

# Tarea 3: Descriptor Matches Filter using RANSAC & Hough

Estudiante: Luis Escares

Profesor: Javier Ruiz del Solar

Auxiliar: Patricio Loncomilla

## I. INTRODUCCIÓN

Lo que se realiza en esta tarea es para un grupo de 4 pares de imágenes calcular los puntos de interés DoG y sus descriptores SIFT. Para cada par de imágenes, se calculan los calces iniciales minimizando la distancia entre descriptores. Se implementa una función que calcula la transformación de semejanza a partir de un calce; la función que calcula el consenso de dicha transformación, es decir, la cantidad de calces cuyo error de proyección es menor a un umbral. Se programan las funciones que realizan los métodos de RANSAC y la transformada de Hough. Se programa y explica la función que calcula una transformación afín usando mínimos cuadrados con base en los DMatches filtrados por RANSAC y Hough; y la función que dibuja un romboide que representa la imagen de prueba proyectada en la imagen de referencia. Se aplican dichas funciones a los 4 pares de imágenes, se varían los hiperparámetros de los métodos de RANSAC y Hough. Se presentan todas las imágenes resultantes y se comentan que tan bien funcionan los métodos bajo distintos hiperparámetros. Se programa un script que ejecuta automáticamente todas estas funciones en todas las imágenes entregadas. Llegando finalmente a un script que se ejecuta una sola vez y genera todas las imágenes requeridas por esta tarea.

Las secciones del informe corresponden a: Un marco teórico donde se describe el método RANSAC en general y se describe su aplicación usando como datos las transformaciones de semejanza generadas por cada match de keypoints. Luego, se detalla el método de la transformada de Hough genérica y se expone a cerca de como es su aplicación en la detección de líneas en imágenes. Del mismo modo que en RANSAC, se explica como se aplica la transformada de Hough para encontrar un conjunto de transformación de semejanza que sirvan para definir una transformación afín. Finalmente, se describe como a partir de dicho conjunto de matches filtrados ya sea por RANSAC o por Hough se calcula la transformación afín mediante mínimos cuadrados.

La sección de desarrollo en general presenta las imágenes obtenidas como resultado del procesamiento, el funcionamiento general del script entregados y las partes más importantes del código.

Primero se parte describiendo como se detectan y grafican puntos de interés SIFT en 4 pares de imágenes, se presentan las imágenes con los keypoints graficados y se comenta porque los keypoints DoG son de tipo Blob. Luego, se generan y grafican calces entre puntos en 4 pares de imágenes basados en la menor distancia euclíadiana entre descriptores y se hacen observaciones sobre lo correcto que son localmente pero que algunos calces en el contexto general de la imagen no tienen sentido. Esta es la parte que venía implementada en el código.

Para la parte de las funciones que se implementan, se parte describiendo como funciona el código que crea la transformación de semejanza a partir de un calce. Se describe que es el consenso y como se calcula para RANSAC, para luego describir cómo funciona la implementación realizada de RANSAC que ocupa estas dos funciones descritas. Luego se describe el código que implementa la transformada de Hough. Para finalizar con los códigos de cálculo de transformación afín con mínimos cuadrados usando los matches resultantes de RANSAC y Hough; y de la función que dibuja un romboide que representa la proyección de la imagen de prueba en la de referencia a partir de transformación afín.

En la última sección del desarrollo se presentan las imágenes resultantes de aplicar RANSAC y Hough, se comenta qué tan bien funcionaron. Además, se presentan las imágenes resultantes al variar los hiperparámetros de estos métodos y se comenta con cuales llegan a una proyección aceptable o por qué no llegan a una.

Por último, en la conclusión se hace una mirada más general de los resultados obtenidos y de sus análisis. Se discute hasta qué punto se observan los comportamientos teóricos esperados para la metodología de usar RANSAC o Hough y luego mínimos cuadrados. Se comenta sobre las dificultades superadas para realizar esta experiencia y se finaliza con los aprendizajes y habilidades desarrolladas.

## II. MARCO TEÓRICO

### II-A. RANSAC como transformación de semejanza

El método RANSAC, que significa random sample consensus, es un método para calcular los parámetros de un modelo que explica la distribución de un conjunto de datos que contiene inliers y outliers. Es un método no determinista porque produce parámetros de un modelo razonable con una cierta probabilidad que aumenta

conforme este se ejecuta con más iteraciones. En resumen, a RANSAC entra el conjunto de muestras o puntos, el error tolerable para aceptar que la muestra es explicada por el modelo y un umbral de la mínima cantidad de muestras que son explicadas por un modelo, para que el modelo sea aceptado como razonable.

Existen escenarios en que RANSAC puede no llegar a un modelo que explique los datos, como también, puede darse que existan dos o más instancias de modelos que explican los datos. Por ejemplo, al buscar líneas, puede darse que para un conjunto de puntos realmente el mejor resultado es que existan 2 líneas, en ese caso, RANSAC sólo encuentra una línea.

En cada iteración, el algoritmo de RANSAC toma un pequeño subconjunto de muestras para calcular una hipótesis de modelo, por ejemplo, para detectar líneas toma 2 puntos para calcular el modelo de una recta. Luego, itera sobre cada una de las demás muestras y calcula el error asociado a que esa muestra sea generada por la hipótesis de modelo; si el error es menor al umbral de error nombrado anteriormente, el punto se adhiere en el conjunto de consenso. Una vez recorridos todos los puntos, se guarda el consenso de esa hipótesis. Se repite el proceso tantas veces como iteraciones del algoritmo se configuren. Al finalizar, se busca la hipótesis (los parámetros de un modelo) con más votos de consenso y si esa cantidad es mayor a la definida por el segundo umbral mencionado, se acepta el modelo siendo este el resultado RANSAC.

