VISIÓN POR COMPUTADOR
Trabajo-2

Detección de puntos Harris multiescala

Se ha reutilizado parte del código de la práctica anterior como es el de la pirámide gaussiana. En este caso la necesitamos para reducir a una escala menor las imágenes de las que se quieren obtener los puntos Harris.

Se va a mostrar a continuación la función general que define este ejercicio, **scalePyramid()** no obstante debajo se irán explicando las funciones que se han ido usando en ésta:

```
/**
* Dada una imagen @img y el número de reducciones @reductions obtener los círculos y líneas de los puntos Harris de dicha imagen con las reducciones dadas.
*/
Mat scalePyramid(Mat img, int reductions){
	Mat aux = gaussianFilter(img);
	float k = 0.04, scale;
	int bsize = 3, aperture = 3, wsize = 3, cradius;
	for (int i = 0; i<=reductions; i++){
		Mat img_gray, harris, binary;
		cradius = (wsize+1)*(i+1);
		scale = 1/(i+1)*pow(2,i);

		cvtColor(aux, img_gray, COLOR_BGR2GRAY);

		cornerEigenValsAndVecs( img_gray, harris, bsize, aperture, BORDER_DEFAULT );

		harrisDetector(harris, k);

		binary = supressNoMax(harris, wsize);

		vector<Mat> harrisData = getHarrisData(harris, binary, scale);

		drawCircleAndOrientation(harrisData, img, img_gray, cradius, i);
		remove(harrisData);
		pyramidDown(aux);
	}
	return img;
}
```

Esta función reduce el tamaño de la imagen original a la mitad las veces que le digamos, y en cada una de estas iteraciones calcula los puntos Harris y los pinta en dicha imagen con ayuda de una serie de funciones.

Con la función de OpenCV **cornerEigenValues()** obtendremos la matriz con los autovalores y autovectores de las diferentes capas. Nosotros solo tendremos en cuenta las dos primeras capas para obtener $\lambda 1$ y $\lambda 2$, **harrisDetector()** será la que se encargue de ello:

```
/**
* Obtención de la matriz con valores potenciales de puntos harris mediante la fórmula
Det(@harris) - k*Traza(@harris)^2 siendo @harris la matriz de datos obtenidos anteriormente con
la llamada a la funcion cornerEigenValsAndVecs()
*/
void harrisDetector(Mat &harris, float k){
        float l1, l2, value;
        Mat aux = Mat::zeros(harris.size(), CV_32FC1);
        for( int i = 0; i < harris.cols; i++ ){
                for( int j = 0; j < harris.rows; j++ ){
                l1 = harris.at<Vec6f>(j,i)[0];
                l2 = harris.at<Vec6f>(j,i)[1];
                value =  l1*l2 - k*pow((l1+l2),2);
                aux.at<float>(j,i) = value;
          }
        }
 harris = aux;
}
```

Aquí no solo se obtienen los autovalores sino que se aplica una fórmula para la generación de los nuevos valores Harris en una nueva matriz que será la que se usará posteriormente:

$$value = Det(H) - k * Traza(H)^2$$

Nos queda la supresión de no máximos que se realiza en la función **supressNoMax()** donde dada la matriz anterior de valores Harris y el tamaño del entorno, elimina aquellos puntos que no son determinantes:

```
/**
* Supresión de los no máximos para quedarnos con los puntos más relevantes de @matrix en
función del tamaño del entorno y la obtención de la imagen binaria con los puntos buenos.
*/
Mat supressNoMax(Mat &matrix, int window_size){
        Mat aux, res(matrix.size(), CV_8UC1, 255);
        res.col(res.cols-1) = 0;
        res.row(res.rows-1) = 0;
        res.col(0) = 0;
        res.row(0) = 0;
```

```
	int cnt = 0;
	for(int i=0; i <= matrix.cols - window_size; i++){
		for(int j=0; j <= matrix.rows - window_size; j++){
			if(res.at<uchar>(j+window_size/2, i+window_size/2) == 255){
				aux = matrix(Rect(i, j, window_size, window_size));
				applyLocalMax(aux, res, j, i);
				cnt++;
			}
		}
	}
	return res;
}
```

Recorre la matriz de valores Harris generando las regiones de interés y llamando a la función **applyLocalMax()**, que es la encargada de comprobar si el valor central de dicho entorno es máximo local. Tras recorrer la matriz y todas las regiones de interés tendríamos una nueva con los puntos relevantes marcados con los valores 255.

```
/**
* Dada la matriz de entorno @matrix y las coordenadas del punto central de este, almacenar en la imagen binaria el valor de 255 y el resto 0 si dicho valor en las coordenadas @x e @y es el máximo
local, en caso contrario ponerlo a 0.
*/
void applyLocalMax(Mat matrix, Mat &res, int x, int y){
	int central = matrix.rows/2;
	if(isLocalMax(matrix, central)){
		for(int i=0; i < matrix.cols; i++){
			for(int j=0; j < matrix.rows; j++){
				res.at<uchar>(j+x,i+y) = 0;
			}
		}
		res.at<uchar>(x+central,y+central) = 255;
	}
	else res.at<uchar>(x+central,y+central) = 0;
}
```

Esta función es la que comprueba si dada una región de interés sus coordenadas, el valor central es el máximo local, y de ser así se guardaría en una nueva imagen binaria el valor 255 en la misma posición y el resto de valores del entorno se pondría a 0; en caso contrario la casilla central valdría 0. La función booleana **isLocalMax()** es la que se encarga de la comparación, puede verse más abajo en funciones auxiliares.

