

Informe de prácticas

2

Alejandro García Montoro
agarciamontoro@correo.ugr.es

21 de noviembre de 2015

1. Estimación de una homografía

El objetivo de este ejercicio es estimar la homografía que mejor aproxima dos conjuntos de puntos en correspondencia tomados de sendos planos proyectivos. La idea, entonces, es rectificar uno de ellos para llevarlo al plano del otro. Antes de ver todo el código, veamos matemáticamente qué tenemos que hacer.

Sean x e y dos puntos en coordenadas homogéneas que queremos poner en correspondencia a través de una homografía H . Se tiene que cumplir por tanto la ecuación

$$y = Hx \quad (1)$$

Como vemos en [Szelinski, p.37, eq 2.21] o en *Homography estimation*¹, al ser H una homografía, y normalizando el punto y , de (1) obtenemos las dos siguientes ecuaciones:

$$y_1 = \frac{H_{1,1}x_1 + H_{1,2}x_2 + H_{1,3}}{H_{3,1}x_1 + H_{3,2}x_2 + H_{3,3}} \quad (2)$$

$$y_2 = \frac{H_{2,1}x_1 + H_{2,2}x_2 + H_{2,3}}{H_{3,1}x_1 + H_{3,2}x_2 + H_{3,3}} \quad (3)$$

donde $H_{i,j}$ son los coeficientes de la matriz H ; es decir, nuestras incógnitas. Además, hemos supuesto normalizado el vector x —si $x_3 \neq 1$, basta dividir entre x_3 todas las coordenadas—.

De (2) y (3), reorganizando las igualdades llegamos a las siguientes ecuaciones lineales:

$$ah = 0 \quad (4)$$

$$\tilde{a}h = 0 \quad (5)$$

¹http://cseweb.ucsd.edu/classes/wi07/cse252a/homography_estimation/homography_estimation.pdf

donde

$$\begin{aligned} a &= (-x_1 \quad -x_2 \quad -1 \quad 0 \quad 0 \quad 0 \quad y_1x_1 \quad y_1x_2 \quad y_1) \\ \tilde{a} &= (0 \quad 0 \quad 0 \quad -x_1 \quad -x_2 \quad -1 \quad y_2x_1 \quad y_2x_2 \quad y_2) \\ h &= (H_{1,1} \quad H_{1,2} \quad H_{1,3} \quad H_{2,1} \quad H_{2,2} \quad H_{2,3} \quad H_{3,1} \quad H_{3,2} \quad H_{3,3})^T \end{aligned}$$

Las ecuaciones (4) y (5) las podemos escribir de forma matricial como

$$Ah = 0 \quad (6)$$

donde

$$A = \begin{pmatrix} a \\ \tilde{a} \end{pmatrix}$$

Pero nuestro objetivo es resolver (6) con un número N arbitrario de puntos —al menos cuatro para determinar una solución—, de manera que la matriz A se convertirá en

$$A = \begin{pmatrix} a_1 \\ \tilde{a}_1 \\ a_2 \\ \tilde{a}_2 \\ \vdots \\ a_N \\ \tilde{a}_N \end{pmatrix}$$

El problema de resolver sistemas de ecuaciones del tipo (6) está muy estudiado, y puede ser abordado a partir de la descomposición en valores singulares —SVD por sus siglas en inglés— de la matriz A :

$$A = USV^T = \sum_{i=0}^9 \sigma_i u_i v_i^T$$

La mejor solución encontrada para h —exacta si $\sigma_9 = 0$ y aproximada si $\sigma_9 > 0$ — es el último vector columna de la matriz V .

Tenemos por tanto un algoritmo claro para implementar en la función que queremos diseñar:

1. Definir dos conjuntos de puntos en correspondencia.
2. Para cada par de puntos, definir los coeficientes de a y \tilde{a} .
3. Construir la matriz A con todos los a y \tilde{a} calculados anteriormente.
4. Hacer la descomposición en valores singulares de la matriz A .
5. Reorganizar el último vector columna de V en los coeficientes de la matriz H .

Aclarando el algoritmo que vamos a seguir, veamos el código que lo implementa.

1.1. Selección de puntos

Se han seleccionado dos tipos de conjuntos de puntos en correspondencia entre las imágenes *Tablero1.jpg* y *Tablero2.jpg*.

Primero, se han determinado diez puntos de cada imagen de manera que estuvieran suficientemente repartidos por todo el tablero. La columna izquierda de la siguiente lista muestra los puntos de la imagen *Tablero2.jpg*; a la derecha, sus correspondencias en la imagen *Tablero1.jpg*:

```
1     Point2f(147, 13) --> Point2f(156, 47)
2     Point2f(504, 95) --> Point2f(532, 11)
3     Point2f(432, 444) --> Point2f(527, 466)
4     Point2f(75 , 388) --> Point2f(137, 422)
5     Point2f(227, 133) --> Point2f(238, 139)
6     Point2f(396, 169) --> Point2f(363, 133)
7     Point2f(362, 338) --> Point2f(413, 337)
8     Point2f(192, 308) --> Point2f(229, 327)
9     Point2f(286, 224) --> Point2f(308, 219)
10    Point2f(304, 251) --> Point2f(331, 245)
```

Además, se ha determinado otro conjunto de diez puntos en correspondencia, esta vez todos en las tres casillas de la esquina superior izquierda:

```
1     Point2f(148,14) --> Point2f(156,47)
2     Point2f(174,19) --> Point2f(177,45)
3     Point2f(198,24) --> Point2f(198,43)
4     Point2f(223,30) --> Point2f(221,40)
5     Point2f(141,40) --> Point2f(155,72)
6     Point2f(168,46) --> Point2f(176,70)
7     Point2f(191,50) --> Point2f(197,68)
8     Point2f(217,56) --> Point2f(219,66)
9     Point2f(136,63) --> Point2f(153,95)
10    Point2f(162,69) --> Point2f(174,93)
```