El uso de RANSAC con transformación de semejanza se basa en lo explicado en el párrafo anterior, sólo que en este caso la salida no son parámetros de un modelo sino que el conjunto de DMatches que tienen error menor al error tolerable para que la muestra sea explicada por la hipótesis de modelo. Entonces, a este RANSAC ingresa un conjunto de matches, en adelante se referirá a ellos como DMatch por su objeto en OpenCV, los umbrales y la cantidad de iteraciones. Se toma un DMatch al azar, se le calcula su transformación de semejanza con las siguientes ecuaciones. Estos parámetros calculados son un símil de la hipótesis de modelo mencionada en el párrafo anterior.

$$e = \frac{\sigma_{PRU}}{\sigma_{REF}} \quad (1)$$

$$\theta = \varphi_{PRU} - \varphi_{REF} \quad (2)$$

$$t_X = x_{PRU} - e(x_{REF} \cos(\theta) - y_{REF} \sin(\theta)) \quad (3)$$

$$t_Y = y_{PRU} - e(x_{REF} \sin(\theta) + y_{REF} \cos(\theta)) \quad (4)$$

Luego, se itera sobre cada uno de los DMatch del vector de matches, para cada uno se calcula el error de proyección con la formula:

Si ese error es menor que el umbral para que el dato sea explicado por el modelo, se suma uno al consenso de la hipótesis de modelo; y así sucesivamente se recorren el

$$e_{PROY} = \left\| e \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x_{REF} \\ y_{REF} \end{pmatrix} + \begin{pmatrix} t_X \\ t_Y \end{pmatrix} - \begin{pmatrix} x_{PRU} \\ y_{PRU} \end{pmatrix} \right\|$$

resto de los DMatch. Luego, se toma otro DMatch al azar y se repite el calculo de la transformación de semejanza, de errores de proyección, comparación y consenso. Se toman al azar tantos DMatch como iteraciones se figuren. Al final, se toma la hipótesis de modelo con el mayor consenso y si ese consenso es mayor a la cantidad mínima de votos para considerar un modelo como aceptable, se acepta esa hipótesis. Se recorre una última vez todos los DMatches determinando cuales votaron por la hipótesis de modelo aceptada. Finalmente, estos se retornan como resultado del algoritmo.

## II-B. Transformada de Hough genérica aplicada en detección de lineas

Al igual que RANSAC, se puede generalizar la definición de la transformada de Hough a un método para calcular los parámetros de un modelo que explica la distribución de un conjunto de datos que contiene inliers y outliers. Aunque en la práctica la transformada de Hough se conoce comúnmente como un método de detección de figuras geométricas como líneas, circunferencias y elipses en imágenes.

Un resumen del algoritmo de la transformada de Hough es que ésta parte con un conjunto de puntos en los que se quiere encontrar una figura geométrica, por ejemplo una recta. La figura geométrica a buscar debe ser parametrizable por una cantidad discreta y pequeña de parámetros. Por ejemplo, una recta se parametriza con 2 parámetros m,n y una circunferencia con 3 parámetros (coords. centro y radio). Los parámetros deben ser pensados para que pertenezcan a un espacio discretizable, por ejemplo, comúnmente se pasa a los parámetros de una recta a coordenadas polares para discretizar más fácilmente el ángulo. Entonces, se construye un tabla llamada *acumulador* (o un array multidimensional, es decir, un tensor) en que cada una de sus casillas represente un intervalo de cada parámetro de las dimensiones que definen la casilla. Para la detección de lineas, por ejemplo, una casilla podría estar caracterizada por los  $\theta$  entre 0 y 10 grados y los  $\rho$  entre 0 y 10 por dar algún intervalo de valores para  $\rho$ .

Entonces, en el caso de la detección de líneas, se van recorriendo todos los puntos del conjunto original de datos. Un punto en el espacio x,y corresponde a una recta en el espacio m,n de Hough, lo que representa un problema porque una recta vertical en el espacio x,y requeriría un m infinito en el espacio m,n. Entonces, el punto x,y se pasa a coordenadas polares al espacio  $\theta, \rho$  usando la fórmula:

$$x \cos \theta + y \sin \theta = \rho \quad (5)$$

Luego, hacer Hough para líneas es tomar un punto (x,y) luego ir variando  $\theta$  de x grados en x grados(o radianes), calcular el  $\rho$  e ir registrando un voto en la celda de la tabla *acumulador* según el  $\rho$  resultante caiga en el intervalo de  $\rho$  que define cada celda.

Una vez recorrido todos los puntos, se busca en la tabla las casillas  $(\theta,\rho)$  que sean máximos locales. Estos máximos locales definen una hipótesis de modelo, luego para cada uno de ellos se compara si su cantidad de votos es mayor a un umbral para aceptar como posible un modelo y finalmente resultan los  $(\theta,\rho)$  que tienen votos mayores que ese umbral; cada uno de estos finalmente define una recta.

En esta parte final depende de la aplicación si se permite terminar con más de una recta detectada o sólo una. Lo realmente importante es que en suma la transforma de Hough es la búsqueda de modelo que explica los datos que se transforma en la búsqueda de las intersecciones de las curvas que se van construyendo en el espacio de Hough al recorrer todos los puntos. Es por esto que es fundamental discretizar correctamente el espacio de Hough en adecuados intervalos de ángulo y  $\rho$  los cuales los da la investigación hecha.

### II-C. Transformada Hough de como transformación de semejanza

Al igual que en el caso de RANSAC, para la transformación de semejanza el resultado de aplicar la transformación de Hough no es una, y sólo una, transformación de semejanza. Si no que, es un conjunto de DMatches con los que se va a poder construir una transformación afín de mejor manera.

Entonces, a esta adaptación de la transformada de Hough se le ingresa el vector con los DMatches. Se itera sobre cada DMatch, para cada DMatch se calcula la transformación de semejanza que representa. El espacio de todas las transformaciones de semejanza esta discretizado por los parámetros i, j, k, z y gracias a las investigaciones realizadas la forma robustamente buena para pasar los parámetros e,  $\theta$ ,  $t_X$ ,  $t_Y$  es usando las siguientes formulas:

$$i = \text{floor}\left(\frac{t_X}{dxBin} + 0.5\right) + 500 \quad (6)$$

$$j = \text{floor}\left(\frac{t_Y}{dxBin} + 0.5\right) + 500 \quad (7)$$

$$k = \text{floor}\left(\frac{\theta}{dangBin} + 0.5\right) + 500 \quad (8)$$

$$z = \text{floor}\left(\frac{\log(e)}{\log(2.0)} + 0.5\right) + 500 \quad (9)$$

Los valores de  $dBin$ ,  $dangBin$ , 0.5 y 500 vienen dados de la investigación. Lo importante es que se puede intentar variar un poco y obtener resultados parecidos. Lo que

se debe entender es que estos son valores referenciales, buenos para la mayoría de los casos. Cuando se construyen sistemas específicos, estos valores se pueden ajustar para mejorar los resultados, es decir, hacer una especie de fine tuning.

