

# Informe de proyecto: Implementación del algoritmo de rectificación de Loop & Zhang

Antonio Álvarez Caballero  
Alejandro García Montoro  
[analca3@correo.ugr.es](mailto:analca3@correo.ugr.es)  
[agarciamontoro@correo.ugr.es](mailto:agarciamontoro@correo.ugr.es)

## 1. Descripción del problema

La rectificación de imágenes es el proceso de aplicar sendas homografías a un par de imágenes cuya geometría epipolar conocemos, de modo que las líneas epipolares queden horizontales, paralelas entre sí y con la misma coordenada vertical.

La motivación de la rectificación es simple: al tener las líneas epipolares horizontales y con la misma coordenada vertical, buscar correspondencias entre las dos imágenes es mucho más fácil y eficiente; como sabemos que las correspondencias se encuentran en las mismas líneas epipolares, si suponemos que las imágenes están rectificadas basta buscar en la correspondiente fila de píxeles de la otra imagen, donde se encuentra la línea epipolar del punto que estemos considerando. Si las imágenes no estuvieran rectificadas, el tiempo de ejecución aumentaría considerablemente; la búsqueda en líneas inclinadas —no paralelas con ninguno de los dos ejes— es más compleja computacionalmente.

Así, es claro que tener un par de imágenes rectificadas mejora la eficiencia de muchos de los algoritmos que precisan conocer la geometría epipolar y tienen que buscar correspondencias entre ambas imágenes. Por ejemplo, de esta situación se consigue la mejora computacional de los algoritmos que calculan mapas de disparidad, donde hay que medir el desplazamiento relativo entre los píxeles de una y otra imagen. Esta distancia relativa es lo que conocemos por *disparidad*, y con esta información se puede reconstruir la profundidad incluso sin noción alguna de cámaras; basta un par de imágenes de una misma escena estática.

Este proyecto implementa el método de rectificación propuesto por Charles Loop y Zhengyou Zhang en [2].

Su algoritmo es totalmente geométrico y no precisa conocimiento de las cámaras. Se consigue descomponiendo la homografía que se desea calcular para cada imagen en una proyección, una transformación euclídea y una cizalla, con especial cuidado en minimizar la distorsión del resultado con respecto a las imágenes originales.

Es de hecho este último criterio el que hace del algoritmo una de las mejores soluciones actuales, ya que se consigue una rectificación que mantiene casi todas las propiedades de las imágenes originales.

## 2. Descripción de la solución implementada

La implementación del algoritmo se basa en la descomposición de la homografía  $H$  que se quiere obtener en las siguientes transformaciones:

- $H_p$ : transformación proyectiva.
- $H_r$ : transformación de semejanza.
- $H_s$ : composición de una transformación de cizalla con un escalado y una traslación.

## 2.1. Descomposición

Siguiendo la notación de [2], sea  $H$  la homografía siguiente:

$$H = \begin{pmatrix} u_a & u_b & u_c \\ v_a & v_b & v_c \\ w_a & w_b & w_c \end{pmatrix}$$

Si descomponemos  $H$  en su parte proyectiva y afín —y esta a su vez en una parte de semejanza y otra de cizalla—, obtenemos que

$$H = H_s H_r H_p$$

donde

$$H_s = \begin{pmatrix} s_a & s_b & s_c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}; \quad H_r = \begin{pmatrix} v_b - v_c w_b & v_c w_a - v_a & 0 \\ v_a - v_c w_a & v_b - v_c w_b & v_c \\ 0 & 0 & 1 \end{pmatrix}; \quad H_p = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ w_a & w_b & 1 \end{pmatrix}$$

El cálculo de cada transformación se hace por separado, teniendo en cuenta las matrices calculadas anteriormente y criterios de minimización que se explicarán en cada subsección.

Antes de todo este cálculo, sin embargo, hace falta calcular la geometría epipolar del par de imágenes, que el algoritmo supone conocida. Este paso previo se ha implementado buscando las correspondencias con un detector `BRISK`, calculando la matriz fundamental con la llamada a la función `findFundamentalMat` de *OpenCV*—usando el algoritmo de los 8 puntos y `RANSAC`— y usando las llamadas a `computeCorrespondEpilines` de la misma librería.

## 2.2. Transformación proyectiva

La transformación proyectiva lleva los epipolos  $e$  y  $e'$  al infinito, de manera que tras aplicar  $H_p$  y  $H'_p$ , las líneas epipolares son ya paralelas.

El cálculo de  $w_a$ ,  $w_b$  y  $w'_a$ ,  $w'_b$  se basa en un criterio de minimización de la distorsión que puede resumirse en lo siguiente: buscamos unas transformaciones proyectivas  $H_p$  y  $H'_p$  que sean lo más cercanas que podamos a transformaciones afines.