1.2. Estimación de la homografía

La función de estimación de la homografía toma un vector de pares de puntos e implementa el algoritmo visto en la introducción. Veamos el código:

```
1 Mat Image::findHomography(vector< pair<Point2f,Point2f> > matches){
2     // Build the equations system. See
3     // http://sl.ugr.es/homography_estimation
4     Mat mat_system, sing_values, l_sing_vectors, r_sing_vectors;
5
6     for (unsigned int i = 0; i < matches.size(); i++) {
7         Point2f first = matches[i].first;
8         Point2f second = matches[i].second;
```

```

9     float coeffs[2][9] = {
10        { -first.x, -first.y, -1., 0., 0., 0., second.x*first.x,
11          second.x*first.y, second.x },
12        { 0., 0., 0., -first.x, -first.y, -1., second.y*first.x,
13          second.y*first.y, second.y }
14    };
15
16
17    // Solve the equations system using SVD decomposition
18    SVD::compute( mat_system, sing_values, l_sing_vectors,
19                  r_sing_vectors, 0 );
20
21    Mat last_row = r_sing_vectors.row(r_sing_vectors.rows-1);
22
23    cout << "sigma_9 for own findHomography: " <<
24      sing_values.at<float>(8,0) << endl;
25
}

```

El código es claro: en el bucle construimos la matriz A , calculando para cada par de puntos en correspondencia los coeficientes de las ecuaciones (4) y (5).

Luego, usamos el módulo *SVD* de OpenCV para obtener la descomposición en valores singulares de A .

Basta entonces tomar la última fila de V —OpenCV devuelve V en filas— y darle la forma 3×3 que necesitamos.

1.3. Aplicación de la homografía estimada

Para aplicar la homografía estimada a la imagen basta usar la función *warpPerspective* de OpenCV. Por tanto, se ha definido una función *wrapper* de igual nombre: *Image::warpPerspective*, cuyo código es el siguiente:

```

1 Image Image::warpPerspective(vector< pair<Point2f,Point2f> >
2   keypoints){
3   Mat homography = this->findHomography(keypoints);
4
5   vector<Point2f> corners(4), corners_trans(4);
6
7   corners[0] = Point2f(0,0);
8   corners[1] = Point2f(this->cols(),0);
9   corners[2] = Point2f(this->cols(),this->rows());
10  corners[3] = Point2f(0,this->rows());
11
12  perspectiveTransform(corners, corners_trans, homography);

```

```

12
13     float min_x, min_y, max_x, max_y;
14     min_x = min_y = +INF;
15     max_x = max_y = -INF;
16     for (int j = 0; j < 4; j++) {
17         min_x = min(corners_trans[j].x, min_x);
18         max_x = max(corners_trans[j].x, max_x);
19
20         min_y = min(corners_trans[j].y, min_y);
21         max_y = max(corners_trans[j].y, max_y);
22     }
23
24     Mat dst(Size(max_x-min_x,max_y-min_y), this->image.type());
25
26     // Define translation homography
27     Mat trans_homography = Mat::eye(3,3,homography.type());
28     trans_homography.at<float>(0,2) = -min_x;
29     trans_homography.at<float>(1,2) = -min_y;
30
31     cv::warpPerspective( this->image, dst,
32                         trans_homography*homography, dst.size() );
33
34     return Image(dst);
}

```

El usuario entonces podrá llamar a *Image::warpPerspective* sobre un objeto imagen pasando como argumento un vector de pares de puntos en correspondencia; siendo el primer elemento del par un punto de la imagen y, el segundo, su imagen por la homografía que estimaremos.

Las líneas 2 y 31 son las esenciales de este código: primero estimamos la homografía a partir del vector provisto y, para aplicársela a la imagen, llamamos a la función *cv::warpPerspective*.

Todo el código intermedio, en las líneas 4-29, sirve para determinar el tamaño que tendrá la imagen tras aplicarle la homografía.

Para determinar este tamaño, lo único que hacemos es aplicarle la homografía, con *cv::perspectiveTransform*, a las cuatro esquinas de la imagen —líneas 4-11—.

De las imágenes por la homografía de las esquinas determinamos los límites de las coordenadas *x* e *y*, viendo cuáles son los mínimos y los máximos² —líneas 13-22—.

En la línea 24 definimos entonces el tamaño de la imagen destino, con anchura igual a $x_{max} - x_{min}$ —que no es más que la anchura de la imagen transformada— y altura igual a $y_{max} - y_{min}$ —la altura de la imagen transformada—.

²La constante INF se define en el archivo *consts.hpp* como el último número en el rango de los enteros.

Ahora bien, si aplicáramos tal cual la homografía estimada a la imagen y la pusiéramos en la imagen destino, se saldría de sus límites —las coordenadas calculadas pueden ser negativas—. Por tanto, hace falta aplicar una traslación a la imagen transformada, que definimos en las líneas 27-29 como la matriz

$$T = \begin{pmatrix} 1 & 0 & -x_{min} \\ 0 & 1 & -y_{min} \\ 0 & 0 & 1 \end{pmatrix}$$

En la llamada a *cv::warpPerspective*, por tanto, tenemos que aplicar la homografía y luego la traslación; es decir, tenemos que pasarle la matriz $T \cdot H$, como se ve en la línea 31.

2. Detección de puntos clave

Para la detección de puntos clave, se ha implementado la siguiente función:

```

1 Mat Image::detectFeatures(enum detector_id det_id, vector<KeyPoint>
2   &keypoints){
3     // Declare detector
4     Ptr<Feature2D> detector;
5
6     // Define detector
7     if (det_id == detector_id::ORB) {
8       // Declare ORB detector
9       detector = ORB::create(
10         500,           //nfeatures = 500
11         1.2f,          //scaleFactor = 1.2f
12         4,             //nlevels = 8
13         21,            //edgeThreshold = 31
14         0,             //firstLevel = 0
15         2,             //WTA_K = 2
16         ORB::HARRIS_SCORE, //scoreType = ORB::HARRIS_SCORE
17         21,            //patchSize = 31
18         20             //fastThreshold = 20
19       );
20     }
21     else{
22       // Declare BRISK and BRISK detectors
23       detector = BRISK::create(
24         55,    // thresh = 30
25         8,     // octaves = 3
26         1.5f  // patternScale = 1.0f
27       );
28     }
29     // Declare array for storing the descriptors
30     Mat descriptors;
```

```

31     // Detect and compute!
32     detector->detect(this->image, keypoints);
33     detector->compute(this->image, keypoints, descriptors);
34
35     return descriptors;
36 }

```

El código es sencillo. Veamos las tres partes esenciales de las que consta.

