

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

Memoria Práctica 2

Ignacio Martín Requena

23 de noviembre de 2016

Índice

1 Ejercicio 1	3
1.1 Enunciado	3
1.2 Comentarios sobre el desarrollo	3
1.3 Salidas obtenidas	6
2 Ejercicio 2	7
2.1 Enunciado	7
2.2 Comentarios sobre el desarrollo	7
2.3 Salidas obtenidas	9
3 Ejercicio 3	10
3.1 Enunciado	10
3.2 Comentarios sobre el desarrollo	10
3.3 Salidas obtenidas	11

Índice de figuras

1.1. Lectura imagenes	4
1.2. Matriz harris a vector ordenado	5
1.3. Dibujado de circulo en cada punto Harris	5
1.4. Puntos Harris obtenidos	6
2.1. Parámetros de entrada función ejercicio 2	7
2.2. Código principal usando detector KAZE	7
2.3. Cálculo de los keypoints y descriptores	8
2.4. Función para determinar los puntos en comun entre dos imagenes	8
2.5. Correspondencias usando KAZE, crossCheck activado y limite de correspondencia 1	9
2.6. Correspondencias usando AKAZE, crossCheck activado y limite de correspondencia 1	9
2.7. Datos obtenidos ejercicio 2	9
3.1. Código de la función principal que crea el mosaico	10
3.2. Código de la función que calcula todo lo necesario para unir una imagen de entrada con el mosaico principal	11
3.3. Mosaico ETSIIT	11
3.4. Salida texto mosaico ETSIIT	12
3.5. Mosaico Yosemite	12
3.6. Salida texto mosaico yosemite	13

1. Ejercicio 1

1.1. Enunciado

Escribir una función que extraiga la lista potencial de puntos Harris en una imagen a distintas escalas. Para ello construiremos una pirámide Gaussiana de la imagen con 4 escalas usando sigma=1. Sobre cada nivel de la pirámide usar la función OpenCV cornerHarris que devuelve el mapa de valores del criterio Harris en cada píxel. Seleccionar al menos los 1500 puntos de entre los de mayor valor distribuidos entre las distintas escalas (las escalas más bajas tendrán más puntos: 70-20-10). Mostrar el resultado identificando con un círculo o una cruz sobre la imagen original las coordenadas de los puntos seleccionados (ver circle()). (1.5 puntos)

- a. Calcular la orientación relevante de cada punto Harris, usando un alisamiento fuerte de las derivadas en x e y de las imágenes, en la escala correspondiente, como propone el paper MOPS de Brown&Szeliski&Winder. (Apartado 2.5) y añadir la información del ángulo al vector de información del punto. Pintar sobre la imagen de círculos anterior un radio en cada círculo indicando la orientación estimada en ese punto (1punto)

1.2. Comentarios sobre el desarrollo

En este ejercicio he tenido varios problemas que iré comentando conforme explique el trabajo realizado.

En primer lugar y antes de nada cargo todas las imágenes que son susceptibles a ser utilizadas en toda la práctica de la siguiente forma:

```

//Lectura de todas las imágenes usadas en el trabajo 2
Mat img_mosaico1 = leeImagen("imagenes/mosaico002.jpg", 1);
Mat img_mosaico2 = leeImagen("imagenes/mosaico003.jpg", 1);
Mat img_mosaico3 = leeImagen("imagenes/mosaico004.jpg", 1);
Mat img_mosaico4 = leeImagen("imagenes/mosaico005.jpg", 1);
Mat img_mosaico5 = leeImagen("imagenes/mosaico006.jpg", 1);
Mat img_mosaico6 = leeImagen("imagenes/mosaico007.jpg", 1);
Mat img_mosaico7 = leeImagen("imagenes/mosaico008.jpg", 1);
Mat img_mosaico8 = leeImagen("imagenes/mosaico009.jpg", 1);
Mat img_mosaico9 = leeImagen("imagenes/mosaico010.jpg", 1);
Mat img_mosaico10 = leeImagen("imagenes/mosaico011.jpg", 1);
vector<Mat> imagenes_mosaico;
imagenes_mosaico.push_back(img_mosaico1);
imagenes_mosaico.push_back(img_mosaico2);
imagenes_mosaico.push_back(img_mosaico3);
imagenes_mosaico.push_back(img_mosaico4);
imagenes_mosaico.push_back(img_mosaico5);
imagenes_mosaico.push_back(img_mosaico6);
imagenes_mosaico.push_back(img_mosaico7);
imagenes_mosaico.push_back(img_mosaico8);
imagenes_mosaico.push_back(img_mosaico9);
imagenes_mosaico.push_back(img_mosaico10);

Mat img_yosemite1 = leeImagen("imagenes/yosemite1.jpg", 1);
Mat img_yosemite2 = leeImagen("imagenes/yosemite2.jpg", 1);
Mat img_yosemite3 = leeImagen("imagenes/yosemite3.jpg", 1);
Mat img_yosemite4 = leeImagen("imagenes/yosemite4.jpg", 1);
vector<Mat> imagenes_yosemite;
imagenes_yosemite.push_back(img_yosemite1);
imagenes_yosemite.push_back(img_yosemite2);
imagenes_yosemite.push_back(img_yosemite3);
imagenes_yosemite.push_back(img_yosemite4);

