

**Visión por Computador (2016-2017)**  
GRADO EN INGENIERÍA INFORMÁTICA  
UNIVERSIDAD DE GRANADA

---

Informe Práctica 2: opción de 10 puntos

---

Laura Tirado López

18 de noviembre de 2016

## Índice

<b>1. Detección de puntos Harris multiescala</b>	<b>3</b>
1.1. Lista potencial de puntos Harris . . . . .	3
1.2. Refinamiento de posiciones a nivel de subpíxel . . . . .	8
1.3. Orientación de los puntos Harris . . . . .	8
1.4. Resultados . . . . .	10
<b>2. Detectores KAZE y AZAKE</b>	<b>13</b>
2.1. Resultados del detector KAZE . . . . .	15
2.2. Resultado del detector AKAZE . . . . .	16
<b>3. Creación de un mosaico con N imágenes</b>	<b>16</b>
3.1. Resultados . . . . .	20

## Índice de figuras

1.1. Imagen original junto con puntos Harris . . . . .	12
1.2. Imagen original junto con puntos Harris y su orientación aproximada . . .	13
2.1. Correspondencias con detector KAZE . . . . .	15
2.2. Correspondencias con detector AKAZE . . . . .	16

# 1. Detección de puntos Harris multiescala

## 1.1. Lista potencial de puntos Harris

Primero tenemos que extraer la lista potencial de puntos Harris en una imagen. En este caso vamos a utilizar varias escalas. Para ello construimos una pirámide Gaussiana a partir de la imagen con 4 escalas con las funciones que implementamos en la práctica anterior.

En cada nivel de la pirámide utilizamos la función *cornerEigenValsAndVecs()* con la que extreemos los mapas de autovalores. A partir de estos mapas de autovalores calculamos el valor de Harris con los valores  $\lambda_1$  y  $\lambda_2$  con la siguiente fórmula:  $valorHarris = \lambda_1 * \lambda_2 - k(\lambda_1 + \lambda_2)^2$ . En este caso le hemos dado a  $k$  el valor 0,04. Además, he añadido un umbral para quedarnos sólo con los puntos Harris que estén por encima de ese umbral. De esta forma evitamos coger puntos que no sean significativos y eliminar ruido.

A continuación, hacemos la supresión de valores no máximos. Para la supresión de no máximos primero debemos comprobar si el valor central de un entorno es o no máximo local. Si dicho valor es máximo local, el resto de elementos del entorno toman el valor 0. En caso contrario, el valor central tomaría el valor 0.

```
1  bool maximoLocal(Mat entorno, Mat entorno_bin){  
2  
3      int centro_x = (entorno.rows)/2;  
4      int centro_y = (entorno.cols)/2;  
5  
6      bool maximo = true;  
7  
8      for(int i=0; i < entorno.rows && maximo; ++i){  
9          for(int j=0; j < entorno.cols && maximo; ++j){  
10              if(!(i == centro_x && j == centro_y) &&  
11                  entorno_bin.at<float>(i,j) != 0)  
12                  if(entorno.at<float>(centro_x,  
13                      centro_y) <= entorno.at<float>(i,j)// && entorno_bin.at<float>(i,j) != 0)  
14                      maximo = false;  
15      }  
16      return maximo;  
17  }  
18  
19  void modificarCeros(Mat & m, int x, int y, int entorno){  
20  
21      Mat resultado;  
22  }
```

```

23     for(int i=x-entorno/2; i < x+entorno/2+1; ++i){
24         for(int j=y-entorno/2; j < y+entorno/2+1; ++j){
25
26             if(!(i == x && j == y))
27                 m.at<float>(i,j) = 0;
28         }
29     }
30 }
31
32 Mat supresionNoMaximos(Mat harris, int entorno){
33
34     Mat seleccionados = Mat::ones(harris.rows, harris.cols,
35     CV_32FC1)*255;
36     int longitud = entorno/2;
37
38     for(int i=longitud; i < harris.rows-longitud; ++i){
39         for(int j=longitud; j < harris.cols-longitud; ++j)
40             {
41
42                 Mat roi = harris(Rect(j-longitud, i-
43                     longitud, entorno, entorno));
44                 Mat roi_bin = seleccionados(Rect(j-
45                     longitud, i-longitud, entorno, entorno)
46                     );
47
48                 if(maximoLocal(roi, roi_bin)){
49                     modificarCeros(seleccionados,i,j,
50                         entorno);
51                 }
52
53             seleccionados.at<float>(i,j) = 0;
54         }
55     }
56
57     return seleccionados;
58 }

```

Ahora ordenamos los puntos Harris de mayor a menor para luego seleccionar poder seleccionar N puntos de mayor valor. Como queremos guardar la posición de cada punto, su orientación y la escala a la que pertenece además de su valor he creado una estructura *PuntoHarris* para poder almacenar toda esta información. Además para poder ordenar los valores, he creado un operador para poder utilizar la función *sort()*.

```

1 struct PuntoHarris{
2
3     float valor;
4     Point2f p;