2.1. Declaración y definición del detector

Para dar la posibilidad de usar tanto el detector ORB como el BRISK, la función *Image::detectFeatures* acepta como primer argumento un enumerado del tipo *descriptor_id*, definido en el archivo *consts.hpp* como sigue:

```

1 enum detector_id{
2     ORB,
3     BRISK
4 };

```

Tras definir un puntero genérico del tipo abstracto *Feature2D*, se le asigna un objeto del tipo especificado por el primer argumento, con parámetros elegidos de forma experimental —en los comentarios de cada línea se indica de qué parámetro se trata y cuál es su valor por defecto—.

2.2. Detección de los puntos clave

La detección de los puntos clave es independiente del descriptor usado —de ahí la ventaja de usar la clase abstracta—, y se realiza en la línea 33. Nótese que los puntos clave se guardan en el vector *keypoints*. Este se le pasa por referencia a la función, así que el usuario que llame a esta función tendrá en él la lista de los puntos clave cuando la función devuelva. Esto lo hacemos porque, además de los puntos clave, queremos devolver los descriptores, que pasamos a explicar.

2.3. Descripción de los puntos clave

Para obtener los descriptores de los puntos clave previamente calculados basta llamar a la función *compute*. El conjunto de descriptores es el objeto *Mat* que devuelve la función.

3. Establecimiento de los puntos clave en correspondencia

Ya tenemos una función para calcular puntos clave y descriptores de una imagen, pero eso no nos sirve de mucho si no somos capaces de ponerlos en correspondencia con los descriptores obtenidos de otra imagen.

Se define entonces la función *Image::match*, cuyo código es el siguiente:

```
1 pair< vector<Point2f>, vector<Point2f> > Image::match(Image
2     matched, enum descriptor_id descriptor , enum detector_id
3     detector){
4     // 1 - Get keypoints and its descriptors in both images
5     vector<KeyPoint> keypoints[2];
6     Mat descriptors[2];
7
8
9     // 2 - Match both descriptors using required detector
10    // Declare the matcher
11    Ptr<DescriptorMatcher> matcher;
12
13    // Define the matcher
14    if (descriptor == descriptor_id::BRUTE_FORCE) {
15        // For ORB and BRISK descriptors, NORM_HAMMING should be used.
16        // See http://sl.ugr.es/norm_ORB_BRISK
17        matcher = new BFMatcher(NORM_HAMMING, true);
18    }
19    else{
20        matcher = new FlannBasedMatcher();
21        // FlannBased Matcher needs CV_32F descriptors
22        // See http://sl.ugr.es/FlannBase_32F
23        for (size_t i = 0; i < 2; i++) {
24            if (descriptors[i].type() != CV_32F) {
25                descriptors[i].convertTo(descriptors[i],CV_32F);
26            }
27        }
28    }
29
30    // Match!
31    vector<DMatch> matches;
32    matcher->match( descriptors[0], descriptors[1], matches );
33
34    // 3 - Create lists of ordered keypoints following obtained
35    // matches
36    vector<Point2f> ordered_keypoints[2];
37
38    for( unsigned int i = 0; i < matches.size(); i++ )
39    {
```

```

39     // Get the keypoints from the matches
40     ordered_keypoints[0].push_back(
41         keypoints[0][matches[i].queryIdx].pt );
42     ordered_keypoints[1].push_back(
43         keypoints[1][matches[i].trainIdx].pt );
44 }
45 }
```

Esta función se llama sobre un objeto `Image`, recibe otro objeto `Image` con el que se quiere calcular su relación y devuelve un vector de pares de puntos en correspondencia, siendo el primer elemento del par un punto de la imagen sobre la que se ha llamado la función `y`, el segundo, el punto de la otra imagen que le corresponde.

Además, recibe dos parámetros adicionales para indicar qué tipos de detector y de descriptor usar.

Veamos el código por partes.

3.1. Obtención de los puntos clave

Lo primero que se necesita calcular es el vector de puntos clave `y`, con él, el de descriptores, asociado a cada imagen. Esto se hace con la función explicada en la sección anterior, llamándola sobre las dos imágenes que tenemos —líneas 3-7—. Nótese que tanto la variable `keypoints` como `descriptors` está declarada doble: en ambos, el elemento $i = 0$ es el correspondiente a la imagen sobre la que se ha llamado la función `y`, el $i = 1$, el correspondiente a la imagen pasada como argumento.

3.2. Emparejamiento de puntos

Podemos ya entonces declarar la variable `matcher` que, como en el caso del detector de puntos clave, es un puntero de tipo abstracto, en este caso `DescriptorMatcher`. Esto nos permite generalizar el código para cualquier tipo de emparejador particularizando únicamente la creación del mismo.

Esta creación se hace en las líneas 14-28, donde se le asigna un objeto a la variable `matcher` dependiente del argumento `detector_id`. Nótese que ambos tienen una peculiaridad que hay que tratar:

- En el caso del `BMatcher`, tenemos que usar la distancia *Hamming* si queremos trabajar sobre descriptores *ORB* o *BRISK*. Ver documentación de OpenCV³.

³http://sl.ugr.es/norm_ORB_BRISK

- En el caso del *FlannBasedMatcher*, los descriptores tienen que tener tipo *CV_32F*, así que se convierten si tienen tipo diferente. Ver pregunta en los foros de OpenCV⁴.

Una vez definidos, no hay más que llamar a la función *match* del objeto *DescriptorMatcher* que tengamos definido y dejar que OpenCV trabaje. Obtenemos entonces un vector de variables *DMatch*, que guardan los puntos emparejados y la distancia que hay entre sus descriptores.

