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

Antonio Álvarez Caballero
Alejandro García Montoro
analca3@correo.ugr.es
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. Enfoque de la implementación y detalles de eficiencia

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 ω_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 ω_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:

$$\frac{z^T A z}{z^T B z} + \frac{z^T A' z}{z^T B' z} \tag{1}$$

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

$$A = [e]_\times^T P P^T [e]_\times \qquad A' = F^T P' P'^T F$$
$$B = [e]_\times^T p_c p_c^T [e]_\times \qquad B' = F^T p'_c p'^T_c F$$

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 λ cuyo mínimo es el valor que buscamos. Este mínimo se alcanza cuando su derivada con respecto a λ 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:

```
Vec3d getInitialGuess(Mat &A, Mat &B, Mat &Ap, Mat &Bp){

    Vec3d z_1 = maximize(A, B);
    Vec3d z_2 = maximize(Ap, Bp);

    return (normalize(z_1) + normalize(z_2))/2;
}
```

donde el código realmente interesante está en `maximize`:

```
Vec3d maximize(Mat &A, Mat &B){
    Mat D; // Output of cholesky decomposition: upper triangular matrix.
    if( choleskyCustomDecomp(A, D) ){

        Mat D_inv = D.inv();

        Mat DBD = D_inv.t() * B * D_inv;

        // Solve the equations system using eigenvalue decomposition

        Mat eigenvalues, eigenvectors;
        eigen(DBD, eigenvalues, eigenvectors);

        // Take largest eigenvector
        Mat y = eigenvectors.row(0);

        Mat sol = D_inv*y.t();

        Vec3d res(sol.at<double>(0,0), sol.at<double>(1,0), sol.at<double>(2,0));

        return res;
    }

    // At this point, there is an error!
    Mat eigenvalues;
    eigen(A, eigenvalues);

    cout << "\n\n\n---------- WARNING ----------" << endl;

    return Vec3d(0, 0, 0);
}
```

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 λ de la expresión 1.

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

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

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 α tal que $f(\alpha) \in]-\varepsilon, \varepsilon[$, donde $\varepsilon = 10^{-15}$. La elección de ε 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 homorgafías H_p y H'_p, tan cercanas a una homorgafía afín como es posible, que transforman los epipolos e y e' a puntos del infinito.

2.3. Transformación de semejanza

2.4. Transformación de cizalla

3. Experimentos realizados

Lola es muy guapa y adorable. V impone.

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, 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.