

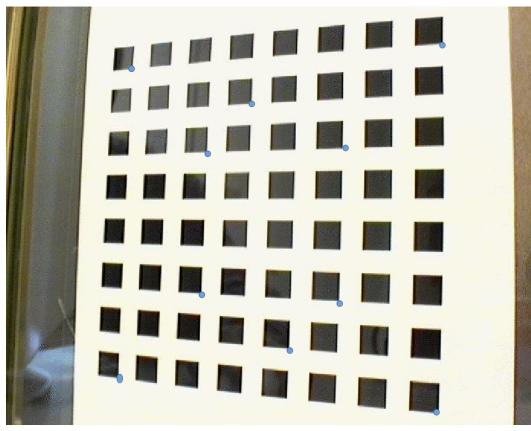
VC: Informe práctica 2

Gustavo Rivas Gervilla

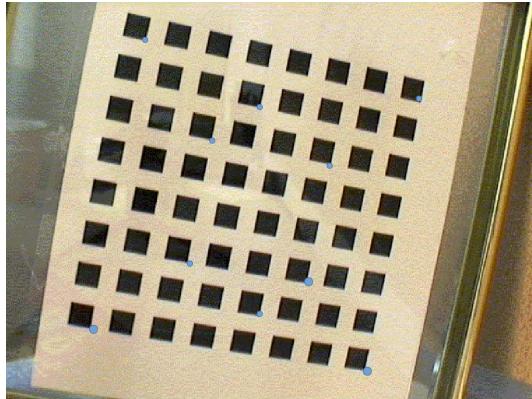


1 Estimación de la homografía

El código para la estimación de la homografía es muy sencillo si tenemos en cuenta lo que hemos visto en teoría junto con el artículo sobre mínimos cuadrados subido a la plataforma DECSAI, para ello lo primero que hemos de hacer es seleccionar 10 puntos en correspondencias en las dos imágenes de modo que estos puntos estén suficientemente distribuidos para que el cálculo de la homografía sea bueno, nosotros hemos seleccionado los siguientes puntos aproximadamente:



(a) Puntos elegidos en Tablero 1



(b) Puntos elegidos en Tablero 2

Figure 1: Puntos seleccionados en cada imagen en azul.

Una vez que tenemos estos puntos no tenemos más que introducirlos en nuestro código, para ello hemos creado una clase muy simple llamada Punto que tiene simplemente dos floats que serán las coordenadas (podríamos usar la clase Point2f de OpenCV como haremos más adelante).

Ahora con estos puntos tenemos que formar la matriz que aparece en las transparencias que es la siguiente:

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

Esto se hace con un bucle for muy básico y que por tanto no lo vamos a mostrar ya que es simplemente seguir la estructura que aquí se marca, creando en nuestro caso una matriz de 20x9, aunque al crear la matriz dónde almacenaremos los coeficientes lo hacemos inicializándola a cero para evitar tener que poner los ceros que tiene la matriz en el bucle, haciendo el código algo más legible (podemos ver el bucle en la función obtenerMatrizCoeficientes del main.cpp).

Ahora de acuerdo con el artículo sobre mínimos cuadrados mencionado no necesitamos más que calcular la descomposición SVD de la matriz y tomar la última columna de V, esto se hace con el siguiente código, usando las funciones implementadas en OpenCV (el código completo es el de la función obtenerMatrizTransformacion de main.cpp):

```
//Obtenemos la descomposicion SVD de la matriz de coeficientes ,
//la matriz que nos interesa es la vT:
SVD::compute(A, w, u, vt);

Mat H = Mat(3, 3, CV_32F);

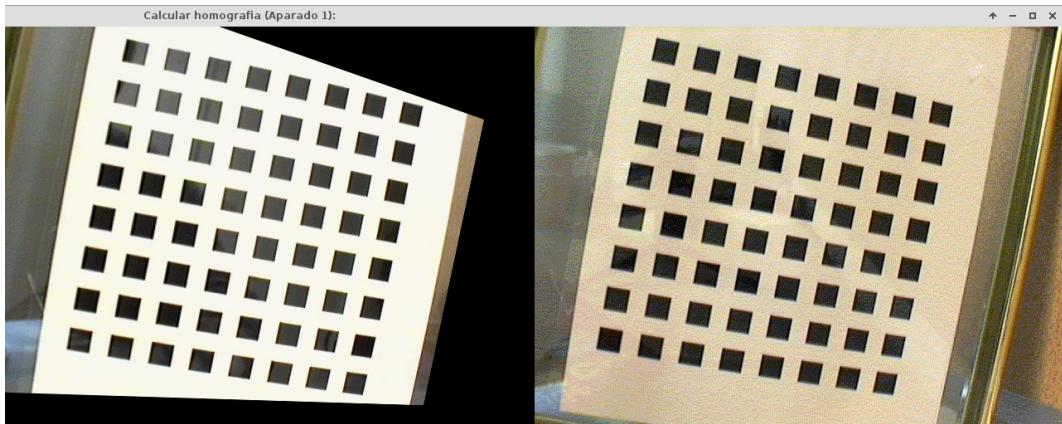
//Construimos la matriz de transformacion con la ultima columna de v
//o lo que es lo mismo la ultima fila de vT:
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        H.at<float>(i, j) = vt.at<float>(8, i * 3 + j);
```

donde A es la matriz de coeficientes calculada previamente como hemos descrito arriba. Yo había pensado en resolver el sistema del que A es matriz de coeficientes ya que OpenCV tiene la función solve pero no podemos hacer esto ya que entre otras cosas no podemos codificar usando esta función la condición de que la solución encontrada tenga norma 1, con lo que podríamos obtener fácilmente la solución trivial algo que evidentemente no nos interesa.

Y ahora no tenemos más que aplicarle la homografía a Tablero1 (que es la que hemos considerado como imagen de partida) para obtener algo parecido a Tablero 2. Esto lo hacemos con el siguiente código:

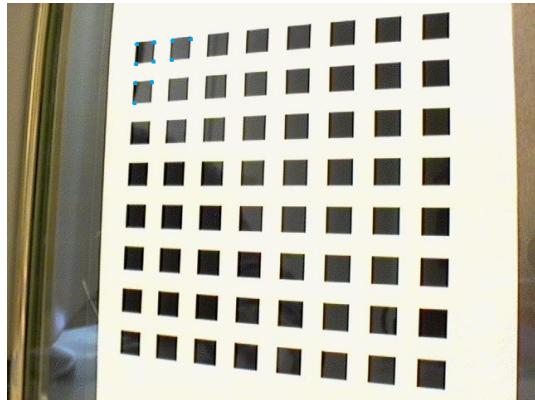
```
Mat tablero1_transformada;
warpPerspective(tablero1, tablero1_transformada, H, Size(tablero2.cols, tablero2.rows));
```

Y aquí tenemos el resultado obtenido comparado con la imagen Tablero2:

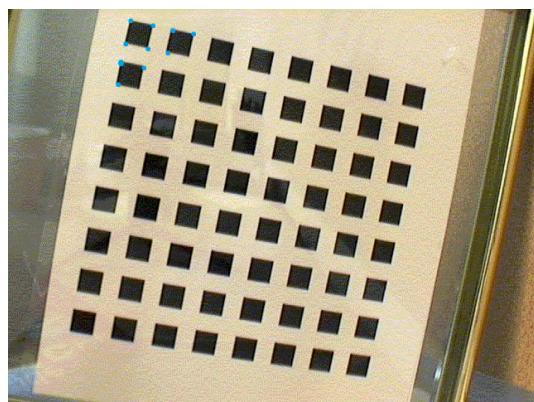


Como podemos ver "la posición del tablero" es muy similar a la del Tablero2, ahora bien lo primero que apreciamos son los cortes negros producto de los puntos que caen fuera del rango de la imagen de destino por la homografía y en segundo lugar notamos lo que no podemos hacer simplemente con geometría y es el color de la imagen, es distinto, nosotros sólo estamos mapeando puntos de la imagen sobre otra no estamos cambiando el valor de dichos píxeles. Además de que por supuesto el resultado no es exacto.