Además, se ha implementado una función auxiliar para usar en los ejemplos cuyo código no hace más que tomar los puntos clave, calcular los descriptores, ejecutar el *matcher* y llamar a la función *cv::drawMatches* para dibujar los puntos en correspondencia.

```

1 void Image::drawMatches(Image other){
2     vector<KeyPoint> keypoints[2];
3     Mat descriptors[2];
4
5     descriptors[0] = this->detectFeatures(detector_id::ORB,
6         keypoints[0]);
7     descriptors[1] = other.detectFeatures(detector_id::ORB,
8         keypoints[1]);
9
10    Ptr<DescriptorMatcher> matcher = new BFMatcher(NORM_HAMMING,
11        true);
12    vector<DMatch> matches;
13    matcher->match( descriptors[0], descriptors[1], matches );
14
15    Mat draw_matches;
16    cv::drawMatches(this->image, keypoints[0], other.image,
17        keypoints[1], matches, draw_matches);
18    Image drawing(draw_matches);
19    drawing.setName("Ejemplo de drawMatches")
20    drawing.draw();
21
22 }
```

3.3. Creación de la lista de puntos emparejados

La información del vector *matches* hay que traducirla a un par de vectores ordenados, que es lo que devuelve nuestra función.

Por tanto, en las líneas 35-42 se definen y llenan estos vectores, usando los índices que proporcionan las entradas del vector *matches*.

⁴http://sl.ugr.es/FlannBase_32F

4. Creación de mosaicos

La creación de un mosaico con dos imágenes se puede generalizar a la creación de un mosaico con N imágenes.

En un principio se implementó la siguiente función para el apartado 4 de la práctica:

```
1 Image Image::createMosaic(Image matched){
2     assert(this->image.type() == matched.image.type());
3
4     // Find homography transforming "matched" image plane into own
5     // plane:
6     pair< vector<Point2f>, vector<Point2f> > matched_points =
7         this->match(matched, descriptor_id::BRUTE_FORCE,
8             detector_id::ORB);
9     Mat homography = cv::findHomography(matched_points.second,
10        matched_points.first, cv::RANSAC, 1);
11
12    // Build the mosaic canvas and declare a header for its left half
13    Mat mosaic(Size(this->cols() + matched.cols(), this->rows(),
14        CV_8UC3, Scalar(0,0,0));
15    Mat left_slot = mosaic( Rect(0,0,matched.cols(),matched.rows()) );
16
17    // Copy own image to the mosaic
18    this->image.copyTo( left_slot );
19
20    // Copy "matched" image to the mosaic applying the homography
21    cv::warpPerspective( matched.image, mosaic, homography,
22        mosaic.size(), INTER_LINEAR, BORDER_TRANSPARENT );
23
24    return Image(mosaic);
25 }
```

Esta primera versión, sin embargo, se mejoró y generalizó después para poder hacer un mosaico con N imágenes, en respuesta al apartado 5 de la práctica. Como esta función sirve para ambos apartados y es, de hecho, la que se usa con los datos de prueba tanto para el caso $N = 2$ como para el caso $N > 2$, es la que vamos a discutir en este informe. Su código completo es el que sigue:

```
1 Image createMosaic_N(vector<Image> &images){
2     // Number of images and middle image index
3     int N = images.size();
4     int mid_idx = N/2; // Integer division, works both with odd and
5     // even numbers.
6
7     // Image and homography types
8     int img_type = images.front().image.type();
```

```

8   int homo_type = CV_64F;
9
10  // Declare homographies vectors
11  vector<Mat> left_homographies, right_homographies;
12
13  // Homographies between images on the left side of the central
14  // one
14  for (int i = 0; i < mid_idx; i++) {
15      // Compute matched points between image i and i+1
16      pair< vector<Point2f>, vector<Point2f> > matched_points =
17          images[i].match(images[i+1], descriptor_id::BRUTE_FORCE,
18                          detector_id::ORB);
19
18      // Find homography transforming image i plane into image i+1
19      // plane
20      left_homographies.push_back(cv::findHomography(matched_points.first,
21                                              matched_points.second, cv::RANSAC, 1));
20 }
21
22 // Homographies between images on the right side of the central
23 // one
23 for (int i = N-1; i > mid_idx; i--) {
24     // Compute matched points between image i and i-1
25     pair< vector<Point2f>, vector<Point2f> > matched_points =
26         images[i].match(images[i-1], descriptor_id::BRUTE_FORCE,
27                         detector_id::ORB);
27
26     // Find homography transforming image i plane into image i-1
27     // plane
28     right_homographies.push_back(cv::findHomography(matched_points.first,
29                                     matched_points.second, cv::RANSAC, 1));
29 }
30
31 int left_size = left_homographies.size();
32 // Compute left homographies composition and store them in the
33 // same vector
33 for (int i = 0; i < left_size; i++) {
34     Mat homo_composition = Mat::eye(3,3,homo_type);
35     for (int j = left_size-1; j >= i; j--) {
36         homo_composition = homo_composition *
37             left_homographies[j];
38     }
39     left_homographies[i] = homo_composition;
39 }
40
41 int right_size = right_homographies.size();
42 // Compute right homographies composition and store them in the
43 // same vector
43 for (int i = 0; i < right_size; i++) {

```

```

44     Mat homo_composition =
45         Mat::eye(3,3,right_homographies[0].type());
46     for (int j = right_size-1; j >= i; j--) {
47         homo_composition = homo_composition *
48             right_homographies[j];
49     }
50     right_homographies[i] = homo_composition;
51 }
52 // Declare a vector with all the homographies without
53 // translation to the mosaic coordinates
54 vector<Mat> homographies;
55 reverse(right_homographies.begin(),right_homographies.end());
56 homographies.insert(homographies.end(),
57     left_homographies.begin(), left_homographies.end());
58 homographies.push_back(Mat::eye(3,3,homo_type));
59 homographies.insert(homographies.end(),
60     right_homographies.begin(), right_homographies.end());
61
62 // Get homography image of the corner coordinates from all the
63 // images to obtain mosaic size
64 vector<Point2f> corners_all(4), corners_all_t(4);
65 float min_x, min_y, max_x, max_y;
66 min_x = min_y = +INF;
67 max_x = max_y = -INF;
68
69 for (int i = 0; i < N; i++) {
70     corners_all[0] = Point2f(0,0);
71     corners_all[1] = Point2f(images[i].cols(),0);
72     corners_all[2] = Point2f(images[i].cols(),images[i].rows());
73     corners_all[3] = Point2f(0,images[i].rows());
74
75     perspectiveTransform(corners_all, corners_all_t,
76         homographies[i]);
77
78     for (int j = 0; j < 4; j++) {
79         min_x = min(min(corners_all[j].x,corners_all_t[j].x),
80             min_x);
81         max_x = max(max(corners_all[j].x,corners_all_t[j].x),
82             max_x);
83
84         min_y = min(min(corners_all[j].y,corners_all_t[j].y),
85             min_y);
86         max_y = max(max(corners_all[j].y,corners_all_t[j].y),
87             max_y);
88     }
89 }
90 int mosaic_cols = max_x - min_x;