La idea es que las líneas  $w = [w_a, w_b, 1]$  y  $w' = [w'_a, w'_b, 1]$  transformen los puntos de la manera más compensada posible; es decir, dado un punto  $p_i = [p_{i,u}, p_{i,v}, 1]$ , la transformación  $H_p$  lo transformará en el punto  $[\frac{p_{i,u}}{\omega_i}, \frac{p_{i,v}}{\omega_i}, 1]$ , donde los pesos  $\omega_i$  pueden medir de alguna manera cuán proyectiva es nuestra transformación: si todos los pesos son idénticos, la transformación es afín.

Buscamos por tanto unas líneas  $w$  y  $w'$  que consigan que los pesos  $\omega_i$  se parezcan todo lo posible.

Toda la discusión matemática se encuentra descrita en [2], transformando el problema propuesto en la minimización de la siguiente expresión:

$$\frac{z^T A z}{z^T B z} + \frac{z^T A' z}{z^T B' z} \quad (1)$$

donde las matrices  $A$ ,  $B$  y sus homólogas con primas son conocidas y el vector  $z = [\lambda, 1, 0]$  depende de un único parámetro  $\lambda$  y es tal que  $w = [e]_t \text{imesz}$  y  $w' = Fz$ .

El cálculo de las matrices  $A$ ,  $B$ ,  $A'$  y  $B'$  depende únicamente de por qué matriz se multiplica:  $[e]_\times$  si queremos calcular  $A$  y  $B$  o la matriz fundamental  $F$  si queremos calcular  $A'$  y  $B'$ :

$$\begin{aligned} A &= [e]_\times^T P P^T [e]_\times & A' &= F^T P' P'^T F \\ B &= [e]_\times^T p_c p_c^T [e]_\times & B' &= F^T p'_c p_c'^T F \end{aligned}$$

Este cálculo se encuentra implementado en la siguiente función, donde la matriz `mult_mat` representa  $[e]_\times$  o  $F$  según si queremos calcular las versiones normales o las primas:

```

1  void obtainAB(const Mat &img, const Mat &mult_mat, Mat &A, Mat &B){
2      int width = img.cols;
3      int height = img.rows;
4
5      int size = 3;
6
7      Mat PPt = Mat::zeros(size, size, CV_64F);
8
9      PPt.at<double>(0,0) = width*width - 1;
10     PPt.at<double>(1,1) = height*height - 1;
11
12     PPt *= (width*height) / 12.0;
13
14     double w_1 = width - 1;
15     double h_1 = height - 1;
16
17     double values[3][3] = {
18         {w_1*w_1, w_1*h_1, 2*w_1},
19         {w_1*h_1, h_1*h_1, 2*h_1},
20         {2*w_1, 2*h_1, 4}
21     };
22
23     Mat pcpct(size, size, CV_64F, values);
24
25     pcpct /= 4;
26     A = mult_mat.t() * PPt * mult_mat;
27     B = mult_mat.t() * pcpct * mult_mat;
28 }
```

Desarrollando la expresión descrita en 1 obtenemos un polinomio en  $\lambda$  cuyo mínimo es el valor que buscamos. Este mínimo se alcanza cuando su derivada con respecto a  $\lambda$  es cero, así que todo nuestro problema se reduce a encontrar esta raíz.

### 2.2.1. Primera aproximación a la raíz

En [2] se da una primera aproximación a esta raíz, cuyo cálculo consiste en minimizar por separado ambos sumandos y tomar la media normalizada de las dos soluciones.

De nuevo, allí se encuentra toda la discusión matemática, que se reduce a hacer lo siguiente para ambos sumandos:

- Hacer la descomposición de Cholesky de la matriz  $A$  (resp.  $A'$ ) —que es simétrica y definida positiva— para obtener una matriz triangular superior  $D$  tal que  $A = D^T D$  (resp.  $A' = D'^T D'$ ).

- Encontrar el vector propio  $y$  (resp.  $y'$ ) asociado al mayor valor propio de  $(D^{-1})^T B D^{-1}$  (resp.  $(D'^{-1})^T B' D'^{-1}$ ).
- Tomar  $z_1 = D^{-1}y$  (resp.  $z_2 = D'^{-1}y'$ )

Se toma entonces como primera aproximación a la raíz el siguiente vector:

$$\frac{1}{2} \left( \frac{z_1}{\|z_1\|} + \frac{z_2}{\|z_2\|} \right)$$

Toda este cálculo se encuentra implementado en la siguiente función:

```

1  Vec3d getInitialGuess(Mat &A, Mat &B, Mat &Ap, Mat &Bp){
2
3      Vec3d z_1 = maximize(A, B);
4      Vec3d z_2 = maximize(Ap, Bp);
5
6      return (normalize(z_1) + normalize(z_2))/2;
7  }
```

donde el código realmente interesante está en `maximize`:

```

1  Vec3d maximize(Mat &A, Mat &B){
2      Mat D; // Output of cholesky decomposition: upper triangular matrix.
3      if( choleskyCustomDecomp(A, D) ){
4
5          Mat D_inv = D.inv();
6
7          Mat DBD = D_inv.t() * B * D_inv;
8
9          // Solve the equations system using eigenvalue decomposition
10
11         Mat eigenvalues, eigenvectors;
12         eigen(DBD, eigenvalues, eigenvectors);
13
14         // Take largest eigenvector
15         Mat y = eigenvectors.row(0);
16
17         Mat sol = D_inv*y.t();
18
19         Vec3d res(sol.at<double>(0,0), sol.at<double>(1,0), sol.at<double>(2,0));
20
21         return res;
22     }
23
24     // At this point, there is an error!
25     Mat eigenvalues;
26     eigen(A, eigenvalues);
27
28     cout << "\n\n\n————— WARNING ———<
29
30     return Vec3d(0, 0, 0);
31 }
```