Veamos ahora lo que obtenemos cuando los puntos en correspondencias seleccionados se toman en 3 cuadrados contiguos de una esquina, es esperable que el resultado sea peor puesto que la información que le damos no es tan buena ya que los puntos están más concetrados y por tanto el área de aplicación que "conocemos" de la homografía es más pequeña dando origen posiblemente a más ambigüedades. Aquí tenemos los puntos elegidos:



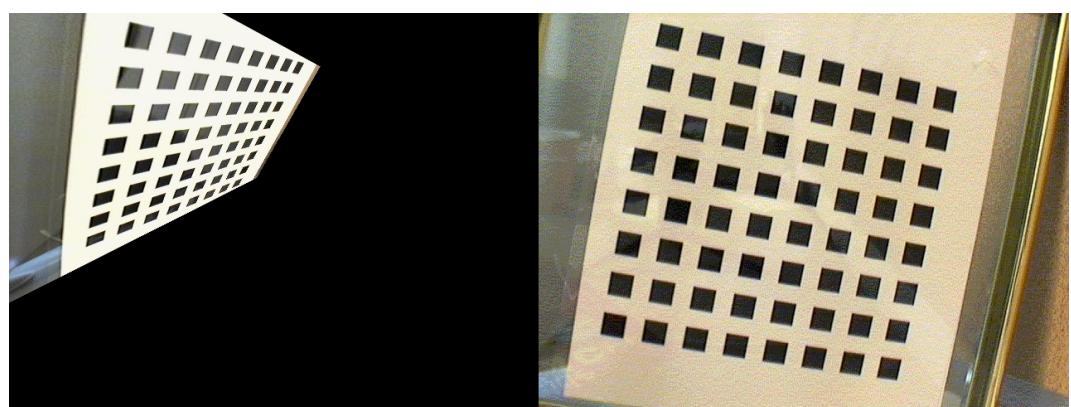
(a) Puntos elegidos en Tablero 1



(b) Puntos elegidos en Tablero 2

Figure 2: Puntos seleccionados en cada imagen en azul.

Y aquí mostramos el resultado que obtenemos al estimar la homografía usando estos puntos en correspondencias:



Es claro que el resultado obtenido es mucho peor, los tres cuadrados donde

hemos tomado puntos quedan más o menos bien, algo que era de esperar ya que la homografía calculada se basa en la información de ellos tres, pero el resto de la imagen no se asemeja prácticamente nada a la de Tablero2, mientras que antes sí que habíamos obtenido un parecido bastante razonable. Como podemos ver se ha producido una cizalla además de un escalado ya que los puntos tomados están muy próximos a los otros con lo cual parece lógico (aunque esto no deja de ser conjetal puesto que lo que estamos haciendo es estimar una homografía de manera aproximada) que lo que está ocurriendo es que se interpreta que se agrupan de este modo todos los puntos de la imagen para que "quepan" en la esquina en correspondencias.

Nota: si se quieren ver los puntos exactos que hemos tomado en la imagen para estimar ambas homografías se pueden ver en la función parte1() del archivo main.cpp.

2 Búsqueda de KeyPoints con los detectores BRISK y ORB

En primer lugar vamos a probar el detector BRISK con los parámetros por defecto, para ello hemos elaborado la siguiente función a la que le pasamos los parámetros que vamos a modificar para el detector BRISK con valor por defecto aquellos que vienen en la documentación de openCV:

```
vector<KeyPoint> obtenerKeyPointsBRISK (Mat im, int umbral = 30) {
    //Creamos el detector
    Ptr<BRISK> ptrDetectorBRISK = BRISK::create(umbral);
    vector<KeyPoint> puntosDetectados;

    //Obtenemos los KP:
    ptrDetectorBRISK->detect(im, puntosDetectados);

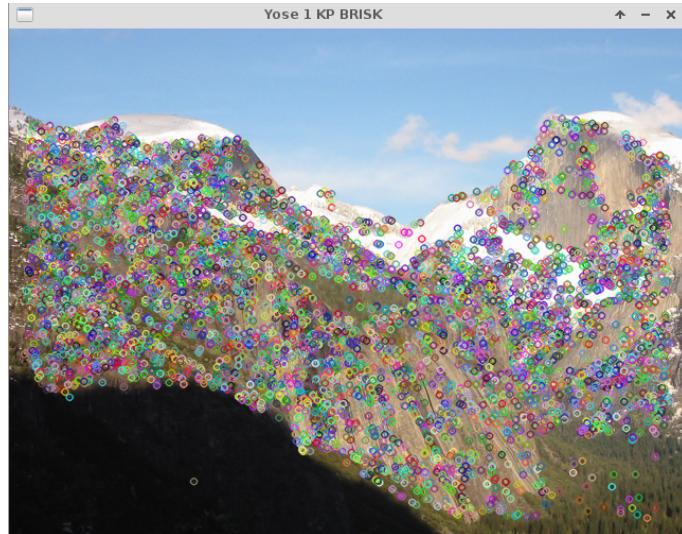
    return puntosDetectados;
}
```

Lo único que hacemos en esta función es crear un puntero a un detector BRISK con los parámetros que le pasamos y extraer con él los KeyPoints (estructura de openCV para obtener puntos de interés de una imagen con un descriptor) de la imagen *im* que le pasamos como parámetro y devolvemos el vector de KeyPoints encontrados por el detector para futuros cálculos, como por ejemplo poder pintarlos en la imagen.

Con estos parámetros tenemos que se obtienen 6218 puntos una cantidad demasiado grande de puntos para nuestros propósitos, además como sabemos en algunos casos, no sólo por el coste computacional, tener tantos datos puede ser contraproducente. Lo que haremos a continuación es modificar los parámetros del detector que creamos para que el número de puntos obtenidos sea aproximadamente 1000 (que es la cantidad que dijimos en clase que sería adecuada). Aquí tenemos todos los puntos obtenidos pintados en la imagen, para ello hacemos uso de la función drawKeypoints como vemos a continuación:

```
puntosDetectados = obtenerKeyPointsBRISK(yose1);
drawKeypoints(yose1, puntosDetectados, yose1KPBRISK);
imshow("Yose_1_KP_BRISK", yose1KPBRISK);
```

En lo sucesivo no vamos a mostrar más este código pese a que cambiemos de imagen o de detector ya que para todos es el mismo, sólo tenemos que obtener el listado de KeyPoints encontrados por el detector que estemos usando en ese momento y pasárselo a la función drawKeypoints junto a la imagen sobre la que queremos mostrarlos (la imagen a la que se los hemos calculado obviamente).



Pues bien ahora vamos a modificar los parámetros del detector creado de modo que obtengamos unos 1000 puntos. Para ello veamos qué párametros tiene este detector:

1. umbral: es la distancia umbral para la medida que hace FAST para seleccionar o no un punto determinado. Lo que hace FAST es trazar un círculo alrededor del píxel de tamaño 16px, entonces seleccionará dicho píxel si hay al menos n píxeles de modo que sean o más claros que $I_p + t$ o más oscuros que $I_p - t$ (siendo I_p la intensidad del píxel estudiado y t el umbral elegido). Con lo cual parece lógico que cuánto más pequeño sea t más puntos serán seleccionados pues es más fácil encontrar píxeles a sus alrededor que cumplan tal condición.
2. octavas: el número de escalas en las que estudiamos la imagen.
3. factor de escala: factor a aplicar al patrón que usamos para estudiar la imagen, básicamente es cómo se dibujan los círculos sobre la imagen para aplicar el criterio FAST. La siguiente imagen lo muestra y ha sido obtenida del enlace que se muestra:

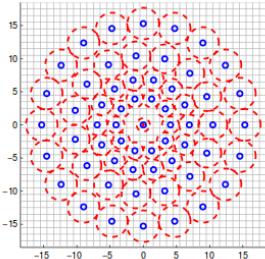
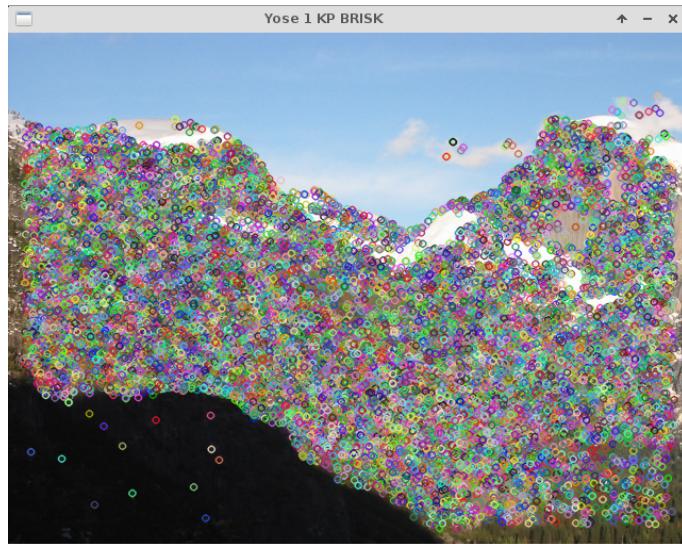


