

*Visión por computador*  
*Trabajo 2*  
*Programación*



## Ejercicios

**1.- Estimación de una homografía:** Usar las imágenes tablero1 y tablero2 incluidas en el fichero de datos para estimar la homografía que relaciona ambas imágenes.

**1.- Seleccionar a mano un conjunto de 10 puntos en correspondencias en ambas imágenes (distribuir dichos puntos de la forma más uniforme posible entre los puntos esquina del tablero).**

Lo primero que he realizado tras cargar las imágenes ha sido seleccionar a mano 10 puntos en correspondencias de ambas imágenes de una forma uniforme. Los 10 puntos elegidos han sido: esquina superior izquierda, esquina superior derecha, esquina inferior izquierda, esquina inferior derecha, lateral izquierdo el cuarto cuadro, lateral derecho el cuarto cuadro, lateral superior el cuarto cuadro, lateral inferior el cuarto cuadro, de la tercera fila el sexto cuadro y de la sexta fila el séptimo cuadro.

**2.- Estimar la homografía (implementar código propio, SIN usar findHomography()) definida entre las imágenes por las parejas de puntos seleccionados**

Una vez seleccionados los 10 puntos en correspondencias del apartado anterior, esos vectores de “Point2d” los he pasado a la función encontrarHomografía.

Esta función encontrar homografía devolverá la matriz de la homografía. Para calcular esta, lo que hago es primero formar la matriz A a partir de las parejas de puntos de los vectores de puntos de correspondencias. La matriz tendrá la siguiente forma:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 & -x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1x_1 & -y'_1y_1 & -y'_1 \\ & & & & & : & & & \\ x_n & y_n & 1 & 0 & 0 & 0 & -x'_nx_n & -x'_ny_n & -x'_n \\ 0 & 0 & 0 & x_n & y_n & 1 & -y'_nx_n & -y'_ny_n & -y'_n \end{bmatrix}$$

**A**  
2n × 9

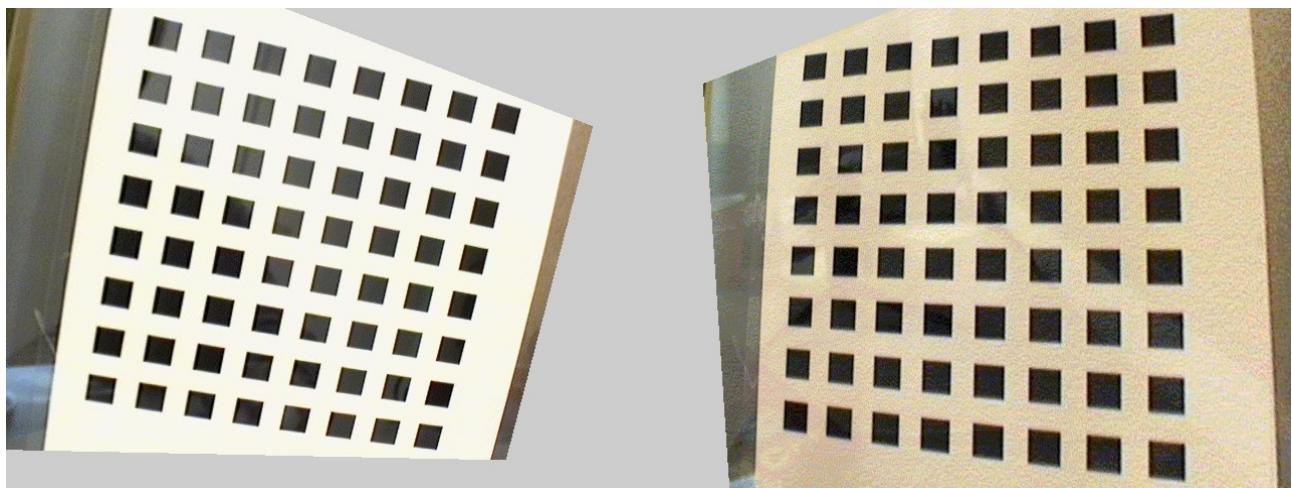
Por lo tanto, una vez montada esta, el siguiente paso a realizar va a ser llamar a la función SVDecomp con la matriz A, y nos devolverá las matrices W, U y Vtraspuesta.

Por lo tanto para montar la homografía como ya se ha estudiado, necesitamos los valores de Vtraspuesta de la última fila. De esta forma con esos 9 valores montamos la homografía que será lo que devolverá nuestra función encontrarHomografía.

**3.-Usar la función warpPerspective() para generar a partir de una de las imágenes y la homografía estimada, la otra imagen. Comentar el resultado obtenido en comparación con la imagen original.**

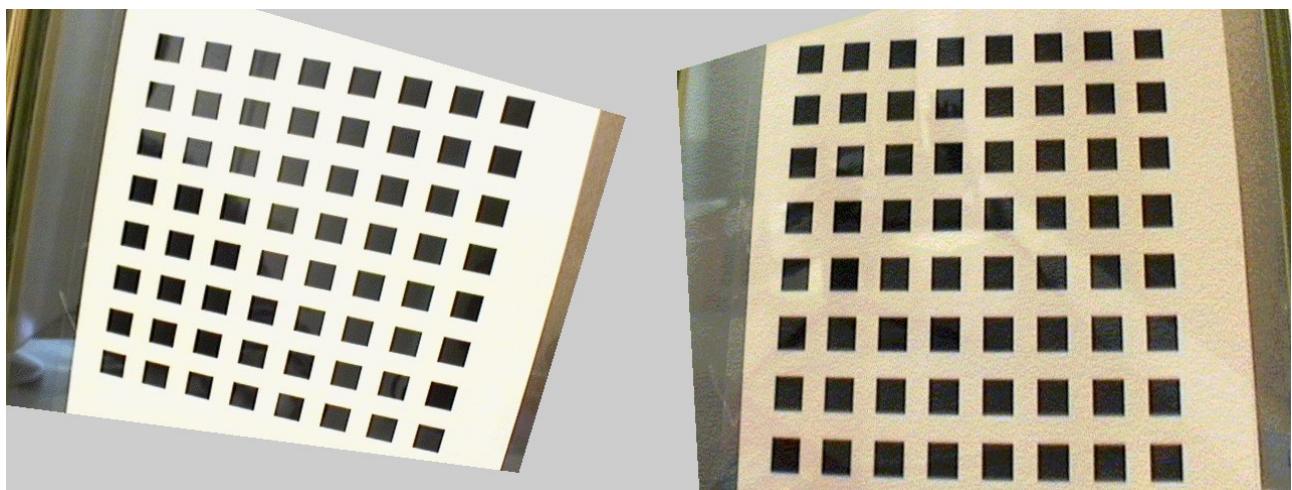
Con la homografía obtenida en el apartado anterior voy a aplicarla sobre una de las imágenes para comprobar que su cálculo ha sido el correcto. Para ello uso la función warpPerspective pasando como argumentos principales, la imagen a convertir, la salida, y la homografía (además de otros).

Como resultado de aplicar las homografías correspondientes a los dos tableros, es decir, sacar las homografías de una respecto de la otra y viceversa, tenemos lo siguiente:



Este es el resultado de aplicar nuestro método de encontrarHomografía.

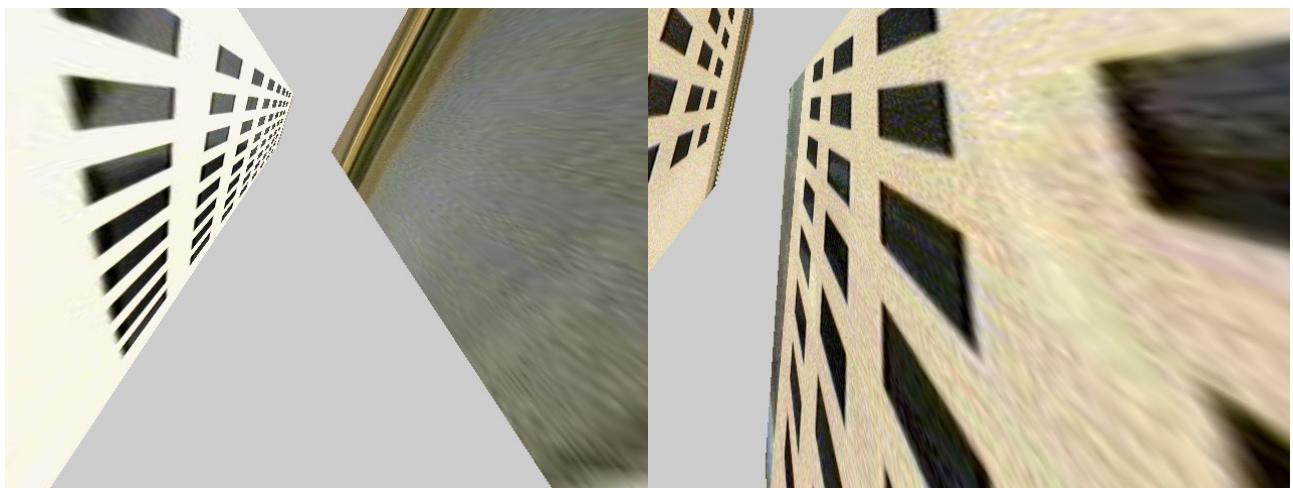
Para comprobar que no se ha realizado incorrectamente, he aplicado la función de OPENCV findHomography con los mismo puntos para ver el resultado que se obtiene con ella, resultando:



Se puede observar que el resultado obtenido es el mismo o muy parecido, por lo que nuestra función de encontrar Homografía es buena.