Aunque *OpenCV* pone a disposición del usuario una función que realiza el cálculo de la descomposición de Cholesky, su uso en este trabajo introducía errores de redondeo que impedían realizar la descomposición. En ocasiones calculaba valores propios muy cercanos a cero —del

orden de  $10^{-5}$ — que tomaba como negativos, comprobando entonces erróneamente que la matriz  $A$  no era definida positiva.

Para evitar estos fallos y tener total control sobre los cálculos, decidimos implementar una función propia que calculara esta descomposición. La función es la que sigue, e implementa las fórmulas descritas en [1].

```
1  bool choleskyCustomDecomp(const Mat &A, Mat &L){
2
3      L = Mat::zeros(3,3,CV_64F);
4
5      for (int i = 0; i < 3; i++){
6          for (int j = 0; j <= i; j++){
7              double sum = 0;
8              for (int k = 0; k < j; k++){
9                  sum += L.at<double>(i,k) * L.at<double>(j,k);
10             }
11
12             L.at<double>(i,j) = A.at<double>(i,j) - sum;
13             if (i == j){
14                 if (L.at<double>(i,j) < 0.0){
15                     if (L.at<double>(i,j) > -1e-5){
16                         L.at<double>(i,j) *= -1;
17                     }
18                     else{
19                         return false;
20                     }
21                 }
22                 L.at<double>(i,j) = sqrt(L.at<double>(i,j));
23             }
24             else{
25                 L.at<double>(i,j) /= L.at<double>(j,j);
26             }
27         }
28     }
29
30     L = L.t();
31
32     return true;
33 }
```

La línea 16 es la que evita los errores de redondeo descritos anteriormente. Cuando se intenta realizar la raíz cuadrada de un elemento, primero se comprueba si es negativo. En caso de serlo, se debe lanzar un fallo, a no ser que el valor sea muy cercano a cero; en esa situación, simplemente tomamos el valor opuesto y seguimos el algoritmo con naturalidad.

En este punto del proceso tenemos una primera aproximación a la raíz que necesitamos, muy cercana a la real según [2] pero que puede ser optimizada.

### 2.2.2. Optimización de la raíz

El problema de encontrar una raíz de un polinomio está intensamente documentado en la literatura, y uno de los mejores algoritmos para abordar este problema es el de Newton-Raphson.

Para implementarlo, hace falta simplemente la función cuya raíz queremos encontrar, su derivada y una primera aproximación.

La función cuya raíz queremos encontrar es la derivada con respecto a  $\lambda$  de la expresión 1.

Esta función, tomando como parámetros  $A$ ,  $B$ ,  $A'$  y  $B'$  es la siguiente:

$$f(\lambda) = \frac{2A'_{0,0}\lambda + A'_{1,0} + A'_{0,1}}{\lambda(B'_{0,0}\lambda + B'_{0,1}) + B'_{1,0}\lambda + B'_{1,1}} - \frac{(2B'_{0,0}\lambda + B'_{1,0} + B'_{0,1})(\lambda(A'_{0,0}\lambda + A'_{0,1}) + A'_{1,0}\lambda + A'_{1,1})}{(\lambda(B'_{0,0}\lambda + B'_{0,1}) + B'_{1,0}\lambda + B'_{1,1})^2} + \frac{2A_{0,0}\lambda + A_{1,0} + A_{0,1}}{\lambda(B_{0,0}\lambda + B_{0,1}) + B_{1,0}\lambda + B_{1,1}} - \frac{(2B_{0,0}\lambda + B_{1,0} + B_{0,1})(\lambda(A_{0,0}\lambda + A_{0,1}) + A_{1,0}\lambda + A_{1,1})}{(\lambda(B_{0,0}\lambda + B_{0,1}) + B_{1,0}\lambda + B_{1,1})^2}$$

Basta entonces implementar esta función, su derivada —cuya expresión no representamos aquí por ser demasiado larga— y el método de Newton-Raphson para optimizar la primera aproximación de la raíz que obtuvimos en el paso anterior. Todo este cálculo numérico de notación obtusa pero razonamiento evidente, se encuentra encapsulado en la siguiente función:

```

1  void optimizeRoot(const Mat &A, const Mat &B,
2                  const Mat &Ap, const Mat &Bp,
3                  Vec3d &z) {
4
5      double lambda = z[0];
6
7      z[0] = NewtonRaphson(A,B,Ap,Bp, lambda);
8      z[1] = 1.0;
9      z[2] = 0.0;
10 }