Terminada la supresión de no máximos el paso siguiente es la extracción de los valores (x, y, escala) y refinar su posición. Para esto primero obtener la estructura de datos que vamos a guardar en un vector de Mat con ayuda de la función **getHarrisData()**:

```
/**
* Obtención de la estructura de datos usada para tratar los puntos harris(coordenadas, valor, escala) de @matrix.
```

```
*/
vector<Mat> getHarrisData(Mat matrix, Mat binary, int level){
	vector<Mat> data;
	float value;
	int cnt = 0;
	Mat harrisPoints, harrisX, harrisY, scale;
	for(int i = 0; i < matrix.cols; i++){
		for(int j = 0; j < matrix.rows; j++){
			if(binary.at<uchar>(j, i) == 255){
				value = matrix.at<float>(j, i);
				harrisPoints.push_back(value);
				harrisX.push_back(j);
				harrisY.push_back(i);
				scale.push_back(1/pow(2,level));
				cnt++;
			}
		}
	}
	data.push_back(harrisPoints);
	data.push_back(harrisX);
	data.push_back(harrisY);
	data.push_back(scale);
	return data;
}
```

Dependiendo de los valores 255 que hemos obtenido de la supresión de no máximos y almacenados en la imagen binaria, vamos a ir buscándolos en la posición correspondiente de la matriz de Harris que teníamos. En el vector de Mat tendremos de momento, 4 matrices con el valor del punto Harris, coordenada X, coordenada Y y su escala.

Todo esto va a ser necesario para la función **drawCircleAndOrientation()** que es la encargada de interpretar todos estos datos y pintar los círculos y líneas con su orientación en la imagen que se ha proporcionado:

```
/**
* Generación de los círculos y dirección del gradiente de la imagen @original teniendo en cuenta el radio @radius, nivel @level y la estructura de datos de los puntos Harris (x, y, valor) en @matrix
*/
void drawCircleAndOrientation(vector<Mat> &matrix, Mat &original, Mat img_gray, int radius, int level){
	Mat aux, gx, gy, angles, mag;
	int totalPoints = 0;
	float counterMax, harrisX, harrisY, harrisdX, harrisdY;
	vector<Point2f> corners;

	switch(level){
		case 0: counterMax = 0.7;
		break;
```

```
                case 1: counterMax = 0.2;
                break;
                case 2: counterMax = 0.1;
                break;
        }

        getBestPoints(matrix, img_gray, corners, totalPoints);

        Sobel(img_gray, gx, CV_32FC1, 1, 0);
        Sobel(img_gray, gy, CV_32FC1, 0, 1);
        phase(gx, gy, angles);

        for(int i = 0; i < 1500*counterMax && i < totalPoints ; i++){
            harrisX=corners[i].y;
            harrisY=corners[i].x;

            float ang = angles.at<float>(harrisX,harrisY);
           float dx=radius*cos(ang);
           float dy=radius*sin(ang);

           harrisdX = ( corners[i].y ) * pow(2,level) + dx;
           harrisdY = ( corners[i].x ) * pow(2,level) + dy;

          circle(original, Point2f(harrisY*pow(2,level), harrisX*pow(2,level)), radius,
Scalar(0,0,255), 1, CV_AA , 0);
           line(original, Point2f(harrisY*pow(2,level),harrisX*pow(2,level)),Point2f(harrisdY,
harrisdX),Scalar(0,0,255), 1, CV_AA);
        }
}
```

Esta función es amplia por lo que vamos a explicarla en diferentes pasos:

1. Dependiendo de la escala de la imagen generamos el parámetro de ponderación de los puntos que se van a seleccionar; si se van a usar 3 imágenes con diferentes escalas una opción válida es 70%,20%,10% de los puntos respectivamente.
2. Obtener los mejores puntos, esto se hace en la función **getBestPoints()**:

```
/**
* Obtención de los mejores puntos @corners y cantidad @totalPoints, tras ordenar de mayor a
menor los
puntos Harris almacenados en @matrix.
*/
void getBestPoints(vector<Mat> matrix, Mat img_gray, vector<Point2f> &corners, int
&totalPoints){
        Mat aux;
        float harrisX, harrisY;
        Size winSize = Size( 5, 5 ), zeroZone = Size( -1, -1 );
        TermCriteria criteria = TermCriteria( TermCriteria::EPS + TermCriteria::COUNT, 40, 0.001
```

```
);

        sortIdx(matrix[0],aux,CV_SORT_EVERY_COLUMN + CV_SORT_DESCENDING);

        for(int i = 0; i < matrix[0].rows && matrix[0].at<float>(aux.at<int>(i,0),0) > 0; i++){
                harrisX = matrix[1].at<int>(aux.at<int>(i,0),0);
                harrisY = matrix[2].at<int>(aux.at<int>(i,0),0);
                 Point2f point(harrisY, harrisX);
                corners.push_back(point);
                totalPoints++;
        }
        cornerSubPix(img_gray, corners, winSize, zeroZone, criteria );
}
```

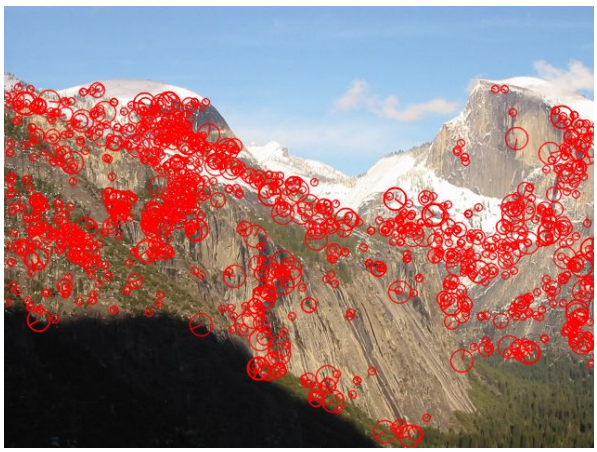
Esta función se ejecuta antes de acotar los puntos que se van a usar en cada escala, ya que se trata de ordenar de forma descendente los valores de éstos con **sortIdx()**. Una vez que estén ordenados se llama a **cornerSubPix()** que devuelve los índices de los puntos de forma más exacta y que posteriormente se pinten con mayor precisión.

