Trabajo 2.

VC. Detección de puntos relevantes y Construcción de panoramas.

Alberto Armijo Ruiz. 4 GII.

Ejercicio 1.

Antes de comenzar el primer apartado, debemos crear la pirámide con 5 niveles, para ello, utilizaremos la función implementada en la práctica anterior. Para este apartado, utilizaremos la imagen “yosemite1.jpg” para calcular sus keypoints.

Apartado A.

En este apartado, deberemos crear una serie de funciones. Primer, crearemos la función **calculateMatrixHarrys(matrizEigenValues)**, dentro de esta función, recorremos cada una de las filas y columnas de la matriz que nos devuelve la función **cv2.eigenValsAndVecs()**. Utilizaremos los valores *lambda1* y *lambda2* de cada autovalor para calcular el valor del criterio Harris en cada pixel de la imagen, que se calcula como criterio(H) = $\det(H)-k*\text{traza}(H)^2$; donde k es una constante con valor k=0.04.

Una vez hemos obtenido esta matriz, pasaremos a suprimir los no-máximos de la matriz. Para ello, se han creado tres funciones, **comprobarMaximoCentral()**, **cambiarACero()** y **suprimirNoMaximos()**, que utiliza las dos anteriores.

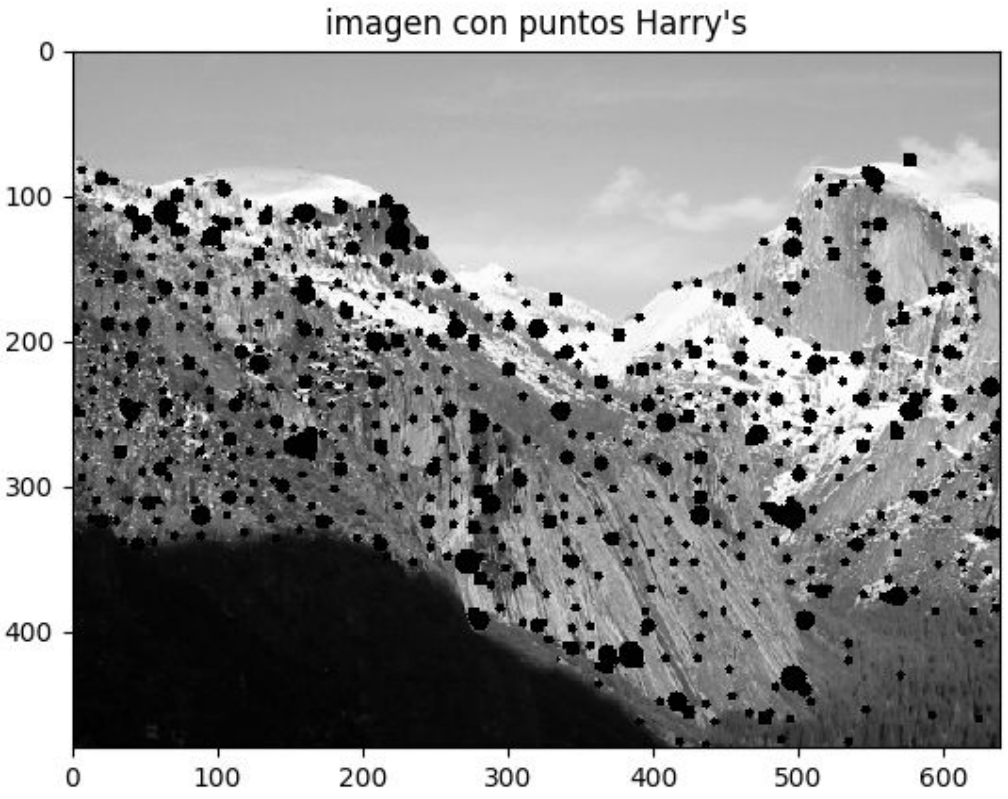
La función **comprobarMaximoCentral()**, comprueba que el punto que se le pasa es el máximo dentro de su ventana, si esto es así, devuelve True; para ello comprueba si el índice central y el índice del máximo son iguales. La función **cambiarACero()**, pone a cero todos los píxeles dentro de la ventana especificada por el punto central y el tamaño de la ventana en una matriz binaria que utilizamos para suprimir los no-máximos.

