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

Memoria Práctica 3

Ignacio Martín Requena

9 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. Generar la matriz de una cámara finita P a partir de valores aleatorios en [0,1] . Verificar si representa una cámara finita y en ese caso quedársela.

Para este primer apartado se ha implementado la función GenerarMatrizP(Mat &P) la cual genera una matriz de cámara finita P a partir de valores aleatorios.

Para ello en primer lugar generamos 12 numeros aleatorios que son los que compondran la matriz P 3x4 . Como P se puede descomponer como el resultado de [M | m], teniendo entonces M dimensiones 3x3, podemos comprobar directamente si el determinante de M es mayor que 0 y, en caso de que lo sea, P será una matriz válida.

1.2. Suponer un patrón de puntos del mundo 3D compuesto por el conjunto de puntos con coordenadas (0,x1,x2) y (x2,x1,0), para x1=0.1:0.1:1 y x2=0.1:0.1:1. Esto supone una rejilla de puntos en dos planos distintos ortogonales

La función implementada para realizar este segundo apartado es vector <Point3d> GenerarPatron3D(), la cual devuelve un patrón 3D generado.

Esta función lo que hace es generar los puntos p1= (0,x1,x2) y p2 = (x2,x1,0) siendo x1 y x2 valores que aumentan de 0,1 en 0,1 hasta llegar a 1. De esta forma crearemos 200 puntos.

1.3. Proyectar el conjunto de puntos del mundo con la cámara simulada y obtener las coordenadas píxel de su proyección.

Para empezar realizaremos tres tareas: transformar los puntos 3D del apartado anterior en matrices 4x1, generar los puntos proyectados y generar las coordenadas pixel usando los puntos proyectados.

La primera de estas tareas la realiza la función vector<Mat>generarMatricesDePuntos (vector<Point3d>punt) la cual recibe un vector de puntos 3D y crea para cada punto 3D del vector una matriz de 4x1 de la forma: [p[i].x,p[i].y,p[i].z,1].

Ahora necesitamos generar los puntos proyectados, de esto se encarga la función vector<Mat>generarPuntosProyectados(Mat P, vector<Mat>mat) cuya labor es multiplicar la matriz p generada en el primer apartado y los puntos 3D transformados en matrices 4x1, lo que devolverá puntos 3D.

Acabamos generando las coordenadas pixel. Estas son el resultado de dividir cada coordenada X e Y de un punto por su correspondiente componente Z en cada uno de los puntos del vector obtenido en el apartado anterior. Lo que obtendremos, por tanto, será un vector de matrices de dimensiones 2x1.

1.4. Implementar el algoritmo DLT para estimar la cámara P a partir de los puntos 3D y sus proyecciones en la imagen.

Este algoritmo lo vamos a dividir en tres pasos:

- **Calculo de K y R a partir de M**

  A partir de la matriz original P y sabiendo que esta es la composición de P=[M|m] calculamos las matrices R y K. Esta tarea la realiza la función calcularRotaciones(M,K,R). Para la implementación de esta función se ha seguido lo expuesto en el apartado de Descomposición de P del tema *Computer Vision: Cameras* explicado en clase, concretamente las diapositivas 45 y 46.

  Básicamente lo que hacemos es generar las matrices Qx,Qy y Qz, a partir de calculando c y s a partir de M para Qx, a partir de M*Qx para Qy y a partir de M*Qx*Qy para Qz. Obtenido esto podemos determinar K como M*Qx*Qy*Qz y R como la transpuesta de Qx*Qy*Qz.