Entonces, se recorren todos los DMatch calculando los i, j, k, z y haciendo el voto en esa casilla. Luego, se determina la casilla con más votos. Luego se pregunta si esa cantidad de votos es mayor al umbral mínimo para considerar como aceptable un modelo, el cual es el segundo umbral que se entrega. Por último se vuelve a recorrer los DMatch para determinar cuales votaron por ese cuarteto de índices, los que votaron son retornados como el resultado de Hough para transformaciones de semejanza.

### II-D. Transformación afín con mínimos cuadrados

Entonces, bajo un paradigma de fuerza bruta se puede llegar y tomar todos los DMatches, hacer mínimos cuadrados con lo que se espera que resalte una transformación afín. Esto esta mal porque ignora outliers e ignora que existan más de dos líneas a las que pertenezcan los puntos. En suma lo que anteriormente se realiza con RANSAC o HOUGH es construir un filtro que atrapa outliers. Dejando pasar un conjunto de DMatches lo suficientemente grande para definir un modelo o una transformación afín que sea aceptable de explicar los DMatches que la generaron (siendo mayor a un umbral).

En fin, nuevamente pensando en fuerza bruta, se podría decir que cada DMatch define una transformación afín, usando la formula:

$$\begin{pmatrix} x_{REF} & y_{REF} & 0 & 0 & 1 & 0 \\ 0 & 0 & x_{REF} & y_{REF} & 0 & 1 \end{pmatrix} (m_{XX} \ m_{XY} \ m_{YX} \ m_{YY} \ t_X \ t_Y)^T = \begin{pmatrix} x_{PRU} \\ y_{PRU} \end{pmatrix}$$

Pero esto estaría erróneo porque no considera que cada DMatch, por ser una muestra, tiene asociado un ruido. Entonces, para lidiar con el ruido usamos mínimos cuadrados sobre este conjunto filtrado de DMatches. Entonces, para calcular la transformación afín se 'aumenta' la ecuación anterior quedando de esta forma:

$$\begin{pmatrix} x_{REF}^0 & y_{REF}^0 & 0 & 0 & 1 & 0 \\ 0 & 0 & x_{REF}^0 & y_{REF}^0 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{REF}^n & y_{REF}^n & 0 & 0 & 1 & 0 \\ 0 & 0 & x_{REF}^n & y_{REF}^n & 0 & 1 \end{pmatrix} (m_{XX} \ m_{XY} \ m_{YX} \ m_{YY} \ t_X \ t_Y)^T = \begin{pmatrix} x_{PRU}^0 \\ y_{PRU}^0 \\ \vdots \\ x_{PRU}^n \\ y_{PRU}^n \end{pmatrix}$$

Entonces, el sistema se resuelve como cualquier sistema de mínimos cuadrados. Resultando el vector x que contiene los parámetros que definen la transformación afín. Es decir, que definen un modelo que mejor explica los DMatches filtrados a pesar de que estos posean ruido.

$$Ax = b \quad (10)$$

$$x_{sol} = (A^T A)^{-1} A^T b \quad (11)$$

### III. DESARROLLO

De aquí en adelante la imagen de REFERENCIA es la imagen input 1 y la imagen de PRUEBA es la imagen input 2. Por consiguiente, se expande a que keypoints1 son los puntos de interés de la imagen de REFERENCIA y keypoints2 son los puntos de interés de la imagen de PRUEBA.

El script implementado procesa los 4 pares de imágenes cuando es ejecutado. Esto lo hace con un ciclo for de 4 iteraciones. En cada una de ellas procesa un par de imágenes. El llamado a cada par de imágenes se implementa guardando el nombre de las imágenes en un arreglo de *strings*. Luego, la dirección de la url de la imagen se concatena usando el signo + dentro del método *imread()*.

#### III-A. Detección y gráfico de puntos de interés SIFT

Los keypoints SIFT son de tipo Blop por lo que teóricamente no sólo se espera encontrar keypoints en esquinas. Si no que también, se espera que se detecten en píxeles que son máximos (o mínimos) locales basados en una operación matemática que se aplica sobre la imagen. En las siguientes imágenes se observan 500 keypoints detectados en los cuatro pares de imágenes.

El código ya venía implementado explicándolo brevemente. Se crean dos arreglos dinámicos de objetos KeyPoint llamados keypoints1 y keypoints2 cuya función será guardar los keypoints detectados. Se crean dos objetos Mat llamados descriptors1 y descriptors2 cuya función será guardar descriptores calculados. Luego, se crea un objeto de la clase DescriptorExtractor llamado descriptorExtractor que es el objeto encargado de analizar la imagen y detectar los puntos de interés y calcula los descriptores SIFT mediante el método *detectAndCompute*. Finalmente, se dibujan los puntos de interés con la función *drawKeypoints*. Dicha función recibe la imagen original el arreglo con los keypoints, una referencia a una imagen de output que sera la imagen donde estén dibujados los keypoints.

```

1 vector<KeyPoint> keypoints1;
2 vector<KeyPoint> keypoints2;
3 Mat descriptors1, descriptors2;
4 Ptr<DescriptorExtractor> descriptorExtractor = xfeatures2d
5   ::SIFT::create(500);
6 descriptorExtractor->detectAndCompute(
7   input1, Mat(), keypoints1, descriptors1);
8 descriptorExtractor->detectAndCompute(
9   input2, Mat(), keypoints2, descriptors2);
10 drawKeypoints(input, keypoints1, output_original);
11 drawKeypoints(input, keypoints2, output_original);

```

En las siguientes imágenes se observa que se detectan puntos de interés tanto en esquinas como en zonas que a simple vista son planas, pero son máximos locales bajo una operación matemática sobre la imagen. Lo que



Figura 1. 500 DoG Keypoints detectados en imágenes 006



Figura 2. 500 DoG Keypoints detectados en imágenes 084

coincide con lo esperado teóricamente.