La función **suprimirNoMaximos()** crea una matriz binaria para comprobar que puntos son máximos y cuáles no, para ello, crear al principio una matriz binaria con todos sus valores a 255. Tras esto, recorre esta matriz comprobando si un píxel que tiene valor 255 es máximo dentro de su ventana, esto es así, todos los puntos de la ventana pasan a ser 0 dentro de la matriz binaria para evitar comprobar píxeles innecesariamente y el valor del píxel, su posición, y su escala se guardan dentro de los puntos Harris.

Tras esto, debemos ordenar los puntos Harris y quedarnos solamente con los mejores, en este caso, yo he optado por quedarme por *500/(2*escala)*, es decir, de la primera escala nos quedamos 250, de la segunda 125, etc... Para ordenar cada uno de los puntos, se ha utilizado la función **sorted()** de python, que nos permite especificar cuál queremos que sea el dato utilizado para ordenar mediante el parámetro *key.*

Por último, solo nos queda dibujar los puntos Harris en la escala imagen original, para ello se ha creado la función **dibujarPuntos()**, esta función recibe como parámetros las coordenadas de los puntos y su escala en un principio, después se utilizará un parámetro más para dibujar la orientación de cada punto. Dentro de esta función, lo primero que debemos hacer es calcular la posición de cada punto en la escala original, para ello, multiplicamos los puntos de escalas más pequeños por 2^{escala} Una vez tenemos esto,

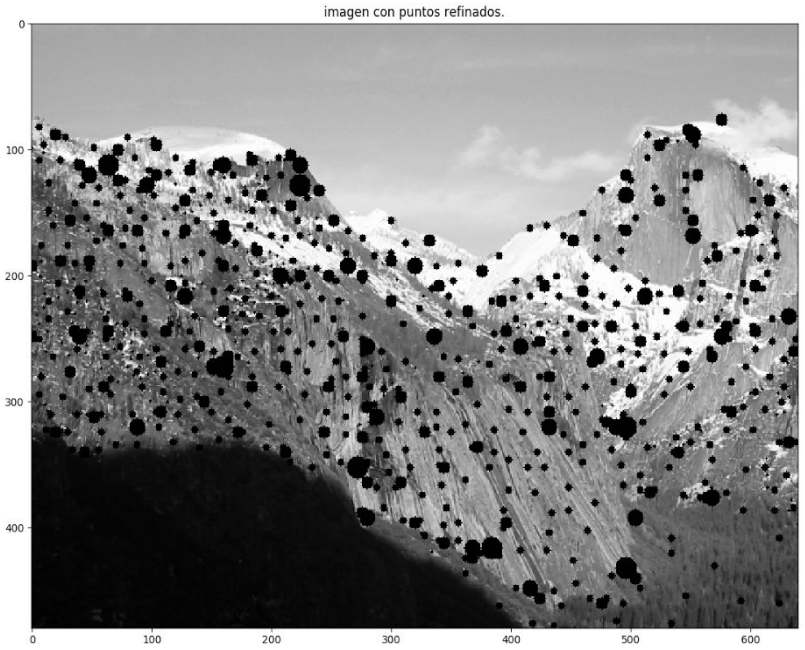
pasamos a dibujar un círculo en la posición de cada punto en una copia de la imagen original. El resultado obtenido es el siguiente:



Como se puede ver, el resultado es bastante bueno, ya que se detectan bastantes puntos, y ninguno de ellos está en el cielo. Es posible que alguno de los puntos Harris encontrados no se haya dibujado en su posición exacta, ya que al hacer el cambio de escala los puntos de escalas más pequeñas pueden tener cierto ruido.

Apartado B.

En este apartado, se refinará la posición de cada punto Harris con la función cv2.cornerSubPix(). Para el refinamiento, tendremos que pasar cada uno de los puntos a float32, eso se puede hacer mediante np.array(puntos,np.float32), y tendremos que especificar un criterio para el parámetro criteria. Para esto, se ha utilizado el que hay en uno de los tutoriales de opencv https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_features_harris/py_features_harris.html .El resultado tras el refinamiento es el siguiente.



