Visión por Computador (2016-2017)
Grado en Ingeniería Informática
Universidad de Granada

Informe Práctica 3: opción de 11 puntos

Laura Tirado López

3 de enero de 2017

Índice

Índice de figuras

1. Estimación de la matriz de una cámara a partir del conjunto de puntos en correspondencias

1.1. Generación de una cámara aleatoria finita

Para la generación de la cámara, generamos 12 valores aleatorios en el intervalo $[0, 1]$ para construir la matriz 3×4 de nuestra cámara. Verificamos si es una cámara finita comprobamos que su determinante sea distinto de 0. En caso de que no represente una cámara finita, volvemos a generar los valores hasta generar una.

```
Mat generarCamaraAleatoria(){

        Mat camara = Mat(3,4,CV_32F);
        bool correcta = false;

        while(!correcta){

                //Generamos los valores de la matriz
                for(int i=0; i < 3; ++i)
                        for(int j=0; j < 4; ++j)
                                camara.at<float>(i,j) =
                                   static_cast <float> (rand()+1)
                                   / static_cast <float> (RAND_MAX
                                   );

                //Tomamos la submatriz para calcular el
                   determinante
                Mat submatriz = Mat(camara, Range::all(), Range
                   (0,3));

                if(determinant(submatriz) != 0)
                        correcta = true;
        }

        return camara;
}
```

1.2. Generación de puntos, proyección y obtención de las coordenadas píxel

El conjunto de puntos que vamos a generar es el conjunto de puntos cuyas coordenadas son $(0, x_1, x_2), (x_2, x_1, 0)$ tal que $x_1, x_2 \in$ [0,1, 1]. En mi caso, he generado los puntos tomando los valores 0,1, 0,2, 0,3, ... , 1,0 para x_1 y x_2 con todas las posibles combinaciones.

```
vector<Mat> generarPuntosMundo(){

        vector<Mat> puntos_mundo;
        Mat m;

        //Creamos los puntos del mundo como matrices columna de 4
           componentes (siendo la cuarta siempre 1)
        for(int i=1; i <= 10; ++i){
                for(int j=1; j <= 10; ++j){

                        m = Mat(4,1,CV_32F, 1.0);
                        m.at<float>(0,0) = 0.0;
                        m.at<float>(1,0) = i*0.1;
                        m.at<float>(2,0) = j*0.1;
                        puntos_mundo.push_back(m);

                        m = Mat(4,1,CV_32F, 1.0);
                        m.at<float>(0,0) = j*0.1;
                        m.at<float>(1,0) = i*0.1;
                        m.at<float>(2,0) = 0.0;
                        puntos_mundo.push_back(m);

                }
        }

        return puntos_mundo;
}
```

Una vez hemos generado el conjunto de puntos, para proyectar los puntos simplemente multiplicamos las coordenadas del punto por la matriz de la cámara finita que generamos anteriormente. Para el cálculo de las coordenadas píxel homogeneizamos la componente z de los puntos.

```
vector<Mat> obtenerPuntosProyectados(vector<Mat> puntos, Mat
   camara){

        vector<Mat> puntos_proyectados;

        //Aplicamos la matriz camara al punto
        for(int i=0; i < puntos.size(); ++i)
                puntos_proyectados.push_back(camara * puntos.at(i)
                   );

        //Homogeneizamos la tercera componente de cada punto
        for(int i=0; i < puntos_proyectados.size(); ++i)
                puntos_proyectados.at(i) = puntos_proyectados.at(i
                   ) / puntos_proyectados.at(i).at<float>(2,0);

```

```
13          return puntos_proyectados;
14  }
```

1.3. Estimación de la cámara a partir de los puntos y sus proyecciones

Para la estimación de la cámara mediante el algoritmo DLT primero calculamos la matriz de coeficientes y a continuación usamos la función *compute()*. La matriz de la cámara estimada la construimos tomando los valores de la última columna de v, que coinciden con los de la última fila de la cuarta matriz que nos devuelve la función *compute()*.

La matriz de coeficientes tiene el siguiente esquema

$$\begin{pmatrix} x & y & z & 0 & 0 & 0 & x_i x & x_i y & x_i z \\ 0 & 0 & 0 & x & y & z & y_i x & y_i y & y_i z \end{pmatrix}$$