```

La función interesante es NewtonRaphson, cuyo código es el siguiente:

```

1  double NewtonRaphson(const Mat &A, const Mat &B,
2                      const Mat &Ap, const Mat &Bp,
3                      double init_guess){
4      double current = init_guess;
5      double previous;
6
7      double fx = function(A,B,Ap,Bp, current);
8      double dfx = derivative(A,B,Ap,Bp, current);
9
10     int iterations = 0;
11
12     do {
13         previous = current;
14         current = current - fx / dfx;
15
16         fx = function(A,B,Ap,Bp, current);
17         dfx = derivative(A,B,Ap,Bp, current);
18
19         iterations++;
20     } while (abs(fx) > 1e-15 && iterations < 150);
21     // Double-precision values have 15 stable decimal positions
22
23     return current;
24 }

```

donde `function` y `derivative` son las funciones  $f(\lambda)$  y  $\frac{d}{d\lambda}f(\lambda)$ , cuyo código se puede encontrar en el fichero `utils.cpp`.

Del algoritmo de Newton-Raphson cabe mencionar su criterio de parada: se busca un elemento  $\alpha$  tal que  $f(\alpha) \in ]-\varepsilon, \varepsilon[$ , donde  $\varepsilon = 10^{-15}$ . La elección de  $\varepsilon$  se debe a la precisión de los `double`: son 15 las posiciones decimales que el estándar considera estables —de hecho, la precisión es de 15.95 posiciones decimales, así que hay casi un decimal más de precisión—; como esto es lo mínimo que podemos medir con rigor, es un buen valor como criterio de parada. En todo caso, si este valor no se alcanzara —en ocasiones la raíz no cambia de una iteración a otra y aún no se ha entrado en el intervalo  $] -\varepsilon, \varepsilon[$ —, el algoritmo se detiene tras 150 iteraciones.

### 2.2.3. Construcción de la homografía

Una vez se ha optimizado la raíz con el método de Newton-Raphson, basta tomar las siguientes líneas:

$$w = [e]_{\times} z$$

$$w' = Fz$$

y usar la primera y segunda coordenadas de las mismas para construir la matriz, como se hace en el siguiente extracto del código de `main.cpp`:

```

1 // Get w
2 Mat w = e_x * Mat(z);
3 Mat wp = fund_mat * Mat(z);
4
5 w /= w.at<double>(2,0);
6 wp /= wp.at<double>(2,0);
7
8 cout << "w = " << w << endl;
9 cout << "wp = " << wp << endl;
10
11 // Get final H_p and Hp_p matrix for projection
12 Mat H_p = Mat::eye(3, 3, CV_64F);
13 H_p.at<double>(2,0) = w.at<double>(0,0);
14 H_p.at<double>(2,1) = w.at<double>(1,0);
15
16 Mat Hp_p = Mat::eye(3, 3, CV_64F);
17 Hp_p.at<double>(2,0) = wp.at<double>(0,0);
18 Hp_p.at<double>(2,1) = wp.at<double>(1,0);

```

En este momento tenemos sendas homografías  $H_p$  y  $H'_p$ , tan cercanas a una homografía afín como es posible, que transforman los epipolos  $e$  y  $e'$  a puntos del infinito.

### 2.3. Transformación de semejanza

Las homografías  $H_r$  y  $H'_r$  simplemente rotan el epipolo de su posición tras aplicar la homografía proyectiva a una posición alineada con la dirección  $[1, 0, 0]$ . Además, aplican un desplazamiento vertical a ambas imágenes para que las líneas epipolares correspondientes tengan la misma coordenada vertical.

Para el cálculo de las homografías  $H_r$  y  $H'_r$  sólo tenemos una incógnita: el desplazamiento vertical que hay que aplicarle a ambas imágenes para mantenerlas en un sistema de coordenadas correcto.

Esto es así porque la homografía  $H_r$  —y su homóloga  $H'_r$ —, una vez conocidas las líneas  $w$

y  $w'$  están definidas como sigue:

$$H_r = \begin{pmatrix} F_{32} - w_b F_{33} & w_a F_{33} - F_{31} & 0 \\ F_{31} - w_a F_{33} & F_{32} - w_b F_{33} & F_{33} + v'_c \\ 0 & 0 & 1 \end{pmatrix} \quad H'_r = \begin{pmatrix} w'_b F_{33} - F_{23} & F_{13} - w'_a F_{33} & 0 \\ w'_a F_{33} - F_{13} & w'_b F_{33} - F_{23} & v'_c \\ 0 & 0 & 1 \end{pmatrix}$$

De ahí desconocemos únicamente  $v'_c$ , y este valor se toma de manera que la menor coordenada vertical de cualquier punto de las imágenes tras aplicar  $H_p$  (resp.  $H'_p$ ) sea cero.

La idea por tanto es la siguiente: aplicamos  $H_p$  a las esquinas de la primera imagen,  $H'_p$  a las esquinas de la segunda y calculamos la menor coordenada vertical de los dos conjuntos de esquinas proyectadas. El opuesto de este valor será la traslación que haya que hacer para que la mínima coordenada vertical sea cero.