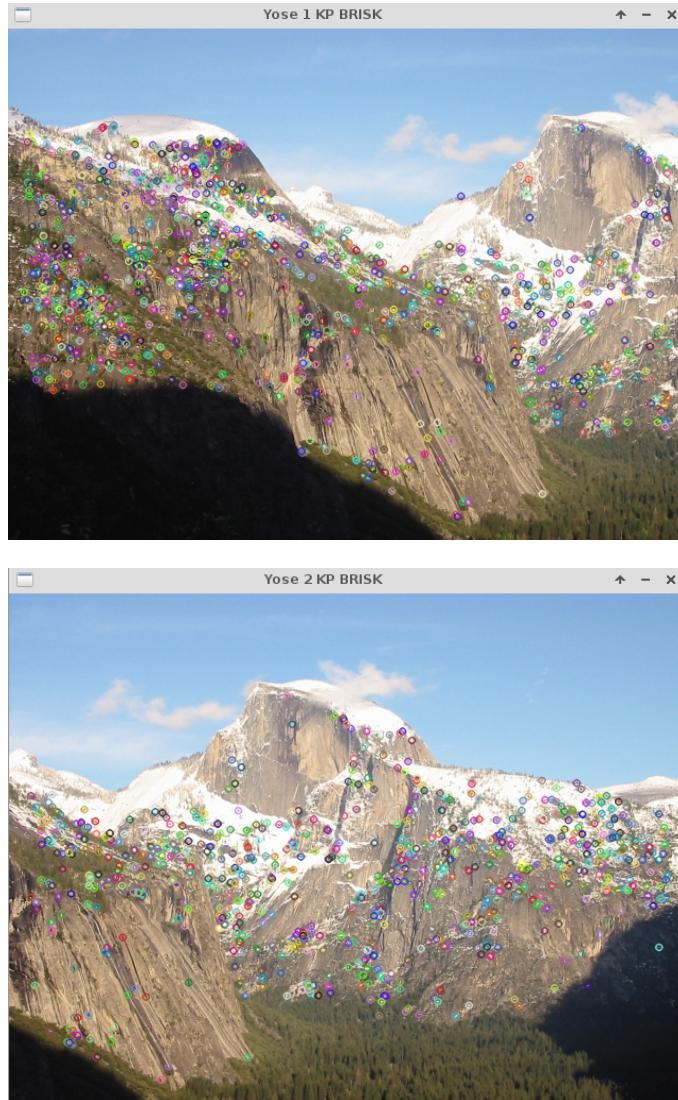
Figure 3: <http://www.asl.ethz.ch/people/lestefan/personal/iccv2011.pdf>

Pues bien como se dijo en clase y cómo tiene sentido nosotros sólo vamos a jugar con el umbral ya que en definitiva es lo que establece que un punto se seleccione o no. Por ello en la función sólo le pasamos como parámetro el umbral.

Lo primero que hacemos es bajar el umbral de 30 a 10 para confirmar algo que ya sabíamos, el número de puntos detectados crece enormemente, ahora se detectan 14416 puntos:



Entonces ahora lo que vamos a hacer es ir subiendo el umbral hasta obtener los 1000 puntos aproximadamente que buscábamos. El umbral elegido ha sido 65, evidentemente en las dos imágenes no se obtienen la misma cantidad de puntos dependerá de la intensidad de cada una, más concretamente en Yosemitel hemos obtenido 1060 puntos mientras que en Yosemitel2 hemos obtenido 817.



Fijémonos en cómo se ve de forma muy gráfica cómo funciona el detector, y es que cuando teníamos un umbral bajo con una mínima diferencia con los puntos del alrededor ya se seleccionaba el punto, por eso con umbral 10 se detectaban puntos en las nubes y en la región oscura de Yosemité1 que son regiones muy homogéneas en cuanto a intensidad. En cambio en cuanto hemos subido el umbral los puntos de estas regiones han desaparecido por completo y sólo se han obtenido puntos de regiones con una mayor variación de intensidad.

Al contrario que lo que sucedía con BRISK cuando aplicamos los parámetros por defecto de ORB obtenemos muchos menos puntos, sólo 500, esto se debe a que el detector tiene un parámetros que controla cuál es el máximo número de características que queremos obtener y por defecto es 500. Para usar este detector hemos declarado la siguiente función a la que le pasamos igual que antes los parámetros del detector que nosotros vamos a modificar.

```

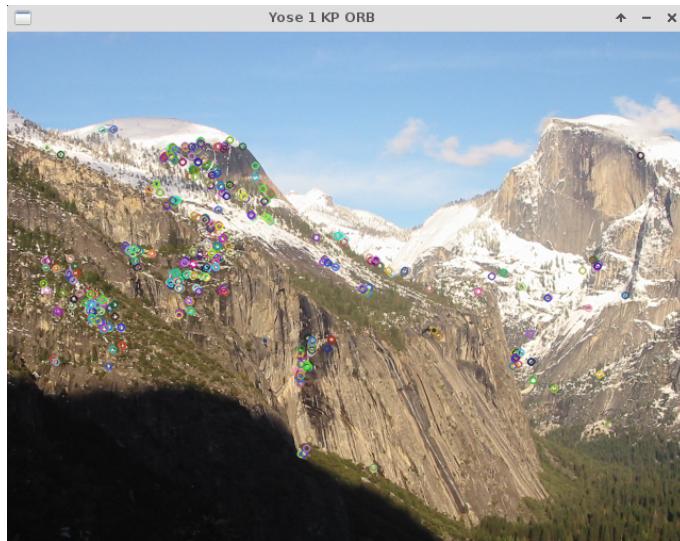
vector<KeyPoint> obtenerKeyPointsORB (Mat im, int num_caracteristicas = 500,
int tipo_marcador = ORB::HARRIS_SCORE, int umbral_FAST = 20){
    //Creamos el detector:
    Ptr<ORB> ptrDetectorORB = ORB::create(num_caracteristicas, 1.2f, 8, 31, 0, 2,
    tipo_marcador, 31, umbral_FAST);
    vector<KeyPoint> puntosDetectados;

    //Obtenemos los KP:
    ptrDetectorORB->detect(im, puntosDetectados);

    return puntosDetectados;
}

```

Aquí tenemos dibujados los puntos detectados con un descriptor ORB con los valores por defecto:

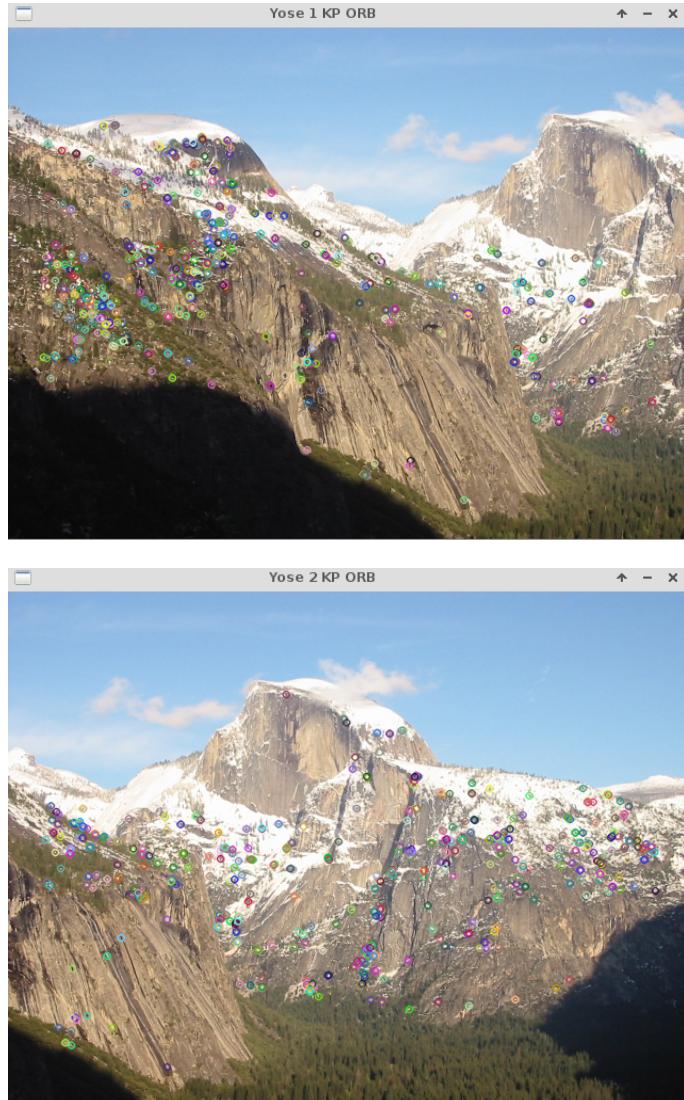


Veamos los parámetros que tiene el constructor de este tipo de detector:

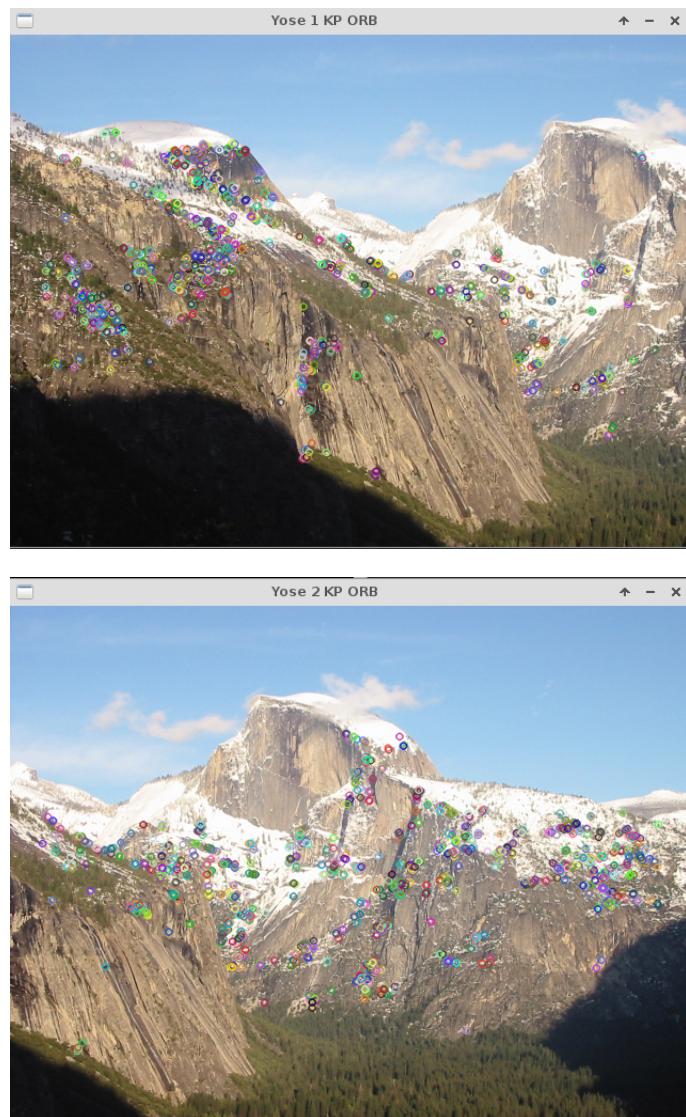
1. número máximo de puntos a detectar.
2. factor de escala: que indica el submuestreo que se realiza entre un nivel de la pirámide y la siguiente. Dice que un umbral igual a dos reduciría el score de los matches obtenidos, con lo cual nosotros lo vamos a dejar con 1.2 que es el valor que viene por defecto.
3. número de niveles de la pirámide que se forma para buscar puntos, también lo dejaremos con el valor por defecto.
4. umbral de borde: es el tamaño del borde donde no se detectan puntos y nos dice que debe coincidir con el tamaño del área que se usa para obtener los descriptores BRIEF, con lo cual como ninguno de estos dos parámetros nos afectan los dejaremos por defecto.

5. tamaño del área para obtener los descriptores BRIEF.
6. WTA_K parámetro relacionado con el detector BRIEF, como nuestro propósito con este detector es sólo obtener KeyPoints lo dejamos también por defecto.
7. tipo de criterio: es el criterio o la medida que se usa para decidir si un punto se selecciona o no, puede ser el HARRIS que es más estable o el FAST como se usa en el detector BRISK que es más rápido computacionalmente.
8. umbral para FAST: el umbral para FAST explicado antes cuando hemos hablado de los parámetros del detector BRISK.

Lo primero que vamos a probar es a aplicar el mismo criterio que aplica BRISK con el mismo umbral para ver si los resultados obtenidos son los mismos para ellos ponemos el criterio como ORB::FAST_SCORE y el umbral a 65 igual que antes. Los resultados obtenidos nos son los mismos ya que obtenemos 678 y 570 puntos, así que no obtenemos la misma cantidad de puntos que cuando usábamos BRISK con el mismo umbral.



Lo que vamos a hacer dado que dice que los puntos obtenido con el criterio Harris son más estables es usar dicho criterio y regular el umbral hasta obtener unos mil puntos igual que antes. Hemos puesto el umbral a 35 de modo que fuese lo más alto posible pero que obtuviésemos suficiente puntos. Con estos parámetros hemos obtenido 994 puntos para Yosemite1 y 999 puntos para Yosemite2, ahora es la segunda imagen en la que se obtienen más puntos con el mismo criterio.



Comparativa final de ambos métodos:

Ya hemos visto cómo ambos detectores pese a teóricamente usar el mismo mecanismo para discriminar puntos (FAST con el mismo umbral) detectaban una cantidad de puntos distinta, indicativo de que no sólo eso influye en la detección de puntos, también dependerá de cómo se muestre o de cómo se haga la medida para discriminar puntos.

En cuanto a los experimentos realizados con los distintos detectores para obtener una buena cantidad de puntos observo que es algo más complicado aumentar la cantidad de puntos con ORB, como si el criterio utilizado fuese más exigente.

Pese a que la densidad de puntos, como hemos comentado antes, es distinta entre ambos descriptores, al menos en lo obtenido, los puntos obtenidos por BRISK están más dispersos por la imagen, no concentrándose tanto en zonas determinadas, al menos eso percibo visualmente.

Otra cosa a señalar es que aunque la cantidad de puntos obtenidos no sea la misma sí que se suelen encontrar por las mismas zonas de la imagen tanto con ORB como con BRISK, es decir, que los puntos que busca son similares.

Por último con lo que he obtenido yo para los parámetros elegidos prefiero los resultados obtenidos por BRISK ya que hay mayor cantidad de puntos en las zonas comunes a Yosemite1 y a Yosemite2 con lo cual a la hora de buscar correspondencias para elaborar el mosaico creo que serán mejor estos datos.