```

```

82     int mosaic_rows = max_y - min_y;
83
84     // Create mosaic canvas
85     Size mosaic_size(mosaic_cols, mosaic_rows);
86     Mat mosaic(mosaic_size, img_type, Scalar(0,0,0));
87
88     // Define translation homography
89     Mat trans_homography = Mat::eye(3,3,homo_type);
90     trans_homography.at<double>(0,2) = -min_x;
91     trans_homography.at<double>(1,2) = -min_y;
92
93     for (size_t i = 0; i < homographies.size(); i++) {
94         Mat curr_homography = trans_homography * homographies[i];
95         cv::warpPerspective( images[i].image, mosaic,
96             curr_homography, mosaic_size, INTER_LINEAR,
97             BORDER_TRANSPARENT );
98     }
99

```

El código es largo pero muy sencillo; vayamos por partes.

4.1. Definiciones previas

Atendamos ahora al siguiente trozo de código de la función anterior:

```

1 // Number of images and middle image index
2 int N = images.size();
3 int mid_idx = N/2; // Integer division, works both with odd and
4 // even numbers.
5
6 // Image and homography types
7 int img_type = images.front().image.type();
int homo_type = CV_64F;

```

Aquí definimos como N el número de imágenes y guardamos, en mid_idx , el índice de la imagen que tomaremos como referencia. Como vamos a realizar una proyección plana, para minimizar las deformaciones geométricas dejaremos fija la imagen central y calcularemos las homografías que lleven todas las demás a esta. Si la que dejáramos fija fuera la primera imagen, la última sufriría una deformación demasiado grande. Con el código actual, las deformaciones geométricas son las mínimas en proyección plana.

Nótese además que la definición de mid_idx la hacemos con una división de enteros: si N es impar, el índice será la parte entera de $N/2$; esto es, el índice —de 0 a $N - 1$ — que tendrá nuestra imagen. Si N es par, el índice será exactamente $N/2$; así, de las dos posibles imágenes centrales, elegimos la de la derecha. Esto es, si N es par, la imagen *central* dejará siempre a la

izquierda una imagen más que a la derecha.

Después, declaramos variables que guardan los tipos de dato con los que estamos trabajando. La variable *homo_type* será muy importante a la hora de multiplicar homografías definidas a mano —con objetos *Mat* en los que tenemos que indicar el tipo— con homografías devueltas por *cv::findHomography*, que son del tipo *homo_type*.

4.2. Definición de homografías

Veamos ahora cómo definir todas las homografías que necesitamos.

Primero calculamos homografías entre imágenes adyacentes con el siguiente código:

```
1 // Homographies between images on the left side of the central one
2 for (int i = 0; i < mid_idx; i++) {
3     // Compute matched points between image i and i+1
4     pair<vector<Point2f>, vector<Point2f>> matched_points =
5         images[i].match(images[i+1], descriptor_id::BRUTE_FORCE,
6                         detector_id::ORB);
7
8     // Find homography transforming image i plane into image i+1
9     // plane
10    left_homographies.push_back(cv::findHomography(matched_points.first,
11                                                 matched_points.second, cv::RANSAC, 1));
12
13    }
14
15 // Homographies between images on the right side of the central one
16 for (int i = N-1; i > mid_idx; i--) {
17     // Compute matched points between image i and i-1
18     pair<vector<Point2f>, vector<Point2f>> matched_points =
19         images[i].match(images[i-1], descriptor_id::BRUTE_FORCE,
20                         detector_id::ORB);
21
22     // Find homography transforming image i plane into image i-1
23     // plane
24     right_homographies.push_back(cv::findHomography(matched_points.first,
25                                     matched_points.second, cv::RANSAC, 1));
26 }
```

Como vemos se calculan por separado las homografías entre imágenes a la izquierda de la imagen central y a la derecha de esta, ya que el sentido de la homografía es distinto. A la izquierda, calculamos la homografía que lleva la imagen I_i a la imagen I_{i+1} . A la derecha, la que lleva la imagen I_{j+1} a la imagen I_j .

Podríamos haberlas hecho todas a la vez y con el mismo sentido, tomando luego inversas para las de la derecha, pero esto facilita el cálculo de la homografía composición siguiente, que lleva cada imagen a la imagen central.

Para calcular esta composición, usamos el siguiente código:

```

1 int left_size = left_homographies.size();
2 // Compute left homographies composition and store them in the same
   vector
3 for (int i = 0; i < left_size; i++) {
4     Mat homo_composition = Mat::eye(3,3,homo_type);
5     for (int j = left_size-1; j >= i; j--) {
6         homo_composition = homo_composition * left_homographies[j];
7     }
8     left_homographies[i] = homo_composition;
9 }
10
11 int right_size = right_homographies.size();
12 // Compute right homographies composition and store them in the
   same vector
13 for (int i = 0; i < right_size; i++) {
14     Mat homo_composition = Mat::eye(3,3,right_homographies[0].type());
15     for (int j = right_size-1; j >= i; j--) {
16         homo_composition = homo_composition * right_homographies[j];
17     }
18     right_homographies[i] = homo_composition;
19 }
```

Centrémonos en la primera parte, que calcula las homografías a la izquierda de la imagen central: tenemos un vector en el que la posición i guarda la homografía H_i , que lleva la imagen I_i en la I_{i+1} . Como queremos calcular la homografía que lleva la imagen i en la central, necesitamos hacer el producto

$$H_{mid_idx-1} \cdot H_{mid_idx-2} \cdots H_{i+1} \cdot H_i$$

así que, para cada i , iteramos de forma inversa desde la última homografía, que es la H_{mid_idx-1} hasta la i , haciendo el producto en el orden adecuado.