**4.- Repetir los puntos anteriores pero seleccionando nuevos 10 puntos, en este caso todos extraídos de 3 cuadrados contiguos de una misma esquina. Comparar el resultado obtenidos con el anterior y valorar las diferencias encontradas.**

Para la realización de este ejercicio he seleccionado otros 10 puntos de la esquina superior derecha y he aplicado lo mismo que en los apartados anteriores. Además para este apartado también he usado como modo de comparar, la función findHomography. A continuación muestro los resultados:



Este ha sido el resultado de aplicar mi función `encuentraHomografía` el cual se puede justificar ya que los puntos no son dispersos alrededor de toda la imagen.



Y este ha sido el resultado que nos proporciona `findHomography`, que se puede apreciar que es mejor que el proporcionado por nuestro método. La justificación como he indicado anteriormente, podría ser que `findHomography` realiza otros procesos diferentes pero tampoco obtiene un resultado tan bueno como cuando los puntos son por toda la imagen.

**2.- Usar los detectores BRIST y ORB de OpenCV sobre las imágenes de Yosemite para extraer y pintar sobre las mismas las regiones relevantes encontradas por los mismos (mirar el contenido de la estructura keyPoint asociada a cada punto detectado). Comenzar usando los parámetros por defecto de los constructores de SIFT y SURF e ir modificando los umbrales de detección hasta que obtengamos resultados razonables). Comparar los resultados obtenidos por ambos detectores y justificar los parámetros usados**

Como se aprecia en la imagen, podemos ver cuales son los parámetros de cada detector, valores que los he puesto con los parámetros por defecto que nos proporcionaría cada detector en caso de no pasarle parámetros algunos.

```
//Parámetros para el detector BRISK (por defecto)
int thresh = 30;
int octaves = 3;
float patternScale = 1.0f;

//Parámetros para el detector ORB (por defecto)
int nfeatures = 500;
float scaleFactor = 1.200000048F;
int nlevels = 8, edgeThreshold = 8, firstLevel = 0, WTA_K = 2, scoreType = 0;
int patchSize = 31, fastThreshold = 20;

//0 ambos, 1 BRISK, 2 ORB
int detector = 0;
Ejercicio2Trabajo2(imagenes_yosemite[0], imagenes_yosemite[1], 0, thresh, octaves,
    patternScale, nfeatures, scaleFactor, nlevels, edgeThreshold, firstLevel, WTA_K,
    scoreType, patchSize, fastThreshold);
```

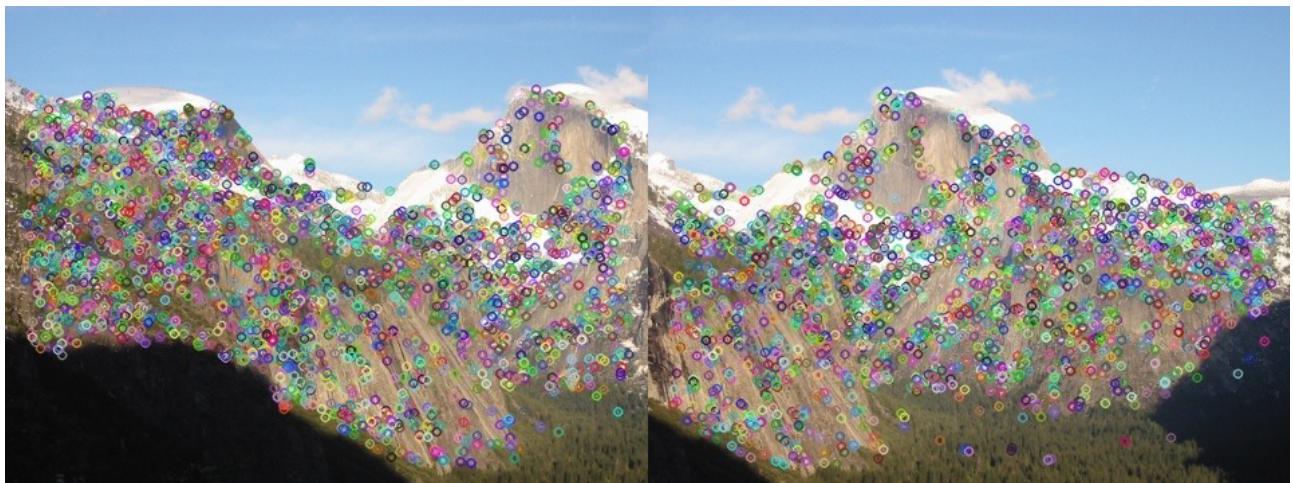
De esta forma lo que realizo a continuación es una llamada a las funciones que me usan estos detectores sobre las imágenes deseadas de forma que los pasos que hacen estas funciones son:

```
//Creamos el detector con los parámetros deseados
Ptr<FeatureDetector> BRISKD = BRISK::create(nthresh, noctaves, npatternScale);
//Hacemos la detección de los puntos clave en la imagen
BRISKD->detect(img, keypoints);
//Sacamos los descriptores
BRISKD->compute(img, keypoints, descriptors);
//Pintamos los keypoints en la imagen y los pasamos a brisk
drawKeypoints(img, keypoints, brisk);
```

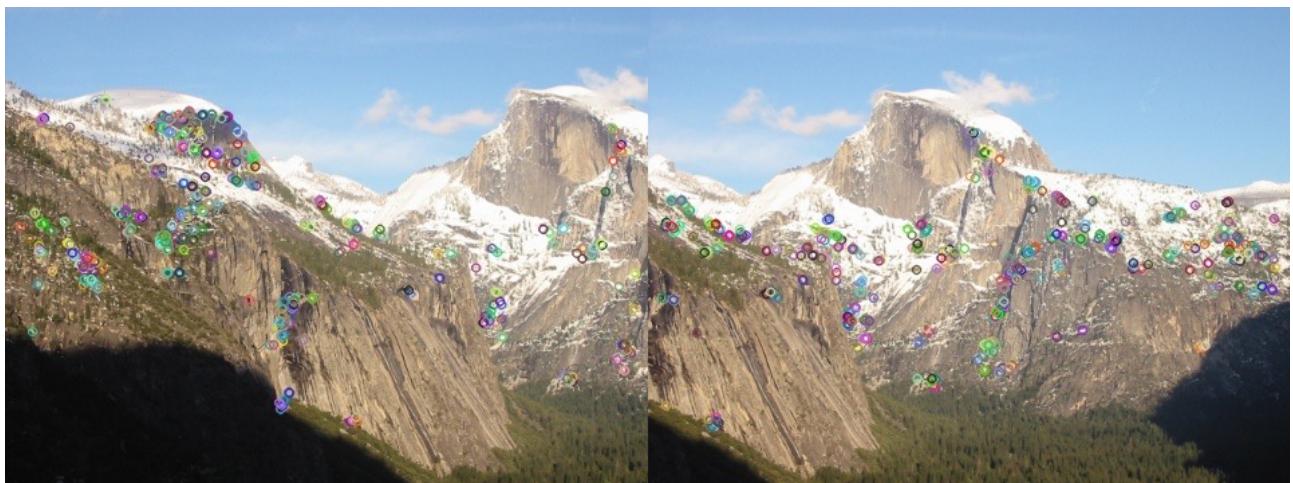
```
//Creamos el detector con los parámetros deseados
Ptr<FeatureDetector> ORBD = ORB::create(nnfeatures, nscaleFactor, nnlevels,
    nedgeThreshold, nfirstLevel, nWTA_K, nscoreType, npatchSize, nfastThreshold);
//Hacemos la detección de los puntos clave en la imagen
ORBD->detect(img, keypoints);
//Sacamos los descriptores
ORBD->compute(img, keypoints, descriptors);
//Pintamos los keypoints en la imagen y los pasamos a orb
drawKeypoints(img, keypoints, orb);
```

Se puede ver que la diferencia entre ambos es los parámetros que les pasamos, ya que se usarían de forma similar.