3 Descriptores y matching

En primer lugar vamos a ver ambos criterios sobre los puntos detectados por BRISK. Aquí tenemos el código para obtener los matches mediante Fuerza Bruta + cross check + BRISK:

```
Mat obtenerDescriptoresBRISK (Mat im, int umbral = 30) {
    //Creamos el detector:
    Ptr<BRISK> ptrDetectorBRISK = BRISK::create(umbral);
    vector<KeyPoint> puntosDetectados;
    Mat descriptores;

    //Obtenemos los KP:
    ptrDetectorBRISK->detect(im, puntosDetectados);

    //Obtenemos los descriptores para estos KP:
    ptrDetectorBRISK->compute(im, puntosDetectados, descriptores);

    return descriptores;
}

vector<DMatch> obtenerMatchesFuerzaBrutaBRISK (Mat im1, Mat im2, int umbral){
    vector<KeyPoint> puntosDetectados1, puntosDetectados2;
    Mat descriptores1, descriptores2;
    vector<DMatch> matches;

    //Creamos el matcher con Fuerza Bruta activandole el flag para el cross check.
    BFMatcher matcher = BFMatcher(NORM_L2, true);

    //Obtenemos los Key Points con BRISK:
    puntosDetectados1 = obtenerKeyPointsBRISK(im1, umbral);
    puntosDetectados2 = obtenerKeyPointsBRISK(im2, umbral);

    //Obtenemos los descriptores de los puntos obtenidos en cada imagen.
    descriptores1 = obtenerDescriptoresBRISK(im1, umbral);
    descriptores2 = obtenerDescriptoresBRISK(im2, umbral);

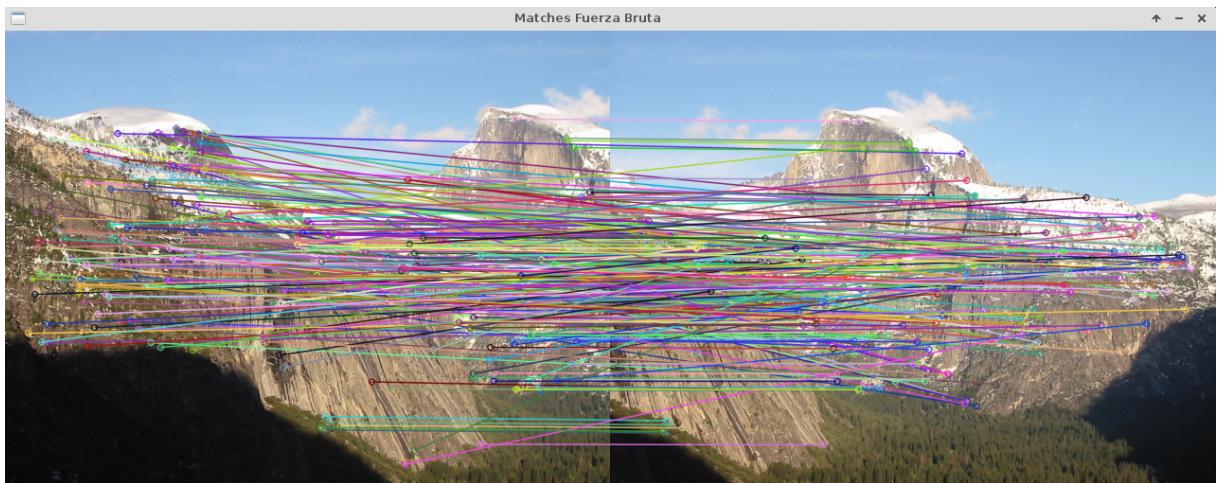
    //clock_t t_inicio= clock();
    //Calculamos los matches entre ambas imagenes:
    matcher.match(descriptores1, descriptores2, matches);
    //printf("FB ha tardado: %.2fs\n", (double)(clock() - t_inicio)/CLOCKS_PER_SEC);

    return matches;
}
```

Se obtienen en total 519 matches y aquí podemos verlos:



La cantidad de matches encontrados es demasiado grande como para poder ver bien la corrección de los mismos por lo tanto lo que vamos a hacer es mostrar sólo unos pocos para la evaluación, luego para el código general mostraremos todos. Aquí mostramos la mitad de los matches encontrados:



Como podemos observar en aquellos puntos detectados en zonas comunes a ambas imágenes los matches que se han creado son buenos, pero en cambio hay muchísimos matches que se establecen entre puntos de las zonas que no son comunes a ambas imágenes.

Veamos ahora lo que sucede cuando usamos el matcher basado en FLANN. En esta ocasión usando el mismo los mismo KeyPoints hemos obtenido 1060 matches, es decir, este mecanismo es mucho menos estricto, aunque como sabemos es más rápido computacionalmente.

```
vector<DMatch> obtenerMatchesFlannBRISK (Mat im1, Mat im2, int umbral){
```

```

vector<KeyPoint> puntosDetectados1, puntosDetectados2;
Mat descriptores1, descriptores2;
vector<DMatch> matches;

//Creamos el matcher con FLANN.
FlannBasedMatcher matcher;

//Obtenemos los Key Points con BRISK:
puntosDetectados1 = obtenerKeyPointsBRISK(im1, umbral);
puntosDetectados2 = obtenerKeyPointsBRISK(im2, umbral);

//Obtenemos los descriptores de los puntos obtenidos en cada imagen.
descriptores1 = obtenerDescriptoresBRISK(im1, umbral);
descriptores2 = obtenerDescriptoresBRISK(im2, umbral);

//Convertimos las matrices de descriptores a CV_32F para el correcto funcionamiento
descriptores1.convertTo(descriptores1, CV_32F);
descriptores2.convertTo(descriptores2, CV_32F);

//clock_t t_inicio = clock();
matcher.match(descriptores1, descriptores2, matches);
//printf("FLANN ha tardado: %.2fs\n", (double)(clock() - t_inicio)/CLOCKS_PER_SEC);

return matches;
}

```



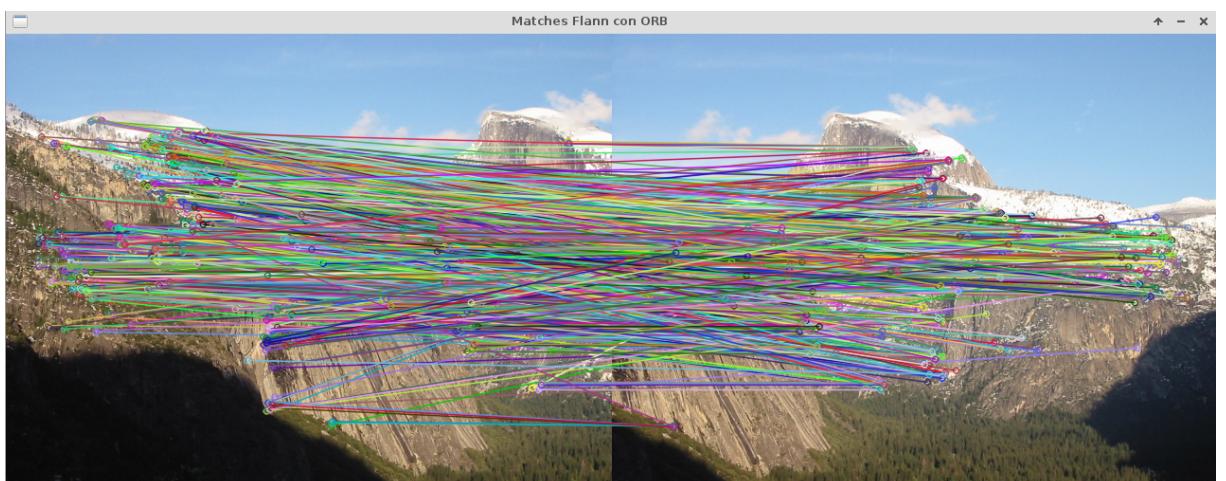
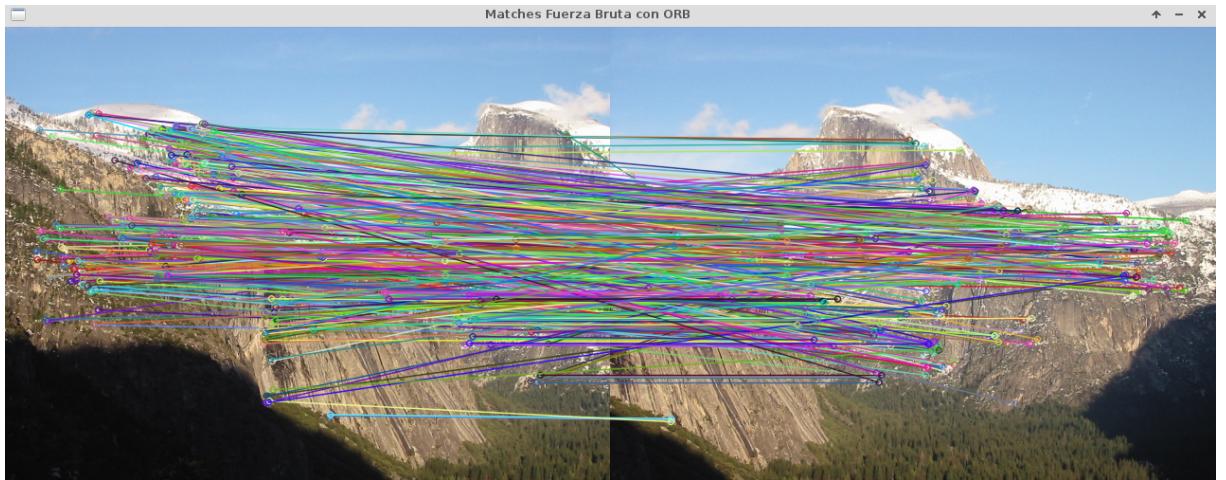
Veamos sólo la mitad de matches:



Aquí al igual que sucedía con Fuerza Bruta se producen mucho matches erróneos e incluso es muy probable que en mayor cantidad, en primer lugar porque los matches encontrados son más y en segundo lugar porque como ya hemos dicho el criterio ha de ser más relajado. En consecuencia nosotros utilizaremos los matches obtenidos por Fuerza Bruta para los apartados siguientes ya que creo que nos darán unos datos más adecuados para la estimación de homografías.