siendo (x, y, z) las coordenadas del punto y (x_i, y_i) sus coordenadas píxel proyectadas.

```
1   Mat estimarCamara(vector<Mat> puntos, vector<Mat> proyecciones){
2
3           Mat A, w, u, vt;
4           int f = 2 * puntos.size();
5           int c = 12;
6
7           //Estimamos los coeficientes de la matriz
8           A = Mat(f,c,CV_32F, 0.0);
9           Mat punto_actual, punto_proyectado_actual;
10
11          //Rellenamos la matriz segun el esquema que ha de tener al
                resolver el sistema de ecuaciones
12          for(int i=0; i < f; i=i+2){
13
14                  punto_actual = puntos.at(i/2);
15                  punto_proyectado_actual = proyecciones.at(i/2);
16
17                  A.at<float>(i,0) = punto_actual.at<float>(0,0);
18                  A.at<float>(i,1) = punto_actual.at<float>(1,0);
19                  A.at<float>(i,2) = punto_actual.at<float>(2,0);
20                  A.at<float>(i,3) = 1.0;
21
22                  A.at<float>(i,8) = -punto_proyectado_actual.at<
                        float>(0,0) * punto_actual.at<float>(0,0);
23                  A.at<float>(i,9) = -punto_proyectado_actual.at<
                        float>(0,0) * punto_actual.at<float>(1,0);
24                  A.at<float>(i,10) = -punto_proyectado_actual.at<
                        float>(0,0) * punto_actual.at<float>(2,0);
```

```
25                  A.at<float>(i,11) = -punto_proyectado_actual.at<
                        float>(0,0);
26
27                  A.at<float>(i+1,4) = punto_actual.at<float>(0,0);
28                  A.at<float>(i+1,5) = punto_actual.at<float>(1,0);
29                  A.at<float>(i+1,6) = punto_actual.at<float>(2,0);
30                  A.at<float>(i+1,7) = 1.0;
31
32                  A.at<float>(i+1,8) = -punto_proyectado_actual.at<
                        float>(1,0) * punto_actual.at<float>(0,0);
33                  A.at<float>(i+1,9) = -punto_proyectado_actual.at<
                        float>(1,0) * punto_actual.at<float>(1,0);
34                  A.at<float>(i+1,10) = -punto_proyectado_actual.at<
                        float>(1,0) * punto_actual.at<float>(2,0);
35                  A.at<float>(i+1,11) = -punto_proyectado_actual.at<
                        float>(1,0);
36          }
37
38          //Obtenemos la descomposicion SVD de la matriz de
                coeficientes
39          SVD::compute(A,w,u,vt);
40
41          Mat camara_estimada = Mat(3,4,CV_32F);
42
43          //Construimos la matriz de transformacion con la ultima
                columna de v (la ultima fila de vt)
44          for(int i=0; i < 3; ++i)
45                  for(int j=0; j < 4; ++j)
46                          camara_estimada.at<float>(i,j) = vt.at<
                                float>(11,i*4+j);
47
48          return camara_estimada;
49  }
```

1.4. Resultados

Para estimar el error calculamos la distancia entre las dos matrices con la norma de Frobenius.

```
1   double calcularErrorEstimacion(Mat A, Mat B){
2
3           Mat diferencia = (A/A.at<float>(0,0)) - (B/B.at<float
                >(0,0));
4
5           double sum = 0.0;
6
7           for(int i=0; i < diferencia.rows; ++i)
```

```
8                   for(int j=0; j < diferencia.cols; ++j)
9                           sum += diferencia.at<float>(i,j) *
                               diferencia.at<float>(i,j);
10
11          return sqrt(sum);
12  }
```

Los resultados se muestran en la siguiente imagen:

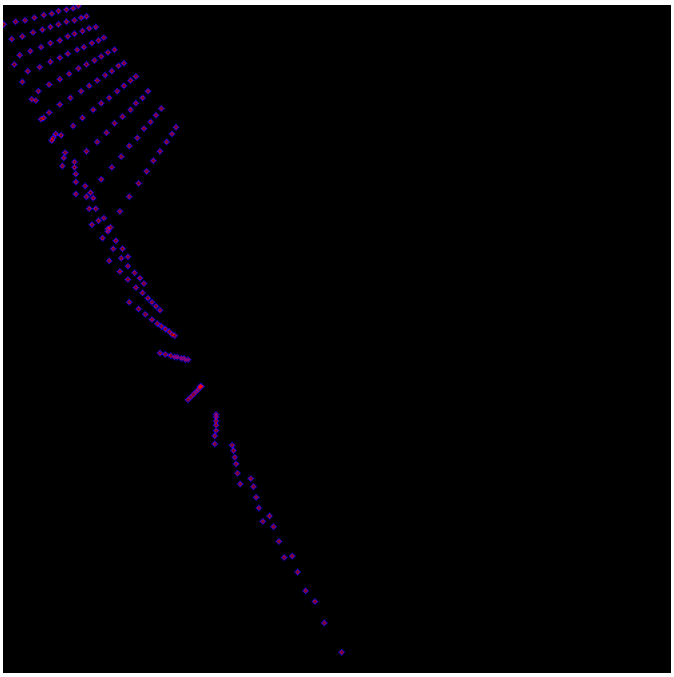


Figura 1.1: Puntos proyectados con cámara simulada (rojo) y con cámara estimada (azul)

Los puntos de la cámara simulada los he dibujado con círculos de un radio más pequeño que los de la cámara estimada para que se puedan ver ambos, dado que todos coinciden de forma bastante exacta.