Por ejemplo, en la figura 1 (RoboCup) y 3 (RoboCUP) se observan puntos de interés en las paredes y en las zonas más brillantes por luz ambiental. En la figura 2 y 4 se observan puntos de interés en esquinas y además en puntos de borde. Todo eso coincide con lo esperado por ser DoG un detector de puntos de interés de tipo Blop.

#### III-B. Generación y gráfico de calces

La función que genera estos matches viene implementada en el código. Explicándola brevemente lo que hace es hacer un match entre los keypoints que tengan menor distancia euclidiana pero de manera simétrica, es decir, los que produzcan una biyección. En el código, esta función primero recorre los kp de la imagen 2 y hace matches entre esos kp y los de la imagen 1, entonces, puede darse el caso de que un kp2 de la imagen 2 haga match con más de un kp de la imagen 1. Luego recorre los kp de la imagen 1 y los match que sobreviven son solo los que resultan en que un y solo un kp de ambas imágenes están pareados, es decir, una biyección.

Entonces, teóricamente lo esperado es ver que de un kp que tenga un kp pareado salga una y solo una linea. Se espera que se vean kp sin líneas, los cuales son los kp que no tienen match 1 a 1.

Los resultados muestran que para las figuras 5, 6, 7 y 8, tiene graficados 98, 179, 222 y 77 matches respectivamente. En ellas se puede observar que los resultados coinciden con lo teórico encontrando kp sin match y para los kp que tienen match sale sólo una linea de ellos. Viendo la imagen en el contexto global se observa que muchos match son incorrectos, lo cual es esperado teóricamente esto se explica porque localmente los descriptores son parecidos. Se observa



Figura 3. 500 DoG Keypoints detectados en imágenes 098

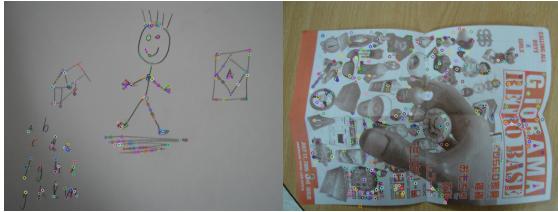


Figura 4. 500 DoG Keypoints detectados en imágenes 101

que el par de imágenes 8 tiene muy pocos matches, lo cual es esperable ¡porque son imágenes totalmente diferentes!.

### III-C. Generación Transformación de Semejanza a partir de un calce

La función que genera la transformación de semejanza a partir de un calce se llama *genTransform*. A ella ingresan un objeto DMatch con **un** match, entran un arreglo con el KeyPoint de la imagen REF y otro con el KeyPoint de la imagen PRU e ingresan los parámetros que definen una transformación de semejanza, que son: *e* (factor de escala entre las dos imágenes),  $\theta$  (la rotación),  $t_x$  y  $t_y$  (traslación relativa entre las dos imágenes).

La salida de esta función es vacía. Esta función sólo modifica los parámetros *e*,  $\theta$ ,  $t_x$  y  $t_y$ . Es por esto que cuando se define la función se les escribe con un signo & al lado. Lo que se observa en el siguiente código.

```
1 void genTransform(..., double &e, double &theta,
2                  double &tx, double &ty){...}
```

Se menciona que en OpenCV la orientación de los KeyPoints está en grados. Por lo que se DEBE manejar el 'overflow/underflow'. Para entenderlo, se busca la documentación de OpenCV de sus KeyPoints[1] que dice que la orientación son *float* en el intervalo [0,360) medido en el sentido horario. Entonces, el problema a solucionar se produce cuando la resta arroja un *float* negativo, es decir, un ángulo negativo.

Si la resta es positiva se retorna el mismo resultado de la resta. Si la resta es negativa se calcula 360-la resta y se retorna eso. Luego, se pasa el ángulo de grado a radian, porque las funciones trigonométricas de OpenCV reciben radianes.

```
1 if(theta < double(0.0))
2 {
3     theta = 360+theta;
```

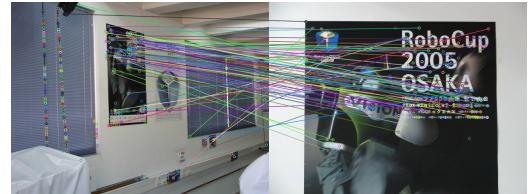


Figura 5. Calces filtrados por symmetrymatcher en imágenes 006



Figura 6. Calces filtrados por symmetrymatcher en imágenes 084

```
4     }
5     theta = theta*M_PI/double(180.0); //pasar grado a radian.
```

Entonces, lo importante de esta sección es explicar como llamar a una coordenada de un objeto Keypoint. Bien como se explico en la tarea anterior, un objeto DMatch tiene 4 parámetros: queryIdx, trainIdx, ImgIdx y distance. Para esta aplicación lo importante es que trainIdx es el índice del Keypoint que hizo el DMatch dentro del arreglo de Keypoints llamado Keypoints1. queryIdx es el índice del Keypoint que hizo el DMatch dentro del arreglo de Keypoints llamado Keypoints2.

Nos llamamos al keypoint de la imagen de REFERENCIA usando: *keypoints1[match.trainIdx]* y al Keypoint de la imagen de PRUEBA con *keypoints2[match.queryIdx]*. El objeto Keypoint tiene varios atributos, para referirnos a las coordenadas del punto se usa: Keypoint.pt.x para la abscisa y keypoint.pt.y para la ordenada del punto. Lo importante es mencionar que teóricamente se esperaría que fueran enteros int, pero no, OpenCV las maneja como float, por eso se definen como se ve en el código a continuación. Finalmente, se calcula  $t_x$  y  $t_y$  usando las formulas expuestas en el marco teórico, lo destacable es recordar que sin y cos reciben radianes.

```
1 float x_ref, y_ref, x_pru, y_pru;
2 x_ref = keypoints1[match.trainIdx].pt.x;
3 y_ref = keypoints1[match.trainIdx].pt.y;
4 x_pru = keypoints2[match.queryIdx].pt.x;
5 y_pru = keypoints2[match.queryIdx].pt.y;
6
7 tx = x_pru - e*(x_ref*cos(theta)) - y_ref*sin(theta));
8 ty = y_pru - e*(x_ref*sin(theta)) + y_ref*cos(theta));
```