La siguiente función encapsula este algoritmo:

```
1  double getTranslationTerm(const Mat &img_1, const Mat &img_2, const Mat &H_p, const Mat &Hp_p){
2      double min_1 = getMinYCoord(img_1, H_p);
3      double min_2 = getMinYCoord(img_2, Hp_p);
4
5      double offset = min_1 < min_2 ? min_1 : min_2;
6
7      return -offset;
8  }
```

donde la función realmente interesante, `getMinYCoord`, está implementada como sigue:

```
1  double getMinYCoord(const Mat &img, const Mat &homography){
2      vector<Point2d> corners(4), corners_trans(4);
3
4      corners[0] = Point2d(0,0);
5      corners[1] = Point2d(img.cols,0);
6      corners[2] = Point2d(img.cols,img.rows);
7      corners[3] = Point2d(0,img.rows);
8
9      perspectiveTransform(corners, corners_trans, homography);
10
11     double min_y;
12     min_y = +INF;
13
14     for (int j = 0; j < 4; j++) {
15         min_y = min(corners_trans[j].y, min_y);
16     }
17
18     return min_y;
19 }
```

Esta función simplemente define las cuatro esquinas como `Point2d`, las guarda en un vector y se lo pasa a la función de *OpenCV* `perspectiveTransform`, que aplica la homografía pasada como último argumento a todos los puntos del vector suministrado. Por último, se devuelve la mínima coordenada vertical.

Con la traslación  $v'_c$  calculada, basta entonces construir las matrices  $H_r$  y  $H'_r$  tal y como se calcularon:

```
1  // Get the translation term
```



```

2  double vp_c = getTranslationTerm(img_1, img_2, H_p, Hp_p);
3
4  cout << "vp_c = " << vp_c << endl;
5
6  // Get the H_r and Hp_r matrix directly
7  Mat H_r = Mat::zeros(3, 3, CV_64F);
8
9  H_r.at<double>(0,0) = fund_mat.at<double>(2,1) - w.at<double>(1,0) * ←
    fund_mat.at<double>(2,2);
10 H_r.at<double>(1,0) = fund_mat.at<double>(2,0) - w.at<double>(0,0) * ←
    fund_mat.at<double>(2,2);
11
12 H_r.at<double>(0,1) = w.at<double>(0,0) * fund_mat.at<double>(2,2) - ←
    fund_mat.at<double>(2,0);
13 H_r.at<double>(1,1) = H_r.at<double>(0,0);
14
15 H_r.at<double>(1,2) = fund_mat.at<double>(2,2) + vp_c;
16 H_r.at<double>(2,2) = 1.0;
17
18 Mat Hp_r = Mat::zeros(3, 3, CV_64F);
19
20 Hp_r.at<double>(0,0) = wp.at<double>(1,0) * fund_mat.at<double>(2,2) - ←
    fund_mat.at<double>(1,2);
21 Hp_r.at<double>(1,0) = wp.at<double>(0,0) * fund_mat.at<double>(2,2) - ←
    fund_mat.at<double>(0,2);
22
23 Hp_r.at<double>(0,1) = fund_mat.at<double>(0,2) - wp.at<double>(0,0) * ←
    fund_mat.at<double>(2,2);
24 Hp_r.at<double>(1,1) = Hp_r.at<double>(0,0);
25
26 Hp_r.at<double>(1,2) = vp_c;
27 Hp_r.at<double>(2,2) = 1.0;

```

## 2.4. Transformación de cizalla

Las composiciones  $H_r H_p$  y  $H'_r H'_p$  culminan el algoritmo de rectificación, ya que alinean los epipolos en la dirección  $[1, 0, 0]$ ; es decir, dejan las líneas epipolares horizontales, paralelas entre sí y a la misma altura.

Sin embargo, la distorsión introducida por las transformaciones proyectivas puede ser reducida, y esto se consigue con una transformación de cizalla que pretende conservar la perpendicularidad y la proporcionalidad de las líneas que unen los puntos medios de los bordes de la imagen. Además, se calcula una escala y una traslación en ambos ejes —con la restricción de que la traslación vertical debe ser igual en ambas imágenes para que las líneas epipolares sigan alineadas— para mantener las imágenes aproximadamente con las mismas áreas y en un sistema de coordenadas razonable.

### 2.4.1. Reducción de la distorsión

Con las restricciones de perpendicularidad y proporcionalidad se puede calcular la transformación de cizalla  $S$ .

De nuevo, toda la discusión matemática se puede encontrar en [2], donde el problema se reduce a encontrar una solución común a dos cuárticas en las incógnitas  $a$  y  $b$ . En [2] se da la solución salvo signo, indicando que la solución en la que  $a$  es positiva es la preferida. Así, basta implementar las expresiones de  $a$  y  $b$  dadas en el artículo y, si  $a$  es negativo, cambiar ambos signos.