En este caso la matriz de la cámara simulada es

$$\begin{pmatrix} 0{,}840188 & 0{,}394383 & 0{,}783099 & 0{,}79844 \\ 0{,}911647 & 0{,}197551 & 0{,}335223 & 0{,}76823 \\ 0{,}277775 & 0{,}55397 & 0{,}477397 & 0{,}628871 \end{pmatrix}$$

y la matriz de la cámara estimada

$$\begin{pmatrix} -0{,}387867 & -0{,}182064 & -0{,}361513 & -0{,}368594 \\ -0{,}420856 & -0{,}0911981 & -0{,}154754 & -0{,}354648 \\ -0{,}128233 & -0{,}255736 & -0{,}220387 & -0{,}290314 \end{pmatrix}$$

El error cometido en la aproximación ha sido $1{,}86863 \times 10^{-6}$, lo cual nos da una buena aproximación ya que como se comprueba en la imagen las proyecciones de los puntos simulados y los estimados son prácticamente coincidentes en su totalidad.

2. Calibración de la cámara usando homografías

2.1. Determinación de las coordenadas de las esquinas

Primero he implementado una función para poder leer las 25 imágenes. Una vez leídas, para determinar si son válidas intentamos obtener las coordenadas de las esquinas usando la función *findChessboardCorners()* para cada una de las imágenes. Si encontramos las esquinas, guardamos dicha imagen como una de las seleccionadas y las coordenadas de sus esquinas. Por último, refinamos todas las coordenadas con la función *cornerSubpix()* y las dibujamos utilizando la función *drawChessboardCorners()*.

```
1  void obtenerEsquinas(vector<Mat> imagenes, vector<vector<Point2f>
       > &esquinas, vector<Mat> &imagenes_calibracion){
2
3          vector<Point2f> esquinas_img_actual;
4
5          for(int i=0; i < 25; ++i){
6                  if(findChessboardCorners(imagenes.at(i), Size
                       (13,12), esquinas_img_actual)){
7                          imagenes_calibracion.push_back(imagenes.at
                               (i));
8                          esquinas.push_back(esquinas_img_actual);
9                  }
10
11                 esquinas_img_actual.clear();
12         }
13
14         cout << "Se han localizado todas las esquinas en " <<
               imagenes_calibracion.size() << " imagenes." << endl;
15
16         //Refinamos las coordenadas
17         for(int i=0; i < imagenes_calibracion.size(); ++i)
18                 cornerSubPix(imagenes_calibracion.at(i), esquinas.
                       at(i), Size(5,5), Size(-1,-1), TermCriteria(
                       CV_TERMCRIT_EPS+CV_TERMCRIT_ITER,30,0.1));
19
```

```
20          //Pintamos las esquinas encontradas
21          for(int i=0; i < imagenes_calibracion.size(); ++i){
22                  cvtColor(imagenes_calibracion.at(i),
                       imagenes_calibracion.at(i), CV_GRAY2BGR);
23                  drawChessboardCorners(imagenes_calibracion.at(i),
                       Size(13,12), Mat(esquinas.at(i)), true);
24          }
25          imshow("Tablero 0", imagenes_calibracion.at(0));
26          imwrite("~/Escritorio/P3/t0.png",imagenes_calibracion.at
               (0));
27          waitKey(0);
28          destroyAllWindows();
29
30          imshow("Tablero 1", imagenes_calibracion.at(1));
31          imwrite("~/Escritorio/P3/t1.png",imagenes_calibracion.at
               (0));
32          waitKey(0);
33          destroyAllWindows();
34
35          imshow("Tablero 2", imagenes_calibracion.at(2));
36          imwrite("~/Escritorio/P3/t2.png",imagenes_calibracion.at
               (0));
37          waitKey(0);
38          destroyAllWindows();
39
40          imshow("Tablero 3", imagenes_calibracion.at(3));
41          imwrite("~/Escritorio/P3/t4.png",imagenes_calibracion.at
               (0));
42          waitKey(0);
43          destroyAllWindows();
44  }
```

2.2. Calibración de la cámara

Para la calibración de la cámara vamos a utilizar la función *calibrateCamera()*.

Primero tenemos que calcular los puntos del objeto, es decir, los puntos teóricos dónde se encuentra el patrón que estamos buscando, en este caso nuestras esquinas del tablero. También creamos dos matrices: la matriz K inicializada a la matriz identidad y la matriz de coeficientes de distorsión inicializada a 0.

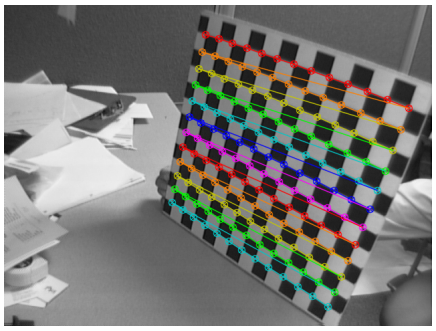
Para las distintas situaciones de calibración según la distorsión le pasamos a la función el criterio correspondiente.

```
1   void errorDistorsionRadialTangencial(Size tam_tablero, vector<Mat>
        imagenes_calibracion, vector<vector<Point2f> >
       esquinas_calibracion){
```