### III-D. Calculo consenso para RANSAC

La función *computeConsensus* recibe objetos y los modifica, al igual que la función anterior mediante el uso de & al igual que *genTransform*. Recibe los mismos objetos que *genTransform* sólo que en vez de recibir un DMatch recibe un arreglo de DMatches llamado

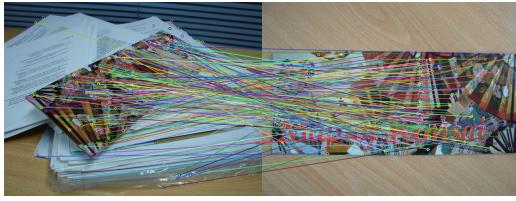


Figura 7. Calces filtrados por symmetrymatcher en imágenes 098

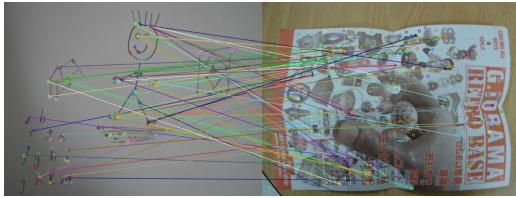


Figura 8. Calces filtrados por symmetrymatcher en imágenes 101

Matches proveniente de la ejecución de la función `symmetrymatcher()`; además, recibe un arreglo de `int` llamado `selected` estos son los índices de los DMatches (del arreglo `Matches`) que votaron por la transformación de semejanza que resultó con más consenso.

```

1 int computeConsensus(vector<DMatch> &matches, ...
2                      vector<int> &selected, ...
3                      double umbralpos, ...)
4 }
```

Entonces, en el fondo a esta función ingresa un DMatch representado por una transformación de semejanza en los parámetros  $e$ ,  $\theta$ ,  $t_x$ ,  $t_y$ . Calcular el consenso es recorrer todos los DMatches e ir calculando el error de proyección de cada DMatch recorrido, esto significa que a menor error de proyección es más probable que la transformación de semejanza haya generado el match, es por eso que se están buscando los DMatch con menor error de proyección. La formula usada para calcular el error de proyección es la expuesta en el marco teórico.

En el código primero se parte definiendo un contador para el consenso de la transformación de semejanza. El consenso es la cantidad de DMatches cuyo error de proyección es inferior a un umbral ingresado como hiperparametro a la función RANSAC y que en este código de llama `umbralpos`. Luego, se vacía la lista que guarda los DMatches seleccionados previamente, esto se hace porque la función `computeConsensus` se llama tantas veces como iteraciones se configuren para RANSAC.

Luego, se definen las coordenadas del los Keypoints REF y PRU. La función de error se implementa por partes para facilitar la comprensión del código. En el marco teórico se observa que la formula es un modulo de un vector, es decir la norma  $L_2$ , lo que se implementa con `sqrt()` y con `pow(double, 2)` que eleva al cuadrado el double del primer argumento.

```

1 int cons = 0;
2 selected.clear();
3 for (int i=0; i<matches.size(); i++)
4 {
5     float x_ref, y_ref, x_pru, y_pru;
6     x_ref = keypoints1[matches[i].trainIdx].pt.x;
7
8     double error, error_x, error_y;
9     error_x= e*(cos(theta)*x_ref - sin(theta)*y_ref)+tx-x_pru;
10    error_y= e*(sin(theta)*x_ref + cos(theta)*y_ref)+ty-y_pru;
11    error = sqrt(pow(error_x,2)+pow(error_y,2));
12
13    if(error < umbralpos)
14    {
15        cons += 1;
16        selected.push_back(i);
17    }
18 }
19 return cons;
```

En el bloque `if` se implementa el umbral de error máximo para aceptar que un DMatch fue generado por la transformación de semejanza que estamos evaluando. Considerando que la selección de los DMatch proviene de Keypoints, que provienen de una imagen digital sacada por una cámara que es un sensor que agrega ruido. Lo importante es preguntarse **¿que significa este umbral?**. Si este numero real positivo (implementado como double) aumenta quiere decir que se es menos exigente con los DMatchs, es decir, se aceptan más DMatchs que fueron generados por la transformación de semejanza, se podría decir que se acepta más ruido. Por el contrario, si se disminuye este umbral significa que se es más exigente para aceptar DMatchs, lo que significa que al final igualmente se va a encontrar una transformación afín (aunque igual existe el caso en que no se encuentre); lo importante, es que se disminuye la capacidad de generalizar de la transformada final que se obtenga, es decir, puede que al sacar otro par de fotos sobre los mismos objetos con incluso la misma cámara sujeta a ruido, es más probable que no se pueda proyectar una imagen sobre la otra usando la transformación afín encontrada como objetivo de esta metodología.

Finalmente, en el código, si el error es menor al umbral, se aumenta en uno al contador de consenso y se ingresa el índice del DMatch que tienen ese error aceptable (índice en el vector `matches`) usando el método `array.push_back(int i)`.

### III-E. Aplicación RANSAC con SIFT

La función que realiza RANSAC recibe los arreglos de Keypoints, DMatch y un arreglo con los DMatch que votaron por la transformación con mayor consenso. Esta función retorna un booleano, sin embargo, lo importante es que modifica el arreglo `accepted` de dichos los DMatch.

La implementación primero parte declarando los hiperparametros, variables y arreglos que se van a usar. La parte más importante del código es el ciclo `for` que se repite tantas veces como numero de iteraciones se configure para ransac con el hiperparametro `ntrials`. Entonces, se toma un índice al azar del arreglo `matches`

con la función rand(). rand() genera un numero aleatorio entre 0 y un máximo definido internamente en cpp. Al aplicarle el operador modulo (%) por el largo del arreglo matches. Se obtienen un numero al azar entre 0 y el largo de dicho arreglo.