3. Obtención de la derivada en X y en Y con Sobel. Una vez que se obtienen se llama a la función **phase()**, pasándole ambas derivadas y escribe una matriz con los ángulos en radianes que forman.

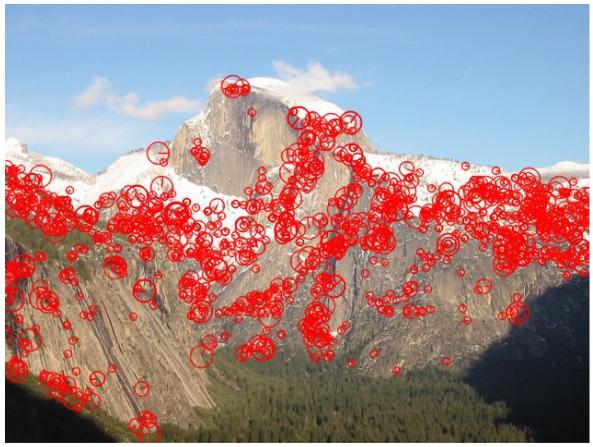
4. Hasta aquí tenemos un vector de puntos en coma flotante ordenado por su valor de mayor a menor por lo que ya se podrían pintar estos puntos. Con un bucle recorriendo este vector y sacando los primeros puntos que queremos pintar (conforme al documento T2 entiendo que alrededor de 1500 puntos en total por lo que se procede a elegir 70% ,20% y 10% de 1500 para cada escala respectivamente). Para pintar círculos llamamos a la función de OpenCV **circle()** pasándole como parámetro el punto en cuestión. Para la dirección del gradiente **line()** pasándole el punto y el punto desplazado formando el ángulo que obtenemos en **phase()**.

Resultados obtenidos

Aplicando el criterio usado para la obtención de puntos Harris y su representación con círculos y líneas para la orientación generamos las siguientes imágenes de yosemite:



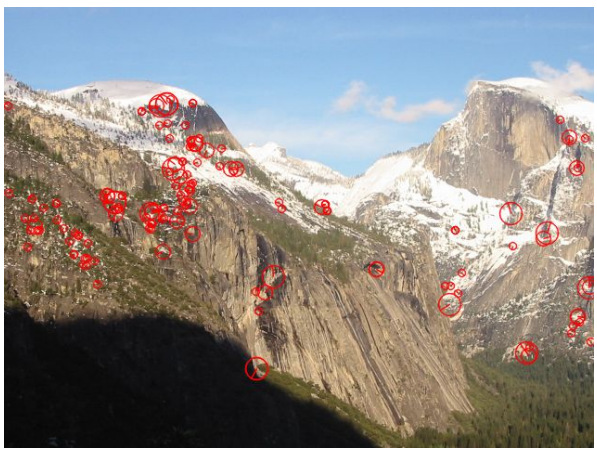
Obtención puntos Harris yosemite1.jpg (p=1500)



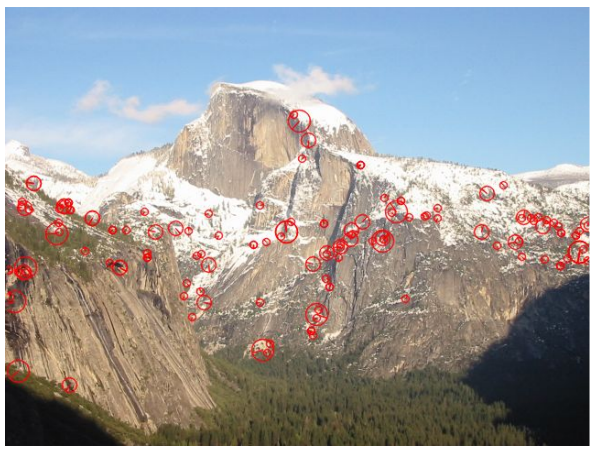
Obtención puntos Harris yosemite2.jpg (p=1500)

Pueden observarse los círculos con radio diferente, cuanto mayor radio, dicho punto se ha pintado desde una escala menor. También podríamos decir que los resultados son buenos ya que hay muchos puntos que estarían en correspondencia con la segunda imagen como por ejemplo se puede ver por la cima de la montaña.

Se puede ver de forma más claro si acotamos más puntos, para 150:



Obtención puntos Harris yosemite1.jpg (p=150)



Obtención puntos Harris yosemite2.jpg (p=150)

Obtención de correspondencias con AKAZE/KAZE

Para este ejercicio se ha usado unos detectores implementados en OpenCV que son AKAZE y KAZE para obtener los keypoints y descriptores de dos imágenes para posteriormente realizar un match entre ellas con parejas de puntos que tengan la menor distancia. La función por la que empezaremos hablando es **myDetector()**.

