = H_0 \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a/2 \\ 0 & 1 & b/2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} a/2 \\ b/2 \\ 1 \end{pmatrix}$$

Si volvemos al sistema de coordenadas normales, la esquina se ha convertido, a través de esta homografía en $P' = (a/2 \ b/2)$. La función `warpPerspective` que proporciona OpenCV se encarga justamente de esto, de aplicar la homografía a las coordenadas de cada punto de una imagen y “traducir” sus coordenadas.

Entendido por encima qué es una homografía, vamos a tratar de hacer un panorama con dos imágenes. Para ello, vamos a necesitar, por supuesto, las dos imágenes y dos homografías: Una de translación al lienzo, H_0 y la homografía de una de las imágenes a otra. El primer paso es definir el tamaño del lienzo. Yo lo calcularé como el doble del máximo número de filas (alto = $2 \cdot maxRow$) y el doble del máximo número de columnas (ancho= $2 \cdot maxCol$). El segundo paso es trasladar una de las imágenes al centro, optaremos por la de la izquierda (aunque da igual). La homografía H_0 se construye como:

$$H_0 = \begin{pmatrix} 1 & 0 & (2 \cdot maxRow/2) - (numfilas(imagenIzquierda)/2) \\ 0 & 1 & (2 \cdot maxCol/2) - (numcols(imagenIzquierda)/2) \\ 0 & 0 & 1 \end{pmatrix} = \\ = \begin{pmatrix} 1 & 0 & maxRow - (numfilas(imagenIzquierda)/2) \\ 0 & 1 & maxCol - (numcols(imagenIzquierda)/2) \\ 0 & 0 & 1 \end{pmatrix}$$

Que trasladaría la imagen izquierda al centro del lienzo (puede verse haciendo las operaciones del mismo modo que en el ejemplo anterior). Ahora nos quedaría encontrar la homografía que lleva la imagen de la derecha a la imagen de la izquierda, con la función `findHomography` de OpenCV y componerla con (multiplicarla por) la homografía de translación H_0 . La teoría del proceso es la siguiente:

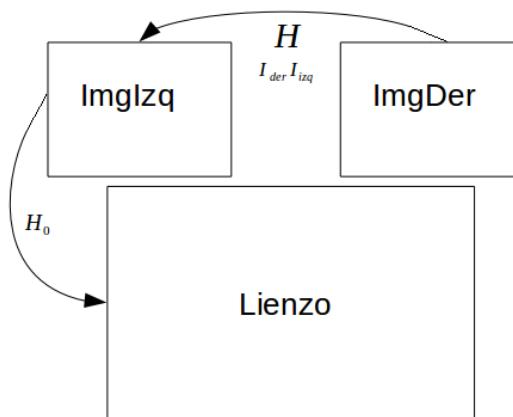


Figura 18. Teoría para la construcción de un mosaico de dos imágenes.

Que se implementa en Python como sigue:

```
"""
Dadas dos imagenes, construye una imagen panoramica
de ambas, para lo que se calcula una homografia de
translacion para poner la imagen de la izquierda y la
homografia entre ambas imagenes, que se multiplica la
de translacion para finalmente colocar la imagen de
la derecha.
"""

def constructPanoramaTwoImages(imgLeft, imgRight,
                                quitleftover=True):
    maxX=0
    maxY=0
```

```

if imgLeft.shape[0]>imgRight.shape[0] :
    maxY=imgLeft.shape[0]
else:
    maxY=imgLeft.shape[0]
if imgLeft.shape[1]>imgRight.shape[1]:
    maxX=imgLeft.shape[1]
else:
    maxX=imgLeft.shape[1]

size_lienzo=[maxX*2,maxY*2]
H0=np.array([
    [1.,0.,(size_lienzo[0]/2)-(maxX/2)],
    [0.,1.,(size_lienzo[1]/2)-(maxY/2)],
    [0.,0.,1.]
])
H=np.eye(3)
result=cv2.warpPerspective(imgLeft,H0.dot(H),
                           (size_lienzo[0],size_lienzo[1]),
                           cv2.BORDER_TRANSPARENT)

H=H.dot(getHomography(imgRight,imgLeft))
cv2.warpPerspective(imgRight, H0.dot(H),
                     (size_lienzo[0],size_lienzo[1]),
                     dst=result,
                     borderMode=cv2.BORDER_TRANSPARENT)
if(quitleftover):
    result=quitLeftOver(result)
return result