Por lo tanto lo que realizamos es crear el detector con los parámetros deseados, realizar la detección de los puntos clave de la imagen, guardando el vector con esos puntos clave y seguidamente el computo para obtener los descriptores de la imagen. Por último y para mostrarlos en las siguientes imágenes, será pintar los puntos clave sobre la imagen. De esta forma los resultados obtenidos son:



Para el detector BRISK con los parámetros por defecto.



Y aquí el resultado del detector ORB con los parámetros por defecto.

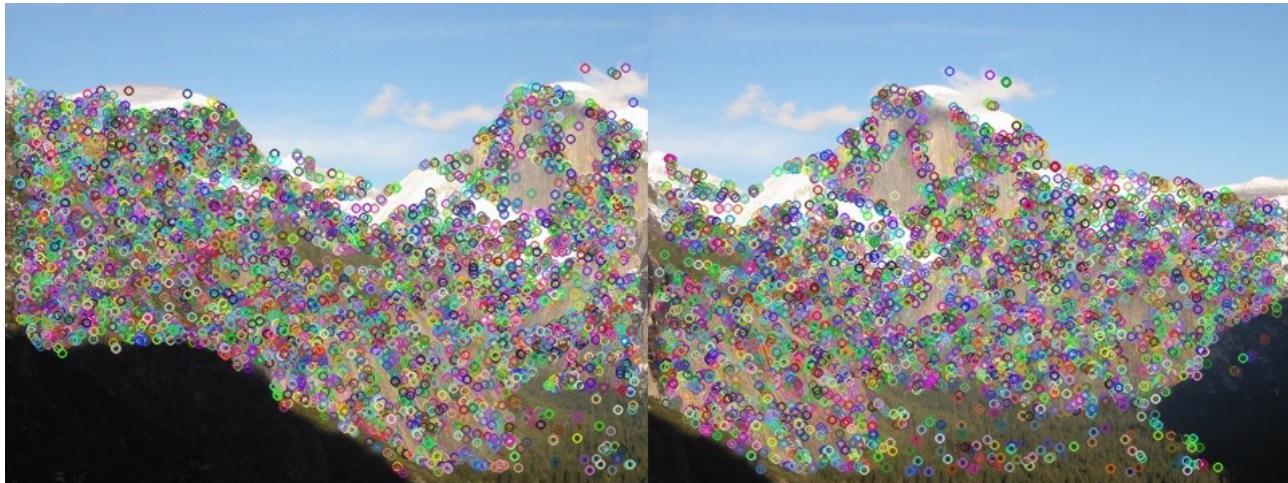
```
Ejecutando Ejercicio 2 Trabajo 2
BRISK:Se han encontrado 2984 keypoints y 2615 keypoints
ORB:Se han encontrado 500 keypoints y 500 keypoints
Fin Ejecucion Ejercicio 2 Trabajo 2
```

Se puede ver que el número de puntos clave encontrados por cada detector es bastante distinto.

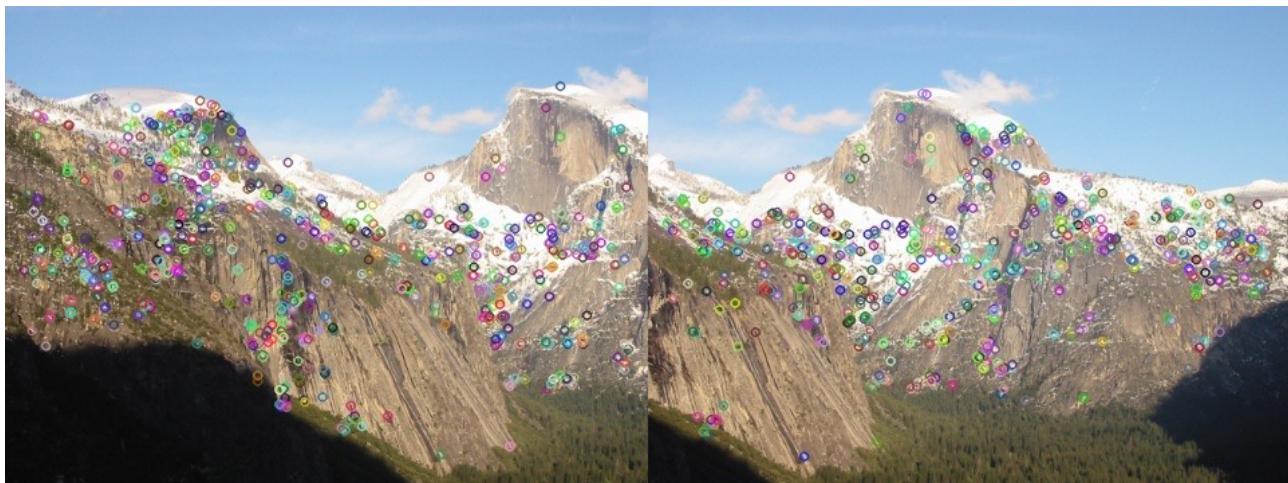
Ahora probaré a realizar cambios en los parámetros por defecto de ambos detectores para ver el comportamiento de estos.

```
Ejercicio2Trabajo2(imagenes_yosemite[0], imagenes_yosemite[1], 0, 40, 4, 1.5f, 300,
1.000000048F, 6, 6, -2, 2, -1, 25, 15);
```

De esta forma he modificado los parámetros de ambos detectores, y el resultado obtenido será:



Para el detector BRISK.



Para el detector ORB.

```
Ejecutando Ejercicio 2 Trabajo 2
BRISK:Se han encontrado 5234 keypoints y 4705 keypoints
ORB:Se han encontrado 662 keypoints y 653 keypoints
Fin Ejecucion Ejercicio 2 Trabajo 2
```

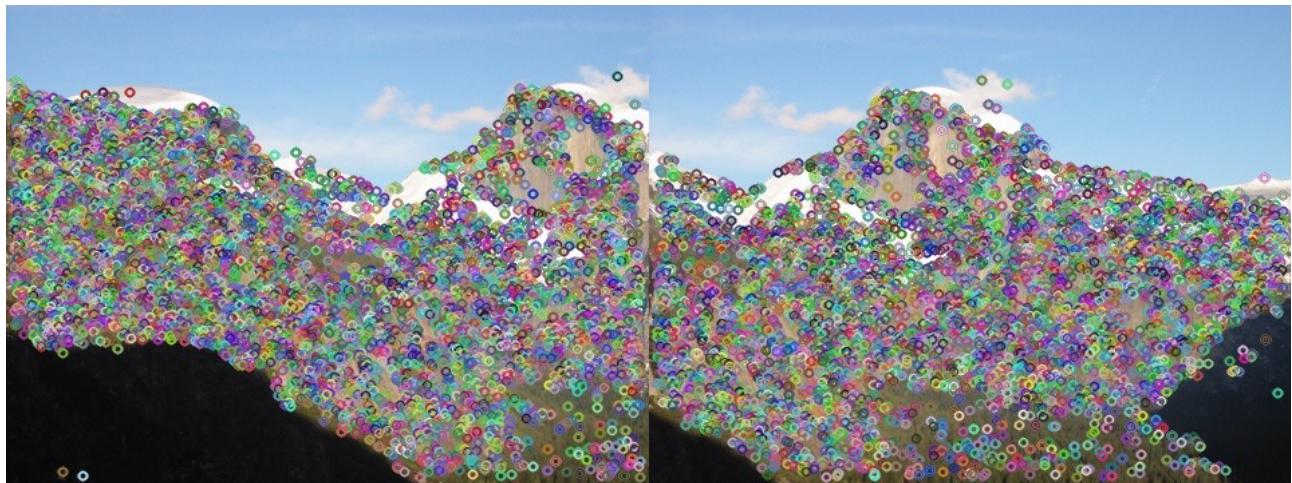
Se puede apreciar como con estos parámetros lo que hemos conseguido es un aumento de detección de puntos clave para ambos detectores.