```

```

5     int escala;
6     int orientacion;
7 }
8
9 bool operador(PuntoHarris a, PuntoHarris b){
10
11     return (a.valor > b.valor);
12 }
```

Después de ordenar los puntos seleccionamos el número de puntos que consideremos.

```

1 void calcularPuntosHarris (vector<Mat> &piramide, vector<Mat> &
2     harris, vector<PuntoHarris> &p_harris, int escala, int npuntos)
3 {
4
5     vector<Mat> derivadas_x, derivadas_y, valores_harris,
6         imagenes_gris;
7
8     for(int i=0; i < piramide.size(); ++i)
9         piramide.at(i).convertTo(piramide.at(i), CV_8U);
10
11    //Para cada escala
12    for(int i=0; i < escala; ++i){
13
14        //Calculamos los mapas de autovalores
15        Mat imagen_gris, harris_aux, dx_aux, dy_aux,
16            abs_dx_aux, abs_dy_aux;
17        cvtColor(piramide.at(i), imagen_gris, CV_BGR2GRAY)
18            ;
19        harris_aux = Mat::zeros(piramide.at(i).size(),
20            CV_32FC(6));
21        cornerEigenValsAndVecs(imagen_gris, harris_aux, 3,
22            5, BORDER_DEFAULT);
23        imagenes_gris.push_back(imagen_gris);
24
25        //Calculamos los valores Harris
26        Mat aux = Mat::zeros(piramide.at(i).size(),
27            CV_32FC1);
28        for(int j=0; j < piramide.at(i).rows; ++j){
29            for(int k=0; k < piramide.at(i).cols; ++k)
30            {
31                float lambda1 = harris_aux.at<
32                    Vec6f>(j,k)[0];
33                float lambda2 = harris_aux.at<
34                    Vec6f>(j,k)[1];
35                float valor_harris = lambda1 *
36                    lambda2 - 0.04f*pow(lambda1+
37                    lambda2,2);
```

```

25             if(valor_harris > 0.001) //<-
26                 Ponemos un umbral para evitar
27                 ruido
28                 aux.at<float>(j,k) =
29                     valor_harris;
30             }
31         }
32         valores_harris.push_back(aux);
33     }
34     //Supresion de no maximos
35     harris.push_back(supresionNoMaximos(aux, 7));
36     //Calculamos las derivadas en x e y con el
37     //operador Sobel
38     Sobel(imagen_gris, dx_aux, CV_16S, 1, 0, 5);
39     convertScaleAbs(dx_aux, abs_dx_aux);
40     derivadas_x.push_back(abs_dx_aux);
41     Sobel(imagen_gris, dy_aux, CV_16S, 0, 1, 5);
42     convertScaleAbs(dy_aux, abs_dy_aux);
43     derivadas_y.push_back(abs_dy_aux);
44 }
45
46 //Seleccionamos los 1500 puntos de mayor de valor de las
47 //distintas escalas
48 vector<PuntoHarris> puntos;
49 PuntoHarris punto;
50 vector<Point2f> puntos_refinados;
51 TermCriteria criteria = TermCriteria(CV_TERMCRIT_EPS +
52                                         CV_TERMCRIT_ITER, 40, 0.001);
53
54 //Guardamos todos los puntos de todas las escalas
55 for(int i=0; i < escala; ++i){
56     for(int j=0; j < harris.at(i).rows; ++j){
57         for(int k=0; k < harris.at(i).cols; ++k){
58             if(harris.at(i).at<float>(j,k) ==
59                 255){
60                 punto.valor =
61                     valores_harris.at(i).at
62                         <float>(j,k);
63                 punto.p.x = (float)k*pow
64                     (2,i);
65                 punto.p.y = (float)j*pow
66                     (2,i);
67                 punto.escala = i;
68                 punto.orientacion = atan(
69                     derivadas_y.at(i).at<
70                         float>(j,k)/derivadas_x
71                         .at(i).at<float>(j,k));
72             }
73         }
74     }
75 }