```

Como podemos apreciar, para calcular la homografía vamos a utilizar la función `getHomography`, que usa a su vez la función `findHomography` de OpenCV. Tal y como vimos en la comparativa de los descriptores, ganaba KAZE. De entre las combinaciones de KAZE, la que usaba DescriptorMatcher y BFMatcher sin crosscheck eran las que cometían menos errores (en una comparativa visual). En principio, vamos a usar la combinación KAZE+BFMatcher sin crosscheck, aunque también usaremos AKAZE+BFMatcher con crosscheck en un experimento al final del documento para verificar que no hay diferencias relevantes, al contrario de lo que pudiera parecer a tenor de lo analizado. El código para calcular la homografía sería el siguiente:

```

"""
La funcion H calcula una homografia de imagenIzq a imagenRight
usando KAZE+BFMatcher sin crosscheck.
"""

def getHomography(imgLeft,imgRight):
    kp1,des1=getKeyPointsAndDescriptorsKAZE(imgLeft)
    kp2,des2=getKeyPointsAndDescriptorsKAZE(imgRight)
    matches=getMatchesBF(des1,des2,crosscheck=False)
    p1=np.float32([(kp1[m.queryIdx].pt) for m in matches])
    p2=np.float32([(kp2[m.trainIdx].pt) for m in matches])
    H, mask = cv2.findHomography(p1, p2, cv2.RANSAC, 1.0)
    return H

```

Si quisiéramos usar otro detector, habría que cambiar esta función.

Una vez colocadas todas las imágenes sobre el lienzo nos quedará espacio sobrante, que quitaremos con la ayuda de la función `boundingRect` de OpenCV esta función devuelve las coordenadas de la esquina superior

izquierda de dónde empieza la imagen, un alto y un ancho. El problema es que sólo funciona con imágenes en grises. Crearemos una función auxiliar `quitLeftOver` que le mandará la versión en escala de grises del panorama a `boundingRect` y recortaremos la imagen original a color con los datos que devuelve. El código de dicha función auxiliar es:

```
"""
Dada una imagen con pixeles que
sobran devuelve la imagen sin
esos pixeles
"""

def quitLeftOver(image):
    imageGray=cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)
    x,y,w,h = cv2.boundingRect(imageGray)
    newImg = image[y:y+h,x:x+w]
    return newImg
```

Vamos a ver el resultado de la construcción de un panorama con dos imágenes:

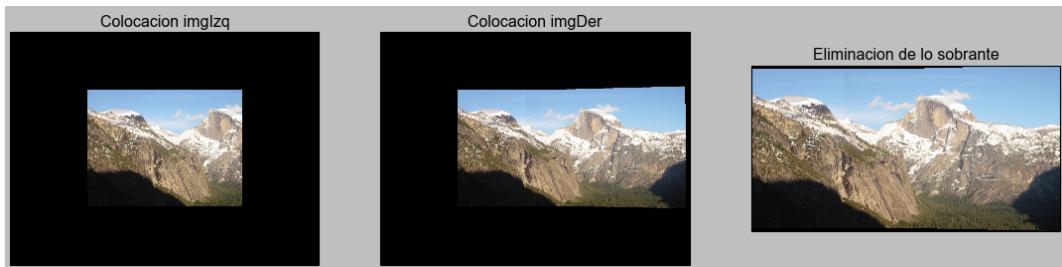


Figura 19. Pasos de la construcción del panorama con dos imágenes.