Otra prueba realizada ha sido con los parámetros:

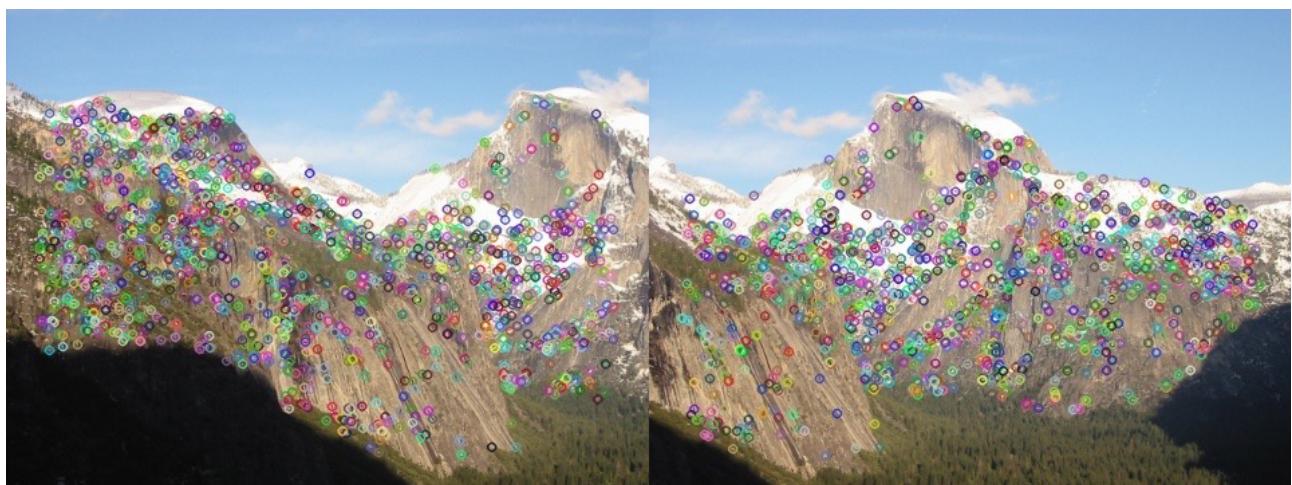
```
Ejercicio2Trabajo2(imagenes_yosemite[0], imagenes_yosemite[1], 0, 20, 2, 0.5f, 700,
1.400000048F, 10, 10, -1, 3, 1, 35, 25);
```

De forma que si en el anterior ejemplo, se aumentaba un poco cada parámetro, ahora lo que se hace es disminuirlos comparados con los de por defecto.

El resultado obtenido será:



Para el detector BRISK.



Para el detector ORB.

```
Ejecutando Ejercicio 2 Trabajo 2
BRISK:Se han encontrado 1596 keypoints y 1372 keypoints
ORB:Se han encontrado 310 keypoints y 313 keypoints
Fin Ejecucion Ejercicio 2 Trabajo 2
```

Se puede apreciar como ha disminuido considerablemente el número de puntos clave encontrado por ambos detectores con estos parámetros.

Por lo tanto se podría concluir con que en función del número de puntos clave que nos gustaría obtener, se deberían variar los parámetros de estos detectores.

**3.- Con los resultados de detección obtenidos en el punto anterior extraer el descriptor asociado a cada punto y establecer la lista de puntos en correspondencias existentes en las imágenes ( Ayuda: usar la clase DescriptorMatcher y BFMatcher). Valorar la calidad de los resultados obtenidos bajo los criterios BruteForce+crossCheck y FlannBasedMatcher().**

Como parámetro de este ejercicio tengo “factor\_correspondencia” que más adelante explicaré en que consiste en mis pruebas.

Lo que voy a hacer en este ejercicio es una vez leída las imágenes y realizado cualquiera de los dos detectores del ejercicio anterior, será llamar a mi función PuntosEnCorrespondencia, con las dos imágenes, los descriptores y keypoints obtenidos por los detectores, un booleano llamado crossCheck para indicar si queremos que se haga por fuerza bruta o validación cruzada, la imagen de salida que devolverá el método, un vector para devolver las correspondencias “buenas”, y el factor\_correspondencia ya nombrado.

Una vez llamada a dicha función, la función realiza lo siguiente:

```

vector<DMatch> matches;

//Realizamos la búsqueda de las correspondencias
BFMatcher matcher = BFMatcher::BFMatcher(NORM_L2, crossCheck);
matcher.match(descriptors1, descriptors2, matches);

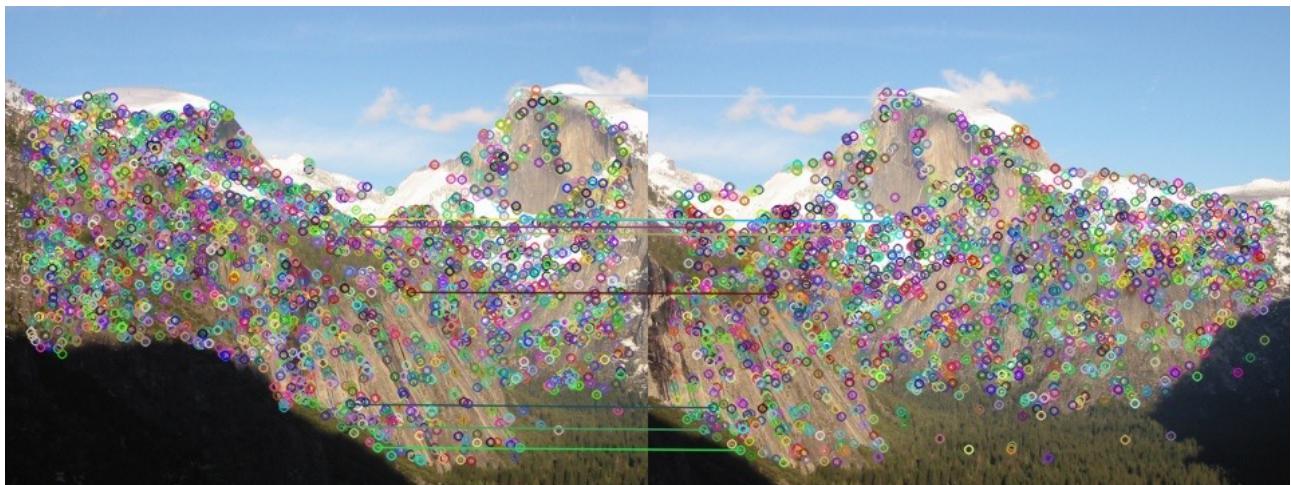
float min_dist;
//Lo ordeno para que sea mas eficiente
sort(matches.begin(), matches.end());
min_dist = matches[0].distance;
//Buscamos las mejores correspondencias
for (int i = 0; i < matches.size(); i++){
    if (matches[i].distance <= factor_correspondencia * min_dist){
        matchesbuenos.push_back(matches[i]);
    }
    else {
        //en el caso de que no consiga un mínimo de 4 correspondencias if
        if (matchesbuenos.size() < 4){
            //aumento el factor de correspondencia
            factor_correspondencia++;
            i--;
        }
        else {
            break;
        }
    }
}
cout << "He pasado de " << matches.size() << " correspondencias a "
<< matchesbuenos.size() << " Valor final de factor_correspondencia: "
<< factor_correspondencia << endl;

//Pintamos los puntos de correspondencia
drawMatches(img1, keypoints1, img2, keypoints2, matchesbuenos, salida);

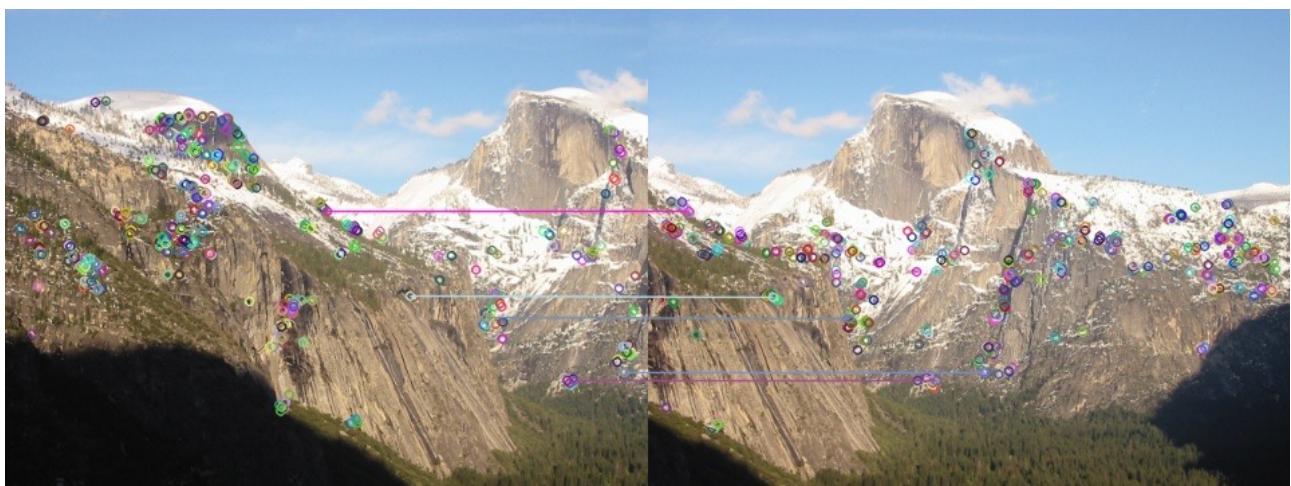
```