- **Calculo de T**

  Una vez obtenida la matriz K, T la deducimos como K^{-1}*m

- **Obtención de P a partir de K, R y T**

  Para ello construimos una matriz 3x4 de la siguiente forma (función obtenerP(K, R, T);):

$$P = K * \begin{pmatrix} r11 & r21 & r31 & t11 \\ r12 & r22 & r32 & t21 \\ r13 & r23 & r33 & t31 \end{pmatrix}$$

1.5. Calcular el error de la estimación usando la norma de Frobenius (cuadrática)

Para el cálculo de este error aplicamos la fórmula:

$$\sum_{i=1}^{n}\sum_{j=1}^{n}(Pij - PP_{ij})^2$$

Obteniendo la siguiente salida:

```
alumno@VC32b:~/workspace/c++/practicavc/make$ make
ejecutando  practicavc  ....
./practicavc

EJERCICIO 1
 P (aleatoria)=
[0.060565587, -0.60148162, -0.19788112, 0.62877017;
 -0.12573405, -0.5024206, 0.54621011, 0.52418745;
 -0.38441104, 0.40486339, -0.043105587, 0.58438003]

 P (algoritmo DLT)=
[0.060565561, -0.60148156, -0.19788112, 0.62877017;
 -0.12573408, -0.50242054, 0.54621005, 0.52418739;
 -0.38441098, 0.4048633, -0.043105572, 0.58438003]

El error cuadratico de Frobenius: 2.75474e-14
```

Figura 1.1: Salida error Frobenius

Como podemos ver el error de la estimación usando Frobenius es bastante bajo, por lo que aparentemente podemos afirmar que los resultados son buenos.

2. Calibración de la cámara usando homografías

2.1. Escribir una función que sea capaz de ir leyendo las sucesivas imágenes en chessboard.rar y determine cuáles son válidas para calibrar una cámara. Usar las 25 imágenes tiff que se incluyen en el fichero datos. Usar la función cv::findChessboardCorners(). Determinar valores precisos de las coordenadas de las esquinas presentes en las imágenes seleccionadas usando cv::cornerSubpix(). Pintar sobre la imagen los puntos estimados usando la función cv::drawChessboardCorners() (ver código de ayuda en la documentación de OpenCV)

Después de la lectura de las imagenes chessboard vamos a ir recorriendolas una a una para determinar si una imagen es o no valida para calibrar una camara, para ello en primer lugar transformamos la imagen a una con escala de grises y, seguidamente, aplicamos la función findChessboardCorners de la siguiente forma:

```
1 bool found = findChessboardCorners(gray_image, board_sz, corners,
      CALIB_CB_ADAPTIVE_THRESH + CALIB_CB_NORMALIZE_IMAGE +
      CALIB_CB_FAST_CHECK);
2 if (found) {
3         devol = true;
4         //Obtenemos los corners
5         cornerSubPix(gray_image, corners, Size(11, 11), Size(-1,
              -1), TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER,
              30, 0.1));
```

```
6             //Dibujamos sobre la imagen
7             drawChessboardCorners(gray_image, board_sz, corners, found
                 );
8             sal.push_back(gray_image);
9     }
```

Como podemos ver, la funcion findChessboardCorners recibe la imagen del tablero, el tamaño de este (en nuestro caso 13x12), el vector donde se almacenen las esquinas encontradas y una serie de flags para usar un thresholding adaptativo, normalizar el gamma de la imagen antes de aplicar el thresholding y hacer un checkeo rápido buscando corners para acabar de forma rápida si la imagen no nos sirve.

Por último solamente quedaría que, en caso de que la imagen si nos valga, obtener los corners en concreto y dibujarlos sobre la imagen

En la salida vamos a obtener que 4 de las 25 imágenes son válidas para calibrar la cámara, en concreto las imágenes numero 9, 11, 17 y 20. Estas imágenes con sus detecciones de esquinas son:

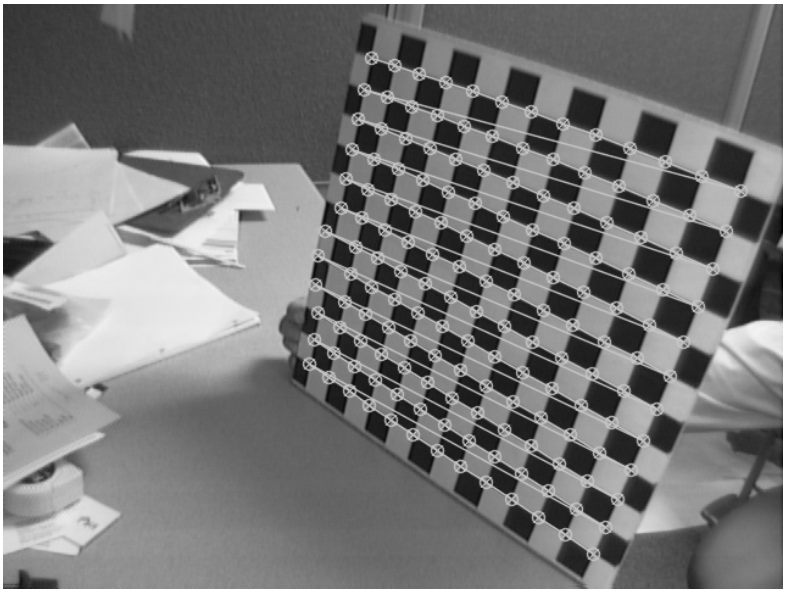

Figura 2.1: Detección imagen 9

Figura 2.2: Detección imagen 11

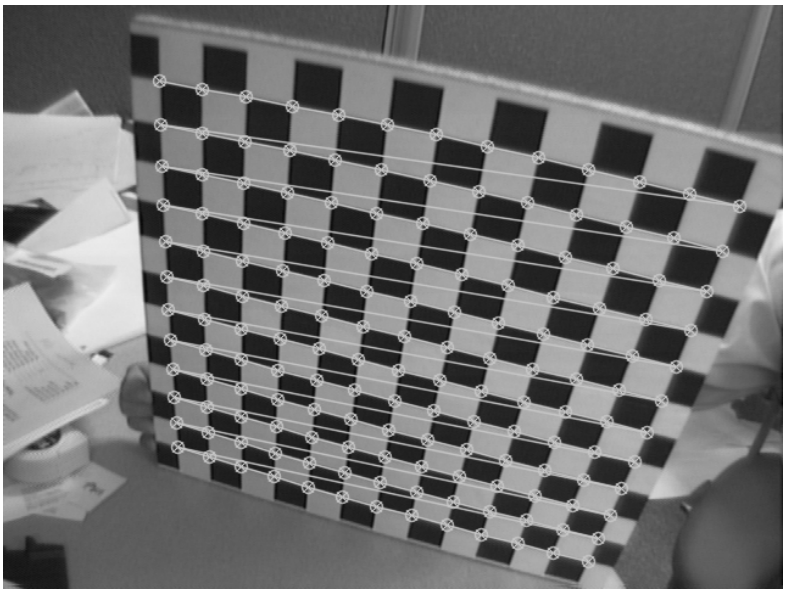

Figura 2.3: Detección imagen 17

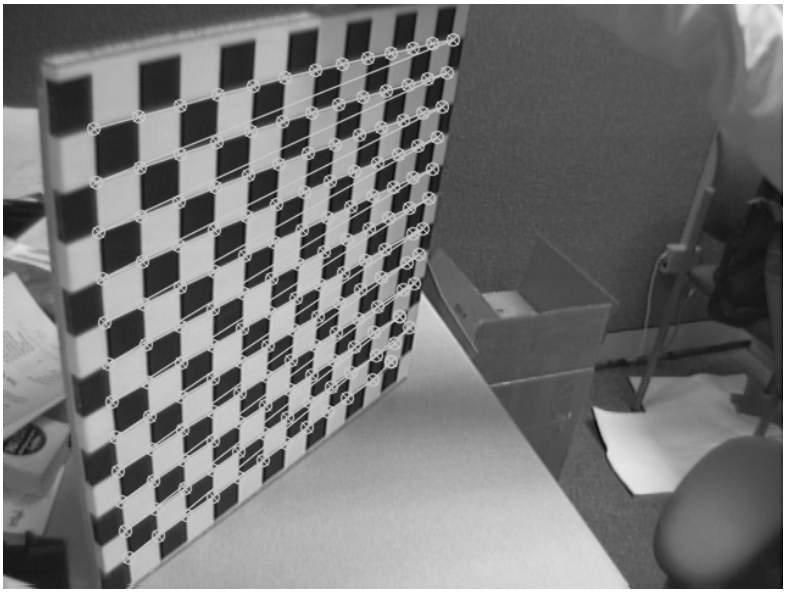

Figura 2.4: Detección imagen 20

2.2. Usando las coordenadas de los puntos extraídos en las imágenes seleccionadas del punto anterior, calcular los valores de los parámetros intrínsecos y extrínsecos de la cámara para cada una de dichas imágenes. Usar la función cv::calibrateCamera(). Suponer dos situaciones: a) sin distorsión óptica y b) con distorsión óptica. Valorar la influencia de la distorsión óptica en la calibración y la influencia de la distorsión radial frente a la distorsión tangencial.

En este apartado el grueso de la cuestión gira en torno a la función de opencv calibrateCamera, la cual se define de la siguiente manera:

```
1          calibrateCamera(object_points, image_points, image.size(),
               intrinsic, distCoeffs, rvecs, tvecs, [FLAGS]);