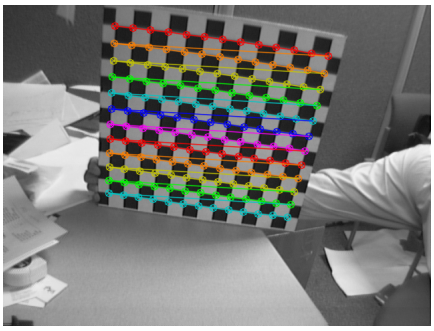
```
2
3           double error;
4
5           vector<Point3f> esquinas_teoricas;
6
7           //Obtenemos los puntos teoricos donde ha de estar el
               patron que estamos buscando
8           for( int i=0; i < tam_tablero.height; ++i)
9           for( int j = 0; j < tam_tablero.width; j++)
10                  esquinas_teoricas.push_back(Point3f(float(j),
                       float(i), 0));
11
12          //Copiamos los puntos teoricos tantas veces como conjuntos
                de puntos reales tengamos
13          vector<vector<Point3f> > puntos_objeto;
14          puntos_objeto.resize(imagenes_calibracion.size(),
               esquinas_teoricas);
15
16          Mat K = Mat::eye(3,3,CV_64F);
17          Mat coef_distorsion = Mat::zeros(8, 1, CV_64F);
18          vector<Mat> rvecs, tvecs;
19
20          error = calibrateCamera (puntos_objeto,
               esquinas_calibracion, imagenes_calibracion.at(0).size()
               , K, coef_distorsion, rvecs, tvecs,
               CV_CALIB_RATIONAL_MODEL);
21
22          cout << "Los parametros de distorsion y la K calculados
               suponiendo que tenemos ambos tipos de distorsiones son:
                " << endl;
23          for (int i=0; i < 8; ++i)
24          cout << coef_distorsion.at<double>(i,0) << " ";
25
26          cout << endl;
27          cout << "r: " << endl;
28          mostrarMatriz(rvecs.at(0));
29          cout << "t: " << endl;
30          mostrarMatriz(tvecs.at(0));
31          cout << "K: " << endl;
32          mostrarMatriz(K/K.at<double>(0,0));
33
34          cout << "Calculando todos los coeficientes de distorsion
               el error es " << error << "." << endl;
35  }
```

2.3. Resultados

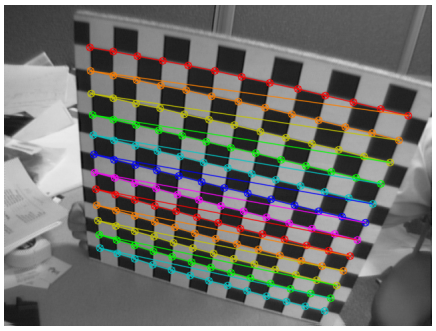
De las 25 imágenes iniciales sólo se han podido detectar todas las esquinas en cuatro de ellas, es decir, sólo cuatro imágenes serán válidas para poder calibrar la cámara:



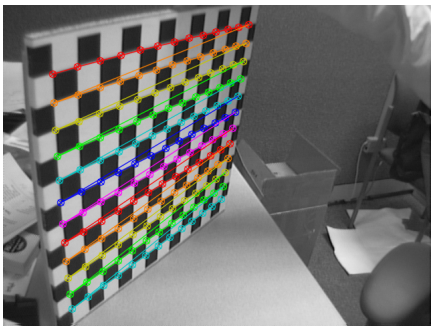
(a) Esquinas del tablero 0



(b) Esquinas del tablero 1



(c) Esquinas del tablero 2



(d) Esquinas del tablero 3

Podemos observar como se han encontrado todas las esquinas correctamente.

Respecto a la calibración hemos obtenido los siguientes resultados.

- Calibración sin distorsión:
  - Coeficientes de calibración: 0 0 0 0 0 0 0 $4{,}00193 \times 10^{-322}$
  - r:
$$\begin{pmatrix} 0{,}354497 \\ 0{,}66675 \\ 0{,}360386 \end{pmatrix}$$
  - t:
$$\begin{pmatrix} -0{,}610978 \\ -8{,}35414 \\ 25{,}8131 \end{pmatrix}$$

- K:
$$\begin{pmatrix} 1 & 0 & 0{,}456671 \\ 0 & 0{,}996352 & 0{,}381506 \\ 0 & 0 & 0{,}00143635 \end{pmatrix}$$
- Error: 1,3259

- Calibración con distorsión radial:
  - Coeficientes de calibración: 38,9284 −167,313 0 0 −66,8058 39,2345 −157,688 −113,078
  - r:
$$\begin{pmatrix} 0{,}328098 \\ 0{,}689583 \\ 0{,}346572 \end{pmatrix}$$
  - t:
$$\begin{pmatrix} -0{,}0760378 \\ -7{,}46708 \\ 24{,}287 \end{pmatrix}$$
  - K:
$$\begin{pmatrix} 1 & 0 & 0{,}461632 \\ 0 & 1{,}00128 & 0{,}367618 \\ 0 & 0 & 0{,}00152173 \end{pmatrix}$$
  - Error: 0,162782

- Calibración con distorsión tangencial:
  - Coeficientes de calibración: 0 0 0,0166309 −0,000849824 0 0 0 $4{,}00193 \times 10^{-322}$
  - r:
$$\begin{pmatrix} 0{,}397029 \\ 0{,}66327 \\ 0{,}374353 \end{pmatrix}$$
  - t:
$$\begin{pmatrix} -0{,}563327 \\ -9{,}61283 \\ 25{,}5256 \end{pmatrix}$$
  - K:
$$\begin{pmatrix} 1 & 0 & 0{,}451938 \\ 0 & 0{,}996606 & 0{,}427516 \\ 0 & 0 & 0{,}00142785 \end{pmatrix}$$
  - Error: 1,30197