Es decir, primero con ayuda de BFMatcher sacamos las correspondencias de ambos descriptores. Seguidamente ordeno ese vector con las correspondencias. Saco del primero la mínima distancia, que va a ser el parámetro que use para seleccionar las correspondencias más prometedoras. De esta forma recorro todas las correspondencias mientras que se cumpla la condición de que la mínima distancia y el factor de correspondencia permita seleccionar correspondencias. De esta forma las correspondencias que tengan una distancia muy lejana las descartaremos por intuir que serán malas. En el caso en el que se vaya a dejar de buscar correspondencias pero no tengamos un mínimo de 4

correspondencias “buenas” se aumentará el factor de correspondencia para permitir realizar mosaicos con el mínimo de 4 puntos. Por último lo que voy a hacer es pintar las correspondencias en una imagen de salida que mostrará que correspondencias se han deducido en ambas imágenes. De esta forma podemos obtener con un factor de correspondencia de 1 y crossCheck activado:



Esta imagen corresponde con el detector BRISK



Esta imagen corresponde con el detector ORB

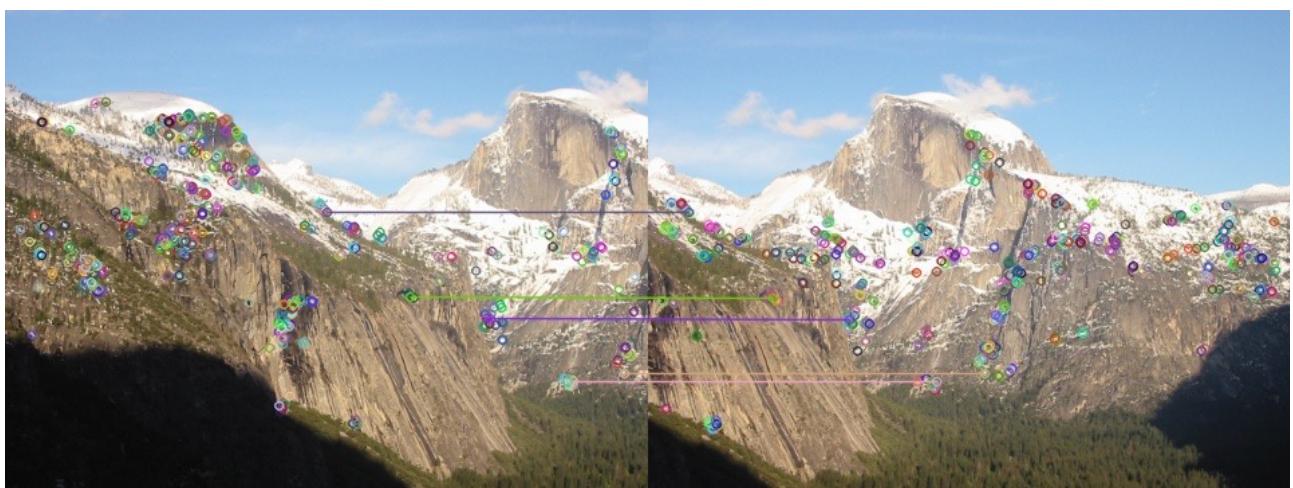
```
Ejecutando Ejercicio 3 Trabajo 2
He pasado de 1640 correspondencias a 9 Valor final de factor_correspondencia: 2
He pasado de 275 correspondencias a 5 Valor final de factor_correspondencia: 13
Fin Ejecucion Ejercicio 3 Trabajo 2
```

Se puede ver como el número de correspondencias buenas ha sido 9 para BRISK y 5 para ORB, teniendo que aumentar el factor de correspondencia a 2 y 13 respectivamente para conseguir el mínimo de 4 exigido.

Otra ejecución ha sido con un factor de correspondencia de 10, y crossCheck activado, consiguiendo:



Para el detector BRISK.



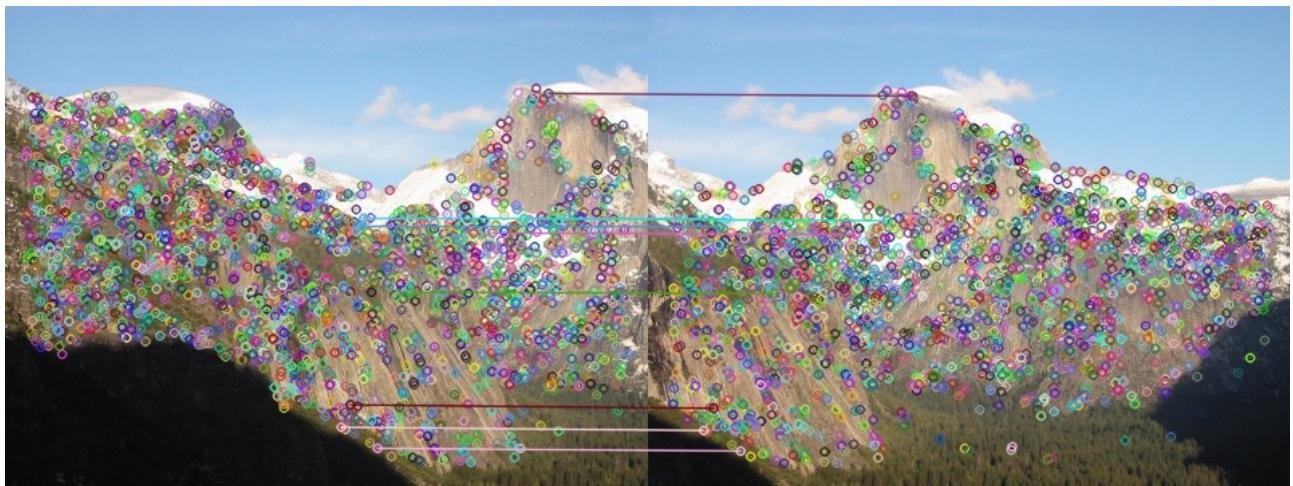
Para el detector ORB.