La parte correspondiente a las imágenes de la derecha, gracias al bucle que hicimos anteriormente al revés, se hace exactamente igual, pues el índice i contiene la homografía entre la imagen $N - i$ y la anterior —pensar, por ejemplo, en $i = 0$: la posición 0 contiene la homografía de la última imagen a la penúltima—.

Siguiendo con el código vemos, que después, *linealizamos* todas las homografías en un sólo vector, que en la posición i contendrá la homografía que lleve la imagen i a la imagen central. Esto se hace de una forma muy sencilla:

```

1 // Declare a vector with all the homographies without translation
   to the mosaic coordinates
2 vector<Mat> homographies;
3 reverse(right_homographies.begin(),right_homographies.end());
4
```

```

5 homographies.insert(homographies.end(), left_homographies.begin(),
6   left_homographies.end());
7 homographies.push_back(Mat::eye(3,3,homo_type));
7 homographies.insert(homographies.end(), right_homographies.begin(),
8   right_homographies.end());

```

A continuación, hacemos una generalización del código que escribimos en la implementación de la función *Image::warpPerspective*, calculando para cada imagen las coordenadas de las esquinas tras aplicarle la transformación sufrida por la homografía correspondiente. Iterando sobre todas esas esquinas calculamos las coordenadas mínimas y máximas en cada eje y así podemos definir el tamaño del mosaico y la homografía translación que moverá la imagen central a su posición en el mosaico. El código es el siguiente:

```

1 // Get homography image of the corner coordinates from all the
2   images to obtain mosaic size
3 vector<Point2f> corners_all(4), corners_all_t(4);
4 float min_x, min_y, max_x, max_y;
5 min_x = min_y = +INF;
5 max_x = max_y = -INF;
6
7 for (int i = 0; i < N; i++) {
8   corners_all[0] = Point2f(0,0);
9   corners_all[1] = Point2f(images[i].cols(),0);
10  corners_all[2] = Point2f(images[i].cols(),images[i].rows());
11  corners_all[3] = Point2f(0,images[i].rows());
12
13 perspectiveTransform(corners_all, corners_all_t,
14   homographies[i]);
14
15 for (int j = 0; j < 4; j++) {
16   min_x = min(min(corners_all[j].x,corners_all_t[j].x), min_x);
17   max_x = max(max(corners_all[j].x,corners_all_t[j].x), max_x);
18
19   min_y = min(min(corners_all[j].y,corners_all_t[j].y), min_y);
20   max_y = max(max(corners_all[j].y,corners_all_t[j].y), max_y);
21 }
22 }
23 int mosaic_cols = max_x - min_x;
24 int mosaic_rows = max_y - min_y;
25
26 // Create mosaic canvas
27 Size mosaic_size(mosaic_cols, mosaic_rows);
28 Mat mosaic(mosaic_size, img_type, Scalar(0,0,0));
29
30 // Define translation homography
31 Mat trans_homography = Mat::eye(3,3,homo_type);
32 trans_homography.at<double>(0,2) = -min_x;
33 trans_homography.at<double>(1,2) = -min_y;

```

Después de todos los cálculos y definiciones previas, podemos por fin crear el mosaico, iterando sobre todas las imágenes y aplicándole la composición de la homografía que la lleva a la central con la homografía traslación que lleva la central al mosaico. El código se reduce a llamar a *cv::warpPerspective* sobre todas las imágenes con sus homografías correspondientes multiplicadas por la de traslación:

```
1 for (size_t i = 0; i < homographies.size(); i++) {  
2     Mat curr_homography = trans_homography * homographies[i];  
3     cv::warpPerspective( images[i].image, mosaic, curr_homography,  
4                           mosaic_size, INTER_LINEAR, BORDER_TRANSPARENT );  
5 }  
6 return Image(mosaic);
```

5. Análisis de resultados

5.1. Estimación de una homografía

En la estimación de la homografía, no es tan importante cómo de bien se ajuste la homografía a los puntos sino cómo de bien se ajusten los puntos entre sí. Podemos ver esto con los dos conjuntos de puntos definidos sobre *Tablero1.jpg* y *Tablero2.jpg*.

Con el conjunto de puntos repartidos por toda la imagen, el resultado de la estimación de la homografía es el que vemos en la figura 1.

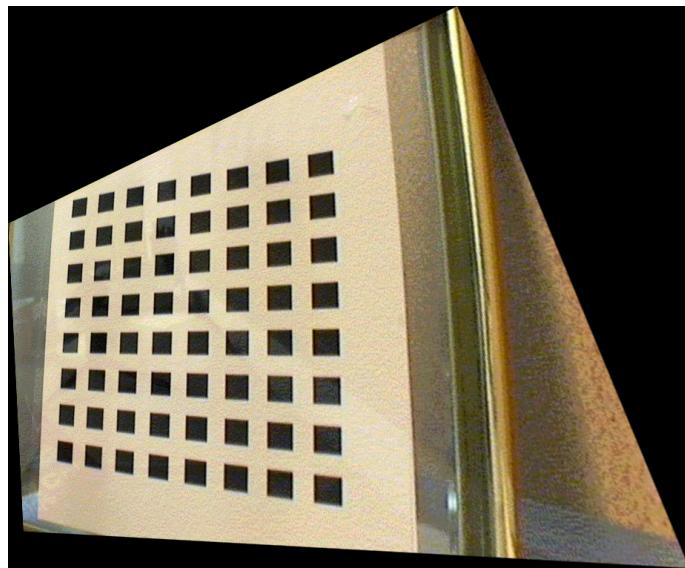


Figura 1: Primera transformación del tablero 2 en el 1.

Además, el valor de σ_9 , que denota el grado de bondad de la solución —mejor cuanto más cercano a cero sea—, es $\sigma_9 = 0,309116$.

Si aplicamos la misma función con el otro conjunto de puntos —todos en las casillas adyacentes de la esquina superior izquierda—, tenemos el resultado que muestra la figura 2.