Luego, se general la transformación de semejanza de ese Dmatch elegido al azar llamando a la función *genTransform*. Se calcula el consenso de ese Dmatch usando la función *computeConsensus*. Luego, en el arreglo bestSelected se guarda el consenso de ese Dmatch y en el arreglo indices\_probados, se guarda el índice del DMatch probado.

```

1 for(int i=0; i< ntrials; i++)
2 {
3     int idx_hip = rand()%matches.size();
4     genTransform(matches[idx_hip],kp1,kp2,e,theta,tx,ty);
5     int cantidad_concenso=computeConsensus(matches..umbralpos);
6     bestSelected.push_back(cantidad_concenso);
7     indices_probados.push_back(idx_hip);
8 }
```

Una vez probados todos los intentos, es destacable que nada asegura que se haya calculado el consenso de todos los DMatch. En fin, con otro ciclo for se recorre el arreglo bestSelected para determinar el índice del DMatch con más consenso (se guarda en la variable *indice\_matches\_mejor\_concenso*) y saber cual es el mejor consenso (se guarda en la variable *mejor\_concenso*).

La segunda parte importante del código es el bloque<sup>11</sup> if con que se comprueba si la cantidad de votos es suficiente para aceptar la hipótesis de modelo. Si la variable *mejor\_concenso* es mayor que la variable *umbralcons* se acepta el modelo. Se vuelve a generar la T. semejanza usando el DMatch con el índice de la variable *indice\_matches\_mejor\_concenso*. Se vuelve a llamar a la función *computeConsensus* para actualizar el arreglo selected a los Dmatch que votaron por la T de sem.

Lo importante del último for es que selected contiene los índices (dentro del array matches) de los DMatch que votaron por la T de sem. Entonces, este último for arma el arreglo accepted copiando los Dmatch desde el arreglo matches usando los índices del arreglo selected. Y así finalmente termina el código de RANSAC.

```

1 for(int k=0; k<selected.size(); k++)
2 {
3     accepted.push_back(matches[selected[k]])
4 }
```

### III-F. Aplicación Transformada Hough con SIFT

La transformada de Hough se implementa con la función *hough4d*. La función recibe lo mismo que lo expuesto para RANSAC. También retorna un booleano, que se presume en principio la tarea estaba diseñada para que si ese booleano era False significaba que ni RANSAC ni

Hough habían encontrado un conjunto DMatch aceptable. Lo que se implementa en con el primer bloque if en la función *calcAfin*. En fin, lo que hace *hough4d* es modificar el arreglo accepted.

Lo primero que hacer *hough4d* es crear la tabla acumuladora de los votos usando el objeto *SparceMat*, primero se crea un arreglo que define las dimensiones de esta arreglo de 4-dimensional. Lo importante es que al escribir:

```

1 int hsize[]={1000, 1000, 1000, 1000};
2 SparseMat sm(4, hsize, CV_32F);
```

Genera un objeto *SparceMat* de 4 dimensiones en que en cada dimensión tiene 1.000 casillas.

El ciclo for principal de *hough4d* va recorriendo cada DMatch, para cada Dmatch le genera su transformación de semejanza y usándola calcula los índices i, j, k, z que son los que definen en que casilla registrar el voto.

```

1 for(int m=0; m <matches.size(); m++)
2 {
3     genTransform(matches[m],kp1,kp2,e,theta,tx,ty);
4     int i,j,k,z;
5     i=floor((tx/dxBin)+0.5)+500;
6     j=floor((ty/dxBin)+0.5)+500;
7     k=floor((theta/dangBin)+0.5)+500;
8     z=floor((log(e)/log(2.0))+0.5)+500;
9
10 }
```

El voto se registra creando una arreglo con las coordenadas de la casilla. Luego, se llama a la casilla con *sm.ref<float>(idx)* y con ++ se aumenta en 1 su valor.

```

1 int idx[4];
2 idx[0] = i;
3 idx[1] = j;
4 idx[2] = k;
5 idx[3] = z;
6 sm.ref<float>(idx)++;
```

En la segunda etapa se determina cual es la casilla de la *SparceMat* llamada *sm* que tiene más votos con la función *minMaxLoc()*, que la recorre y retorna el máximo valor de votos y un array que representa las coordenadas o los índices de la casilla con más votos.

```

1 double minvotos, maxvotos;
2 int sm_maxvotos_Idx[4], sm_minvotos_Idx[4];
3 cv::minMaxLoc(sm, &minvotos, &maxvotos,
4 ...sm_minvotos_Idx, sm_maxvotos_Idx);
```

Por último, se pasa a la etapa en que se prueba si la cantidad de votos es aceptable con un bloque if al igual que el método anterior. Luego el for interno, vuelve a recorrer todos los DMatch calculando sus índices i, j, k, z y determinando si votaron por la casilla de más votos usando un if con 4 comparaciones (que coincidan los 4 índices). Finalmente, los que pasan este ultimo if son

agregados al array accepted.

```

1 if(maxvotos > umbralvotos)
2 {
3     int i_accepted = sm_maxvotos_Idx[0];
4     ...
5     int z_accepted = sm_maxvotos_Idx[3];
6     for(int j=0; j<matches.size(); j++)
7     {
8         genTransform(matches[j],kp1,kp2, e, theta, tx, ty);
9         int i_candidate = floor((tx/dxBin)+0.5)+500;
10        ...
11        int z_candidate = floor((log(e)/log(2.0))+0.5)+500;
12        if(i_candidate == i_accepted &&
13           j_candidate == j_accepted &&
14           k_candidate == k_accepted &&
15           z_candidate == z_accepted)
16        {
17            accepted.push_back(matches[j]);
18        }
19    }
20 }
```

### III-G. Calculo de Transformación afín con mínimos cuadrados

La función que calcula la transformación afín usando mínimos cuadrados recibe un arreglo con los DMatch provenientes de RANSAC o Hough y recibe los dos arreglos con los Keypoints SIFT detectados en las imágenes. La función retorna un objeto Mat llamado *transf* que contiene los parámetros que definen la transformación afín.