```

```

60                               puntos.push_back(punto);
61                           }
62                       }
63                   }
64               }
65
66           //Ordenamos en orden ascendente
67           sort(puntos.begin(), puntos.end(), operador);
68
69           //Guardamos los n primeros
70           for(int i=0; i < npuntos; ++i){
71               p_harris.push_back(puntos.at(i));
72               puntos_refinados.push_back(puntos.at(i).p);
73           }
74
75           //Refinamos la posicion de los puntos
76           for(int i=0; i < p_harris.size(); ++i){
77
78               //Guardamos el punto en un vector auxiliar
79               vector<Point2f> p_aux;
80               p_aux.push_back(puntos_refinados.at(i));
81
82               //Utilizamos la imagen de la piramide
83               //correspondiente con el punto escogido
84               if(p_harris.at(i).escala == 0)
85                   cornerSubPix(imagenes_gris.at(0), p_aux,
86                                 Size(5,5), Size(-1,-1), criteria);
87
88               else if(p_harris.at(i).escala == 1)
89                   cornerSubPix(imagenes_gris.at(1), p_aux,
90                                 Size(5,5), Size(-1,-1), criteria);
91
92               else if(p_harris.at(i).escala == 2)
93                   cornerSubPix(imagenes_gris.at(2), p_aux,
94                                 Size(5,5), Size(-1,-1), criteria);
95
96               else
97                   cornerSubPix(imagenes_gris.at(3), p_aux,
98                                 Size(5,5), Size(-1,-1), criteria);
99
100              //Actualizamos el valor de las coordenadas de los
101              //puntos
102              p_harris.at(i).p.x = p_aux.at(0).x;
103              p_harris.at(i).p.y = p_aux.at(0).y;
104
105              p_aux.clear();
106          }
107      }

```

En esta función calculamos también las posiciones a nivel de sub-píxel y las derivadas en  $x$  e  $y$  que se explican en los siguientes apartados.

## 1.2. Refinamiento de posiciones a nivel de subpíxel

Para refinar las posiciones de los puntos de Harris utilizamos la función `cornerSubPix()` con los puntos que hemos calculado en el apartado anterior. Esto lo he implementado en la misma función para calcular los puntos Harris justo después de seleccionar los N puntos, de forma que sólo refinamos los puntos que vamos a utilizar.

## 1.3. Orientación de los puntos Harris

Para calcular la orientación de los puntos Harris tenemos que obtener los valores de las derivadas en  $x$  e  $y$  de los píxeles. Para ello utilizamos la función `Sobel()`, ya que es el operador que emplea `cornerHarris()`. Para obtener la derivada en  $x$  le damos el valor 1 al parámetro  $dx$  y el valor 0 al parámetro  $dy$ . Para obtener la derivada en  $y$  le damos los valores contrarios.

El valor de la derivada en  $x$  coincide con el valor del  $\cos(\alpha)$  y el valor de la derivada en  $y$  con el valor del  $\sin(\alpha)$ , por lo que para calcular el ángulo simplemente tenemos que calcular la arcotangente a partir de estos dos valores.

Para dibujar línea de orientación he utilizado la función `line()`, pasándole como punto inicial las coordenadas del punto Harris y calculando las coordenadas del punto final a partir del radio de las circunferencias con las que señalamos los puntos harris, las coordenadas del punto inicial y el ángulo de orientación.

```
1 void mostrarPuntosHarris(Mat imagen, vector<PuntoHarris> &puntos,
2                           bool orientacion){
3
4     float x2, y2;
5
6     //Para cada escala mostramos los puntos de un color y un
7     //tamaño distintos
8     for(int i=0; i < puntos.size(); ++i){
9
10         if (puntos.at(i).escala == 0){
11
12             //Dibujamos el círculo con centro en las
13             //coordenadas del punto de Harris
14             circle(imagen, puntos.at(i).p, 10*(1+puntos.
15                 at(i).escala), Scalar(0,0,255));
16
17             if(orientacion){
```

```

14          //Dibujamos un segmento que nos
15          indica la direccion del
16          gradiente
17          x2 = puntos.at(i).p.x + cos(puntos
18          .at(i).orientacion)*10*(1+
19          puntos.at(i).escala);
20          y2 = puntos.at(i).p.y + sin(puntos
21          .at(i).orientacion)*10*(1+
22          puntos.at(i).escala);
23          line(imagen,puntos.at(i).p, Point(
24          x2,y2), Scalar(0,0,255));
25      }
26
27      else if (puntos.at(i).escala == 1){
28
29          //Dibujamos el circulo con centro en las
30          coordenadas del punto de Harris
31          circle(imagen,puntos.at(i).p,10*(1+puntos.
32          at(i).escala), Scalar(0,255,0));
33
34          if(orientacion){
35              //Dibujamos un segmento que nos
36              indica la direccion del
37              gradiente
38              x2 = puntos.at(i).p.x + cos(puntos
39              .at(i).orientacion)*10*(1+
40              puntos.at(i).escala);
41              y2 = puntos.at(i).p.y + sin(puntos
42              .at(i).orientacion)*10*(1+
43              puntos.at(i).escala);
44              line(imagen,puntos.at(i).p, Point(
45              x2,y2), Scalar(0,255,0));
46          }
47
48      else if (puntos.at(i).escala == 2){
49
50          //Dibujamos el circulo con centro en las
51          coordenadas del punto de Harris
52          circle(imagen,puntos.at(i).p,10*(1+puntos.
53          at(i).escala), Scalar(255,0,0));
54
55          if(orientacion){
56              //Dibujamos un segmento que nos
57              indica la direccion del
58              gradiente
59              x2 = puntos.at(i).p.x + cos(puntos
60              .at(i).orientacion)*10*(1+
61              puntos.at(i).escala);
62              y2 = puntos.at(i).p.y + sin(puntos
63              .at(i).orientacion)*10*(1+
64              puntos.at(i).escala);
65              line(imagen,puntos.at(i).p, Point(
66              x2,y2), Scalar(0,0,255));
67          }
68
69      }
70
71  }