Se va a mostrar a continuación el código de toda la función que define este ejercicio, al ser tan larga se procederá a explicar paso a paso y las subfunciones usadas:

```
/**
* Uso del detector AKAZE/KAZE y obtener los keypoints y descriptores para @img1 y @img2
donde posteriormente se buscará el match entre ellos según el número de correspondencias
@correspondences. Basicamente genera la secuencia de correspondencias y el panorama de juntar
éstas en las dos imágenes.
*/
vector<Mat> myDetector(Mat img1, Mat img2, int correspondences, int mode = BOTH, int type =
TYPE_AKAZE ){
	vector<Mat> result;
	Mat img1_gray, img2_gray, descriptors1, descriptors2, correspondence, mosaic;
	vector<KeyPoint> keyP1, keyP2, good_keyP1, good_keyP2;
	vector<Point2f> points1, points2;
	vector<DMatch> good_matches, matches;
	int index, queryIdx, trainIdx;

	cvtColor(img1, img1_gray, COLOR_BGR2GRAY);
	cvtColor(img2, img2_gray, COLOR_BGR2GRAY);

	if(type == TYPE_AKAZE){
		Ptr<AKAZE> detector = AKAZE::create();
		detector->detectAndCompute(img1_gray, noArray(), keyP1, descriptors1);
		detector->detectAndCompute(img2_gray, noArray(), keyP2, descriptors2);
		BFMatcher matcher(NORM_HAMMING, true);
		matcher.match(descriptors1, descriptors2, matches);
	}
	else if(type == TYPE_KAZE){
		Ptr<Feature2D> detector = KAZE::create();
		detector->detectAndCompute(img1_gray, noArray(), keyP1, descriptors1);
		detector->detectAndCompute(img2_gray, noArray(), keyP2, descriptors2);
		Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create("BruteForce");
		matcher->match(descriptors1, descriptors2, matches);
	}

	Mat distances(matches.size(), 1, CV_32FC1), aux;
	for( unsigned i = 0; i < matches.size(); i++ )
		distances.at<float>(i,0) = matches[i].distance;

	sortIdx(distances,aux,CV_SORT_EVERY_COLUMN + CV_SORT_ASCENDING);
	for( int i = 0; i < correspondences; i++ ){
		index = aux.at<int>(i,0);
		good_matches.push_back(matches[index]);
```

```
                queryIdx = matches[index].queryIdx;
                trainIdx = matches[index].trainIdx;

                good_keyP1.push_back(keyP1[queryIdx]);
                good_keyP2.push_back(keyP2[trainIdx]);

                good_matches[i].queryIdx = i;
                good_matches[i].trainIdx = i;

                points1.push_back(good_keyP1[i].pt);
                points2.push_back(good_keyP2[i].pt);

        }
        if(mode == BOTH || mode == ONLY_CORRESPONDENCES){
                drawMatches(img1, good_keyP1, img2, good_keyP2, good_matches,
correspondence);
                result.push_back(correspondence);
        }

        if(mode == BOTH || mode == ONLY_PANORAMA){
                blendImages(img1, img2, mosaic, points1, points2);
                result.push_back(mosaic);
        }
        return result;
}
```

1. Esta función está implementada de forma que en función de un parámetro opcional podamos elegir qué detector usar, si **KAZE** o **AKAZE** (AKAZE por defecto).
2. Tras haber seleccionado el detector se obtienen los keypoints y descriptores de cada una de las imágenes por separado.
3. Inicializamos el objeto de tipo **BFMatcher**, Bruteforce, y con el parámetro crosscheck a verdadero, también usamos **NORM_HAMMING** ya que AKAZE usa por defecto descriptores binarios. Con este objeto llamamos a la función implementada por OpenCV **match()** pasando como parámetro los descriptores de cada imágenes y obteniendo las correspondencias en un vector de tipo **DMatch**.
4. Como antes hemos mencionado queremos quedarnos con las parejas de puntos que tienen menor distancias, es por ello que creamos un vector auxiliar donde se guardan las distancias de las correspondencias y se ordenan con ayuda de **sortIdx()**. De esta forma tenemos un vector de índices que nos indica qué posiciones del vector de correspondencias tienen menor distancia.
5. Un parámetro que recibe la función **myDetector()** es *correspondences*, que indica el número de correspondencias que queremos tener en cuenta para el par de imágenes.
6. Crearemos un nuevo vector donde se guardarán las mejores correspondencias, tantas como la variable mencionada *correspondences* defina, y otros dos vectores para los keypoints de éstas. Es importante indicar que aquí también se crea y

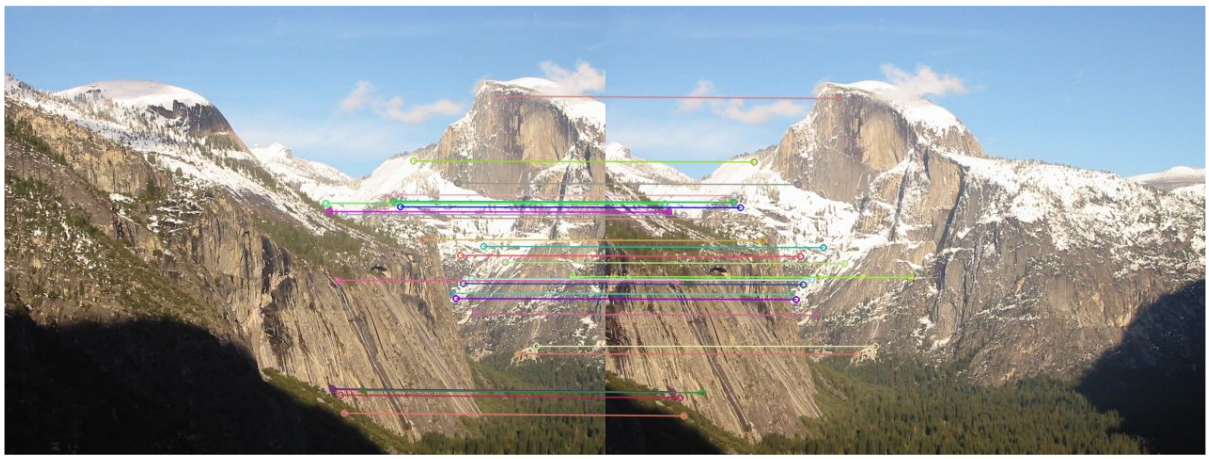
guarda el valor de los puntos de los keypoints buenos en dos vectores(para los de la primera y segunda imagenrespectivamente) de puntos tipo **Point2f** que serán usados y explicados posteriormente para la parte de panoramas del ejercicio 3.