El grado de bondad de esta slución, por otro lado, es $\sigma_9 = 0,0219701$

Como vemos, aunque la homografía calculada para el segundo conjunto de datos es mucho mejor, el resultado es desastroso. Las tres esquinas de donde se tomaron los datos encajan perfectamente con la imagen *Tablero1.jpg*, pero cuanto más nos acercamos a la esquina inferior derecha, peor es la transformación. Esto evidencia el comportamiento de nuestro algoritmo: es mucho mejor coger puntos dispersos y que la homografía encaje peor con ellos que coger puntos muy juntos para los que la homografía esté definida perfectamente.

Cualquier mínimo error en la selección de estos puntos implicará un error

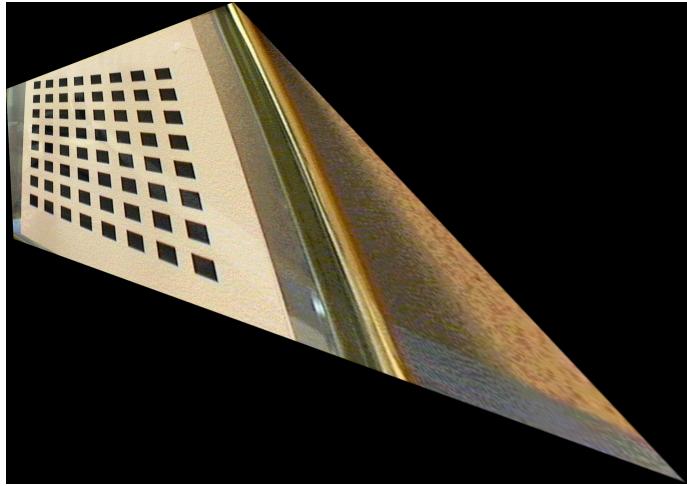


Figura 2: Segunda transformación del tablero 2 en el 1.

enorme en los puntos alejados. Sin embargo, si dispersamos los puntos, el error general de la transformación será mucho más estable ante los errores locales que cometamos en la selección de puntos.

5.2. Detección de puntos clave

Los resultados de la detección de puntos clave depende mucho de los parámetros con los que se inicialicen los detectores. La figura 3 muestra el resultado de los detectores *ORB* y *BRISK* sobre las dos primeras imágenes de Yosemite, usando los valores por defecto.

Como vemos, *BRISK* encuentra muchos puntos clave y *ORB* bastantes menos.

Si modificamos los parámetros de ambos inicializadores, basados en un proceso de experimentación en el que vemos qué parámetros mejoran qué puntos, podemos llegar a la versión que finalmente se usa en el código, y que es la que muestra la figura 4. Como vemos, se reduce significativamente el número de puntos clave detectados por *BRISK* para quedarnos con los más destacados y aumentamos un poco los que detecta *ORB*. Los parámetros usados son los que vimos en la sección en el que se analiza este código.

5.3. Emparejamiento de puntos clave

En esta sección probamos el algoritmo de emparejamiento con las técnicas *BruteForce* y *FlannBased*.

En las imágenes 5 y 6 de Yosemite podemos ver ambos emparejadores actuando sobre unos mismos datos de entrada. Aunque la diferencia no es muy grande, parece que los falsos positivos son más numerosos con la técnica *FlannBased*.

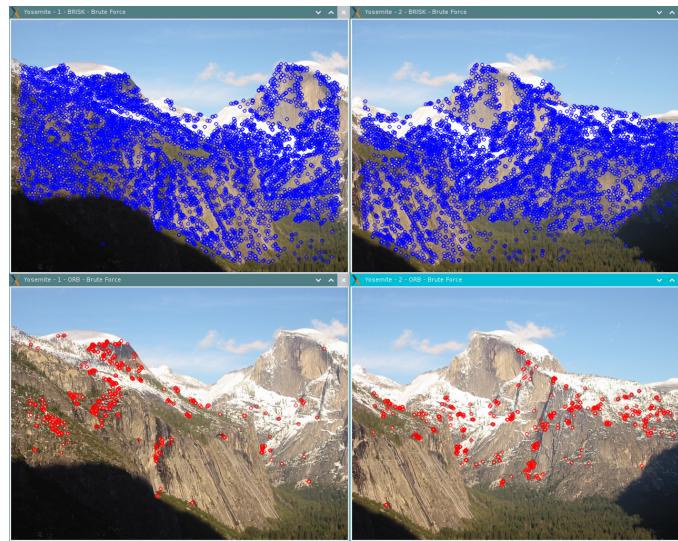


Figura 3: Detección con BRISK y ORB con parámetros por defecto.

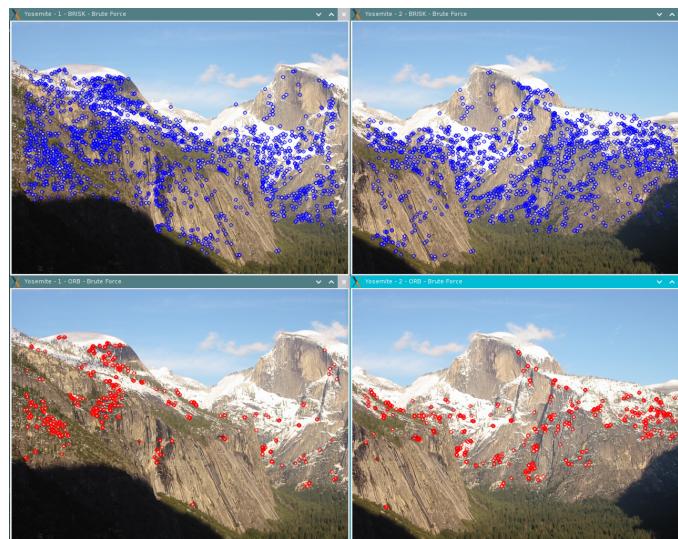


Figura 4: Detección con BRISK y ORB con parámetros cambiados.

En las imágenes 7 y 8, de la ETSIIT, podemos ver con más claridad cómo *FlannBased* tiene muchos más falsos positivos que *BruteForce* —todas las líneas deberían ser prácticamente horizontales, pues el cambio de vista es en esa dirección, pero en la figura 8 se ven muchas diagonales—.