```

Figura 1.1: Lectura imágenes

Para continuar llamando a la función que ejecuta el ejercicio 1. Esta función realiza las siguientes acciones:

- A la imagen pasada como parámetro, se le aplica la función de OpenCV cornerHarris para obtener en cada posición de la matriz resultado un valor Harris, cuanto mayor sea el valor obtenido más probabilidad hay de que ese valor sea una esquina.
- Una vez obtenidos la lista de puntos Harris procedemos a normalizarlos para obtener unos valores mas simplificados (función normalize de OpenCV)
- Ahora deberíamos, de esos valores obtenidos, eliminar aquellos que no son máximos locales con el fin de que no salgan puntos Harris en posiciones cercanas. Después de intentarlo durante mucho tiempo no he conseguido realizar la función no-maximo debido a no poder solucionar un fallo de segmentación. Por tanto, los resultados finales no serán del todo correctos y algunos puntos Harris se superpondrán.
- Ahora, la matriz que contiene los puntos Harris la transformaremos en una lista

ordenada de mayor a menor valor. Esto lo hacemos para, una vez ordenada, poder seleccionar solamente aquellos puntos con un buen valor Harris.

```
//Ordenar los valores de la matriz normalizada de mayor a menor
for( int j = 0; j < dst_norm.rows ; j++ ){
    for( int i = 0; i < dst_norm.cols; i++ ){
        punto.value = dst_norm.at<float>(j,i);
        punto.p.x = i;
        punto.p.y = j;

        vectorsort_aux.push_back(punto);
    }
}
std::sort(vectorsort_aux.begin(), vectorsort_aux.end(), sortfunc);
```

Figura 1.2: Matriz harris a vector ordenado

- Por último nos quedaría seleccionar solamente los 1500*0.7 primeros valores de la lista creada en el punto anterior

Una vez realizado todo el proceso con la imagen original lo vamos a volver a repetir dos veces mas, cada una a partir de la imagen anterior escalada a la mitad (para ello usamos la función pyrDown de OpenCV) y seleccionando el 20 % y el 10 % de los puntos Harris obtenidos respectivamente.

Por último dibujamos un círculo en cada posición del vector que contiene todos los puntos Harris seleccionados.

```
/// Dibujar circulos alrededor de las esquinas
for(uint j = 0; j < vectorsort.size()-1; j++ ){
    circle( im_aux, vectorsort.at(j).p, 5, Scalar(0), 1, 4, 0 );
```

Figura 1.3: Dibujado de circulo en cada punto Harris

La estructura principal del código de obtención de puntos Harris ha sido obtenida de un ejemplo de la documentación de OpenCV.¹

¹http://docs.opencv.org/2.4/doc/tutorials/features2d/trackingmotion/harris_detector/harris_detector.html

1.3. Salidas obtenidas



Figura 1.4: Puntos Harris obtenidos

Como hemos comentado anteriormente, a los puntos Harris faltaría añadirle un método para eliminar los no-maximos locales, por lo que aunque en el resultado se muestran los puntos Harris de mayor valor este no es un resultado de calidad.