2          cout << endl << "distCoeffs con distorsion= " <<
              distCoeffs << endl << endl;
3
4          Mat imageUndistorted;
5          undistort(image, imageUndistorted, intrinsic, distCoeffs);
6
7          cout << endl << "distCoeffs sin distorsion= " <<
              distCoeffs << endl << endl;
8
9          sal.push_back(imageUndistorted);
```

Obtendremos dos salidas de coeficientes, una con distorsion óptica y otra sin, determinadas por los flags de calibrateCamera (la imagen sin ningun tipo de distorsion se obtiene

añadiendo los flags CV_CALIB_FIX_K1 or CV_CALIB_FIX_K2 or CV_CALIB_FIX_K3 or CV_CALIB_FIX_K4 or CV_CALIB_FIX_K5 or CV_CALIB_FIX_K6 or CV_CALIB_ZERO_TANGENT_DIST).

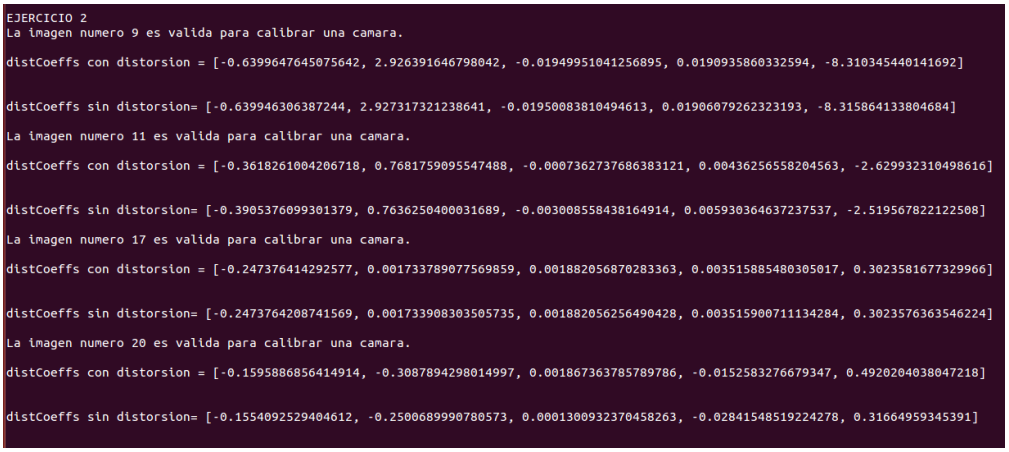

Figura 2.5: Coeficientes con y sin distorsion

Como vemos, lo resultados son muy parecidos, por lo que, o a simple vista el hecho de que haya distorsión óptica o no en nuestro caso no influye o quizá la comparación no es la correcta.