Este cálculo se encuentra en la función `getS`, que sigue la misma notación que [2]:

```

1  Mat getS(const Mat &img, const Mat &homography){
2      int w = img.cols;
3      int h = img.rows;
4
5      Point2d a((w-1)/2, 0);
6      Point2d b(w-1, (h-1)/2);
7      Point2d c((w-1)/2, h-1);
8      Point2d d(0, (h-1)/2);
9
10     vector<Point2d> midpoints, midpoints_hat;
11     midpoints.push_back(a);
12     midpoints.push_back(b);
13     midpoints.push_back(c);
14     midpoints.push_back(d);
15
16     perspectiveTransform(midpoints, midpoints_hat, homography);
17
18     Point2d x = midpoints_hat[1] - midpoints_hat[3];
19     Point2d y = midpoints_hat[2] - midpoints_hat[0];
20
21     double coeff_a = (h*h*x.y*x.y + w*w*y.y*y.y) / (h*w * (x.y*y.x - x.x*y.↵
22         y));
23     double coeff_b = (h*h*x.x*x.y + w*w*y.x*y.y) / (h*w * (x.x*y.y - x.y*y.↵
24         x));
25
26     Mat S = Mat::eye(3, 3, CV_64F);
27     S.at<double>(0,0) = coeff_a;
28     S.at<double>(0,1) = coeff_b;
29
30     Vec3d x_hom(x.x, x.y, 0.0);
31     Vec3d y_hom(y.x, y.y, 0.0);
32
33     if( coeff_a < 0 ){
34         coeff_a *= -1;
35         coeff_b *= -1;
36
37         S.at<double>(0,0) = coeff_a;
38         S.at<double>(0,1) = coeff_b;
39     }
40
41     Mat EQ18 = (S * Mat(x_hom)).t() * (S * Mat(y_hom));
42     cout << R0J0 << "EQ18 " << EQ18 << RESET << endl;
43
44     Mat EQ19 = ((S * Mat(x_hom)).t() * (S * Mat(x_hom))) / ((S * Mat(y_hom)↵
45         ).t() * (S * Mat(y_hom))) - (1.*w*w)/(1.*h*h);
46     cout << R0J0 << "EQ19 " << EQ19 << RESET << endl;
47
48     cout << "S = " << S << endl;
49
50     return S;
51 }

```

En la línea 5 se definen los puntos medios de la imagen, se proyectan con la homografía calculada hasta ahora en 16 y, después, se definen las soluciones  $a$  y  $b$  tal y como se declara en el artículo.

Para obtener la solución en la que  $a$  es positivo, basta ver si no se cumple esta restricción y, en ese caso, tomar  $-a$  y  $-b$ .

Hemos añadido por último, a partir de la línea 39, que los valores calculados sean realmente

soluciones. Así, implementamos las expresiones 18 y 19 de [2] e imprimimos los resultados, que deben ser cero. En efecto, se obtienen valores del orden de  $10^{-6}$ , como veremos en la valoración de los resultados.

#### 2.4.2. Escalado y traslación

Las composiciones  $SH_rH_p$  y  $S'H'_rH'_p$  culminan el algoritmo de rectificación minimizando la distorsión. Sin embargo, hay que aplicar un escalado y una traslación de manera que mantengamos áreas y sistemas de coordenadas parecidos a los originales.

Mientras se aplique el mismo factor de escala a ambas imágenes y la traslación vertical sea igual, hay libertad para escoger estos valores.

Tal y como se indica en [2], uno de los criterios para el cálculo del factor de escala puede ser el de mantener la suma de las áreas de ambas imágenes. Así se ha implementado en este proyecto, aunque se ha impuesto una nueva restricción<sup>1</sup>: la transformación no puede invertir la imagen.

Para mantener el cálculo de las áreas se sigue el siguiente proceso:

- Se calcula  $A$ , suma de las áreas de las imágenes originales.
- Se aplica homografía  $SH_rH_p$  (resp.  $S'H'_rH'_p$ ) a las esquinas de las imágenes originales y se obtienen sus proyectadas
- Se calcula  $A'$ , área del contorno definido por las cuatro esquinas proyectadas.
- Se toma como factor de escala  $s = \sqrt{\frac{A}{A'}}$ .

Definimos entonces la matriz  $W_2$  como un escalado con ese factor de escala:

$$W_2 = \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Después de comprobar si la homografía  $W_2SH_rH_p$  invierte la imagen —y, en caso afirmativo, redefinir  $W_2$  con  $-s$  como factor de escala—, se calcula una traslación horizontal  $x$  y  $x'$  independiente en cada imagen —de manera que la menor coordenada horizontal sea cero— y una traslación vertical común  $y$  —de forma que la menor coordenada vertical de ambas imágenes sea cero—. Con estas traslaciones se define entonces  $W_1$  y  $W'_1$  como sigue:

$$W_1 = \begin{pmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{pmatrix} \quad W'_1 = \begin{pmatrix} 1 & 0 & x' \\ 0 & 1 & y \\ 0 & 0 & 1 \end{pmatrix}$$