Y con la salida:

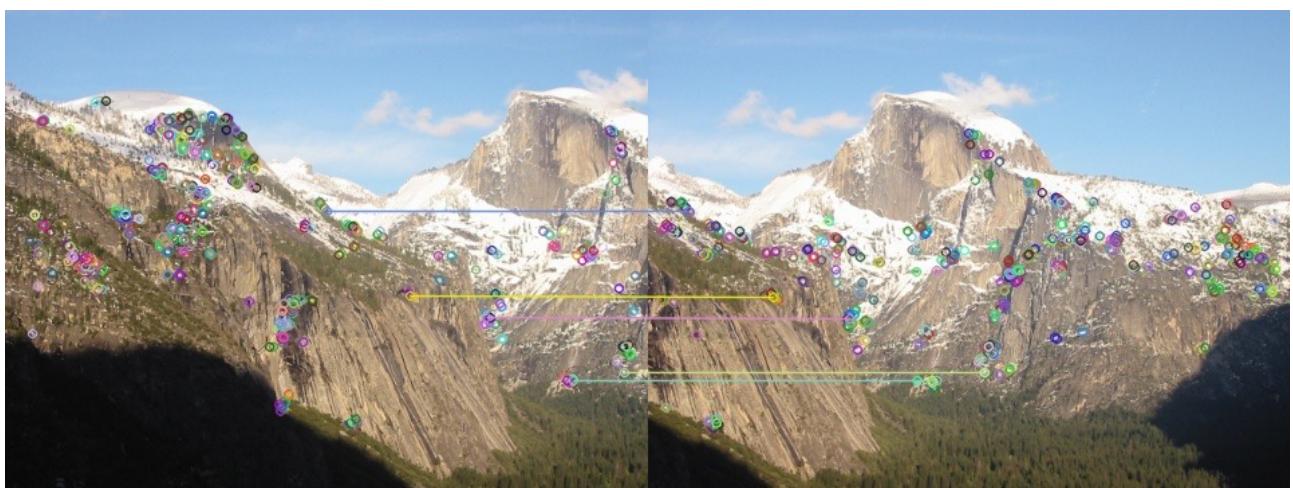
```
Ejecutando Ejercicio 3 Trabajo 2
He pasado de 1640 correspondencias a 553 Valor final de factor_correspondencia:
10
He pasado de 275 correspondencias a 5 Valor final de factor_correspondencia: 13
Fin Ejecucion Ejercicio 3 Trabajo 2
```

de forma que tenemos 553 y 5 correspondencias para BRISK y ORB respectivamente, junto con un factor de correspondencia que para el caso de ORB ha tenido que ser aumentado hasta 13, para llegar a nuestro mínimo establecido.

Para un factor de correspondencia de 1 y crossCheck desactivado, obtenemos:



Para el detector BRISK.



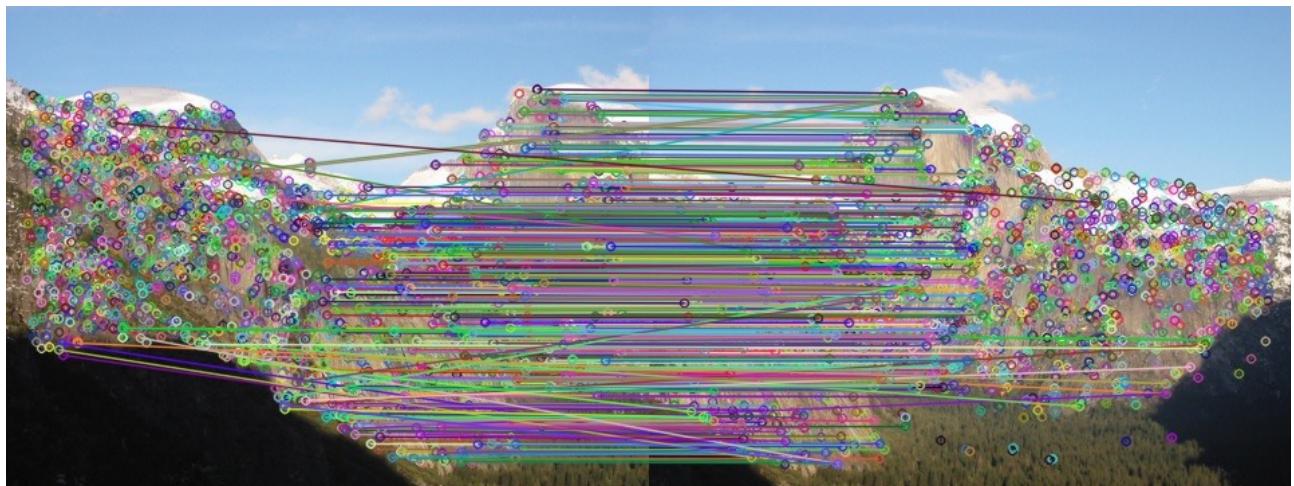
Para el detector ORB.

Con una salida:

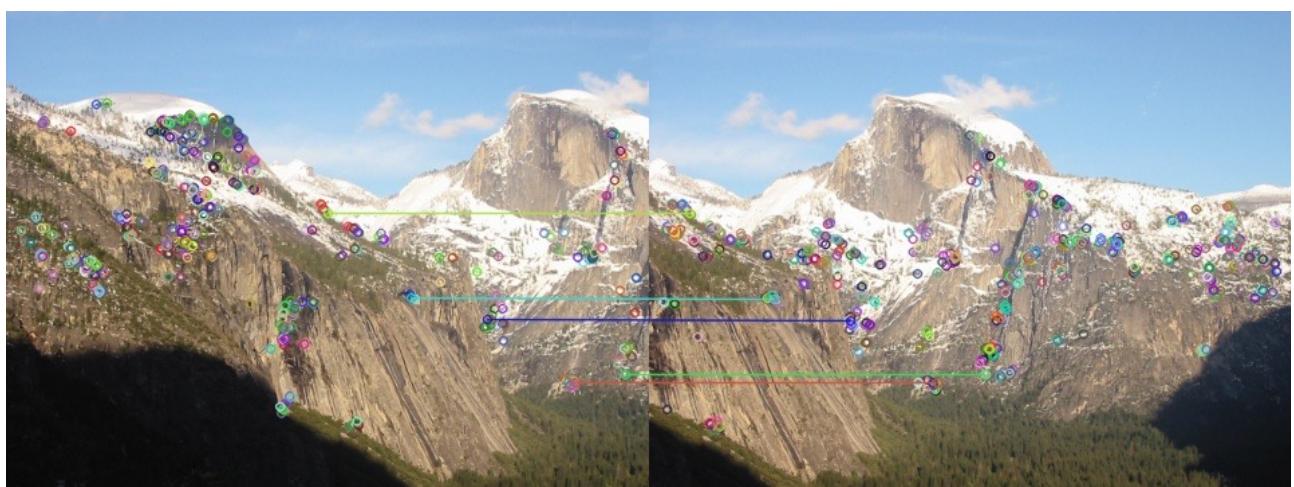
```
Ejecutando Ejercicio 3 Trabajo 2
He pasado de 2984 correspondencias a 9 Valor final de factor_correspondencia: 2
He pasado de 500 correspondencias a 5 Valor final de factor_correspondencia: 13
Fin Ejecucion Ejercicio 3 Trabajo 2
```

De forma que hemos obtenido 9 y 5 correspondencias para BRISK y ORB, con un factor de correspondencia de 2 y 13 respectivamente. Este resultado es igual al obtenido con crossCheck activado.

Ahora para un factor de correspondencia de 10 y crossCheck desactivado obtenemos:



Para el detector BRISK.



Para el detector ORB.

Y una salida:

```
Ejecutando Ejercicio 3 Trabajo 2
He pasado de 2984 correspondencias a 601 Valor final de factor_correspondencia:
10
He pasado de 500 correspondencias a 5 Valor final de factor_correspondencia: 13
Fin Ejecucion Ejercicio 3 Trabajo 2
```

De forma que el número de correspondencias ha sido 601 y 5 para el detector BRISK y ORB.  
En este caso hemos obtenido más correspondencias, para BRISK, que con crossCheck activado.

4.- Escribir una función que tome como entrada dos imágenes relacionadas por una homografía, sus listas de keypoints en correspondencia calculados de acuerdo al punto anterior y estime la homografía entre ellas usando el criterio RANSAC con la función `findHomography(p1,p2,CV_RANSAC,1)`. Crear un mosaico con ambas imágenes. ( Ayuda: 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 (ayuda: usar `borderMode=BORDER_TRANSPARENT`)).

Los parámetros usados para este ejercicio son el factor de correspondencia, comentado en el punto anterior, con valor a 1 y rows de la pantalla junto a cols de la pantalla, para crear la salida del mosaico con unas dimensiones lo suficientemente grandes.

Por lo tanto, lo primero que realizo será preparar el mosaico, es decir voy a inicializar el mosaico a las dimensiones pasadas por parámetros. Una vez creado el “contenedor”, voy a añadir una de las imágenes al mosaico con la función `warpPerspective`. Apuntar que la Homografía que le paso a esta función, en este punto del programa va a ser para trasladar la imagen colocada.

Cuando ya he introducido la primera imagen en el mosaico llamaré a la función “`crearMosaico`” a la que le pasaré la siguiente imagen junto al mosaico y al factor de correspondencia deseado. En la función de `crearMosaico` voy a sacar los keypoints de ambas imágenes, usando para esto, por ejemplo, el detector BRSIK. Después de los keypoints, voy a sacar los puntosEnCorrespondencias con la función creada en el ejercicio anterior.