- Calibración con distorsión radial y tangencial:
  - Coeficientes de calibración: 31,9236 −141,75 −0,000448144 −0,000133183 −54,1531 32,2301 −133,979 −92,8565
  - r:
$$\begin{pmatrix} 0{,}327967 \\ 0{,}690309 \\ 0{,}346501 \end{pmatrix}$$
  - t:
$$\begin{pmatrix} -0{,}0787165 \\ -7{,}46073 \\ 24{,}2972 \end{pmatrix}$$
  - K:
$$\begin{pmatrix} 1 & 0 & 0{,}461284 \\ 0 & 1{,}00063 & 0{,}367029 \\ 0 & 0 & 0{,}00152025 \end{pmatrix}$$
  - Error: 0,161846

Como podemos ver, la distorsión afecta claramente a la calibración de la cámara ya que el error al no tener en cuenta ningún tipo de distorsión es considerablemente mayor al error teniendo en cuenta distorsión radial. Sin embargo, la diferencia entre el error sin distorsión y con distorsión tangencial es bastante similar lo que nos hace deducir que la distorsión tangencial no tiene mucha influencia en la calibración de esta cámara. Esto lo podemos comprobar al calibrar la cámara teniendo en cuenta ambas distorsiones. Observamos que el error teniendo en cuenta ambas distorsiones frente al error con distorsión radial es prácticamente igual por lo que confirmamos que la influencia de la distorsión tangencial es mucho menor que la de la distorsión radial en este caso.

3. Estimación de la matriz fundamental F

3.1. Descriptor BRISK

Para obtener puntos en correspondencia en este caso vamos a utilizar el descriptor BRISK. Primero tenemos que obtener la lista de KeyPoints y a continuación la lista de correspondencias. Este proceso es análogo al implementado en prácticas anteriores con los descriptores KAZE y AKAZE.

3.2. Cálculo de F

Una vez tenemos la lista de correspondencias procedemos al cálculo de F utilizando la función *findFundamentalMat()*.

```
Mat obtenerMatrizFundamental(vector<KeyPoint> v1, vector<KeyPoint>
    v2, vector<Point2f> &c1, vector<Point2f> &c2, vector<DMatch>
   matches, vector<unsigned char> &buenos_malos){

        for (int i=0; i < matches.size(); ++i){
                c1.push_back(v1[matches[i].queryIdx].pt);
                c2.push_back(v2[matches[i].trainIdx].pt);
        }

        //Calculamos la matriz fundamental
        Mat F = findFundamentalMat(c1, c2, CV_FM_8POINT+
           CV_FM_RANSAC,1,0.99, buenos_malos);

        int numero_descartes = 0;

        //Vemos cuantas parejas de puntos en correspondencias han
           sido descartadas por RANSAC
        for (int i=0; i < buenos_malos.size(); ++i)
                if (buenos_malos.at(i) == 0)
                        numero_descartes++;

        cout << "RANSAC ha descartado " << numero_descartes << "
           parejas en correspondencias." << endl;

        return F;
}
```

3.3. Líneas epipolares

Una vez hemos calculado la matriz fundamental F pasamos al cálculo de las líneas epipolares. Esto lo hacemos mediante la función *computeCorrespondEpilines()*.

A la hora de dibujarlas, dibujamos las líneas epipolares de la primera imagen sobre la segunda imagen y viceversa evaluándolas en $x = 0$ y en $x = n^{o} columnas de la imagen$.

```
void dibujarLineasEpipolares(Mat &vmort1, Mat &vmort2, vector<
   Point2f> &v1, vector<Point2f> &v2, Mat &F, vector<Vec3f> &l1,
   vector<Vec3f> &l2, vector<unsigned char> &buenos_malos){

        //Calculamos las lineas epipolares para los puntos de cada
            imagen
        computeCorrespondEpilines(v1, 1, F, l1);
        computeCorrespondEpilines(v2, 2, F, l2);

        Vec3f l;
        int pintadas = 0;
```