7. Tenemos todo lo necesario, las dos imágenes, las mejores correspondencias y los keypoints de estas, por lo que se procede a pintarlas mediante la función **drawMatches()**.

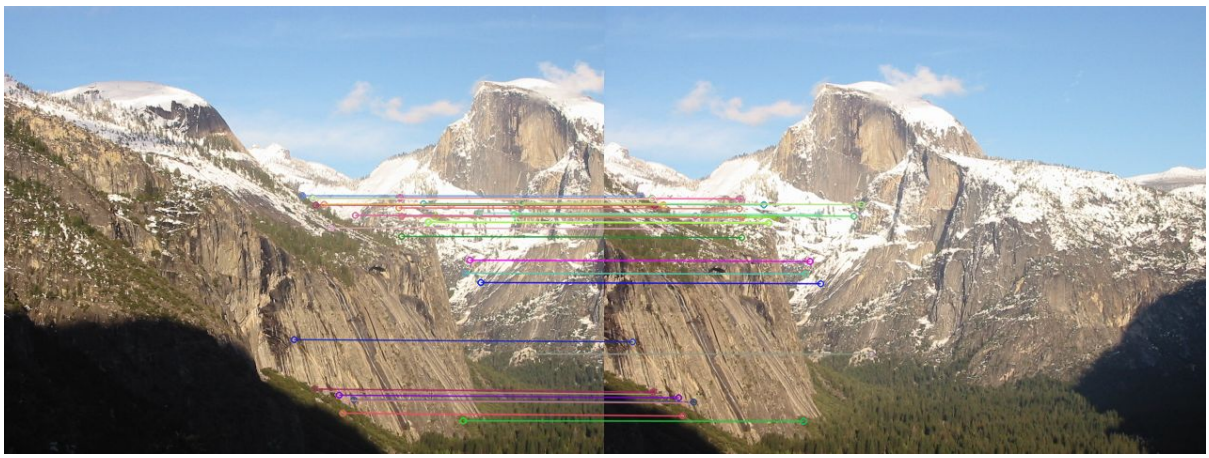
Esta función también se llama en el ejercicio 3, es por ello que para poder definir el objetivo de la llamada a esta función se ha creado un enumerado ONLY_CORRESPONDENCES y ONLY_PANORAMA, el primero para indicar que solo queremos mostrar el match y las dos imágenes y el segundo para indicar que solo queremos mostrar el panorama que explicaremos más adelante. El parámetro BOTH es para obtener ambas imágenes resultantes.

Resultados obtenidos

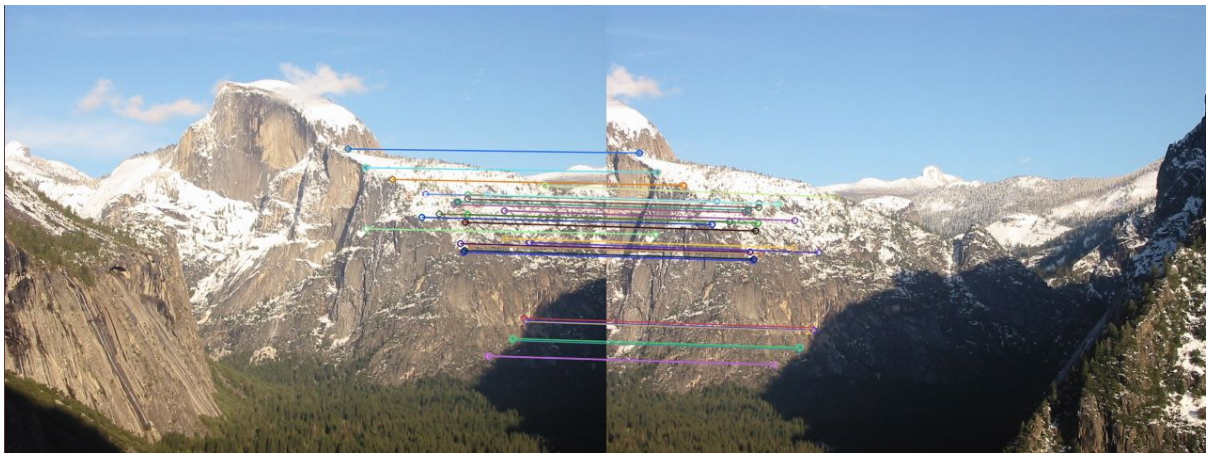
Siguiendo los pasos anteriores y eligiendo un número de correspondencias de 30 para las imágenes de yosemite 1, 2 y 3 con AKAZE/KAZE.