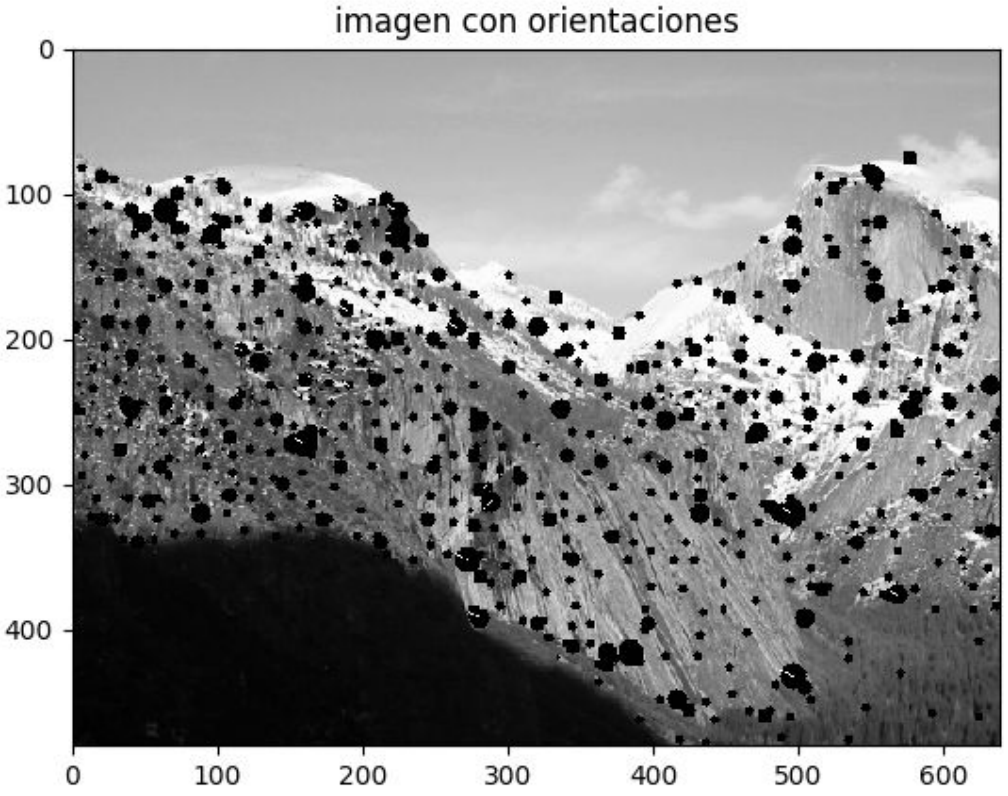
En este caso, no ha habido grandes cambios con respecto a la imagen original.

Apartado C.

Para calcular las orientaciones, calcularemos la orientación calculando la arcotangente de los gradientes de la imágen. Para calcular la arcotangente podemos utilizar la función **np.arctan2()**.

Para calcular los gradientes, podemos utilizar las funciones de la pasada práctica, para calcularlos utilizaremos un *sigma=4.5*. Una vez tenemos el gradiente en x y en y, obtenemos solamente los valores de la imagen en los puntos Harris, tras esto calculamos la orientación de cada uno de ellos. Todo este proceso se realiza en la función **calcularOrientación()**.

Tras esto, debemos dibujar la orientación con dentro de los puntos Harris, para ello, dentro de la función **dibujarPuntos()** debemos utilizar la función **cv2.line()** para dibujar una línea que vaya desde las coordenadas del punto hasta la periferia del círculo que hemos dibujado. Para ello, obtendremos el punto en la periferia sumándoles a las coordenadas del punto el desplazamiento en el eje x e y. El resultado es el siguiente.