2. Ejercicio 2

2.1. Enunciado

Usar los detectores OpenCV (KAZE/AKAZE) sobre las imágenes de Yosemite.rar. Extraer extraer sus listas de keyPoints y establecer las correspondencias existentes entre ellas (Ayuda: usar la clase DescriptorMatcher y BFMatcher de OpenCV). Valorar la calidad de los resultados obtenidos bajo el criterio de correspondencias OpenCV Brute-Force+crossCheck. (1.5 puntos)

2.2. Comentarios sobre el desarrollo

Para realizar este ejercicio en primer lugar creamos una función con los siguientes parámetros de entrada:

```
/**************************************************************************/  
/* EJERCICIO 2 */  
/**************************************************************************/  
  
//0 ambos, 1 kaze, 2 akaze  
int detector = 0;  
Ejercicio2Trabajo2(imagenes_yosemite[0], imagenes_yosemite[1], detector);
```

Figura 2.1: Parámetros de entrada función ejercicio 2

Como podemos ver la función recibe las dos imágenes a relacionar y un entero que determina si usamos el detector KAZE, el AKAZE o ambos.

En cuanto a la función de este ejercicio esta tiene dos partes: un calculo de los puntos en correspondencia mediante el detector KAZE y otro cálculo similar pero usando el detector AKAZE. Como la metodología a seguir en ambos ha sido la misma salvo el cambio de detector únicamente explicaré lo concerniente al detector KAZE.

Calculo de los puntos en correspondencia mediante el detector AKAZE

```
if (detector == 0 || detector == 1){  
    //Paso el detector KAZE:  
    kazeFuncion(img1, keypoints1, kaze1, descriptor1);  
    kazeFuncion(img2, keypoints2, kaze2, descriptor2);  
    //Muestro los resultados del detector Kaze  
    cout << "KAZE:Se han encontrado " << keypoints1.size() << " keypoints y " << keypoints2.size() << " keypoints" << endl;  
  
    Mat img_salida;  
    vector<DMatch> matchesbuenos;  
    //Este es el mismo resultado de la función con los imágenes, descriptores, keypoints y el flag si queremos crossCheck  
    PuntosEnComun(img1, img2, descriptor1, descriptor2, keypoints1, keypoints2, crossCheck, img_salida, matchesbuenos, limite_correspondencias);  
    pintal(img_salida, "KAZE");  
}
```

Figura 2.2: Código principal usando detector KAZE

Esta función realiza lo siguiente: en primer lugar calculamos la lista de puntos clave y los descriptores de cada una de las dos imágenes para, una vez obtenidos estos valores pintar los puntos clave en la imagen original

```

//Creamos el detector con los parámetros deseados
Ptr<KAZE> KAZE = KAZE::create();
//Hacemos la detección de los puntos clave en la imagen
KAZE->detect(img, keypoints);
//Obtenemos los descriptores
KAZE->compute(img, keypoints, descriptors);
//Pintamos los keypoints en la imagen
drawKeypoints(img, keypoints, kaze);

```

Figura 2.3: Cálculo de los keypoints y descriptores

Ahora resta detectar y pintar los puntos en común entre ambas imágenes. Esto lo realiza la función PuntosEnComun que recibe las **dos imágenes**, los **descriptores** y **keypoints** obtenidos anteriormente por los detectores, un booleano llamado **crossCheck** que indica si queremos que los valores obtenidos se realicen mediante el criterio de fuerza bruta o el de validación cruzada, la **imagen de salida** que devolverá las correspondencias entre ambas imágenes, **un vector** para devolver las correspondencias que hay entre ambas imágenes y un límite **límite_correspondencia** que explicaré mas adelante.

```

vector<DMatch> matches;
//Realizamos la búsqueda de las correspondencias
BFMatcher matcher(NORM_L2, crossCheck);
matcher.match(descriptors1, descriptors2, matches);

//Ordenamos las correspondencias para buscar las mejores de forma más eficiente
sort(matches.begin(), matches.end());
float min_dist = matches[0].distance;

//Buscamos las mejores correspondencias
for (uint i = 0; i < matches.size(); i++){
    if (matches[i].distance <= limite_correspondencias * min_dist){
        matchesbuenos.push_back(matches[i]);
    }
    else if (matchesbuenos.size() < 4){ //en el caso de que no consiga un mínimo de 4 correspondencias
        //Aumento el factor de correspondencia
        limite_correspondencias++;
        i--;
    }
}
cout << "He pasado de " << matches.size() << " correspondencias a " << matchesbuenos.size() << ". Valor limite_correspondencias: " << limite_correspondencias << endl;
//Pintamos los puntos de correspondencia
drawMatches(img1, keypoints1, img2, keypoints2, matchesbuenos, salida);