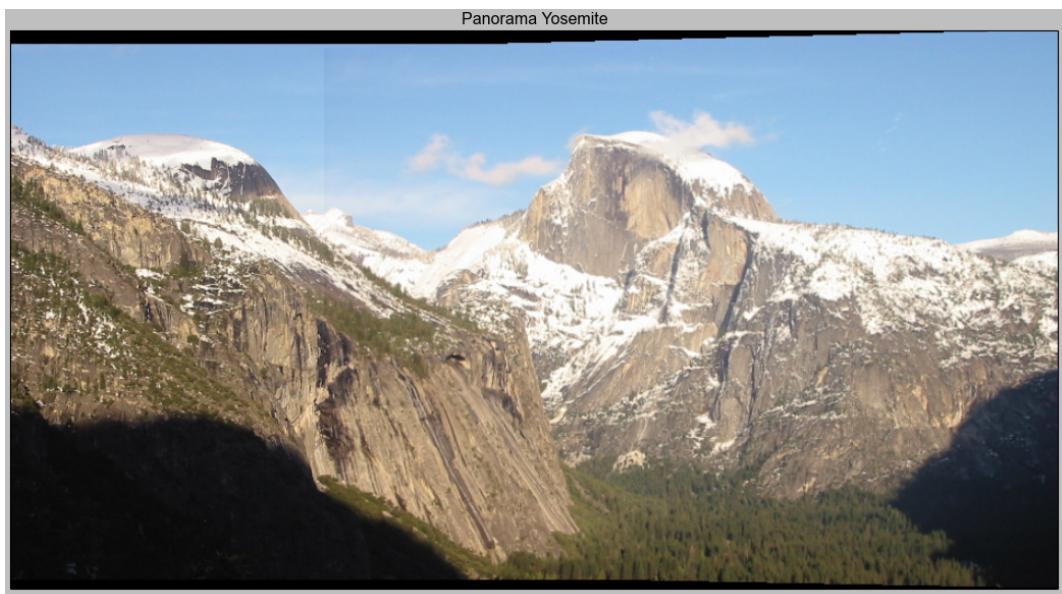


Figura 20. Resultado final del panorama de dos imágenes.

Construcción de un panorama con N imágenes.

Para construir un panorama con N imágenes, la primera idea que se nos viene a la cabeza es, usando lo implementado anteriormente, ir de izquierda a derecha (o de derecha a izquierda) uniendo dos imágenes que dan como resultado un panorama. A dicho resultado “coserle” la siguiente imagen y así sucesivamente. El código de esta **mala idea** sería el siguiente:

```
"""
Construye un panorama de izquierda a derecha. Es la primera idea
pero no es una buena idea, el panorama está construido mal.
"""
def constructPanoramaMultipleImagesBAD(listOfImages,quitleftover=True):
    result=listOfImages[0]
    for imagen in listOfImages:
        result=constructPanoramaTwoImages(result,imagen)
    return result
```

El resultado de hacer esto es lo que vemos en las siguientes figuras:

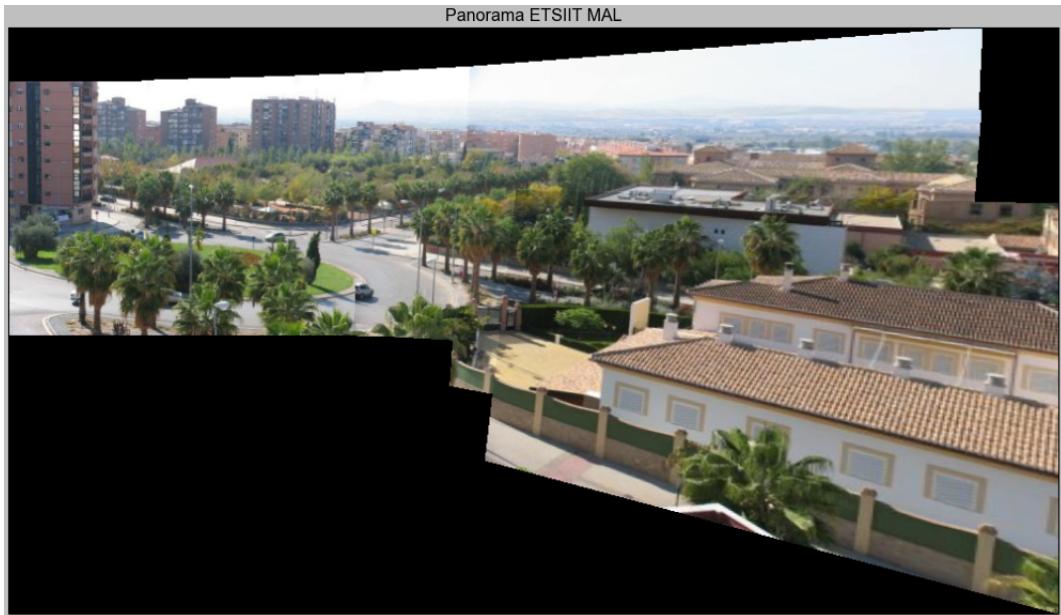


Figura 21. Panorama de N imágenes **MAL** construído.