3. Estimación de la matriz fundamental F

3.1. Obtener puntos en correspondencias sobre las imágenes Vmort[*].pgm de forma automática usando las funciones de BRISK/ORB

Para este apartado se ha usado parte de la practica anterior. En concreto las funciones de detección de puntos en correspondencia Akaze, kaze y puntosEnComun explicadas en la práctica anterior, por lo que pasaremos directamente a la valoración de los resultados:

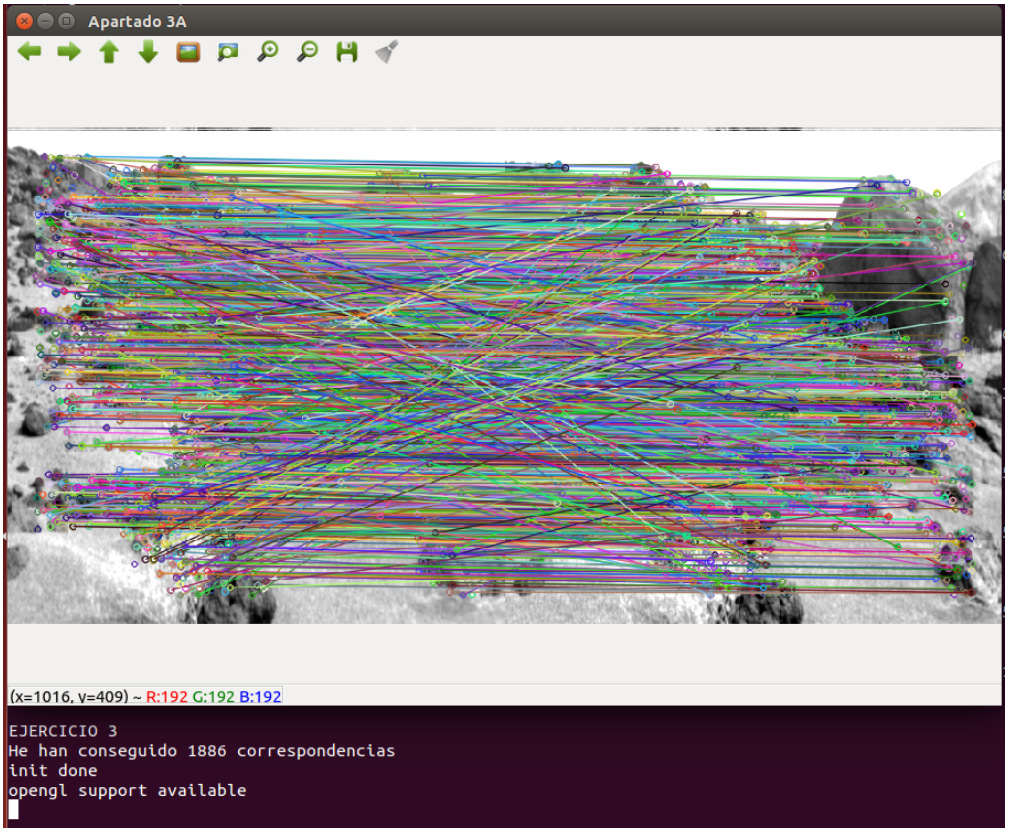

Figura 3.1: Correspondencias usando Akaze

Hemos obtenido 1886 correspondencias, lo que es un numero bastante elevado.

3.2. Calcular F por el algoritmo de los 8 puntos + RANSAC (usar un valor pequeño para el error de RANSAC)

Para este apartado vamos, en primer lugar, a transformar la lista de keypoints que el descriptor Akaze nos proporcionó en el apartado anterior en una lista de Point2f con el objetivo de facilitarnos los cálculos. Una vez hecho esto dibujamos un circulo en la imagen que corresponda en cada punto keypoint anteriormente convertido a Point2f.