```

Figura 2.4: Función para determinar los puntos en comun entre dos imagenes

En primer lugar realizamos la búsqueda de correspondencias y las ordenamos. Una vez tenemos todas las correspondencias vamos a determinar cuales de ellas son las mejores, para ello usamos el límite correspondencia. Lo que hace este límite es que solamente elijamos aquellas correspondencias cuya distancia sea cercana, las correspondencias De esta forma las que tengan una distancia muy lejana las descartaremos por intuir que serán malas. Por ultimo un añadido if de control para que si el numero de correspondencias buenas obtenidas es bajo relajar la restricción de la distancia aumentando el límite de correspondencia.

Ya sólo quedaría dibujar las correspondencias con la función drawMatches de OpenCV.

2.3. Salidas obtenidas



Figura 2.5: Correspondencias usando KAZE, crossCheck activado y límite de correspondencia 1

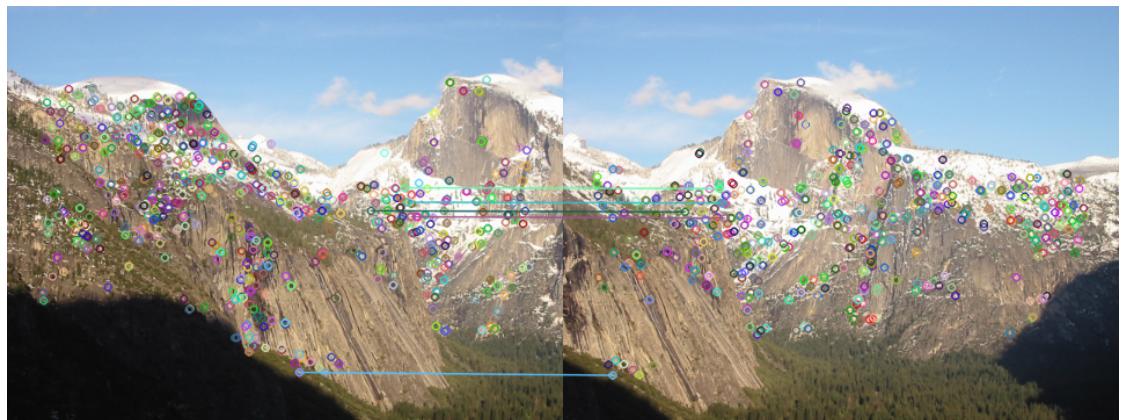


Figura 2.6: Correspondencias usando AKAZE, crossCheck activado y límite de correspondencia 1

```
Ejecutando Ejercicio 2 Trabajo 2
KAZE:Se han encontrado 1238 keypoints y 1086 keypoints
He pasado de 718 correspondencias a 718. Valor limite_correspondencias: 50
AKAZE:Se han encontrado 1396 keypoints y 1159 keypoints
He pasado de 746 correspondencias a 6. Valor limite_correspondencias: 50
Fin Ejecucion Ejercicio 2 Trabajo 2
```

Figura 2.7: Datos obtenidos ejercicio 2

Como podemos ver, el límite de correspondencia influye a la hora de determinar las correspondencias en el caso de AKAZE, mostrándonos sólo las 5 más prometedoras.

3. Ejercicio 3

3.1. Enunciado

(OPCION: 10 puntos) Escribir una función que forme un Mosaico de calidad a partir de $N > 3$ imágenes relacionadas por homografías, sus listas de keyPoints calculados de acuerdo al punto anterior y las correspondencias encontradas entre dichas listas. Estimar las homografías entre ellas usando la función `findHomography(p1,p2,CV_RANSAC,1)`. 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: mirar el flag `BORDER_TRANSPARENT` de `warpPerspective` para comenzar). (4 puntos)

3.2. Comentarios sobre el desarrollo

La función de este ejercicio recibirá como argumentos un vector con todas las imágenes que forman el mosaico, las dimensiones máximas del mosaico y el límite de correspondencia explicado en el punto anterior.