Las partes importantes del código son: El bloque *if* que captura el caso en que el método usado (RANSAC o Hough) no logran llegar a un resultado. Entonces, retornan un arreglo de DMatch vacío. El bloque *if* captura esto preguntando si el largo del arreglo de DMatch es cero. Si lo es, se imprime un mensaje en pantalla y se retorna el mat de *transf* lleno de ceros. Entonces, al dibujarlo, las coordenadas resultantes dan todas cero y no se dibujan líneas en la imagen de referencia, pero igual se guarda la misma imagen de referencia sin modificar.

```

1 if(matches.size()==0)
2 {
3     cout<<"Ni RANSAC ni Hough encuentran un conjunto matches
4     ...realmente correctos para definir una t. afín entre
5     ...las 2 imágenes"<<endl;
6     return transf;
7 }
```

Esta función *calcAfin* en resumen es el proceso de *Aumentación*, es decir, es un ciclo *for* que va tomando los puntos de los DMatch que lograron consenso y los va metiendo en las matrices A y b que se crean con las dimensiones justas con base a dicha cantidad de DMatch.

Es innecesario detallar como se llaman las coordenadas *x\_ref*, *y\_ref*, *x\_pru* y *y\_pru* porque ya se explico en las secciones de *genTransform* y *computeConsensus*. Lo destacable es que al llamar a los coeficientes de un *Mat* como por ejemplo: *A.at(2\*i+1,2)*, OpenCV es flexible y permite operaciones algebraicas que resulten en un *int*.

Finalmente, el problema de mínimos cuadrados se resuelve con la formula explicada en el marco teórico.

Resultando *x* como el vector de 6 x 1 que contiene los coeficientes *m<sub>XX</sub>*, *m<sub>XY</sub>*, *m<sub>YX</sub>*, *m<sub>YY</sub>*, *t<sub>X</sub>*, *t<sub>Y</sub>*. Es destacable que *m<sub>XY</sub>* y *m<sub>YX</sub>* no tienen porque ser iguales. Se asignan estos coeficientes a la matriz *transf* en un orden específico que facilita el entendimiento del código al implementar la proyección de puntos entre imágenes en la función *drawProjAfin*. Por último se retorna la Mat *transf*.

```

Mat A = Mat::zeros(2*matches.size(), 6, CV_32FC1);
Mat x = Mat::zeros(6, 1, CV_32FC1);
Mat B = Mat::zeros(2*matches.size(), 1, CV_32FC1);
for (int i = 0; i < matches.size(); i++)
{
    x_ref = keypoints1[matches[i].trainIdx].pt.x;
    ...
    y_pru = keypoints2[matches[i].queryIdx].pt.y;
    A.at<float>(2*i,0) = x_ref;
    A.at<float>(2*i,1) = y_ref;
    ...
    A.at<float>(2*i+1,5) = 1.0;
    B.at<float>(2*i,0) = x_pru;
    B.at<float>(2*i,1) = y_pru;
}
x = ((A.t() * A).inv()) * A.t() * B;
transf.at<float>(0,0) = x.at<float>(0,0);
...
transf.at<float>(1,2) = x.at<float>(5,0);
```

### III-H. Graficando romboídes a partir de Transformación afín

La función que se encarga de graficar el romboide que representa la proyección de la imagen de prueba dentro de la imagen de referencia se llama *drawProjAfin*. Esta función tiene como entrada la imagen de referencia y la imagen de prueba en objetos *Mat*; además, le entra un objeto *Mat* que es el que contiene los coeficientes que definen la transformación afín calculada en la sección anterior. Como salida tiene un objeto *Mat* de las dimensiones de la imagen de referencia, la salida es la imagen de referencia con el romboide dibujado.

En el código la imagen de referencia es el *Mat input1* y la imagen de prueba es el *Mat input2*. Lo primero que se hace es definir 4 objetos *Point* que representan las esquinas de la imagen de referencia. La notación es contra intuitiva, por ejemplo, en *Point id\_pru(input2.cols, 0.0)*. Lo importante es que la abscisa es la cantidad de columnas y la ordenada es la cantidad de filas. Para definir los nombres de los puntos se usa *s* para parte superior, *i* para inferior, *d* para el lado derecho y *l* para el izquierdo de la imagen.

```

1 Point si_pru(0.0, input2.rows);
2 Point sd_pru(input2.cols, input2.rows);
3 Point id_pru(input2.cols, 0.0);
4 Point ii_pru(0.0, 0.0);
```

Luego, el código proyecta estas esquinas de la imagen de prueba usando la transformación afín con la siguiente formula:

$$\begin{pmatrix} x_{PRU} \\ y_{PRU} \end{pmatrix} = \begin{pmatrix} m_{XX} & m_{XY} \\ m_{YX} & m_{YY} \end{pmatrix} \begin{pmatrix} x_{REF} \\ y_{REF} \end{pmatrix} + \begin{pmatrix} t_X \\ t_Y \end{pmatrix}$$

La cual se implementa en 8 líneas con el siguiente código. Lo destacable es el llamado a las componentes del objeto matriz *Mat* usando *transf.at<float>(1,0)* y que se llama a las coordenadas de los puntos usando *Point.x* y *Point.y*.

```

1 si_x_ref=(transf.at<float>(0,0)*si_pru.x) +
2   (transf.at<float>(0,1)*si_pru.y)+transf.at<float>(0,2);
3 ii_y_ref=(transf.at<float>(1,0)*ii_pru.x) +
4   (transf.at<float>(1,1)*ii_pru.y)+transf.at<float>(1,2);
5

```

Por último, se arman nuevos objetos *Point* que son los que se grafican en la imagen de referencia. Para dibujar el romboide sé utilizar 4 veces la función *line()* [2] que dibuja una línea entre dos objetos *Point*. Esta función recibe la imagen en donde graficar los puntos, llamada *output* en el código, los dos puntos en cuestión, el color de la línea mediante un objeto *Scale(r,g,b)*, y otros 3 parámetros que definen el grosor, si es línea segmentada o continua, y el número de bits fraccionarios en las coordenadas del punto. Finalmente, se retorna la imagen *output* con las líneas ya dibujadas.

```

1 Point si_ref(si_x_ref, si_y_ref);
2 Point ii_ref(ii_x_ref, ii_y_ref);
3 line(output, si_ref, sd_ref, Scalar(0, 255,0),1,8,0);
4 ii_line(output, ii_ref, si_ref, Scalar(0, 255,0),1,8,0);
5 return output;
6
7
8

```

### III-I. Pruebas de RANSAC

En las siguientes 4 imágenes se grafican calces correspondientes los DMatch que resultan de RANSAC. Lo esperado es graficar muchos menos calces que los que salen de la función *symmetrymatcher* lo que se comprueba visualmente. Lo importantes, es que se comprueba que los calces son 1 a 1 y que en la generalidad de la imagen son mucho más correctos además de serlo localmente.