```

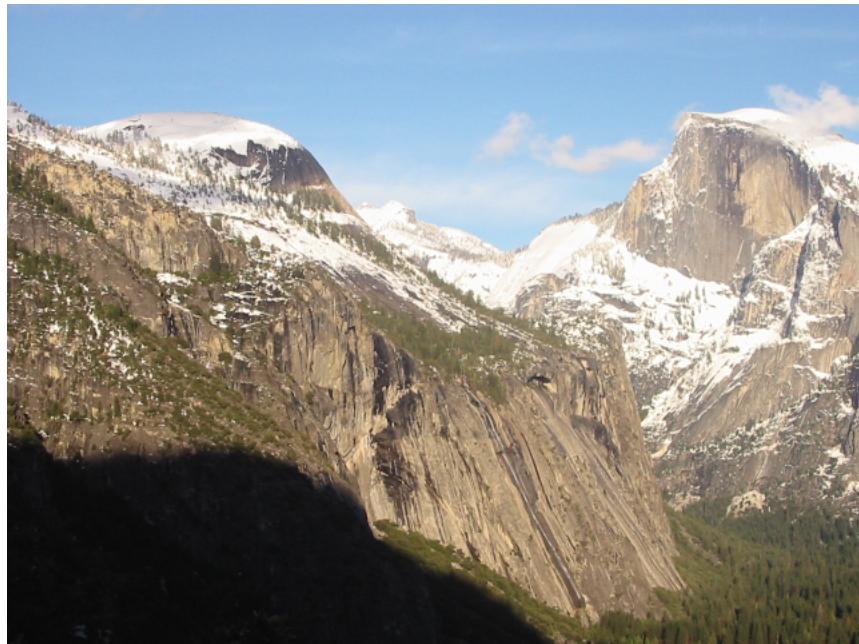
```

43           .at(i).orientacion)*10*(1+
44           puntos.at(i).escala);
45           y2 = puntos.at(i).p.y + sin(puntos
46           .at(i).orientacion)*10*(1+
47           puntos.at(i).escala);
48           line(imagen,puntos.at(i).p, Point(
49               x2,y2), Scalar(255,0,0));
50       }
51   }
52
53   else {
54
55       //Dibujamos el circulo con centro en las
56       //coordenadas del punto de Harris
57       circle(imagen,puntos.at(i).p,10*(1+puntos.
58           at(i).escala), Scalar(0,255,255));
59
60       if(orientacion){
61           //Dibujamos un segmento que nos
62           //indica la direccion del
63           //gradiente
64           x2 = puntos.at(i).p.x + cos(puntos
65           .at(i).orientacion)*10*(1+
66           puntos.at(i).escala);
67           y2 = puntos.at(i).p.y + sin(puntos
68           .at(i).orientacion)*10*(1+
69           puntos.at(i).escala);
70           line(imagen,puntos.at(i).p, Point(
71               x2,y2), Scalar(0,255,255));
72       }
73   }
74
75   if(orientacion)
76       mostrarImagen("Puntos de Harris con orientacion
77           sobre la imagen original",imagen);
78
79   else
80       mostrarImagen("Puntos de Harris sobre la imagen
81           original",imagen);
82 }

```

## 1.4. Resultados

Vamos a mostrar primero los resultados al obtener los mapas de valores:



(a) Imagen original a escala 1



(b) Mapa de valores de Harris para la imagen a escala 1

Como podemos ver los puntos Harris que obtenemos coinciden con puntos de la montaña, que se distribuyen por toda su superficie. Gracias al umbral mencionado antes, evitamos tomar puntos Harris que puedan darse en la zona del cielo o de la sombra que se ve

en la parte inferior izquierda de la imagen. Los puntos que obtenemos parecen bastante buenos.

A continuación mostramos el resultado con los puntos de Harris de mayor valor de las distintas escalas:

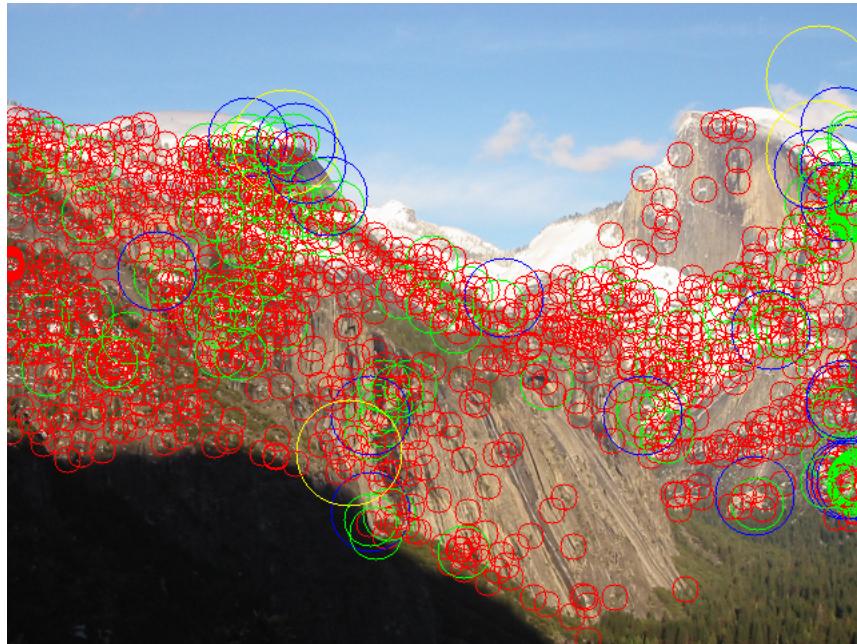


Figura 1.1: Imagen original junto con puntos Harris

El tamaño de los círculo se ha dibujado de forma proporcional a la escala, siendo los de menor tamaño correspondientes a la escala más grande y los mayores a la escala más pequeña. Para diferenciarlos mejor, los he dibujado con distintos colores. Como hemos indicado antes los puntos Harris se sitúan sobre la superficie de la montaña y podemos ver como delinean más o menos su contorno.

Por último, mostramos los puntos de Harris junto con un radio indicando la orientación aproximada:

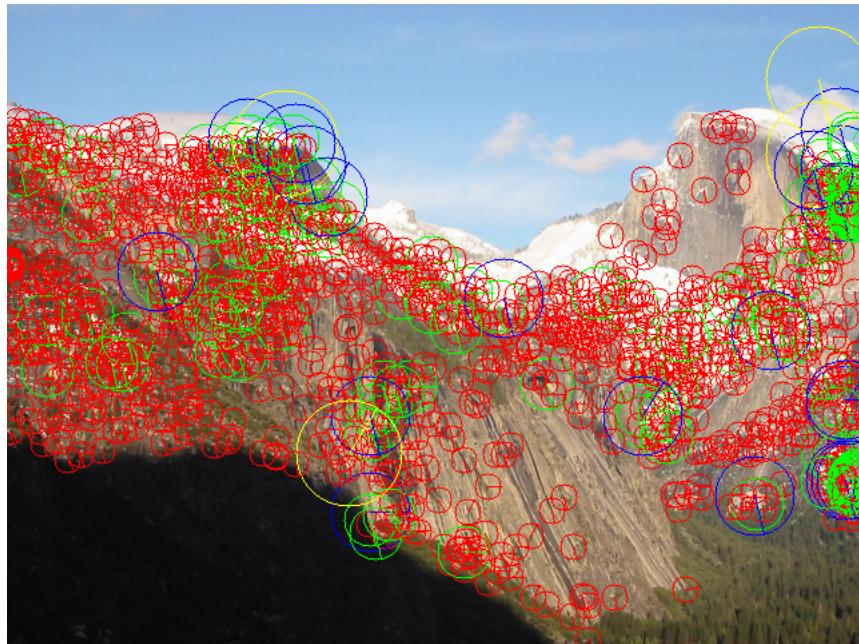


Figura 1.2: Imagen original junto con puntos Harris y su orientación aproximada

Los resultados obtenidos parecen ser bastante buenos, dado que los puntos Harris se sitúan delineando los bordes de la montaña y los picos, evitando las zonas planas de la imagen.

## 2. Detectores KAZE y AZAKE

Ambos son detectores definidos en OpenCV. El algoritmo KAZE se basa en la idea del viento y como éste se basa en procesos no lineales a gran escala. La idea es detectar características 2D en un espacio no lineal de gran escala para obtener una localización mejor y más exacta y distintiva. Para ello utiliza una matriz Hessiana para la detección de puntos blob. El algoritmo AKAZE es una versión acelerada del algoritmo KAZE, es decir, funciona de la misma forma pero a mayor velocidad. Para ello utiliza otro método de creación del espacio no lineal que es más rápido que el empleado pro KAZE.