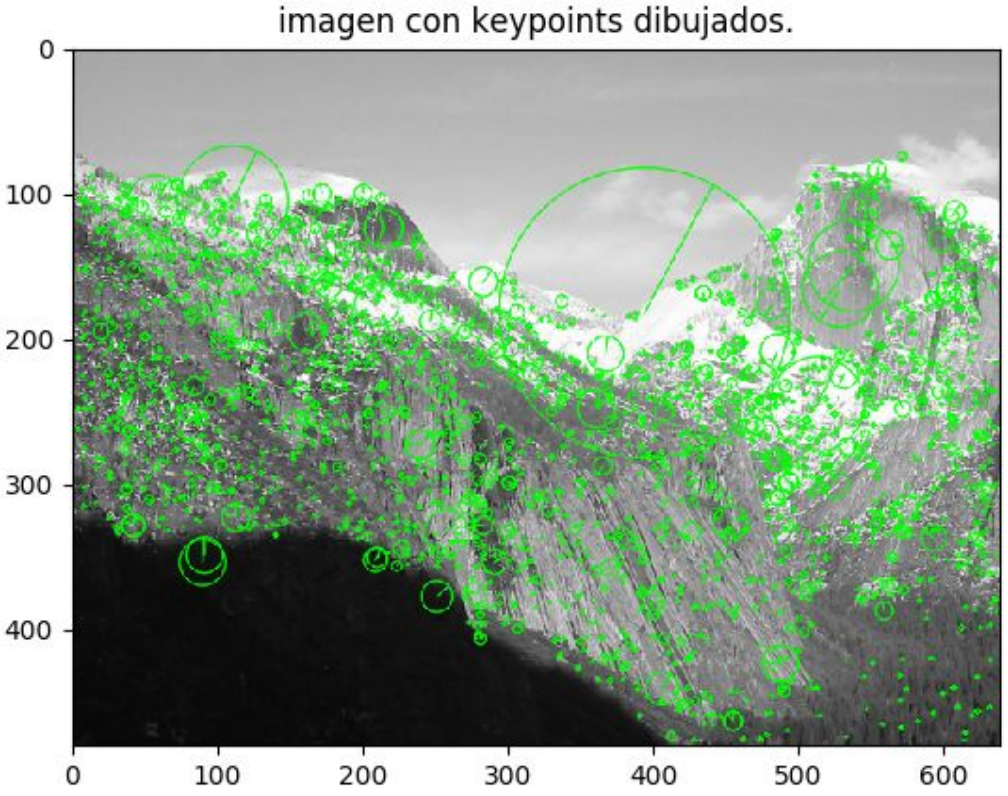


Apartado D.

Para este apartado, crearemos una función que cree objetos del tipo KeyPoint, para ello, solo necesitamos declarar un objeto del tipo *cv2.KeyPoint,* el cual contiene las coordenadas del punto, su escala, y su orientación. Esto se realiza en la función **createKeypoints()**.

Ahora calcularemos los descriptores de cada punto, para ello, utilizaremos la función **detectAndCompute()** de opencv. Antes de poder utilizar esta operación, debemos crear un objeto del tipo *SIFT*, para ello, debemos utilizar la función, **cv2.xfeatures2d.SIFT_create()**. Tras esto, podemos utilizar la función **sift.detectAndCompute()**, pasándole la imagen en blanco y negro; esta función nos devuelve sus propios keypoints, y los descriptores. Este proceso se realiza en la función **calculateDescriptors()**.

El resultado es el siguiente:



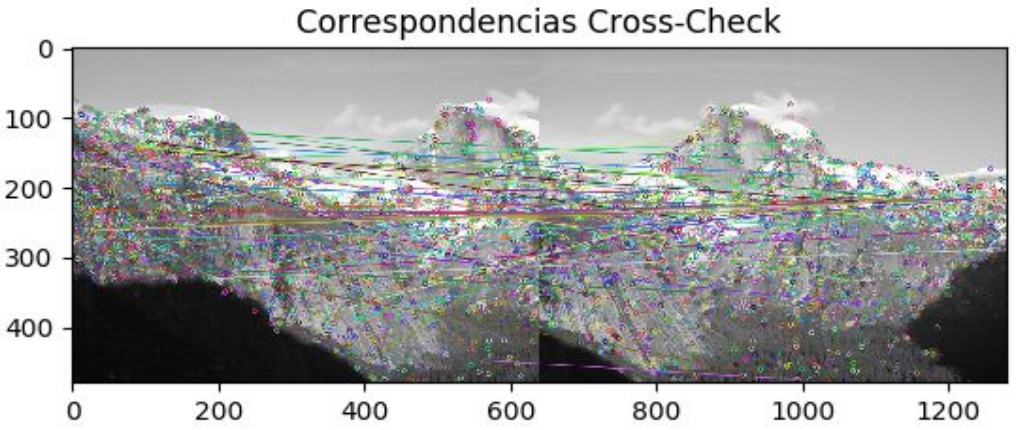
Como se puede ver, esta función devuelve un número mayor de keypoints que los calculados anteriormente, pero los calculados siguen estando ahí.

Ejercicio 2.

Para este apartado, se han creado dos funciones, una por cada tipo de criterio utilizados para seleccionar las correspondencias entre imágenes.

Para el criterio cross-check, se ha creado la función **correspondenciasCrossCheck()**; esta función recibe los descriptores y los keypoints de dos imágenes, con esto, crea un objeto del tipo *BFMatcher* y se ejecuta el método **match()** para calcular las correspondencias. Para el cálculo de las correspondencias, utilizamos la distancia euclídea (*normType=cv2.NORM_L2*) y seleccionamos ponemos el parámetro *croosCheck* a True.