Una vez hecho esto obtenemos F usando la función findFundamentalMat que proporciona OpenCV y aplica RANSAC de la siguiente forma:

```
1          F = findFundamentalMat(Mat(points1), Mat(points2),
               CV_FM_RANSAC, 3, 0.999);
```

Obteniendo la matriz F:

```
F = [-8.794428482743167e-07, 5.258687961328477e-05, -0.01452692991249663;
-8.8035816195988e-05, -0.0002452483088249054, -0.3822938269450242;
0.01738064443420771, 0.4312058609832226, 1]
```

Figura 3.2: Matriz F 3b

3.3. Dibujar las líneas epipolares sobre ambas imágenes (<200).

En primer lugar hacemos uso de la función computeCorrespondEpilines que a partir de los puntos usados en el apartado anterior para generar F y la propia F nos proporciona una lista de parejas de puntos que determinan las lineas obtenidas.

Ahora quedaría dibujar las lineas en la imagen y comprobar como de buenos son nuestros resultados. Esto lo hacemos con la función dibujarLineaEpipolar cuyo contenido es:

```
Mat dibujarLineaEpipolar(Mat img, vector<Vec3f> lines){
        RNG rng(200);
        Scalar color;
        Mat sal = img;
        uint condfin = 200;
        for (uint i=0; i<condfin; i++){
                color = Scalar(rng.uniform(0, 255), rng.uniform(0,
                    255), rng.uniform(0, 255));
                line(sal, Point(0, -lines[i][2] / lines[i][1]),
                   Point(img.cols, -(lines[i][2] + lines[i][0] *
                   img.cols) / lines[i][1]), color);
                }
        return sal;
}
```

El color de cada linea es determinado aleatoriamente y la posición de cada punto en la imagen se determina directamente en la función line.

La salida obtenida en este caso es:

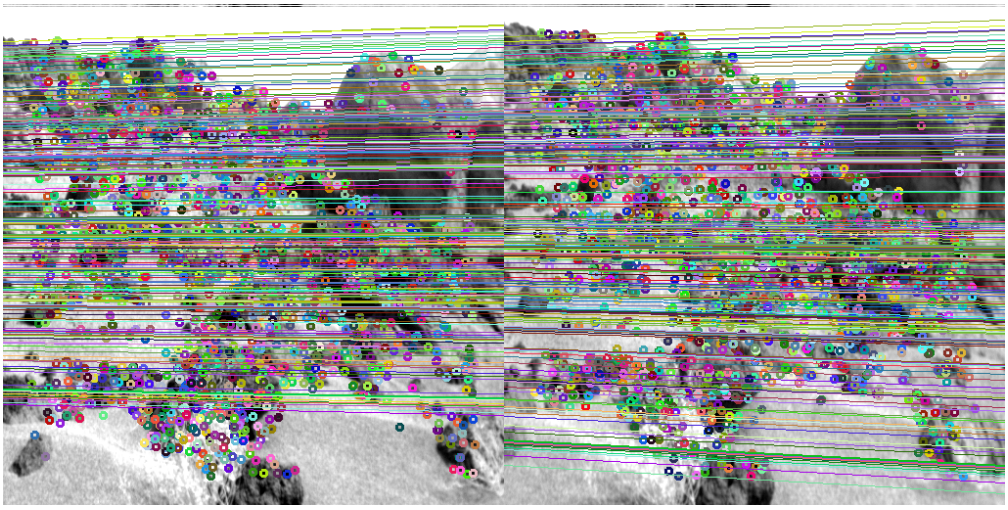

Figura 3.3: Lineas Epipolares

Como podemos observar a simple vista el resultado no es nada bueno, las lineas no siguen ningún patrón esperado. Aquí me ha sido imposible determinar cual es el origen del problema, pero seguramente la raíz de este sea un mal cálculo de la matriz F o un mal cálculo de la posición de las lineas a la hora de pintarlas.