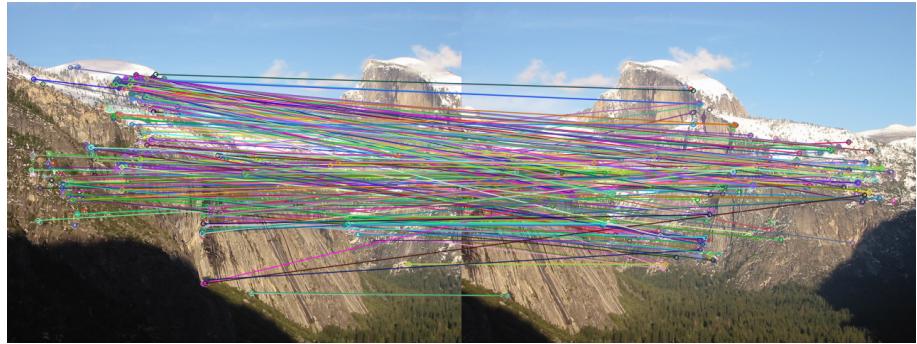


Figura 5: BruteForce+crossCheck en una imagen de Yosemite.

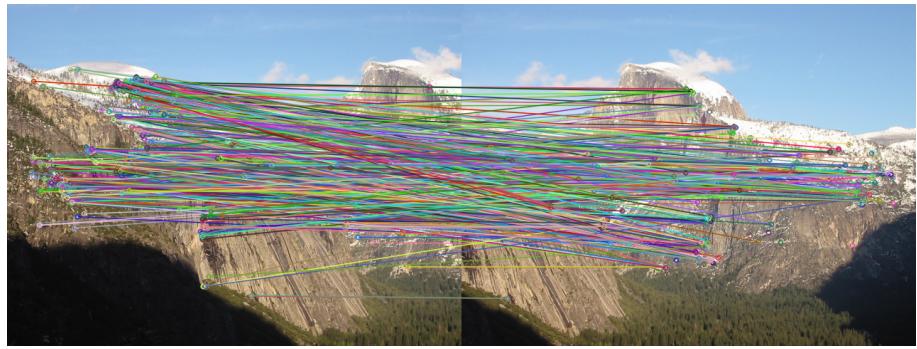


Figura 6: FlannBased en una imagen de Yosemite.



Figura 7: BruteForce+crossCheck en una imagen de la ETSIIT.

5.4. Creación de un mosaico

La creación del mosaico depende intensamente de todas y cada una de las discusiones anteriores, pero no introduce nuevos parámetros ni técnicas cuya discusión sea muy interesante.

Podemos sin embargo ver en acción el resultado de todos los pasos anteriores, observando además que el proceso de ajuste del tamaño del



Figura 8: FlannBased en una imagen de la ETSIIT.

mosaico y de la translación de las imágenes a él funciona correctamente.

En los siguientes ejemplos se ha usado el detector *ORB* —con los parámetros fijados experimentalmente que vimos antes— en conjunción con la técnica *BruteForce* para poner en correspondencia los puntos clave detectados. La decisión sobre la técnica de emparejamiento está clara en virtud de los ejemplos vistos anteriormente. La decisión sobre el detector tiene más que ver con eficiencia —el detector *BRISK* emplea más tiempo en hacer el mismo cálculo—, que con optimización del resultado.

La figura 9 muestra el resultado de hacer un mosaico con sólo 2 imágenes, *yosemite1.jpg* y *yosemite2.jpg*. Vemos que, aunque se nota la línea entre las dos imágenes —la correcta fusión de colores se queda como trabajo pendiente—, la fusión geométrica es perfecta.



Figura 9: Mosaico de Yosemite con dos imágenes.

La figura 10 muestra el resultado de hacer un mosaico con 10 imágenes de la ETSIIT. La geometría, de nuevo, encaja casi perfectamente —vemos algún fallo en alguna unión, pero casi imperceptible—. Aquí, sin embargo, podemos ver otro de los problemas que dejamos como trabajo pendiente: si hay objetos en movimiento —como el coche que está en la rotonda—, pueden aparecer artefactos extraños —vemos que hay sólo medio coche, pues en la foto anterior aún no estaba en esa posición—. En este ejemplo se observa, además, que el algoritmo se comporta bien aunque los cambios de visión sean grandes, como en la foto de la esquina inferior derecha.

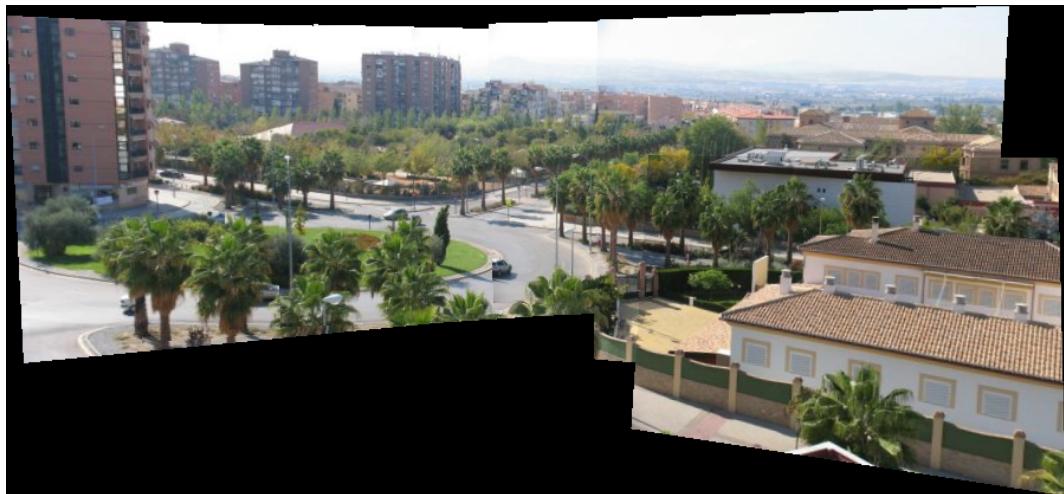


Figura 10: Mosaico de Yosemite con dos imágenes.