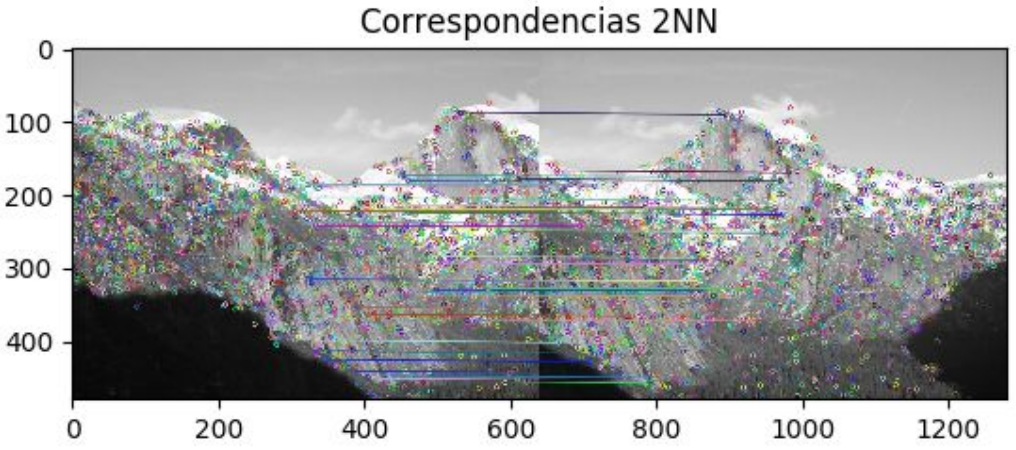
Una vez hemos calculados las correspondencias, elegimos solamente las 50 mejores, según la distancia que haya entre las correspondencias. El resultado que obtenemos es el siguiente:



Como se puede ver, algunas de las correspondencias son correctas; pero también hay casos donde confunde puntos que aparecen en una imagen y no en la otra, esto se debe a que ambas zonas son parecidas, aunque no sean iguales.

Para el segundo criterio, se ha creado la función **correspondencias2NN()**, para este algoritmo, también se ha creado un objeto del tipo *BFMatcher*, pero esta vez no utilizamos cross-check. Tras esto, utilizamos el método **knnMatch()**, pasándole los descriptores de ambas imágenes y el número de vecinos que se quiere analizar, en nuestro caso dos.

Una vez hemos obtenido las correspondencias, debemos ver cuál de ellas son realmente buenas, para ello recorremos las correspondencias obtenidas y solamente nos quedamos con aquellas cuya distancia sea a la derecha sea menor que la distancia a la izquierda multiplicada por una constante (en mi caso he utilizado *0.65* ya que es la que se utiliza en el paper que acompaña a la práctica). Tras quedarnos solamente con las correspondencias que cumplan este requisito, las ordenamos y nos quedamos con las 50 que menor distancia tengan. El resultado obtenido es el siguiente:



Para este criterio, se puede comprobar que hay muchas más correspondencias correctas que fallidas, por lo que es un mejor criterio a la hora de calcular correspondencias entre dos imágenes.

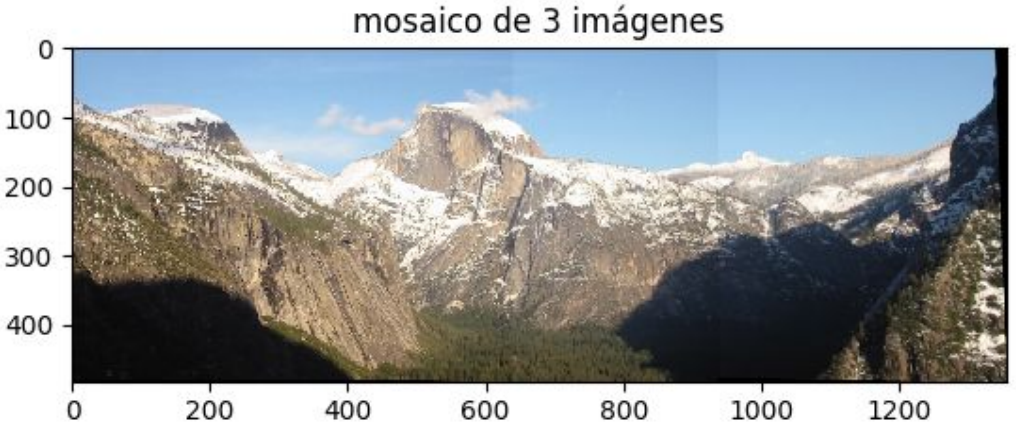
Ejercicio 3.