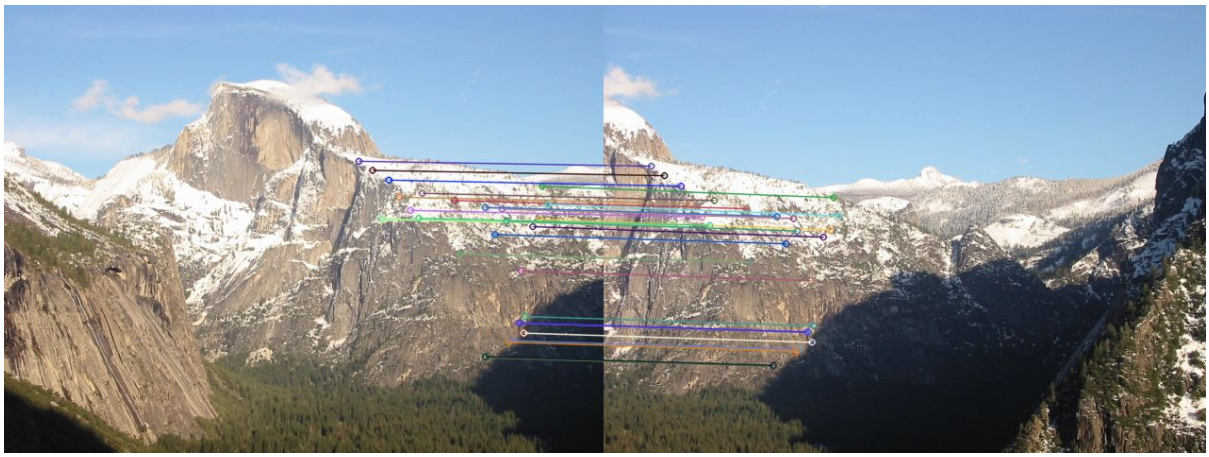
Obtención de correspondencias entre las imágenes yosemite 1 y 2 con AKAZE Bruteforce+Croscheck c=30 time = 0.59s



Obtención de correspondencias entre las imágenes yosemite 1 y 2 con KAZE Bruteforce+Croscheck c=30 time=1.59s



Obtención de correspondencias entre las imágenes yosemite 2 y 3 con AKAZE Bruteforce+Croscheck c=30 time=0.53s



Obtención de correspondencias entre las imágenes yosemite 2 y 3 con KAZE Bruteforce+Croscheck c=30 time=1.62s

Se pueden observar algunas pequeñas diferencias entre los detectores AKAZE/KAZE, una bastante evidente en las dos primeras imágenes, con AKAZE señala una parte de la cima de la montaña como correspondencia mientras que no ocurre con KAZE, pero en general las correspondencias son similares. La diferencia fundamental radica en que AKAZE tarda mucho menos que KAZE en realizar la correspondencia, por ejemplo, para las de yosemite 1 y 2 AKAZE ha sido casi 3 veces más rápido que KAZE (0.59s frente 1.59s).

Para comprobar ambos detectores en el código se ha dejado por defecto AKAZE para las parejas de puntos de yosemite 1 y 2 y KAZE para las parejas de puntos de yosemite 2 y 3.

Creación de un mosaico a partir de N imágenes

Se ha implementado una función que con ayuda de la función de detectores AKAZE/KAZE del ejercicio anterior, genera un mosaico de N imágenes, siendo N>3. La función general que define este ejercicio es **getPanorama()** donde dado un vector de imágenes y el número de correspondencias que queremos coger, obtiene el mosaico o panorama de todas esas imágenes juntas:

```
/**
* Dado el vector de imágenes @images que contiene las diferentes imagenes de un mosaico y
según
el número de correspondencias @correspondences obtener la imagen resultante que es un
panorama de
dichas imágenes.
*/
Mat getPanorama(vector<Mat> images, int correspondences, int type = TYPE_AKAZE){
	int n_images = images.size();
	Mat aux = images[0], panorama;
	for(int i = 0; i < n_images - 1; i++){
		panorama = myDetector(aux, images[i+1], correspondences, ONLY_PANORAMA,
type)[0];
		aux = panorama;
	}
	return panorama;
}
```

Esta función recorre todas las imágenes cogiendo la primera y la siguiente del vector para obtener sus correspondencias llamando a la función del ejercicio anterior **myDetector()**. A continuación se va a mostrar el fragmento de código que se necesita:

```
/**
* Uso del detector AKAZE/KAZE y obtener los keypoints y descriptores para @img1 y @img2
donde posteriormente
se buscará el match entre ellos según el número de correspondencias @correspondences.
Básicamente genera la secuencia de correspondencias y el panorama de juntar éstas en las dos
imágenes.
*/
vector<Mat> myDetector(Mat img1, Mat img2, int correspondences, int mode = BOTH, int type =
TYPE_KAZE ){

	. . . . . .

	if(mode == BOTH || mode == ONLY_PANORAMA){
		blendImages(img1, img2, mosaic, points1, points2);
		result.push_back(mosaic);
	}
	return result;
	. . . . . .
}
```