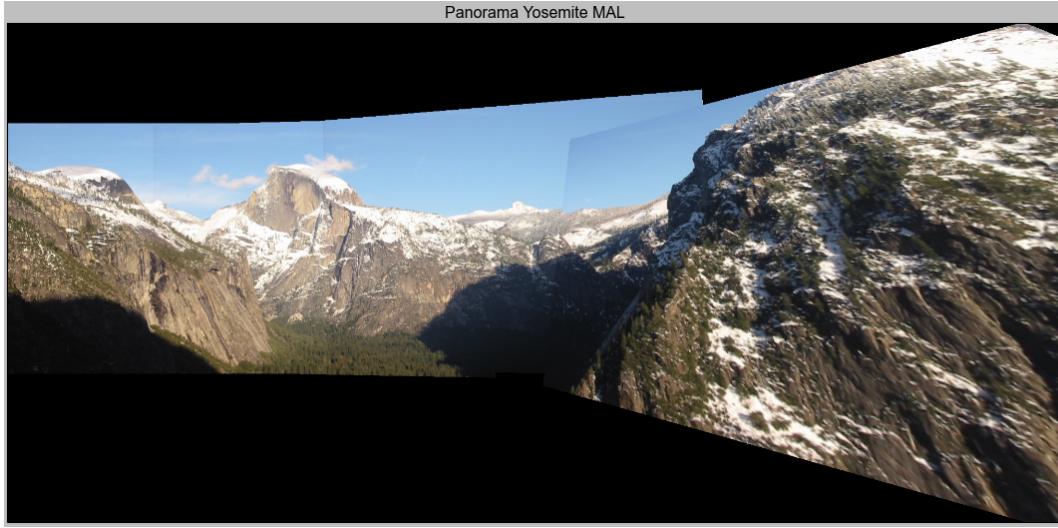


Figura 22. Otro panorama de N imágenes MAL construido.

Tan pronto como vemos estos resultados abandonamos la idea, pues las deformaciones sufridas por las últimas imágenes colocadas en perspectiva son muy fuertes en relación a las que sufren las primeras que se colocan. El sentido común en seguida se pregunta ¿por qué no comienzo por el centro y crezco hacia los lados? Esta es la aproximación correcta, pues se reducirán los errores acumulados. Daremos una vuelta de tuerca más a la teoría que desarrollamos en el panorama de dos imágenes, solo que ahora tenemos $n/2$ por la izquierda y $n/2$ por la derecha. Ahora el tamaño del lienzo será, en alto, el máximo de filas por el número de imágenes y, en ancho, el máximo de columnas por el número de imágenes. Entonces, la composición de homografías se hace así:

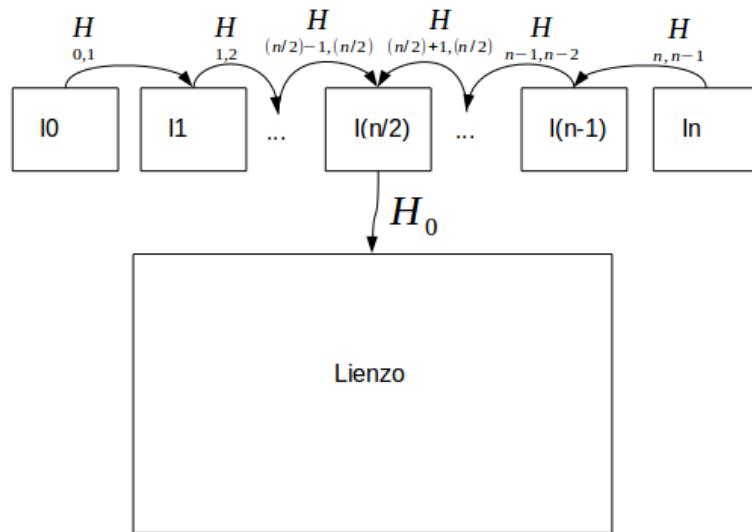


Figura 23. Teoría para la construcción de un panorama de N imágenes.