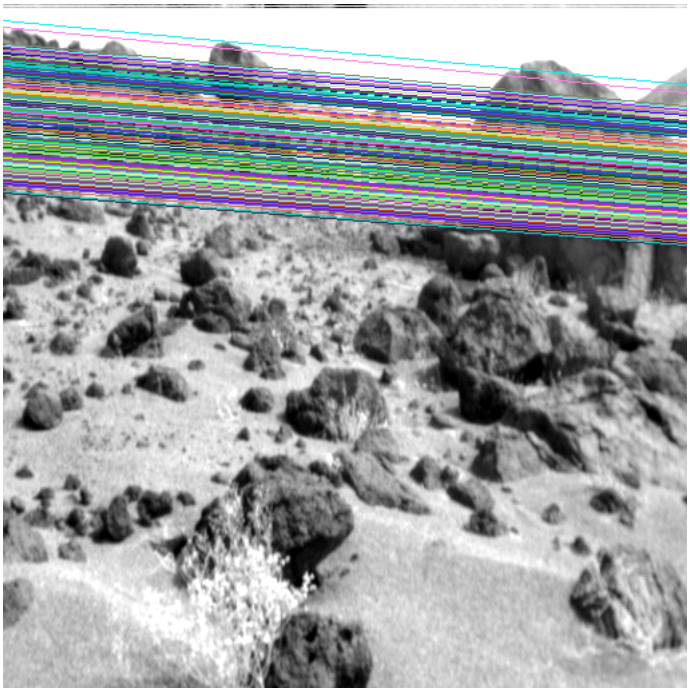
```
9           double c = vmort2.cols;
10
11          //Dibujamos las lineas epipolares evaluandolas en x = 0 y
                x = num_columnas_imagen
12          for (int i=0; i < l1.size() && pintadas <= 200; ++i) {
13                  if (buenos_malos.at(i) == 1) {
14                          l = l1.at(i);
15                          line(vmort2, Point(0, -l[2]/l[1]), Point(c
                                , (-l[2]-l[0]*c)/l[1]), CV_RGB(rand() %
                                 256,rand() % 256 ,rand() % 256));
16                          pintadas++;
17                  }
18          }
19
20          c = vmort1.cols;
21          pintadas = 0;
22
23          for (int i=0; i < l2.size() && pintadas <= 200; ++i) {
24                  if (buenos_malos.at(i) == 1) {
25                          l = l2.at(i);
26                          line(vmort1, Point(0, -l[2]/l[1]), Point(c
                                , (-l[2]-l[0]*c)/l[1]), CV_RGB(rand() %
                                 256,rand() % 256 ,rand() % 256));
27                          pintadas++;
28                  }
29          }
30
31          imshow("Lineas epipolares de los puntos de Vmort2 sobre
                Vmort1", vmort1);
32          imwrite("~/Escritorio/P3/l1.png",vmort1);
33          waitKey(0);
34          destroyAllWindows();
35
36          imshow("Lineas epipolares de los puntos de Vmort1 sobre
                Vmort2", vmort2);
37          imwrite("~/Escritorio/P3/l2.png",vmort2);
38          waitKey(0);
39          destroyAllWindows();
40  }
```

3.4. Resultados

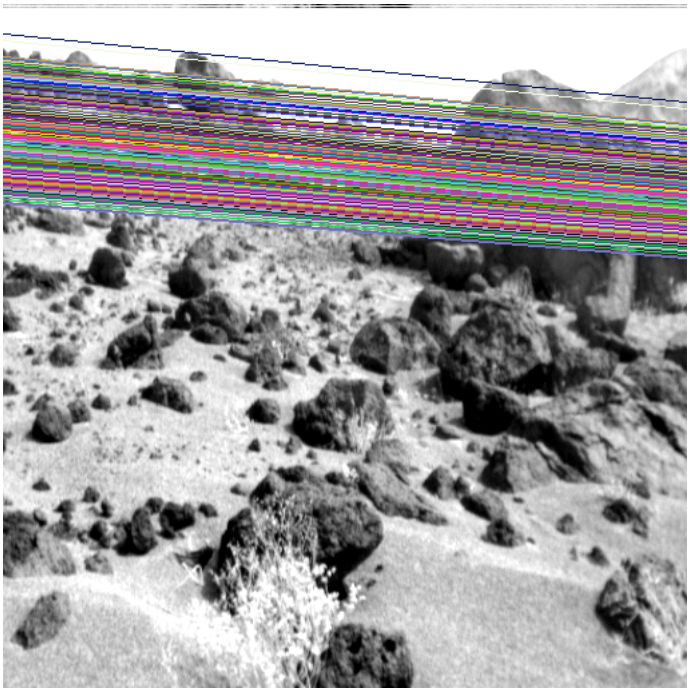
En total, se encuentran 3375 parejas en correspondencias entre las dos imágenes de las cuáles RANSAC descarta 1480 al hacer el cálculo de la matriz F. La matriz F calculada es la siguiente:

$$\begin{pmatrix} 1{,}96796 \times 10^{-7} & -1{,}53736 \times 10^{-5} & 0{,}0101106 \\ 1{,}23909 \times 10^{-5} & 2{,}19241 \times 10^{-6} & -0{,}0990446 \\ -0{,}00955949 & 0{,}0977622 & 1 \end{pmatrix}$$

En las siguiente imágenes se muestran las líneas epipolares:



(a) Líneas epiopolares de los puntos de vmort2 sobre vmort1



(b) Líneas epiopolares de los puntos de vmort1 sobre vmort2

Los errores promedio cometidos para las líneas epipolares de vmort1 y vmort2 han sido

0,393262 y 0,396766 respectivamente.

Como podemos observar las líneas epipolares calculadas son prácticamente las mismas además de que tienen errores muy similares por lo que podemos afirmar que hemos hecho una estimación de la matriz fundamental bastante buena.

4. Cálculo del movimiento de la cámara (R,t) asociado a cada pareja de imágenes calibradas

4.1. Correspondencias entre imágenes

Para el cálculo de las correspondencias entre las imágenes hemos hecho uso del descriptor BRISK de la misma manera que en el apartado anterior.