Como dijimos esta función tenía diferentes opciones dependiendo de lo que queríamos, como en este caso queremos generar el mosaico, la variable que pasaremos a la función será ONLY_PANORAMA. Como puede verse en ese código se llama a la función **blendImages()**:

```
/**
* Con la lista de puntos @p1 y @p2 correspondientes a los keypoints de las images @img1 e
@img2 respectivamente encontrar su homografía y guardar en @mosaic la imagen resultante.
*/
void blendImages(Mat img1, Mat img2, Mat &res, vector<Point2f> p1, vector<Point2f> p2){
	Mat homography1, homography2, homography_final;

	res = Mat::zeros(2*(img1.rows+img2.rows), 2*(img1.cols+img2.cols), img1.type());

	homography1 = findHomography(p2, p1, CV_RANSAC, 1);
	homography1.convertTo(homography1, CV_32FC1);
	homography2 = Mat::eye(3, 3, CV_32FC1);
	homography2.at<float>(Point(2,0)) = (img1.cols+img2.cols)/2;
	homography2.at<float>(Point(2,1)) = 100;

	homography_final = homography2 * homography1;
	warpPerspective(img1, res, homography2, Size(res.cols, res.rows), INTER_LINEAR,
BORDER_TRANSPARENT);
	warpPerspective(img2, res, homography_final, Size(res.cols, res.rows), INTER_LINEAR,
BORDER_TRANSPARENT);

	omitEdges(res);
}
```

Esta función recibe dos vectores correspondientes a los puntos de los keypoints de cada imagen y las dos imágenes. Se crea una nueva matriz que será en la que se pinte el panorama generado por ambas imágenes. Esta matriz en un principio está inicializada a 0 y lo suficientemente grande como para evitar un desbordamiento a lo hora de juntar las imágenes.

A continuación tenemos que encontrar la homografía por lo que buscaremos que puntos de la imagen 2 corresponden con la imagen 1. Tendremos luego otra matriz que corresponde a la homografía con la que trasladaremos la imagen resultante para evitar que si comenzamos con una imagen central en el vector de imágenes, no se pueda ver correctamente en el canvas el panorama.

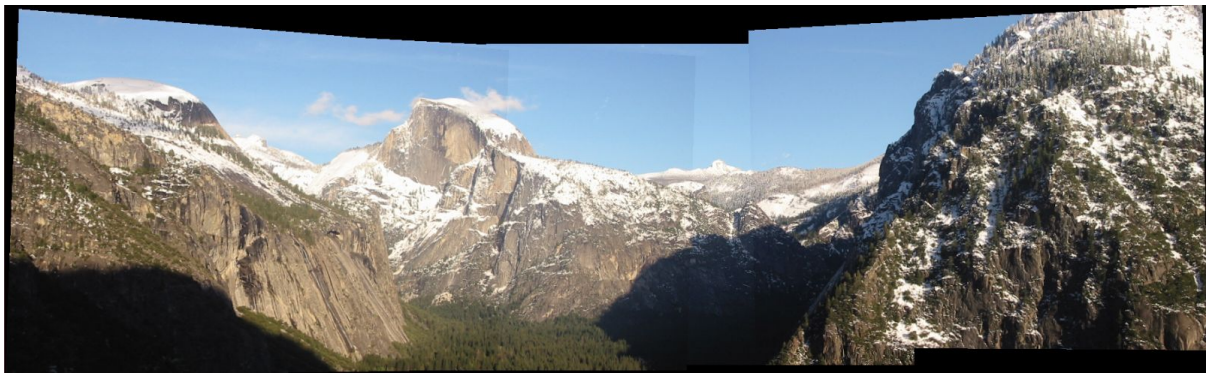
Llamamos dos veces a la función **warpPerspective()** primero para la transformación de la imagen 1 (aplicando homografía 2) y luego para la imagen 2 (aplicando la homografía final resultante de la 1 y 2).

Finalmente usamos la función auxiliar **omitEdges()** para quitar las filas y columnas de 0 y que la imagen no sea de un tamaño excesivo.

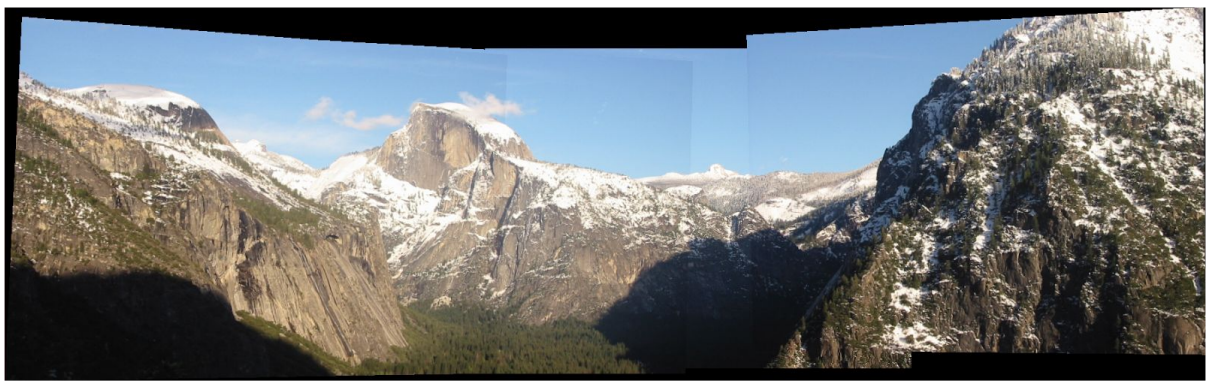
La función **getPanorama()** termina devolviendo la imagen o mosaico generado por las N imágenes del vector.