Para utilizar ambos detectores primero tenemos que obtener las listas de KeyPoints. Para ellos creamos el detector correspondiente y obtenemos la lista de KeyPoints con el método *detect()*. Una vez tenemos la lista de KeyPoints obtenemos los descriptores con el método *compute()*.

```

1 Mat descriptorKaze(Mat imagen){
2
3     //Creamos el detector

```

```

4     Ptr<KAZE> ptrKaze = KAZE::create();
5     vector<KeyPoint> puntosKaze;
6     Mat descriptores;
7
8     //Obtenemos los KeyPoints
9     ptrKaze->detect(imagen, puntosKaze);
10
11    //Obtenemos los descriptores
12    ptrKaze->compute(imagen, puntosKaze, descriptores);
13
14    return descriptores;
15
16 }
17
18 Mat descriptorAkaze(Mat imagen){
19
20     //Creamos el detector
21     Ptr<AKAZE> ptrAkaze = AKAZE::create();
22     vector<KeyPoint> puntosAkaze;
23     Mat descriptores;
24
25     //Obtenemos los KeyPoints
26     ptrAkaze->detect(imagen, puntosAkaze);
27
28     //Obtenemos los descriptores
29     ptrAkaze->compute(imagen, puntosAkaze, descriptores);
30
31     return descriptores;
32
33 }
```

Para calcular las correspondencias tenemos que utilizar la función *match()*. Para ello necesitamos los descriptores que hemos calculado y el matcher de fuerza bruta y cross check que creamos a partir de la clase *BFMatcher*.

```

1 vector<DMatch> obtenerMatchesKaze(Mat imagen1, Mat imagen2){
2
3     Mat descriptor1, descriptor2;
4     vector<DMatch> matches;
5
6     //Creamos el matcher con Fuerza Bruta activando el flag de
7     //Cross Check
8     BFMatcher matcher = BFMatcher(NORM_L2, true);
9
10    //Calculamos los descriptores
11    descriptor1 = descriptorKaze(imagen1);
12    descriptor2 = descriptorKaze(imagen2);
```

```

13     matcher.match(descriptor1,descriptor2,matches);
14
15     return matches;
16 }
17
18 vector<DMatch> obtenerMatchesAkaze(Mat imagen1, Mat imagen2){
19
20     Mat descriptor1, descriptor2;
21     vector<DMatch> matches;
22
23     //Creamos el matcher con Fuerza Bruta activando el flag de
24     //Cross Check
24     BFMatcher matcher = BFMatcher(NORM_L2,true);
25
26     //Calculamos los descriptores
27     descriptor1 = descriptorAkaze(imagen1);
28     descriptor2 = descriptorAkaze(imagen2);
29
30     matcher.match(descriptor1,descriptor2,matches);
31
32     return matches;
33 }
```

## 2.1. Resultados del detector KAZE

Con el detector KAZE y las imágenes de Yosemite.rar se obtienen 711 correspondencias usando el valor por defecto del umbral. Para poder ver las correspondencias más claramente mostraremos sólo algunas.

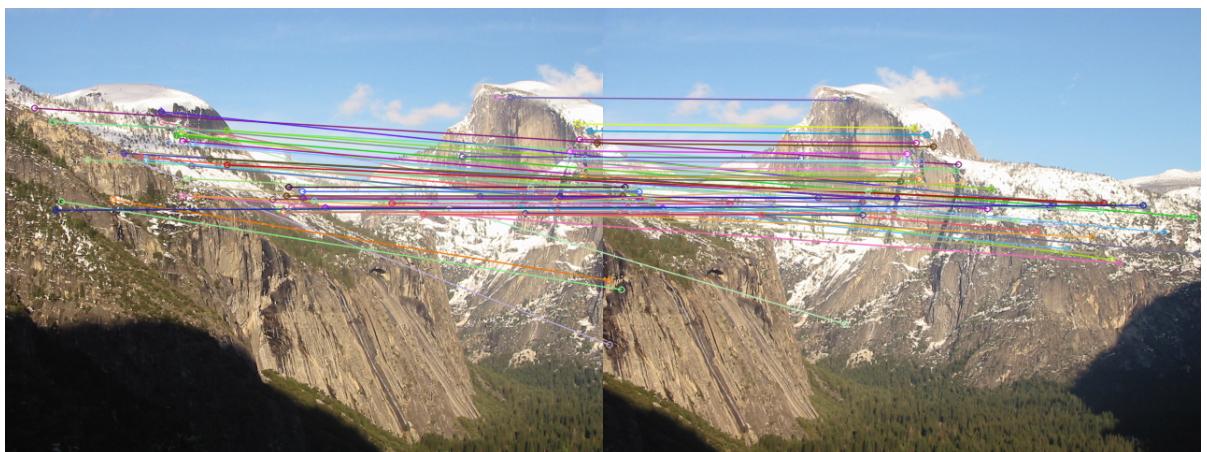


Figura 2.1: Correspondencias con detector KAZE

Como podemos observar se encuentran bastantes correspondencias erróneas entre las partes izquierda de la primera imagen y derecha de la segunda. Aunque también se encuentran muchas correspondencias entre las partes comunes de las imágenes. Teniendo en cuenta que las partes no comunes de las imágenes son bastante similares, es comprensible que encuentre correspondencias. Sin embargo, también empareja puntos que no tienen nada que ver, aunque estos son mínimos.

## 2.2. Resultado del detector AKAZE

Con el detector AKAZE se obtienen 747 correspondencias usando el valor por defecto del umbral. De igual manera, mostraremos sólo algunas correspondencias.