Para este ejercicio, se han creado tres funciones, una llamada **mosaico2Imgs()**, **mosaico3Imgs()** y **crop()**. Esta última función se utiliza para recortar el mosaico cuando ya está formado.

La función **mosaico2Imgs()** se encarga de crear un mosaico a partir de dos imágenes, dentro de esta, se toma la primera imagen como la imagen de referencia, a la cual le asignaremos como homografía la matriz identidad. Tras esto, se calculan las correspondencias entre ambas imágenes, y se calculan los puntos necesarios para obtener la homografía de la segunda imágen.

Una vez hemos calculado la homografía que hay en las dos imágenes, debemos calcular la homografía inversa, qué es la que nos interesa en este caso. Para ello utilizaremos la función de **numpy.linalg.inv()**. Por último, utilizamos la función **cv2.warpPerspective()** para obtener la imagen dentro del mosaico transformada.

Para tres imágenes, el proceso es el mismo, ya que utilizaremos la función **mosaico2Img()** para las dos primeras imágenes,y luego introduciremos las tercera con otra llamada a esta función. El resultado obtenido con la imágenes llamadas “yosemiteXX” es el siguiente:



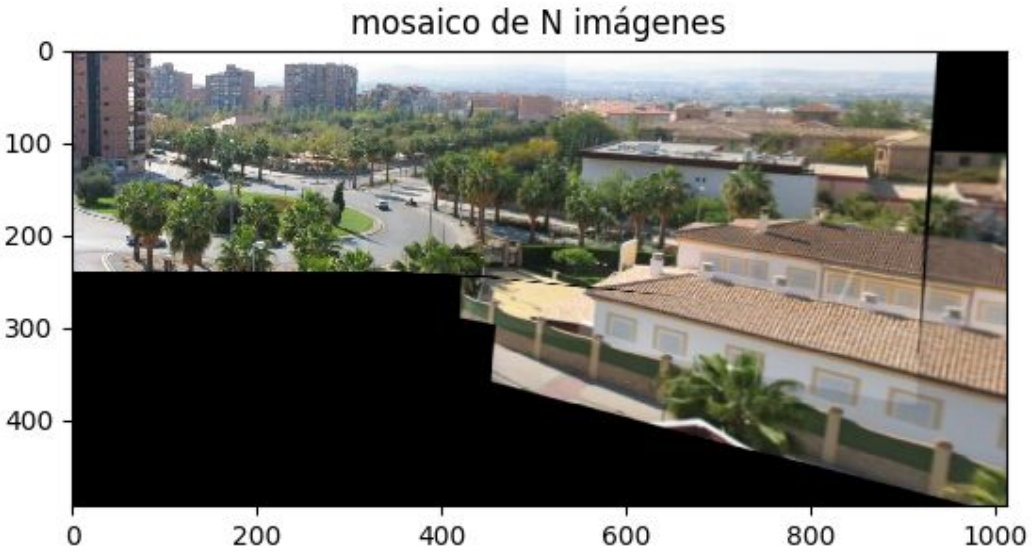
Como se puede ver, el resultado es bastante satisfactorio, el único problema visible es que existen cambios en el color del cielo. Aunque eso podría mejorarse aplicando filtros al cielo.

Ejercicio 4.

Para este ejercicio, se ha creado la función **mosaicoNImgs()**, que recibe como argumento un vector de imágenes.

Esta función comprueba si el tamaño del vector es dos ó tres, y si es el caso llama a las funciones anteriormente implementadas para crear el mosaico. Si el tamaño es mayor, el vector de imágenes se divide en dos y se vuelve a llamar a la función, hasta que el conjunto de imágenes sea de dos ó tres.

Para el siguiente ejemplo se han utilizado las imágenes llamadas “mosaicoXX”, los resultados son los siguientes:



En este caso, obtenemos peores resultado que en el apartado anterior, ya que se pueden ver cortes negros entre imágenes también hay ciertos cambios de color en el cielo y en la casa. El número de imágenes utilizadas para este ejemplo es N=10.