Una vez tenemos las correspondencias “buenas” voy a sacar los keypoints de esas correspondencias, para realizar, seguidamente, la homografía de ambos keypoints. Para finalizar, realizo `warpPerspective` con la homografía encontrada para montar nuestro mosaico.

La ejecución es:

```
Ejecutando Ejercicio 4 Trabajo 2
He pasado de 1718 correspondencias a 37 Valor final de factor_correspondencia: 2
Total de correspondencias buenas para las dos imagenes: 37
Fin Ejecucion Ejercicio 4 Trabajo 2
```

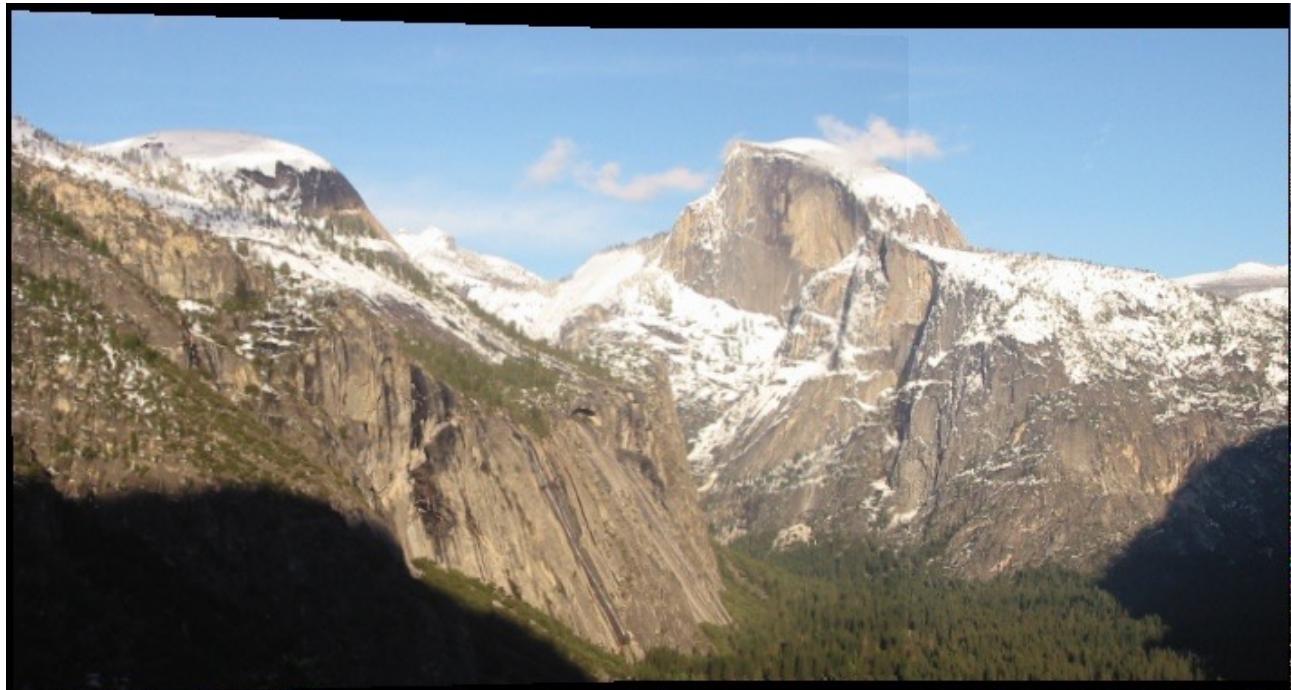
Podemos observar que el número de correspondencias “buenas” es 37 y que se ha aumentado el factor de correspondencia a 2.

El resultado obtenido es:



Una modificación de última hora añadida a mi código ha sido la de recortar las zonas negras. Lo primero que hago es buscar la última fila y la última columna que tienen puntos con color y recorto el resto. Lo mismo hago con la primera fila y primera columna, y recorto.

El resultado por tanto es:



**5.- Generar un mosaico de proyección plana a partir de múltiples imágenes. Estimar las homografías entre imágenes consecutivas y construir el mosaico a partir de ellas. La calidad del registrado geométrico y la menor deformación posible son los objetivos a alcanzar. (No está permitido usar los recursos del módulo stitching de OpenCV).**

Como en el ejercicio anterior, los parámetros para crear el mosaico serán los rows y cols de mi pantalla y el factor de correspondencia.

Primero voy a mostrar las imágenes que van a pertenecer al mosaico que voy a crear. Después voy a llamar a mi función crearSuperMosaico con el vector de imágenes.

Dentro de esta función, lo primero que voy a realizar va a ser colocar la imagen central en el mosaico con la función prepararMosaico comentada en el ejercicio anterior.

Una vez tengamos la imagen colocada en el centro del mosaico voy a recorrer las imágenes que irán a la izquierda de la imagen ya centrada. Voy a ir llamando a la función crearMosaico, comentada en el ejercicio anterior, pasando como parámetros el mosaico actual y la imagen a añadir a este mosaico actual.

La siguiente etapa va a ser recorrer las imágenes pertenecientes a la derecha de la imagen central para hacer lo mismo que con el lado contrario.

De esta forma, ya tendré el mosaico creado.

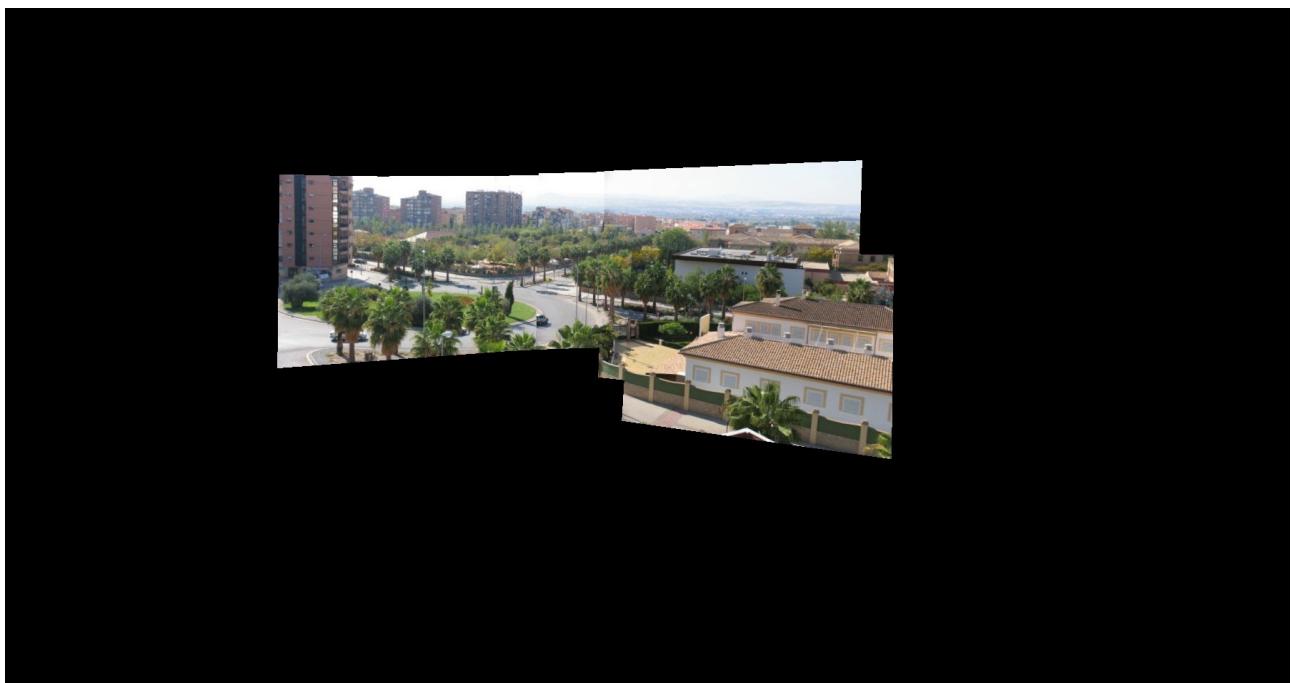
Como resultado tenemos:



Primera tanda de imágenes pertenecientes al mosaico.



Segunda tanda de imágenes pertenecientes al mosaico.