Lo que vamos a hacer ahora es medir el tiempo que tardan cada uno de los criterios para comprobar que efectivamente FLANN es mucho más rápido: En media tras algunas ejecuciones para buscar los matches entre las dos imágenes de Yosemite tenemos que Fuerza Bruta tarda 0.035 segundos mientras que FLANN tarda 0.01 segundos, o sea que tenemos lo que queríamos (el código para calcular estos tiempos está comentados en las funciones obtenerMatchesFuerzaBrutaBRISK y obtenerMatchesFLANNBRISK).

Ahora vamos a obtener los matches cuando trabajamos sobre los puntos detectados por el detector ORB por si al detectar puntos distintos estos puntos permiten encontrar mejor matches al tener otros descriptores (las funciones para esto están en el main.cpp pero no las mostramos porque la estructura es igual a las anteriores sin más que cambiar el detector BRISK por ORB). En esta ocasión hemos obtenido 544 matches con Fuerza Bruta y 994 con FLANN (se siguen obteniendo más puntos con FLANN), en esta ocasión Fuerza Bruta ha detectado algunos matches más:



Los resultados obtenidos son muy similares, en cuanto a bondad, con lo que usaremos los puntos obtenidos por BRISK para los apartados anteriores.

4 Mosaico con dos imágenes

Para construir el siguiente mosaico hemos usado los matches obtenidos por Fuerza Bruta a partir de los puntos de BRISK. Lo único que hemos hecho es colocar, por medio de la matriz identidad, la imagen Yosemite1 en la esquina superior izquierda del mosaico y a continuación llevar al mosaico la Yosemite2 por medio de la homografía obtenida con estos matches.

Hemos creado una función que calcular la homografía que lleva de una imagen a la otra mediante `findHomography` con los matches que hemos dicho y luego no hemos tenido más que colocar la primera imagen donde queríamos mediante una translación y la segunda colocarla allí donde nos lleve la homografía calculada:

```
Mat calcularHomografia (Mat origen , Mat destino) {
    vector<KeyPoint> puntosDetectadosOrigen , puntosDetectadosDestino;
    vector<DMatch> matches;
    vector<Point2f> puntosEnCorrespondenciasOrigen , puntosEnCorrespondenciasDestino;

    //Obtenemos los puntos clave con BRISK en cada imagen
    puntosDetectadosOrigen = obtenerKeyPointsBRISK(origen , 65);
    puntosDetectadosDestino = obtenerKeyPointsBRISK(destino , 65);

    //Obtenemos los matches por fuerza bruta:
    matches = obtenerMatchesFuerzaBrutaBRISK(origen , destino , 65);

    //Obtenemos los puntos en correspondencias entre ambas imagenes:
    for (int i = 0; i < matches.size(); i++){
        puntosEnCorrespondenciasOrigen .push_back(
            puntosDetectadosOrigen [matches [i].queryIdx ].pt );
        puntosEnCorrespondenciasDestino .push_back(
            puntosDetectadosDestino [matches [i].trainIdx ].pt );
    }

    //Calculamos la homografia con los puntos en correspondencias:
    Mat H = findHomography(puntosEnCorrespondenciasOrigen ,
                           puntosEnCorrespondenciasDestino , CV_RANSAC);

    //Pasamos la homografia a 32F:
    H.convertTo(H, CV_32F);

    return H;
}

Mat mosaicoDeDos (Mat origen , Mat destino) {
    //Creamos la imagen donde proyectaremos ambas imagenes:
    Mat mosaico = Mat(550, 1000, origen.type());

    //Colocamos la primera imagen en la esquina superior izquierda por medio de
    //la identidad:
    Mat id = Mat(3,3,CV_32F,0.0);

    for (int i = 0; i < 3; i++)
        id.at<float>(i,i) = 1.0;

    warpPerspective(destino , mosaico , id , Size(mosaico.cols , mosaico.rows) , INTER_LINEAR,
                    BORDER_CONSTANT);

    //Calculamos la homografia que lleva la segunda imagen a la que hemos colocado primera
    //en el plano de proyeccion:
```

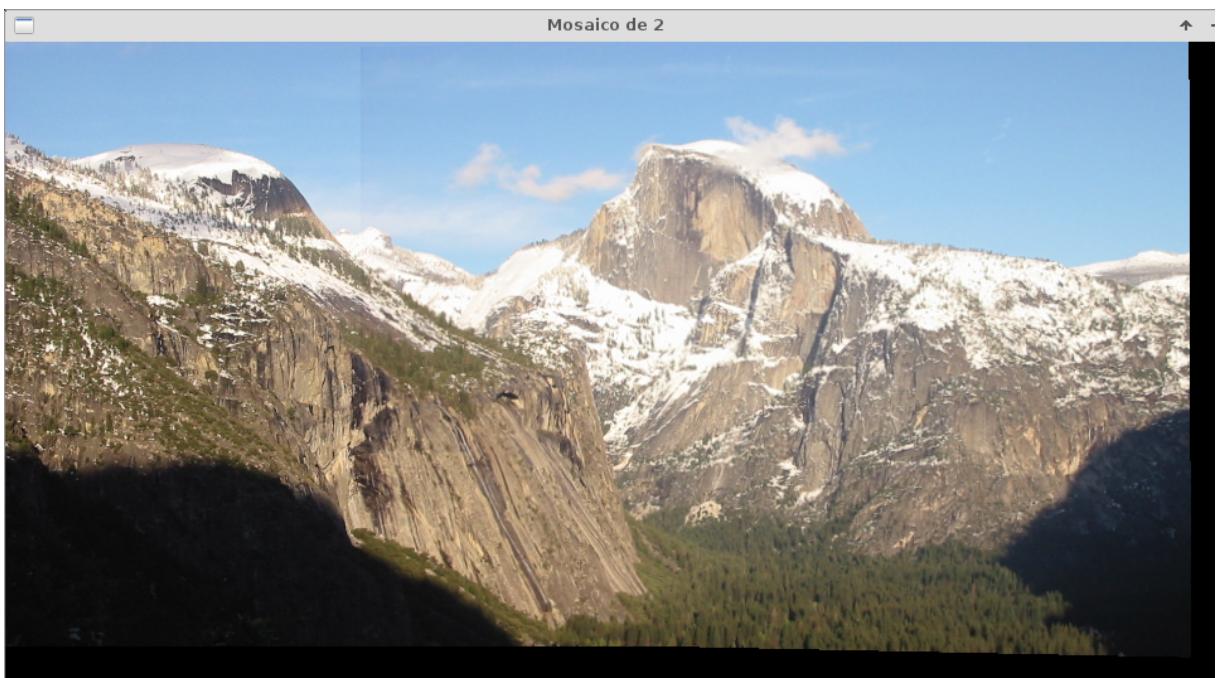
```

    Mat homografia = calcularHomografia(origen, destino);

    //Colocamos la segunda imagen por medio de esa homografia
    //(compuesta con la identidad):
    warpPerspective(origen, mosaico, homografia, Size(mosaico.cols, mosaico.rows),
    INTER_LINEAR, BORDER_TRANSPARENT);

    return mosaico;
}

```



Lo único que podemos observar aquí es que se produce un corte de color debido a las distintas condiciones de iluminación de ambas fotografías. La otra opción sería colocar primero Yosemite2 y que sea Yosemite1 la que solape, el corte estaría presente, pero ahora los colores que se verían serían los de Yosemite1 simplemente.