El código que implementa esta teoría es el siguiente:

```
"""
Dadas N imagenes en una lista, construye una
imagen panoramica con todas, para lo que se calcula
la posicion central y se coloca en el lienzo.
```

A partir de aquí se van componiendo homografías por la derecha y por la izquierda, de manera que el panorama va creciendo hacia ambos lados. Esto es para reducir las deformaciones producidas por las perspectivas.

```

"""
def constructPanoramaMultipleImages(listOfImages,quitleftover=True):
    maxX=0
    maxY=0
    numImages=len(listOfImages)
    imgCentro=numImages/2
    for image in listOfImages:
        if image.shape[0]>maxY:
            maxY=image.shape[0]
        if image.shape[1]>maxX:
            maxX=image.shape[1]

    size_lienzo=[maxX*numImages,maxY*numImages]
    H0=np.array([
        [1.,0.,(size_lienzo[0]/2)-(maxX/2)],
        [0.,1.,(size_lienzo[1]/2)-(maxY/2)],
        [0.,0.,1.]
    ])
    H=np.eye(3)
    result=cv2.warpPerspective(listOfImages[imgCentro], H0.dot(H),
                               (size_lienzo[0],size_lienzo[1]),
                               cv2.BORDER_TRANSPARENT)

    #Crecimiento hacia la derecha
    for i in xrange(imgCentro,numImages-1):
        H=H.dot(getHomography(listOfImages[i+1],listOfImages[i]))
        cv2.warpPerspective(listOfImages[i+1], H0.dot(H),
                           (size_lienzo[0],size_lienzo[1]),
                           dst=result,
                           borderMode=cv2.BORDER_TRANSPARENT)

    H=np.eye(3)
    #Crecimiento hacia la izquierda
    for i in xrange(imgCentro,0,-1):
        H=H.dot(getHomography(listOfImages[i-1],listOfImages[i]))
        cv2.warpPerspective(listOfImages[i-1], H0.dot(H),
                           (size_lienzo[0],size_lienzo[1]),
                           dst=result,
                           borderMode=cv2.BORDER_TRANSPARENT)

    if(quitleftover):
        result=quitLeftOver(result)

    return result

```

Los resultados, como se puede comprobar en las siguientes figuras, aumentan notablemente la calidad del panorama en relación con el modo de construcción anterior:

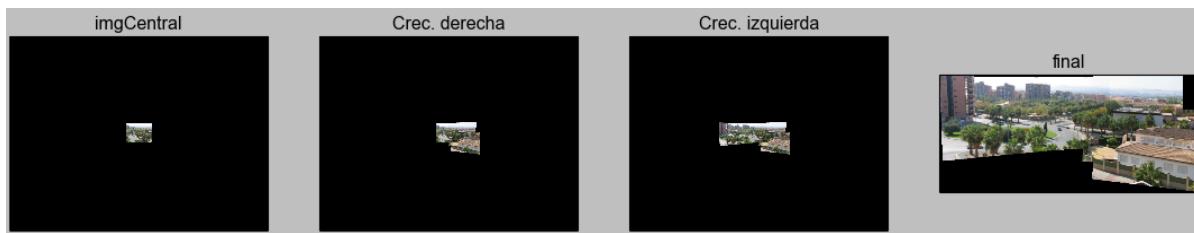


Figura 24. Pasos correctos para la construcción de un panorama de N imágenes.