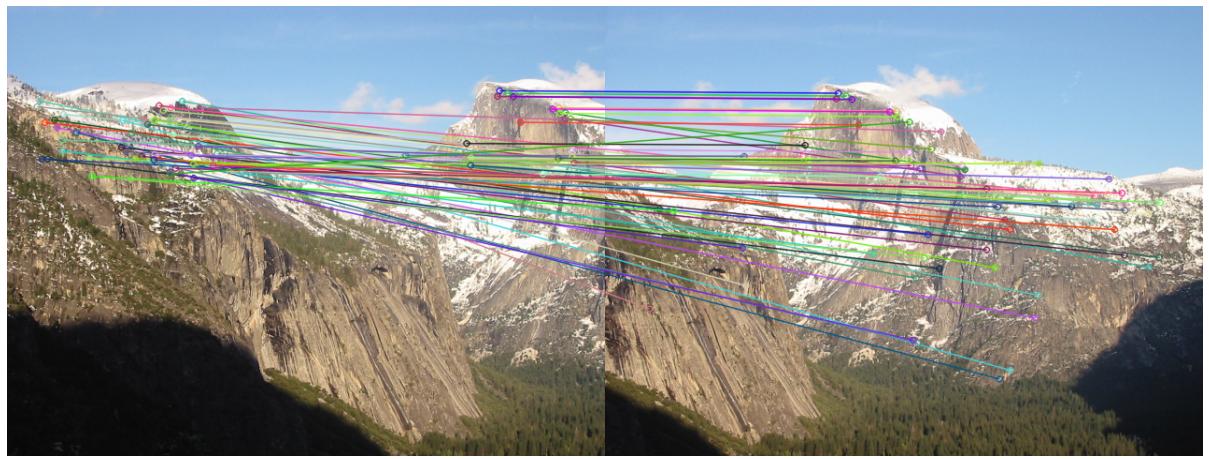


Figura 2.2: Correspondencias con detector AKAZE

Dado que el detector AKAZE funciona de la misma manera que el detector KAZE se obtienen resultados muy similares por no decir casi idénticos. Si podemos destacar que el detector AKAZE calcula las correspondencias de manera más rápida que el detector KAZE al ser éste una versión acelerada del mismo y además encuentra un mayor número de correspondencias.

## 3. Creación de un mosaico con N imágenes

Para la creación del mosaico primero debemos estimar las homografías para cada par de imágenes. Para ello necesitamos las listas de KeyPoints, la lista de correspondencias y la función *findHomography()*. En este caso los KeyPoints y las correspondencias los calcularemos a partir del detector AKAZE. Con esta función determinaremos cuáles son las mejores correspondencias entre cada par de imágenes:

```

1 Mat obtenerHomografia(Mat origen, Mat destino){
2
3     vector<KeyPoint> v1, v2;
4     vector<DMatch> matches;
5     vector<Point2f> p_origen, p_destino;
6     Mat homografia;
7
8     //Obtenemos los vectores de KeyPoints
9     v1 = obtenerKeyPointAkaze(origen);
10    v2 = obtenerKeyPointAkaze(destino);
11
12    //Obtenemos los matches
13    matches = obtenerMatchesAkaze(origen, destino);
14
15    //Guardamos los keyPoints como puntos Point2f
16    for(int i=0; i < matches.size(); ++i){
17        p_origen.push_back(v1[matches[i].queryIdx].pt);
18        p_destino.push_back(v2[matches[i].trainIdx].pt);
19    }
20
21    //Calculamos la homografia
22    homografia = findHomography(p_origen,p_destino,CV_RANSAC);
23
24    homografia.convertTo(homografia, CV_32F);
25
26    return homografia;
27}

```

Una vez tenemos un método para obtener las homografías, primero colocamos la imagen central en el mosaico final utilizando la función *warpPerspective()*. Para esta imagen la matriz de homografía que utilizamos es la matriz identidad con unas modificaciones para trasladar la imagen de manera que el mosaico se forme correctamente. Una vez está colocada vamos calculando sucesivamente las homografías de las imágenes a ambos lados de la imagen central. Por último, eliminamos posibles bordes negros que sobren en la imagen:

```

1 Mat crearMosaicoN(vector<Mat> imagenes){
2
3     //Creamos la imagen donde formaremos el mosaico
4     Mat mosaico = Mat(700, 1100, imagenes.at(0).type());
5
6     //Seleccionamos la posicion de la imagen central de la secuencia
7     int posicion_central = imagenes.size()/2;
8
9     //Colocamos la imagen central del vector en el centro del
10    mosaico
11    Mat colocacionCentral = Mat(3,3,CV_32F,0.0);