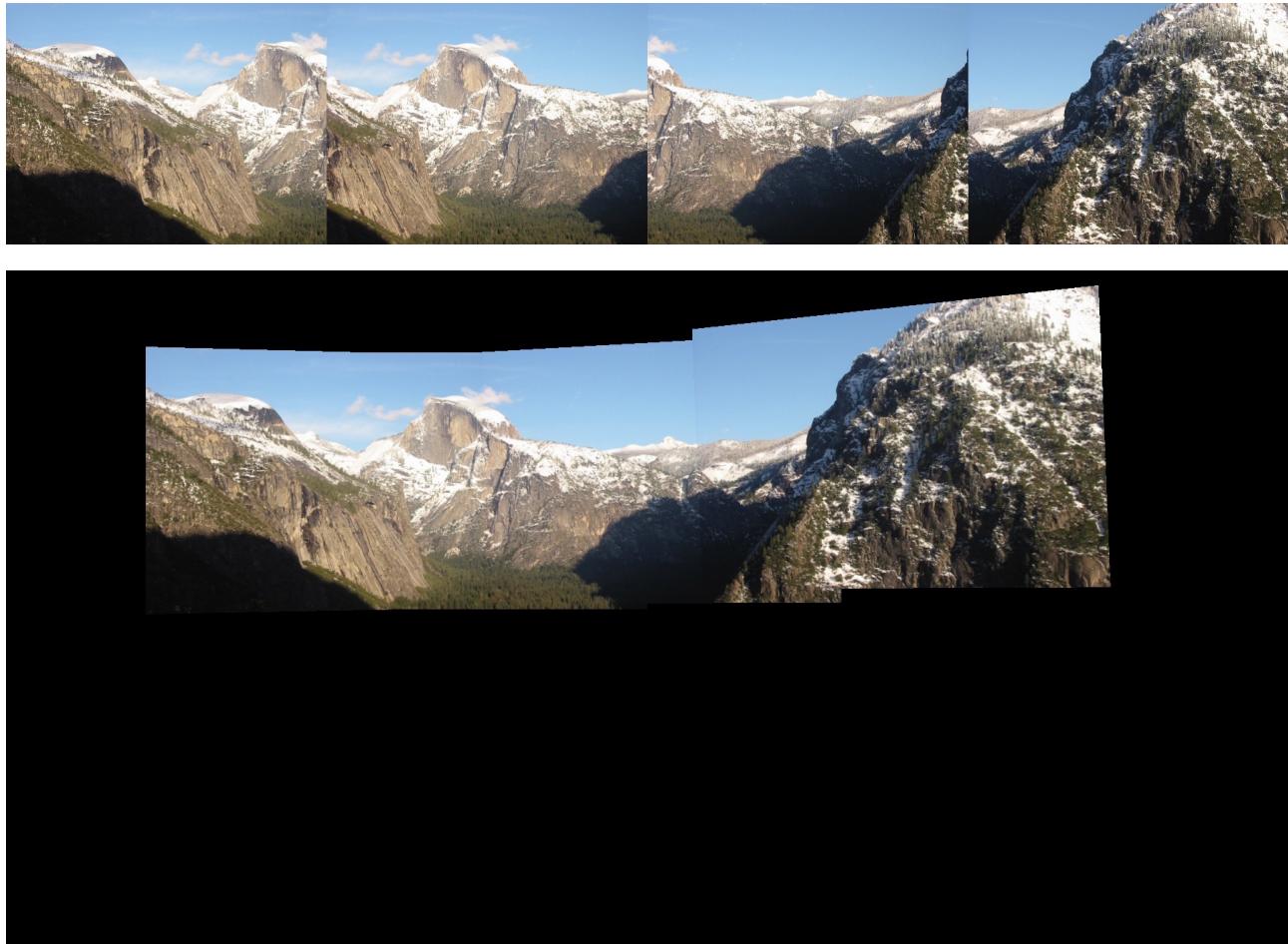


Donde se puede observar como se construye el mosaico correctamente. \*\*RECORTADO ABAJO

La ejecución nos muestra lo siguiente:

```
Ejecutando Ejercicio 5 Trabajo 2
En total hay 10 imagenes.
Imagen 5 aniadida.
He pasado de 819 correspondencias a 19 Valor final de factor_correspondencia: 4
Total de correspondencias buenas para las dos imagenes: 19
Imagen 4 aniadida.
He pasado de 780 correspondencias a 6 Valor final de factor_correspondencia: 8
Total de correspondencias buenas para las dos imagenes: 6
Imagen 3 aniadida.
He pasado de 801 correspondencias a 215 Valor final de factor_correspondencia: 4
Total de correspondencias buenas para las dos imagenes: 215
Imagen 2 aniadida.
He pasado de 896 correspondencias a 204 Valor final de factor_correspondencia: 4
Total de correspondencias buenas para las dos imagenes: 204
Imagen 1 aniadida.
4
He pasado de 827 correspondencias a 64 Valor final de factor_correspondencia: 4
Total de correspondencias buenas para las dos imagenes: 64
Imagen 6 aniadida.
He pasado de 852 correspondencias a 37 Valor final de factor_correspondencia: 4
Total de correspondencias buenas para las dos imagenes: 37
Imagen 7 aniadida.
He pasado de 852 correspondencias a 14 Valor final de factor_correspondencia: 4
Total de correspondencias buenas para las dos imagenes: 14
Imagen 8 aniadida.
He pasado de 1037 correspondencias a 114 Valor final de factor_correspondencia:
4
Total de correspondencias buenas para las dos imagenes: 114
Imagen 9 aniadida.
He pasado de 956 correspondencias a 58 Valor final de factor_correspondencia: 4
Total de correspondencias buenas para las dos imagenes: 58
Imagen 10 aniadida.
5
Fin Ejecucion Ejercicio 5 Trabajo 2
```

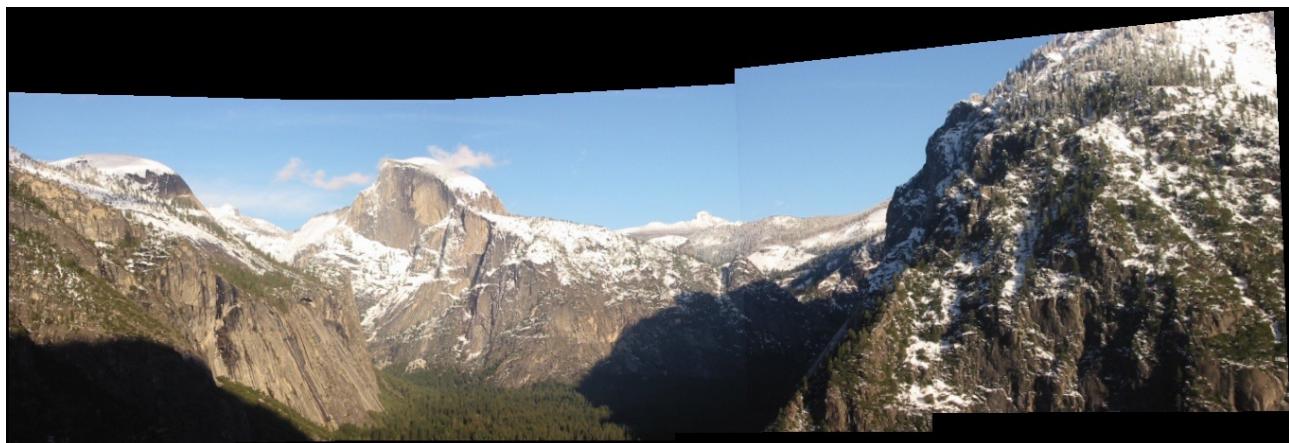
Otro ejemplo de construcción de mosaico es:



```
Ejecutando Ejercicio 5 Trabajo 2
En total hay 4 imagenes.
Imagen 2 aniadida.
He pasado de 1718 correspondencias a 367 Valor final de factor_correspondencia:
4
Total de correspondencias buenas para las dos imagenes: 367
Imagen 1 aniadida.
1
He pasado de 1640 correspondencias a 267 Valor final de factor_correspondencia:
4
Total de correspondencias buenas para las dos imagenes: 267
Imagen 3 aniadida.
He pasado de 2527 correspondencias a 82 Valor final de factor_correspondencia: 4
Total de correspondencias buenas para las dos imagenes: 82
Imagen 4 aniadida.
2
Fin Ejecucion Ejercicio 5 Trabajo 2
```

Por lo que podemos concluir que se crean unos mosaicos con una calidad considerable.

Este mosaico recortado quedaría:



\*\* El anterior mosaico recortado quedaría:



Por lo tanto, los resultados de los mosaicos son muy buenos.