Lo primero que hacemos es mostrar por pantalla todas las imágenes que van a formar parte del mosaico para, a continuación, llamar a la función `crearSuperMosaico`, que será la encargada de ir añadiendo cada una de las imágenes al mosaico de la siguiente forma:

```
cout << "En total hay " << imagenes.size() << " imagenes." << endl;

Mat mosaico;
int indice_comienzo = (imagenes.size()-1)/2;
inicializarMosaico(imagenes[indice_comienzo],mosaico,mirows,micols);
cout << "Imagen " << indice_comienzo+1 << " introducida." << endl;

//Comienzo el mosaico del centro a la izquierda
for (int i = indice_comienzo -1; i >= 0; i--){
    Mat img1 = imagenes[i];
    crearMosaico(mosaico, img1, limite_correspondencias);
    cout << "Imagen " << i+1 << " introducida." << endl;
}
//Sigo el mosaico del centro a la derecha
for (uint i = indice_comienzo +1 ; i < imagenes.size(); i++){
    Mat img1 = imagenes[i];
    crearMosaico(mosaico, img1, limite_correspondencias);
    cout << "Imagen " << i+1 << " introducida." << endl;
}
//Guardo el resultado en lo que devolveré
mosaico_res = mosaico;
```

Figura 3.1: Código de la función principal que crea el mosaico

Esta función en primer lugar inicializa el mosaico con las dimensiones introducidas y colocando la primera imagen (que será la que esté en el centro del vector de imágenes).

A continuación voy recorriendo el vector de imágenes primero hacia la izquierda y luego hacia la derecha y llamando a la función encargada de determinar los keypoints, las correspondencias entre el estado actual del mosaico y la nueva imagen a introducir para acabar buscando la homografía de la lista de keypoints de ambas imágenes. Por último, con la función wrapPerspective realizamos las transformaciones de la imagen que la homografía calculada previamente nos indica.

```

vector<KeyPoint> keypoints1, keypoints2;
Mat descriptors1, descriptors2, empty;

//Primero tenemos que sacar los keypoints, para ello uso kaze
kazeFuncion(img1, keypoints1, empty, descriptors1);
kazeFuncion(mosaico, keypoints2, empty, descriptors2);

Mat img_sal;
vector<DMatch> matchesbuenos;
//Sacamos los puntos en correspondencias
PuntosEnComun(img1, mosaico, descriptors1, descriptors2, keypoints1, keypoints2, 1, img_sal, matchesbuenos, limite_correspondencias);

vector<Point2f> key1, key2;
//Mostramos el numero de correspondencias encontradas y guardamos sus coordenadas
cout << "Total de correspondencias para las dos imágenes: " << matchesbuenos.size() << endl;
for (int i = 0; i < matchesbuenos.size(); i++)
{
    key1.push_back(keypoints1[matchesbuenos[i].queryIdx].pt);
    key2.push_back(keypoints2[matchesbuenos[i].trainIdx].pt);
}

//Buscamos la homografía de las imágenes
/*[R11,R12,T1]
[R21,R22,T2]
[ P , P + 1]*/
Mat H = findHomography(key1, key2, CV_RANSAC);
//Realizamos la transformación
warpPerspective(img1, mosaico, H, mosaico.size(), CV_INTER_LINEAR + CV_WARP_FILL_OUTLIERS, BORDER_TRANSPARENT);

```

Figura 3.2: Código de la función que calcula todo lo necesario para unir una imagen de entrada con el mosaico principal