```

```

11
12     for (int i = 0; i < 3; i++)
13         colocacionCentral.at<float>(i,i) = 1.0;
14
15     //Realizamos una traslacion para colocarla correctamente
16     colocacionCentral.at<float>(0,2) = mosaico.cols/2 - imagenes.at(
17         posicion_central).cols/2;
18     colocacionCentral.at<float>(1,2) = mosaico.rows/2 - imagenes.at(
19         posicion_central).rows/2;
20
21     warpPerspective(imagenes.at(posicion_central), mosaico,
22                     colocacionCentral, Size(mosaico.cols, mosaico.rows),
23                     INTER_LINEAR, BORDER_CONSTANT);
24
25     //Matrices donde se acumularan las homografias a cada uno de los
26     //lados de la imagen central
27     Mat hoizda, h_der;
28
29     //Inicializamos con la homografia que hemos calculado antes:
30     colocacionCentral.copyTo(h_izq);
31     colocacionCentral.copyTo(h_der);
32
33
34     //Vamos formando el mosaico empezando desde la imagen central y
35     //desplazandonos a ambos extremos calculando las homografias
36     //correspondientes
37     for (int i = 1; i <= posicion_central; i++) {
38         if (posicion_central-i >= 0){
39             h_izq = h_izq * obtenerHomografia(imagenes.at(
40                 posicion_central-i), imagenes.at(posicion_central-i+1));
41             warpPerspective(imagenes.at(posicion_central-i), mosaico,
42                             h_izq, Size(mosaico.cols, mosaico.rows), INTER_LINEAR,
43                             BORDER_TRANSPARENT);
44         }
45
46         if (posicion_central+i < imagenes.size()){
47             h_der = h_der * obtenerHomografia(imagenes.at(
48                 posicion_central+i), imagenes.at(posicion_central+i-1));
49             warpPerspective(imagenes.at(posicion_central+i), mosaico,
50                             h_der, Size(mosaico.cols, mosaico.rows), INTER_LINEAR,
51                             BORDER_TRANSPARENT);
52         }
53     }
54
55     //Eliminamos bordes negros que puedan quedar en la imagen
56     vector<Point> p_aux;
57
58     for (int i = 0; i < mosaico.rows; ++i)
59         for (int j = 0; j < mosaico.cols; ++j)

```

```
47             if (mosaico.at<Vec3b>(i, j) != Vec3b(0, 0,
48                                         0))
49                 p_aux.push_back(Point(j, i));
50
51             Rect rectangulo = boundingRect(p_aux);
52             mosaico=mosaico(rectangulo);
53
54         return mosaico;
55     }
```

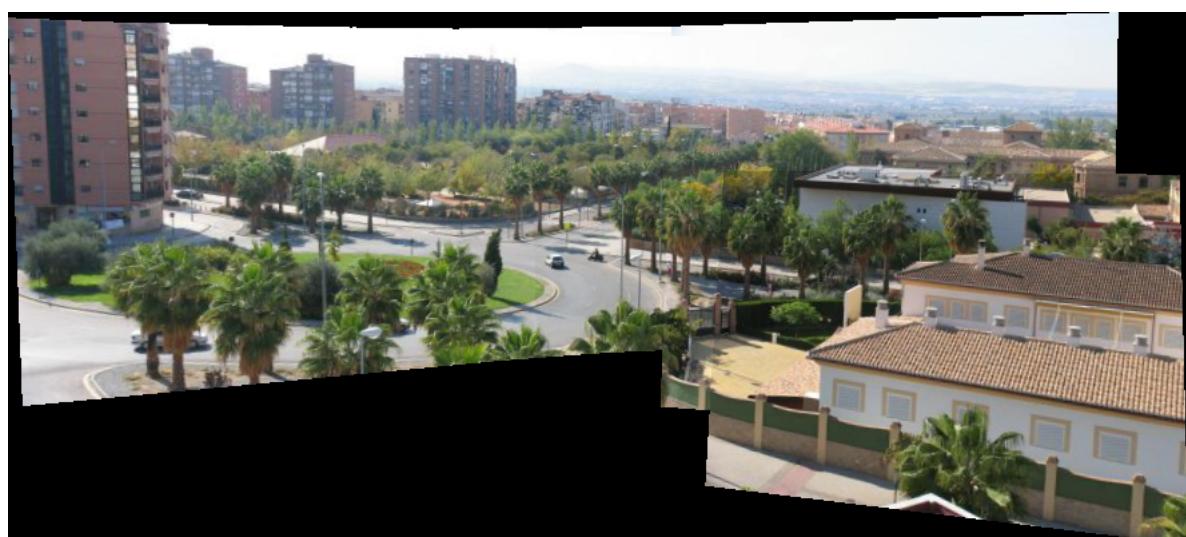
### 3.1. Resultados



(a) Mosaico con cuatro imágenes



(b) Mosaico con siete imágenes



(c) Mosaico con diez imágenes

Como podemos ver, el resultado es bastante bueno. Se observan algunos saltos de color y pequeñas distorsiones. Lo más notable es que podemos ver que los bordes no son totalmente rectos probablemente por el modo en que la homografía coloca las imágenes.