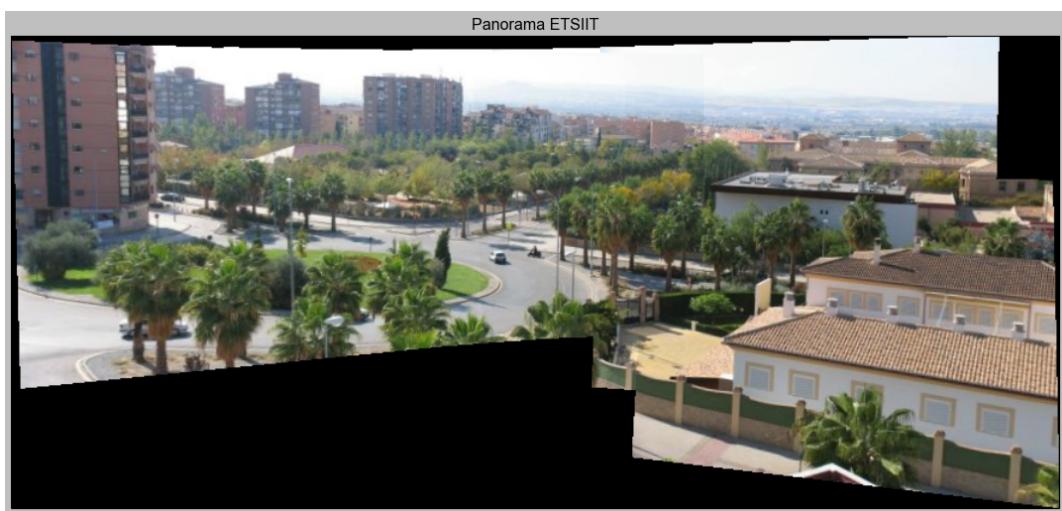


Figura 25. Panorama de N imágenes BIEN construido.

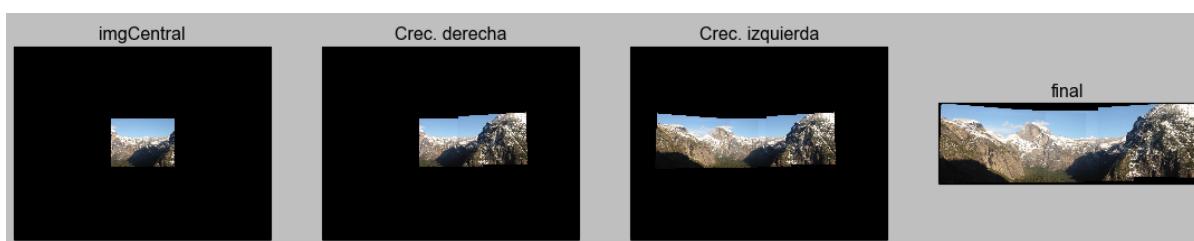


Figura 26. Pasos correctos para la construcción de otro panorama de N imágenes.

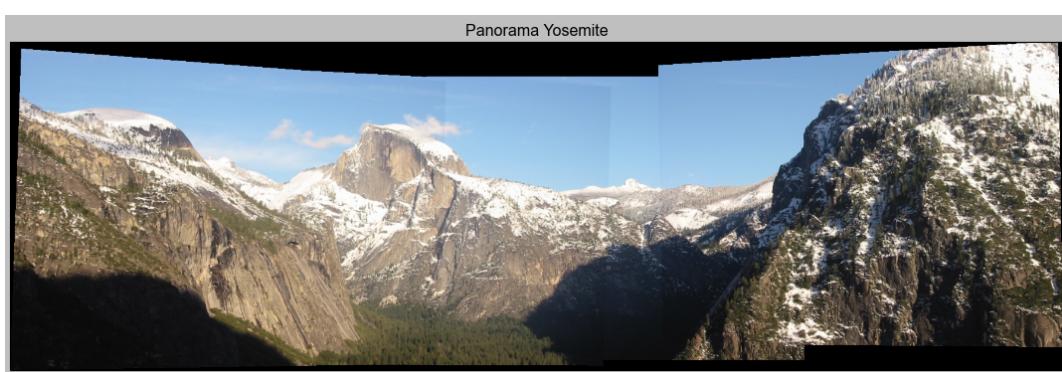


Figura 27. Otro panorama de N imágenes BIEN construido.

Experimento

Dudando de qué pasaría si utilizaba AKAZE como detector, con BFMatcher+crosscheck, decidí cambiarlo en la función `getHomography` y volverlo a probar de cara a construir panoramas, La función queda así.

```
"""
La funcion H devuelve una homografia de imagenIzq a imagenRight
usando AKAZE+BFMatcher con crosscheck
"""

def getHomography(imgLeft,imgRight):
```

```
    kp1,des1=getKeyPointsAndDescriptorsAKAZE(imgLeft)
    kp2,des2=getKeyPointsAndDescriptorsAKAZE(imgRight)
    matches=getMatchesBF(des1,des2,crosscheck=True)
    p1=np.float32([(kp1[m.queryIdx].pt) for m in matches])
    p2=np.float32([(kp2[m.trainIdx].pt) for m in matches])
    H, mask = cv2.findHomography(p1, p2, cv2.RANSAC, 1.0)
    return H
```