Podremos decir más en el siguiente apartado cuando ya hay varias imágenes.

5 Mosaico con varias imágenes

Nuevamente vamos a usar Fuerza Bruta + BRISK. Lo primero que hemos hecho es poner las fotografía en el orden correcto, ya que las fotografía Mosaico005 y Mosaico006 están intercambiadas. Esto lo hemos hecho con el objetivo de obtener los mejores resultados posibles.

La otra cosa que hemos hecho para obtener el mejor resultado posible es hacer lo que ya se dijo en clase, situar la foto más central (con respecto a la realidad) en el centro del mosaico en un plano de proyección paralelo a ella. El resto de imágenes serán proyectadas en el mismo plano, obteniendo así la menor distorsión posible.

```
Mat mosaicoDeN (vector<Mat> imagenes) {
    //Creamos la imagen donde formaremos el mosaico:
    Mat mosaico = Mat(700, 1100, imagenes.at(0).type());

    //Seleccionamos la posicion de la imagen central de la secuencia:
    int posicion_central = imagenes.size()/2;

    //Colocamos la imagen central del vector en el centro del mosaico
    Mat colocacionCentral = Mat(3,3,CV_32F,0.0);

    for (int i = 0; i < 3; i++)
        colocacionCentral.at<float>(i,i) = 1.0;

    //Realizamos una traslacion simplemente:
    colocacionCentral.at<float>(0,2) = mosaico.cols/2 -
    imagenes.at(posicion_central).cols/2;
    colocacionCentral.at<float>(1,2) = mosaico.rows/2 -
    imagenes.at(posicion_central).rows/2;

    warpPerspective(imagenes.at(posicion_central), mosaico, colocacionCentral,
    Size(mosaico.cols, mosaico.rows), INTER_LINEAR, BORDER_CONSTANT);

    //Matrices donde se acumularan las homografias a cada uno de los lados de la
    //imagen central:
    Mat Hizda, Hdcha;
    //Las inicializamos con la homografia que hemos calculado antes, la que seria la H0:
    colocacionCentral.copyTo(Hizda);
    colocacionCentral.copyTo(Hdcha);

    /*
    Vamos formando el mosaico empezando desde la imagen central y desplazandonos a ambos
    extremos.
    En cada iteracion calculamos la homografia que queda cada imagen al mosaico
    componiendo con las que ya se han calculado para las imagenes anteriores.
    */
    for (int i = 1; i <= posicion_central; i++) {
        if (posicion_central-i >= 0){
            Hizda = Hizda * calcularHomografia(imagenes.at(posicion_central-i),
            imagenes.at(posicion_central-i+1));
            warpPerspective(imagenes.at(posicion_central-i), mosaico, Hizda,
            Size(mosaico.cols, mosaico.rows), INTER_LINEAR, BORDER_TRANSPARENT);
        }
        if (posicion_central+i < imagenes.size()){
            Hdcha = Hdcha * calcularHomografia(imagenes.at(posicion_central+i),
            imagenes.at(posicion_central+i-1));
```

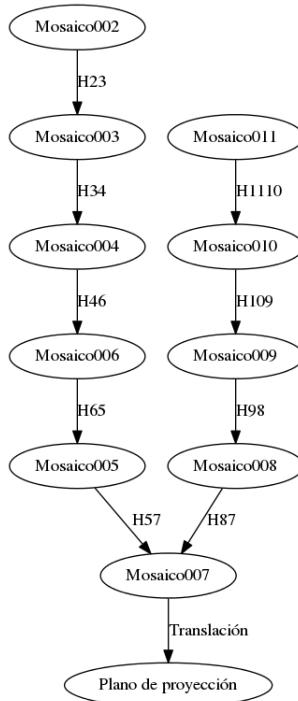
```

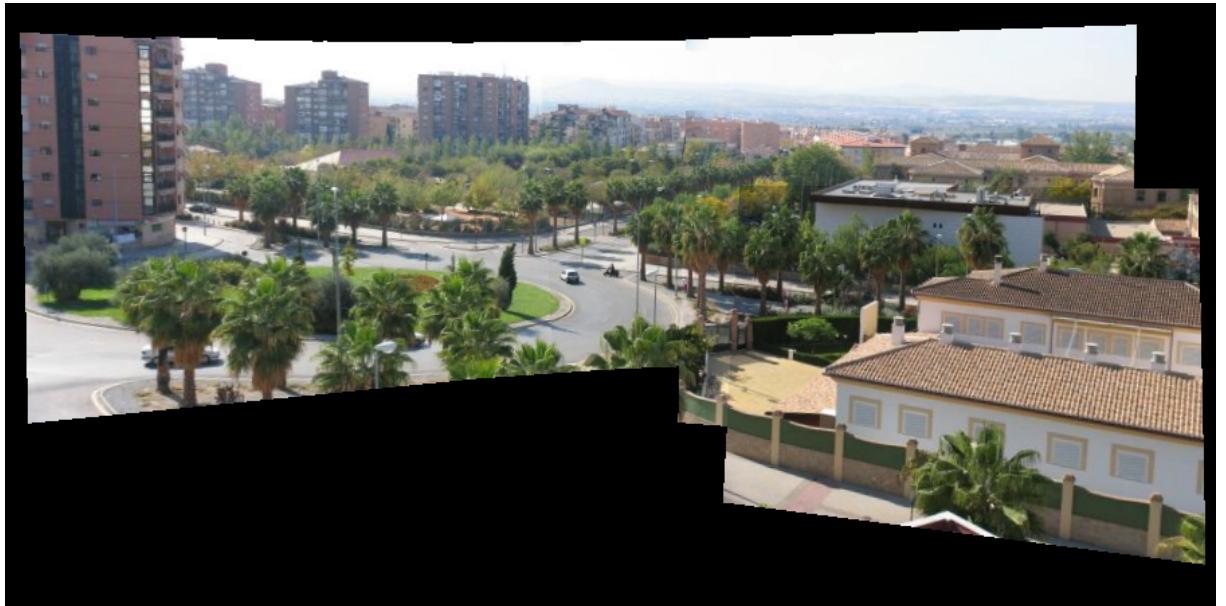
        warpPerspective(imagenes.at(posicion_central+i), mosaico, Hdcha,
Size(mosaico.cols, mosaico.rows), INTER_LINEAR, BORDER_TRANSPARENT);
    }

return mosaico;
}

```

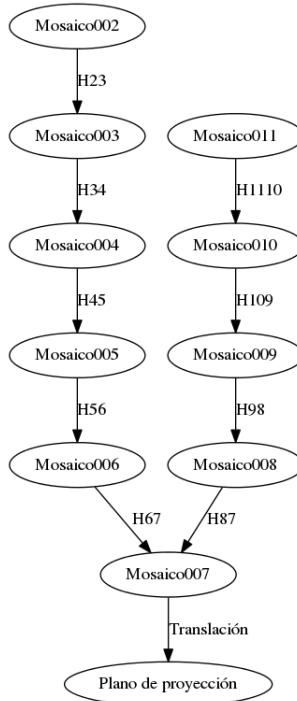
Entonces las homografías que calculamos son las siguientes:



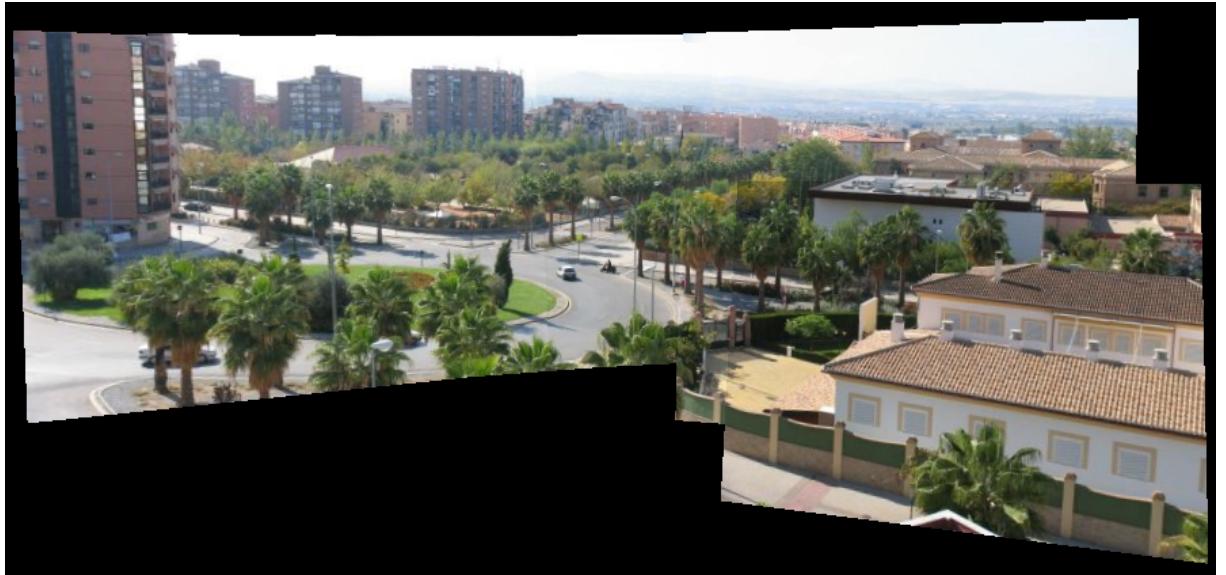


Como podemos ver el resultado es bastante bueno, no apreciamos distorsiones excesivas y sólo se ven algunos pequeños saltos de color inevitables con las técnicas aquí empleadas. También observamos que los bordes no son totalmente rectos probablemente porque los puntos ahí se salen del rango de la imagen o la homografía los coloca de ese modo.

Vamos a hacer ahora un par de experimentos más. En el primero de ellos vamos a nuevamente usar la técnica de usar la imagen central como inicio del mosaico pero en esta ocasión no vamos a alterar el orden de las imágenes, dejando la 5 y la 6 en su lugar para ver cómo afecta esto al mosaico. Ahora las homografías que se van a calcular son las siguientes:



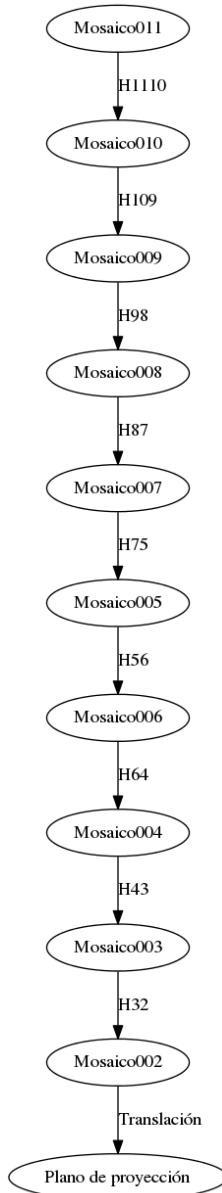
Con lo que los problemas pueden surgir en las homografías H45 y H67 ya que estas imágenes no tienen tantos puntos en correspondencias como los tenían las 4 con la 6 y la 5 con la 7. Veamos el resultado:



Como vemos las diferencias con el mosaico anterior son prácticamente inexistentes esto se debe a que las imágenes señaladas están tomadas con muy

poco movimiento de la una a la otra con lo que hay suficientes puntos en correspondencias (buenos) como para calcular una homografía adecuada.

Por último lo que vamos a hacer es partir desde la primera imagen, Mosaico002, e ir añadiendo el resto a su derecha, de este modo queremos apreciar cómo la deformación sufrida por las imágenes es mayor al usar este plano de proyección, como vimos en teoría. Aquí las proyecciones a calcular son las siguientes:



```
Mat mosaicoDeNMalo (vector<Mat> imagenes) {
```

```

Mat mosaico = Mat(700, 1100, imagenes.at(0).type());

//Colocamos la primera imagen en la parte izquierda del mosaico pero
// no arriba del todo:
Mat colocacionInicial = Mat(3,3,CV_32F,0.0);

for (int i = 0; i < 3; i++)
    colocacionInicial.at<float>(i,i) = 1.0;

//Realizamos una traslacion simplemente:
colocacionInicial.at<float>(1,2) = 100;

warpPerspective(imagenes.at(0), mosaico, colocacionInicial,
Size(mosaico.cols, mosaico.rows), INTER_LINEAR, BORDER_CONSTANT);

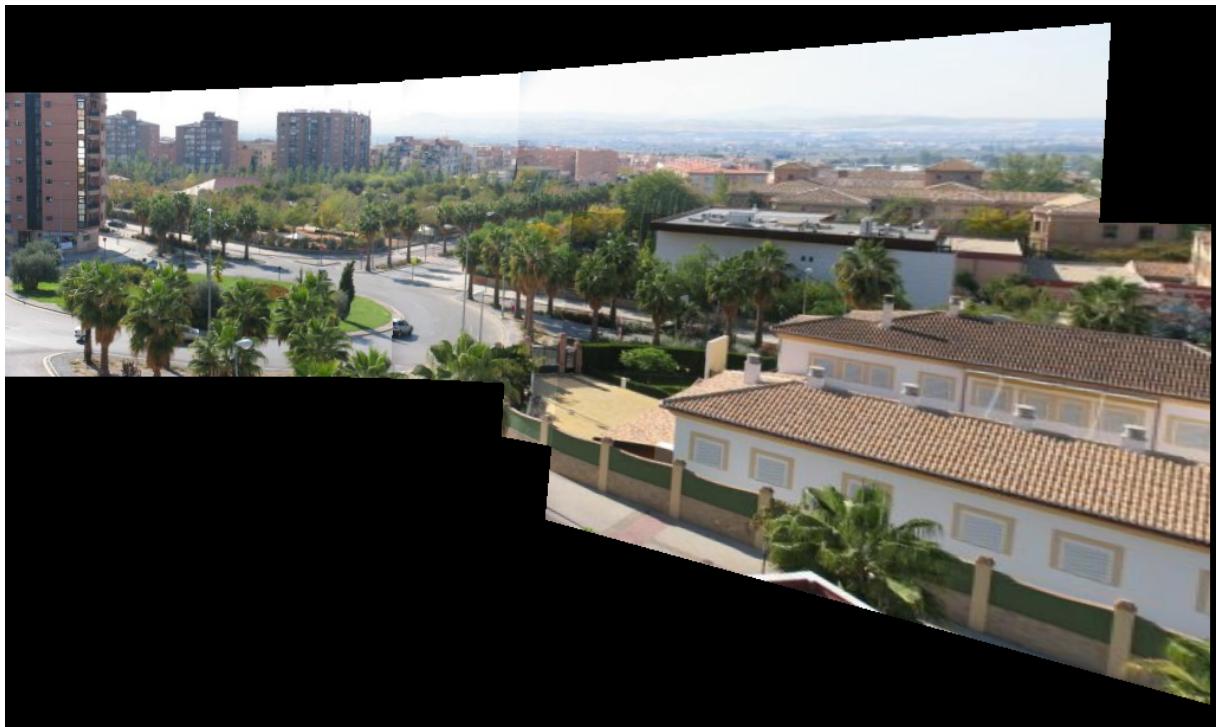
//Matriz donde se acumularan las homografias segun vamos colocando imagenes
//en el mosaico:
Mat H;
//Se inicializa con la homografia calculada anteriormente:
colocacionInicial.copyTo(H);

/*
Vamos recorriendo la secuencia de imagenes hacia la derecha.
Calculamos la homografia que lleva la imagen actual a la anterior y la acumulamos
(componemos) con las previamente calculadas.
*/
for (int i = 1; i < imagenes.size(); i++) {
    H = H * calcularHomografia(imagenes.at(i), imagenes.at(i-1));
    warpPerspective(imagenes.at(i), mosaico, H, Size(mosaico.cols, mosaico.rows),
INTER_LINEAR, BORDER_TRANSPARENT);
}

return mosaico;
}

```

Aquí no hay problemas de que las imágenes tengan pocos puntos en correspondencias ya que las estamos tomando en el mejor orden posible, el problema es, como hemos dicho, el plano de proyección elegido. Lo que obtenemos en esta ocasión es lo siguiente:



Y aquí observamos lo que veníamos buscando, y es que si nos fijamos la casa blanca a la derecha aparece mucho más grande que en los mosaicos anteriores ya que las últimas imágenes de la secuencia sufren una deformación mayor (un estiramiento mayor) que las otras, como queríamos probar.