Tomaremos entonces las transformaciones  $H_s$  y  $H'_s$  como

$$H_s = W_1W_2S \quad H'_s = W'_1W_2S'$$

Todo este proceso —incluyendo el cálculo de  $S$ — se encuentra implementado en la siguiente función:

---

<sup>1</sup>Esta decisión de implementación se tomó tras la fase de prueba, donde comprobamos que había casos en los que la rectificación funcionaba bien pero giraba 180 las imágenes.

```

1  void getShearingTransforms(const Mat &img_1, const Mat &img_2,
2                          const Mat &H_1, const Mat &H_2,
3                          Mat &H_s, Mat &Hp_s){
4
5      Mat S = getS(img_1, H_1);
6      Mat Sp = getS(img_2, H_2);
7
8      double A = img_1.cols*img_1.rows + img_2.cols*img_2.rows;
9      double Ap = 0;
10
11     vector<Point2f> corners(4), corners_trans(4);
12
13     corners[0] = Point2f(0,0);
14     corners[1] = Point2f(img_1.cols,0);
15     corners[2] = Point2f(img_1.cols,img_1.rows);
16     corners[3] = Point2f(0,img_1.rows);
17
18     perspectiveTransform(corners, corners_trans, S*H_1);
19     Ap += contourArea(corners_trans);
20
21     float min_x_1, min_y_1;
22     min_x_1 = min_y_1 = +INF;
23     for (int j = 0; j < 4; j++) {
24         min_x_1 = min(corners_trans[j].x, min_x_1);
25         min_y_1 = min(corners_trans[j].y, min_y_1);
26     }
27
28     corners[0] = Point2f(0,0);
29     corners[1] = Point2f(img_2.cols,0);
30     corners[2] = Point2f(img_2.cols,img_2.rows);
31     corners[3] = Point2f(0,img_2.rows);
32
33     perspectiveTransform(corners, corners_trans, Sp*H_2);
34     Ap += contourArea(corners_trans);
35
36     float min_x_2, min_y_2;
37     min_x_2 = min_y_2 = +INF;
38     for (int j = 0; j < 4; j++) {
39         min_x_2 = min(corners_trans[j].x, min_x_2);
40         min_y_2 = min(corners_trans[j].y, min_y_2);
41     }
42
43     double scale = sqrt(A/Ap);
44     cout << "A = " << A << " \n/ Ap = " << Ap << endl;
45
46     double min_y = min_y_1 < min_y_2 ? min_y_1 : min_y_2;
47
48     // We define W2 as the scale transformation and W1 as the translation
49     // transformation. Then, W = W1*W2.
50
51     Mat W;
52     Mat Wp;
53
54     Mat W_1 = Mat::eye(3, 3, CV_64F);
55     Mat Wp_1 = Mat::eye(3, 3, CV_64F);
56
57     Mat W_2 = Mat::eye(3, 3, CV_64F);
58     Mat Wp_2 = Mat::eye(3, 3, CV_64F);
59
60     W_2.at<double>(0,0) = W_2.at<double>(1,1) = scale;
61     Wp_2.at<double>(0,0) = Wp_2.at<double>(1,1) = scale;
62

```

```

63     if (isImageInverted(img_1, W_2*H_1)){
64         W_2.at<double>(0,0) = W_2.at<double>(1,1) = -scale;
65         Wp_2.at<double>(0,0) = Wp_2.at<double>(1,1) = -scale;
66     }
67
68     corners[0] = Point2d(0,0);
69     corners[1] = Point2d(img_1.cols,0);
70     corners[2] = Point2d(img_1.cols,img_1.rows);
71     corners[3] = Point2d(0,img_1.rows);
72
73     perspectiveTransform(corners, corners_trans, W_2*S*H_1);
74
75     min_x_1 = min_y_1 = +INF;
76     for (int j = 0; j < 4; j++) {
77         min_x_1 = min(corners_trans[j].x, min_x_1);
78         min_y_1 = min(corners_trans[j].y, min_y_1);
79     }
80
81     corners[0] = Point2d(0,0);
82     corners[1] = Point2d(img_2.cols,0);
83     corners[2] = Point2d(img_2.cols,img_2.rows);
84     corners[3] = Point2d(0,img_2.rows);
85
86     perspectiveTransform(corners, corners_trans, Wp_2*Sp*H_2);
87
88     min_x_2 = min_y_2 = +INF;
89     for (int j = 0; j < 4; j++) {
90         min_x_2 = min(corners_trans[j].x, min_x_2);
91         min_y_2 = min(corners_trans[j].y, min_y_2);
92     }
93
94     min_y = min_y_1 < min_y_2 ? min_y_1 : min_y_2;
95
96     W_1.at<double>(0,2) = -min_x_1;
97     Wp_1.at<double>(0,2) = -min_x_2;
98
99     W_1.at<double>(1,2) = Wp_1.at<double>(1,2) = -min_y;
100
101     W = W_1*W_2;
102     Wp = Wp_1*Wp_2;
103
104     H_s = W*S;
105     Hp_s = Wp*Sp;
106
107     cout << "H_s = " << H_s << "\nHp_s = " << Hp_s << endl;
108 }