4.2. Cálculo de la matriz esencial

Para calcular la matriz esencial primero hemos estimado la matriz fundamental y hemos utilizado la siguiente matriz K:

$$F = \begin{pmatrix} 5{,}91452 \times 10^{-8} & -3{,}2595 \times 10^{-7} & -0{,}000742496 \\ 6{,}11622 \times 10^{-7} & 3{,}05802 \times 10^{-7} & 0{,}00292171 \\ 0{,}000204881 & 0{,}00163187 & 1 \end{pmatrix}$$

$$K = \begin{pmatrix} 1839{,}63 & 0 & 1024{,}2 \\ 0 & 1848{,}07 & 686{,}518 \\ 0 & 0 & 1 \end{pmatrix}$$

A partir de estas dos matrices hemos estimado la matriz esencial como

$$E = K^T F K$$

```
1  Mat estimarMatrizEsencial(Mat F, Mat K){
2
3          Mat E1 = K.t() * F;
4          Mat E = E1 * K;
5
6          return E;
7  }
```

4.3. Cálculo del movimiento

Una vez hemos estimado la matriz esencial, para estimar el movimiento(R,t) seguimos los pasos descritos en las diapositivas de teoría:

- Calculamos EE^t
- Normalizamos EE^t con su traza y estimamos la dirección T
- Definimos las matrices de R compatibles con la dirección de T
- Buscamos la combinación de matrices R y T adecuada.

```
1  void estimarMovimiento(Mat E, Mat K, vector<Point2f> corresp_1,
       vector<Point2f> corresp_2){
2
3          //Calculamos E*Et y normalizamos con su traza
4          Mat EEt = E*E.t();
5
6          double traza = 0.0;
7          for (int i=0; i < 3; ++i)
8                  traza += EEt.at<double>(i,i);
9
10         cout << "Traza de EEtrapuesta: " << traza << endl;
11
12         Mat E_norm = E / sqrt(traza/2);
13
14         Mat EEt_norm = E_norm * E_norm.t();
15
16         cout << "EEtrapuesta normalizada: " << endl;
17         mostrarMatriz(EEt_norm);
18
19         //Estimamos la direccion de T
20         Mat T = Mat(1,3, CV_64F);
21         Mat menos_T = Mat(1,3, CV_64F);
22         int fila_donde_despejar;
23
24         //Despejamos T de la fila de EEt_norm con el elemento de
               la diagonal mas pequenio
25         double elem = EEt_norm.at<double>(0,0);
26         for (int i=0; i < 3; ++i)
27                 if (EEt_norm.at<double>(i,i) <= elem) {
28                         fila_donde_despejar = i;
29                         elem = EEt_norm.at<double>(i,i);
30                 }
31
32         T.at<double>(0, fila_donde_despejar) = sqrt(1-elem);
33         T.at<double>(0,(fila_donde_despejar+1)\%3) = -EEt_norm.at<
               double>(fila_donde_despejar, (fila_donde_despejar+1)
               \%3) / sqrt(1-elem);
```

```
34          T.at<double>(0,(fila_donde_despejar+2)\%3) = -EEt_norm.at<
               double>(fila_donde_despejar, (fila_donde_despejar+2)
               \%3) / sqrt(1-elem);
35
36          menos_T.at<double>(0,0) = -T.at<double>(0,0);
37          menos_T.at<double>(0,1) = -T.at<double>(0,1);
38          menos_T.at<double>(0,2) = -T.at<double>(0,2);
39
40
41          //Construimos las rotaciones
42          Mat menos_E_norm = -E_norm;
43          Mat R_E_T = Mat(3,3,CV_64F);
44          Mat R_E_menosT = Mat(3,3,CV_64F);
45          Mat R_menosE_T = Mat(3,3,CV_64F);
46          Mat R_menosE_menosT = Mat(3,3,CV_64F);
47
48          Mat w0 = Mat(1,3, CV_64F);
49          Mat w1 = Mat(1,3, CV_64F);
50          Mat w2 = Mat(1,3, CV_64F);
51
52          Mat R0 = Mat(1,3, CV_64F);
53          Mat R1 = Mat(1,3, CV_64F);
54          Mat R2 = Mat(1,3, CV_64F);
55
56          (E_norm.row(0).cross(T)).copyTo(w0);
57          (E_norm.row(1).cross(T)).copyTo(w1);
58          (E_norm.row(2).cross(T)).copyTo(w2);
59
60          R0 = w0+w1.cross(w2);
61          R1 = w1+w2.cross(w0);
62          R2 = w2+w0.cross(w1);
63
64          (R0).copyTo(R_E_T.row(0));
65          (R1).copyTo(R_E_T.row(1));
66          (R2).copyTo(R_E_T.row(2));
67
68          (E_norm.row(0).cross(menos_T)).copyTo(w0);
69          (E_norm.row(1).cross(menos_T)).copyTo(w1);
70          (E_norm.row(2).cross(menos_T)).copyTo(w2);
71
72          R0 = w0+w1.cross(w2);
73          R1 = w1+w2.cross(w0);
74          R2 = w2+w0.cross(w1);
75
76          (R0).copyTo(R_E_menosT.row(0));
77          (R1).copyTo(R_E_menosT.row(1));
78          (R2).copyTo(R_E_menosT.row(2));
79
80          (menos_E_norm.row(0).cross(T)).copyTo(w0);
```

```
81          (menos_E_norm.row(1).cross(T)).copyTo(w1);
82          (menos_E_norm.row(2).cross(T)).copyTo(w2);
83
84          R0 = w0+w1.cross(w2);
85          R1 = w1+w2.cross(w0);
86          R2 = w2+w0.cross(w1);
87
88          (R0).copyTo(R_menosE_T.row(0));
89          (R1).copyTo(R_menosE_T.row(1));