3.3. Salidas obtenidas



Figura 3.3: Mosaico ETSIIT

```
Ejecutando Ejercicio 3 Trabajo 2
En total hay 10 imagenes.
Imagen 5 introducida.
He pasado de 303 correspondencias a 17. Valor limite_correspondencias: 2
Total de correspondencias para las dos imagenes: 17
Imagen 4 introducida.
He pasado de 319 correspondencias a 43. Valor limite_correspondencias: 2
Total de correspondencias para las dos imagenes: 43
Imagen 3 introducida.
He pasado de 362 correspondencias a 6. Valor limite_correspondencias: 2
Total de correspondencias para las dos imagenes: 6
Imagen 2 introducida.
He pasado de 364 correspondencias a 37. Valor limite_correspondencias: 2
Total de correspondencias para las dos imagenes: 37
Imagen 1 introducida.
He pasado de 338 correspondencias a 35. Valor limite_correspondencias: 2
Total de correspondencias para las dos imagenes: 35
Imagen 6 introducida.
He pasado de 340 correspondencias a 32. Valor limite_correspondencias: 2
Total de correspondencias para las dos imagenes: 32
Imagen 7 introducida.
He pasado de 309 correspondencias a 28. Valor limite_correspondencias: 2
Total de correspondencias para las dos imagenes: 28
Imagen 8 introducida.
He pasado de 401 correspondencias a 21. Valor limite_correspondencias: 2
Total de correspondencias para las dos imagenes: 21
Imagen 9 introducida.
He pasado de 395 correspondencias a 23. Valor limite_correspondencias: 2
Total de correspondencias para las dos imagenes: 23
Imagen 10 introducida.
Fin Ejecucion Ejercicio 3 Trabajo 2
```

Figura 3.4: Salida texto mosaico ETSIIT

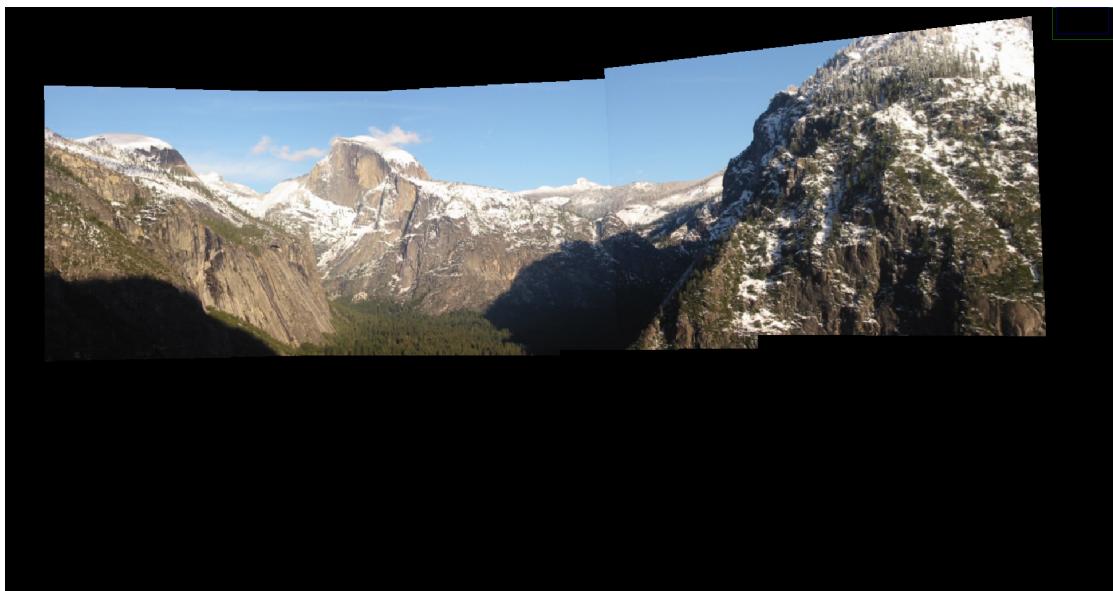


Figura 3.5: Mosaico Yosemite

```
Ejecutando Ejercicio 3 Trabajo 2
En total hay 4 imagenes.
Imagen 2 introducida.
He pasado de 669 correspondencias a 68. Valor limite_correspondencias: 2
Total de correspondencias para las dos imagenes: 68
Imagen 1 introducida.
He pasado de 677 correspondencias a 45. Valor limite_correspondencias: 2
Total de correspondencias para las dos imagenes: 45
Imagen 3 introducida.
He pasado de 1199 correspondencias a 29. Valor limite_correspondencias: 2
Total de correspondencias para las dos imagenes: 29
Imagen 4 introducida.
```

Figura 3.6: Salida texto mosaico yosemite

Como podemos ver ambos resultados son razonablemente buenos.