```

Todo este código simplemente implementa lo descrito anteriormente. Lo único que ha quedado sin explicar es la función `isImageInverted`, que se usa en la línea 63 para tomar  $-s$  en caso de que la transformación invierta la imagen. El código de esta función comprueba que las esquinas no se den la vuelta, midiendo si la diferencia entre la coordenada vertical de la esquina inferior izquierda y la coordenada vertical de la esquina superior izquierda es negativa; en ese caso, es claro que la imagen se ha invertido:

```

1  bool isImageInverted(const Mat &img, const Mat &homography){
2      vector<Point2d> corners(2), corners_trans(2);
3
4      corners[0] = Point2d(0,0);
5      corners[1] = Point2d(0,img.rows);
6
7      perspectiveTransform(corners, corners_trans, homography);

```

```

8
9     return corners_trans[1].y - corners_trans[0].y < 0.0;
10 }

```

### 2.4.3. Homografía final

Una vez se han calculado  $H_p$ ,  $H_r$  y  $H_s$  —y sus homólogas con las primas— basta componerlas para obtener la homografía final que rectifica las imágenes con la menor distorsión posible:

$$H = H_s H_r H_p \quad H' = H'_s H'_r H'_p$$

Por último, se ha añadido código para mostrar las imágenes completas: tras calcular la posición de las esquinas de las imágenes proyectadas por  $H$  y  $H_p$ , se toman sendos objetos `Mat` donde se acomoden bien las nuevas imágenes y se proyectan allí las imágenes rectificadas. El extracto de código en `main.cpp` que implementa este proceso es el siguiente:

```

1     Mat H = H_s * H_r * H_p;
2     Mat Hp = Hp_s * Hp_r * Hp_p;
3
4     // Get homography image of the corner coordinates from all the images
5     vector<Point2d> corners_all(4), corners_all_t(4);
6     double min_x, min_y, max_x, max_y;
7     min_x = min_y = +INF;
8     max_x = max_y = -INF;
9
10    corners_all[0] = Point2d(0,0);
11    corners_all[1] = Point2d(img_1.cols,0);
12    corners_all[2] = Point2d(img_1.cols,img_1.rows);
13    corners_all[3] = Point2d(0,img_1.rows);
14
15    perspectiveTransform(corners_all, corners_all_t, H);
16
17    for (int j = 0; j < 4; j++) {
18        min_x = min(corners_all_t[j].x, min_x);
19        max_x = max(corners_all_t[j].x, max_x);
20
21        min_y = min(corners_all_t[j].y, min_y);
22        max_y = max(corners_all_t[j].y, max_y);
23    }
24
25    int img_1_cols = max_x - min_x;
26    int img_1_rows = max_y - min_y;
27
28    // Get homography image of the corner coordinates from all the images
29    min_x = min_y = +INF;
30    max_x = max_y = -INF;
31
32    corners_all[0] = Point2d(0,0);
33    corners_all[1] = Point2d(img_2.cols,0);
34    corners_all[2] = Point2d(img_2.cols,img_2.rows);
35    corners_all[3] = Point2d(0,img_2.rows);
36
37    perspectiveTransform(corners_all, corners_all_t, Hp);
38
39    for (int j = 0; j < 4; j++) {
40        min_x = min(corners_all_t[j].x, min_x);
41        max_x = max(corners_all_t[j].x, max_x);
42

```

```

43         min_y = min(corners_all_t[j].y, min_y);
44         max_y = max(corners_all_t[j].y, max_y);
45     }
46
47     int img_2_cols = max_x - min_x;
48     int img_2_rows = max_y - min_y;
49
50     // Apply homographies
51     Mat img_1_dst(img_1_rows, img_1_cols, CV_64F);
52     Mat img_2_dst(img_2_rows, img_2_cols, CV_64F);
53
54     warpPerspective( img_1, img_1_dst, H, img_1_dst.size() );
55     warpPerspective( img_2, img_2_dst, Hp, img_2_dst.size() );

```

### 3. Experimentos realizados

### 4. Valoración de resultados

Todo se ve de lujo

### 5. Conclusiones

Este algoritmo es pro. Mejor que el de OpenCV.

## Referencias

- [1] *The Cholesky Banachiewicz and Cholesky Crout algorithms*. [https://en.wikipedia.org/wiki/Cholesky\\_decomposition#The\\_Cholesky.E2.80.93Banachiewicz\\_and\\_Cholesky.E2.80.93Crout\\_algorithms](https://en.wikipedia.org/wiki/Cholesky_decomposition#The_Cholesky.E2.80.93Banachiewicz_and_Cholesky.E2.80.93Crout_algorithms), visitado el 02-02-2016.
- [2] Loop, Charles y Zhengyou Zhang: *Computing Rectifying Homographies for Stereo Vision*. Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, 1:125–131, Abril 1999.