90          (R2).copyTo(R_menosE_T.row(2));
91
92          (menos_E_norm.row(0).cross(menos_T)).copyTo(w0);
93          (menos_E_norm.row(1).cross(menos_T)).copyTo(w1);
94          (menos_E_norm.row(2).cross(menos_T)).copyTo(w2);
95
96          R0 = w0+w1.cross(w2);
97          R1 = w1+w2.cross(w0);
98          R2 = w2+w0.cross(w1);
99
100         (R0).copyTo(R_menosE_menosT.row(0));
101         (R1).copyTo(R_menosE_menosT.row(1));
102         (R2).copyTo(R_menosE_menosT.row(2));
103
104         cout << "La rotacion para E y T:" << endl;
105         mostrarMatriz(R_E_T);
106
107         cout << "La rotacion para E y -T:" << endl;
108         mostrarMatriz(R_E_menosT);
109
110         cout << "La rotacion para -E y T:" << endl;
111         mostrarMatriz(R_menosE_T);
112
113         cout << "La rotacion para -E y -T:" << endl;
114         mostrarMatriz(R_menosE_menosT);
115
116         vector<Mat> rotaciones;
117         rotaciones.push_back(R_E_T);
118         rotaciones.push_back(R_E_menosT);
119         rotaciones.push_back(R_menosE_T);
120         rotaciones.push_back(R_menosE_menosT);
121
122         //Obtenemos la distancia focal en pixels de la matriz de
               calibracion K
123         double f = K.at<double>(0,0);
124
125         int num_corresp = corresp_1.size();
126         double dot1, dot2, Zi, Zd;
127         Mat pi=Mat(1,3,CV_64F);
128         Mat Pi=Mat(1,3,CV_64F);
```

```
129             pi.at<double>(0,2) = 1.0;
130
131             int R_act = 0;
132             Mat R = rotaciones.at(R_act);
133             Mat T_act = Mat(1,3,CV_64F);
134             T.copyTo(T_act);
135
136             int contador = 0;
137             bool encontrado = false;
138             bool cambio;
139
140             //Vemos que combinacion es la adecuada
141             while (!encontrado) {
142
143                     cambio = false;
144
145                     for (int i=0; i < corresp_1.size() && !cambio && !
                            encontrado; ++i) {
146
147                             //Calculamos Zi y Zd
148                             pi.at<double>(0,0) = corresp_1.at(i).x;
149                             pi.at<double>(0,1) = corresp_1.at(i).y;
150
151                             dot1 = (f*R.row(0) - corresp_2.at(i).x*R.
                                    row(2)).dot(T_act);
152                             dot2 = (f*R.row(0) - corresp_2.at(i).x*R.
                                    row(2)).dot(pi);
153
154                             Zi=f*dot1/dot2;
155
156                             Pi=(Zi/f)*pi;
157
158                             Zd = R.row(2).dot(Pi-T_act);
159
160                             //Si ambos negativos cambiamos el signo a
                                    T
161                             if (Zi < 0 && Zd < 0) {
162                                     T_act = -T_act;
163
164                                     if (R_act%2 == 0)
165                                             R_act++;
166                                     else
167                                             R_act--;
168
169                                     R = rotaciones.at(R_act);
170                                     cambio = true;
171                             }
172                             //Si tienen signos distintos cambiamos de
                                    signo a la E
```

```
173                             else if (Zi*Zd < 0){
174                                     R_act = (R_act+2)\%4;
175                                     R = rotaciones.at(R_act);
176                                     cambio = true;
177                             }
178                             //Si los dos positivos, hemos acabado.
179                             else
180                                     encontrado = true;
181                     }
182             }
183
184             cout << "La matriz R es:" << endl;
185             mostrarMatriz(R);
186             cout << "La matriz T es: " << endl;
187             mostrarMatriz(T_act);
188
189     }
```

4.4. Resultados

En este caso, hemos utilizados las imágenes *rdimage.001* y *rdimage.004*. Como resultado final obtenemos las siguientes matrices al estimar el movimiento:

$$R = \begin{pmatrix} 0{,}415084 & -0{,}640161 & 0{,}352441 \\ -0{,}587474 & -0{,}794114 & -0{,}477394 \\ 0{,}476274 & -0{,}219982 & -0{,}634723 \end{pmatrix}$$

$$T = \begin{pmatrix} 0{,}871646 & -0{,}387093 & 0{,}189965 \end{pmatrix}$$