Resultados obtenidos

Usando 100 parejas de puntos con el detector de AKAZE, y cambiando el orden de las 4 imágenes de yosemite para que la central sea la primera (se ha cogido la 3º tomada de izquierda a derecha) generamos un primer panorama y el segundo es el correspondiente a las 10 imágenes mosaico con 50 parejas de puntos (la imagen central es la 4º tomada de izquierda a derecha):

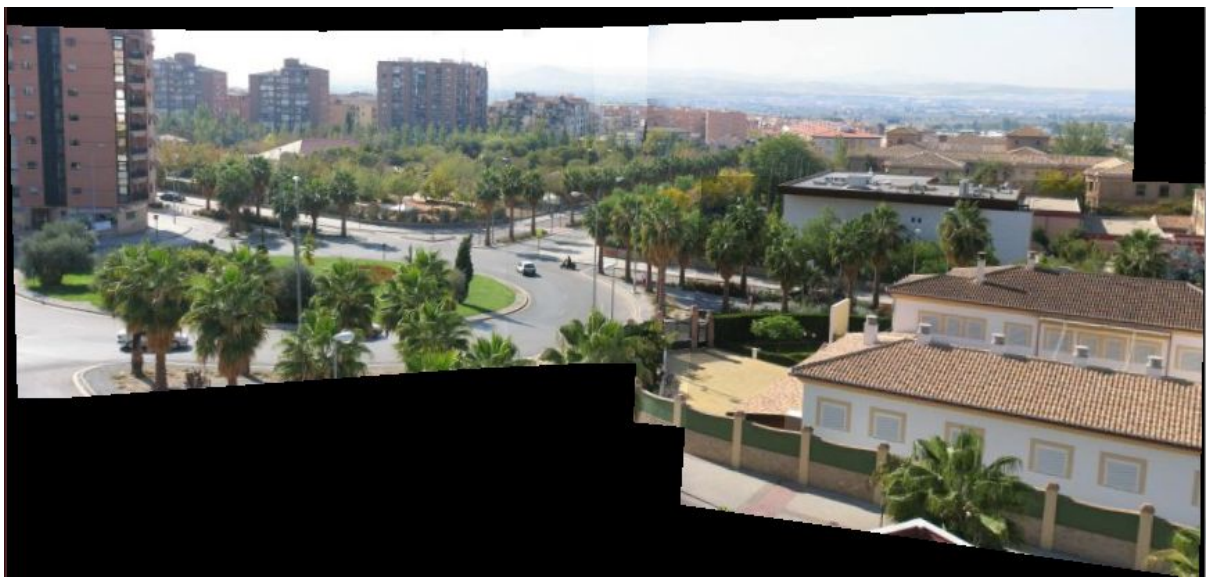


Obtención del panorama yosemite de las 4 imágenes en el orden 3,2,4,1 con AKAZE Bruteforce+Croscheck c=100 time=3.69s



Obtención del panorama yosemite de las 4 imágenes en el orden 3,2,4,1 con KAZE Bruteforce+Croscheck c=100 time=8.58s

Panorama mosaico de las 10 imágenes en el orden 4,3,5,6,7,2,1,8,9,10 AKAZE Bruteforce+Croscheck c=50 time=2.19s



Panorama mosaico de las 10 imágenes en el orden 4,3,5,6,7,2,1,8,9,10 AKAZE Bruteforce+Croscheck c=50 time=6.01s

Se ha probado a crear el panorama con ambos detectores AKAZE/KAZE y los resultados son muy buenos, las imágenes se han juntado correctamente y no se aprecian diferencias para ambos detectores. La diferencia, tal y como se ha mencionado en el ejercicio anterior, se encuentra en el tiempo de ejecución ya que KAZE es más lento que AKAZE.

Funciones auxiliares

A continuación se muestran aquellas funciones que aunque tengan una función importante y necesaria para el funcionamiento del código, tiene un objetivo concreto que no es necesario explicar debido a que es trivial:

```
/**
* Comprobación de si el pixel con coordenadas x e y como @central es un máximo local de la matriz de
entorno @matrix. Devuelve true en caso afirmativo, false en caso contrario.
*/
bool isLocalMax(Mat matrix, int central){
	bool is_max = true;
	float max = matrix.at<uchar>(central,central);
	for(int i=0; i < matrix.cols; i++){
		for(int j=0; j < matrix.rows; j++){
			if(max < matrix.at<uchar>(j,i)) is_max = false;
		}
	}
	return is_max;
}

/**
* Dada una imagen @img, eliminar los bordes que contienen negros para hacer más visible y menos pesada la imagen.
*/
void omitEdges(Mat &img){
	Mat img_gray;
	cvtColor(img, img_gray, COLOR_BGR2GRAY);
	img_gray.convertTo(img_gray, CV_8UC1);
	int cnt = 0, down_row = 0, up_row = img_gray.rows-1, right_col, left_col;
	for(int i = 0; i < img_gray.cols; i++){
		for(int j = 0; j < img_gray.rows; j++){
			int value = (int) img_gray.at<uchar>(j, i);
			if( value > 0){
				if(cnt == 0){
					left_col = i;
					cnt++;
				}
				if(down_row < j) down_row = j;

				if(up_row > j)  up_row = j;

				right_col = i;
			}
```

```
                }
        }
        img = img(Rect(left_col, up_row, right_col-left_col+1, down_row-up_row+1));
}


/**
* Muestra las imágenes que se encuentran en el vector @images en una ventana llamada
* como se establece en @window. Para cambiar de imagen hay que pulsar cualquier tecla.
*/
void showImages(vector<Mat> images, string window = "Ventana"){
        for(unsigned i=0; i<images.size(); i++){
                imshow(window, images[i]);
                waitKey(0);
        }
}

/**
* Libera la memoria de las imágenes que están en el vector @images y a continuación
* la memoria de dicho vector.
*
*/
void remove(vector<Mat> &images){
        for(unsigned i=0; i<images.size(); i++){
                images[i].release();
        }
        images.clear();
}
```