3.4. Verificar la bondad de la F estimada calculando la media de la distancia ortogonal entre los puntos soporte y sus líneas epipolares en ambas imágenes. Mostrar el valor medio del error.

El calculo del error medio se basa en una media de las distancias entre las lineas y los puntos que hemos obtenido en los apartados anteriores en cada una de las dos imágenes. La bondad obtenida es de 153.146, lo que muestra que los resultados no son muy buenos a priori.

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

4.1. Usar las imágenes y datos de calibración dados en el fichero reconstruccion.rar

En el caso de las imágenes la lectura de estas se ha realizado con funciones de OpenCV como ha sido habitual durante todo el curso. En el caso de los archivos .camera se ha tenido que implementar una función llamada LeerCamera para su lectura. En esta función se ha presupuesto que no existen comentarios en el archivo con el fin de facilitar la programación.

4.2. Calcular parejas de puntos en correspondencias entre las imágenes

Este apartado merece poca explicación, ya que lo que vamos a realizar es el calculo de los puntos en común entre las imágenes, cosa que ya hemos implementado en el apartado 3A, por tanto, llamaremos a la función de este apartado y posteriormente convertiremos los keypoints calculados a Point2f con el fin de facilitar los cálculos.

4.3. Estimar la matriz esencial y calcular el movimiento.

La estimación de la Matriz esencial la realizamos como hicimos previamente, esto es, usando la función findFundamentalMat la cual nos devuelve una estimación de F.

La Matriz de movimiento E sabemos que es el resultado de K2.t()*F*K1, así que podemos calcularla directamente. Tanto la matriz R_E como la T_E no he conseguido llegar a ninguna solución razonable.

La salida obtenida sería:

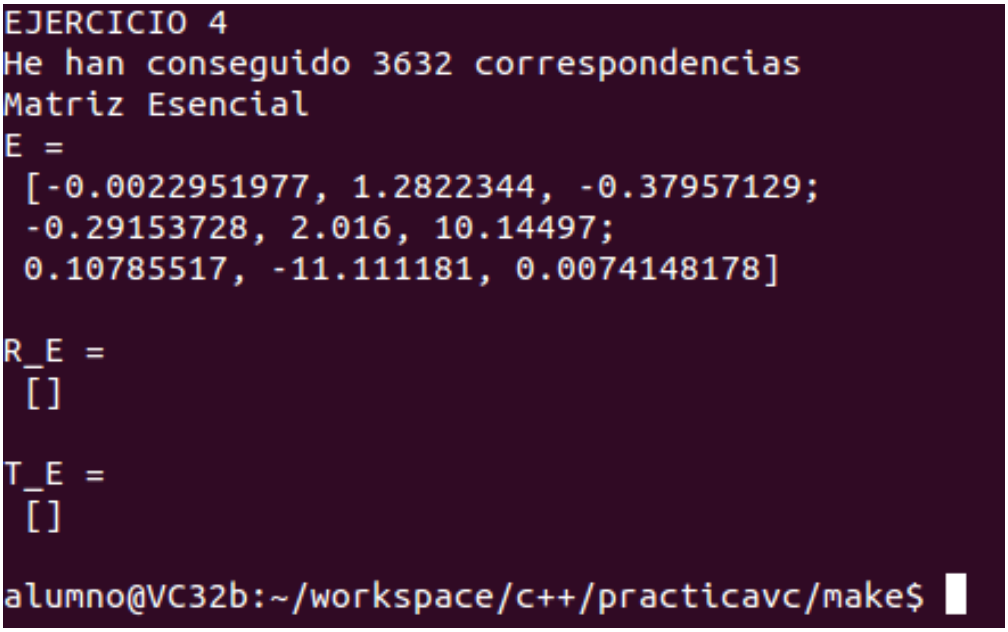

Figura 4.1: Lineas Epipolares