Curiosamente, a pesar de lo que vimos que sucedía en el análisis de los detectores, los resultados obtenidos son idénticos, como muestran las figuras:

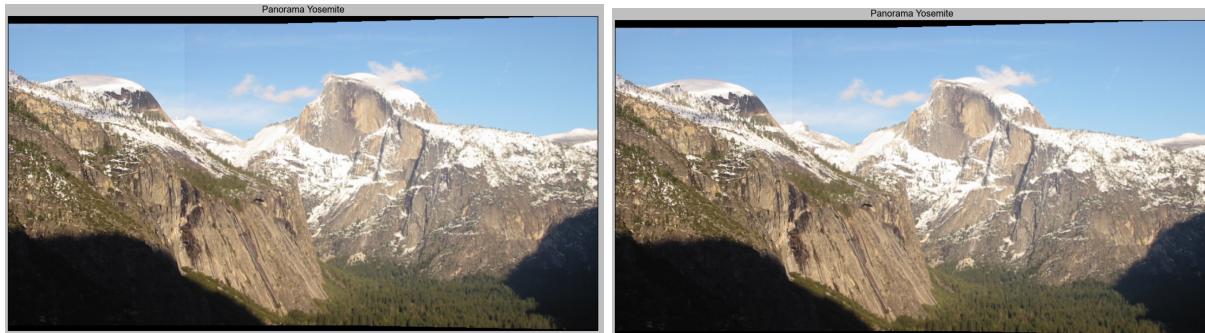


Figura 28. KAZE (izq) vs. AKAZE (der) para dos imágenes.



Figura 29. KAZE (izq) vs. AKAZE (der) para N imágenes (I).

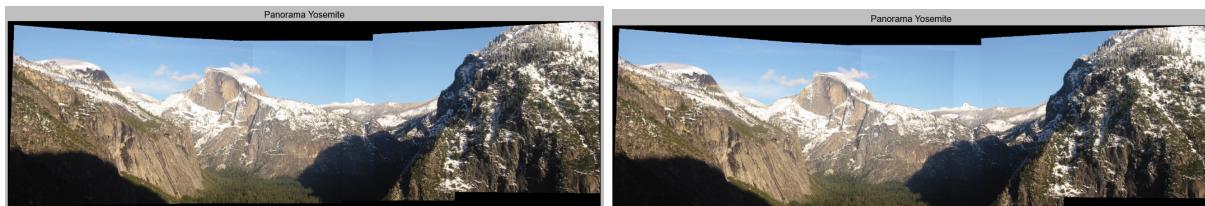


Figura 30. KAZE (izq) vs. AKAZE (der) para N imágenes (II).

Entonces, para este cometido, no sería crítico decidir entre uno u otro detector, pues ambos resultados son de la misma calidad.

Cuestiones técnicas

El proyecto ha sido implementado y probado en Ubuntu 14.04 LTS, con la versión Python 2.7.6. No se ha utilizado ninguna biblioteca no estándar de Python, excepto OpenCV, que hay que tener correctamente instalado y configurado en las variables de entorno. Para ejecutar correctamente el proyecto, hay que ejecutar el comando `python P2.py` (o `python2 P2.py` en caso de no ser la 2.7 la versión de Python por defecto) en una terminal desde el directorio donde se encuentra dicho archivo. Las imágenes que se usan deben estar dentro de la carpeta llamada `imagenes`, que se adjunta y no debería ser modificada.

Referencias

1. *Harris corner detector formalization.* Wikipedia.
https://en.wikipedia.org/wiki/Corner_detection#The_Harris_.26_Stephens_.2F_Plessey_.2F_Shi.E2.80.93Tomasi_corner_detection_algorithms
2. *Multi-Image Matching using Multi-Scale Oriented Patches.* M. Brown, R. Szeliski, S. Winder. Diciembre 2004.
3. *Edge Detection with Gradients: Part 01 - Gradient Orientation & Magnitude.* Vkedco. (desde min. 10)
<https://www.youtube.com/watch?v=j7r3C-otk-U>