Figura 9. Calces RANSAC imágenes RoboCup

Teóricamente, en par de imágenes distintas no se debería graficar ningún Match, lo que se comprueba en la siguiente imagen no hay matches dibujados

A continuación de presentan los romboídes que representan a la imagen de prueba proyecta en la imagen de referencia usando la transformación afín. Teóricamente, RANSAC debería tener problemas en los



Figura 10. Calces RANSAC imágenes páginas amarillas



Figura 11. Calces RANSAC imágenes map of Kansai

pares de imágenes que hay pocos inliers, es decir, pocos DMatch aceptados.

En la imagen 13 del afiche de la RoboCup y en la imagen 14 map de Kansai. La imagen de prueba estaba contenida en la imagen de referencia, por lo que se tenían varios inliers, por lo que se llega a un buen resultado.

En la imagen 15 de las páginas amarillas habían pocos inliers, porque la imagen de prueba no estaba totalmente contenida en la de referencia. Entonces, aquí RANSAC igual logra un resultado, mas no tan ajustado como los casos anteriores.

Por último en las imágenes totalmente diferentes, teóricamente, no se debería encontrar una transformación, y lo que se confirma porque no se logra encontrar y resulta la imagen tal como entro.

### III-J. Variación de hiperparámetros de RANSAC

Al variar el hiperparámetro de *ntrials = 10000*, es decir la cantidad de hipótesis de modelo que prueba RANSAC es importante mencionar que se puede calcular la probabilidad de que RANSAC llegue a un modelo aceptado en base a la cantidad de DMatch que inicialmente ingresa. Anteriormente se mencionó que de la función *symmetrymatcher* salen 98, 179, 222 y 77 matches respectivamente. Bajo los mismo otros hiperparámetros de error se obtiene que seteando el *ntrials* en el doble de Dmatches que entran hay una posibilidad superior al 0.99. Esto significa que teóricamente lo esperado es que si aumentamos los intentos de RANSAC A 20000 siempre obtendremos un resultado e incluso se los disminuimos a su 10% es decir, 1000 también tendremos posibilidades

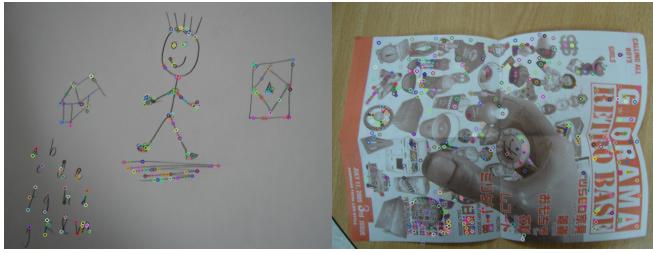


Figura 12. Calces RANSAC imágenes totalmente diferentes



Figura 13. Proyección RANSAC imágenes Robocop

altísimas de obtener un modelo.

Lo anterior se comprueba al variar el hiperparámetro de  $ntrials$ , a los valores mencionados obteniéndose siempre un modelo que lograba una proyección aceptable de imágenes como resultado final.

Al aumentar el umbral de consenso al doble sucede que para las imágenes de las páginas amarillas y del mapa de kansai no se llega a un modelo. Esto se explica porque porque hay menos Dmatch que selecciona RANSAC y nos estamos poniendo más exigentes con el conjunto que aceptamos como generado por la transformada de semejanza.

### III-K. Pruebas de Transformada de Hough

En las siguientes 4 imágenes se grafican calces correspondientes los DMatch que resultan de Hough. Lo esperado es graficar muchos menos calces que los que salen de la función *symmetrymatcher* lo que se comprueba visualmente. Lo importantes, es que se comprueba que los calces son 1 a 1 y que en la generalidad de la imagen son mucho más correctos además de serlo localmente.

Ambos métodos se enfrentan a un umbral de la cantidad de votos para aceptar la hipótesis de modelo. En RANSAC el umbral es  $umbralcons = 30$  y en Hough es de



Figura 14. Proyección RANSAC imágenes maps of Kansay



Figura 15. Proyección RANSAC imágenes páginas amarillas

$umbralvotos = 5$ . Así que lo que se espera para que Hough acepte una hipótesis de modelo pasan menos DMatch, lo que significa que se verán menos líneas graficadas.

Esto se observa en las figuras 17, 18 y 19. Se confirma además que Hough es consistente porque no se observan calces en la figura 19 de las imágenes diferentes, donde en teoría no deberían observarse.

A continuación de presentan las imágenes de referencia con romboide. Teóricamente, Hough no debería tener problemas en los pares de imágenes que hay pocos inliers, por ejemplo, las imágenes donde la de prueba no está contenida en la de referencia.

La dificultad de Hough radica en definir correctamente los la forma en que se dicretizan los parámetros, es decir, los intervalos con los que se arma la casilla de votación. Pero como venía dado del enunciado, gracias a una investigación previa. Esta no fue un problema y se espera



Figura 16. Proyección RANSAC imágenes totalmente diferentes



Figura 18. Calces Hough imágenes páginas amarillas



Figura 19. Calces Hough imágenes map of Kansai.

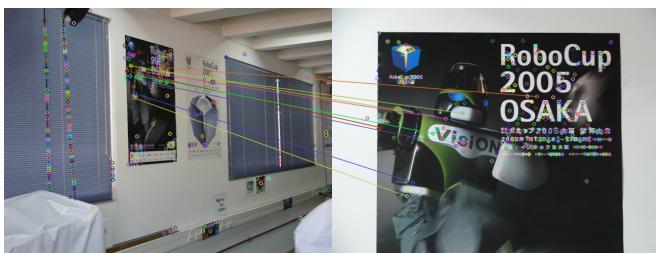


Figura 17. Calces Hough imágenes RoboCup.

que Hough obtenga proyecciones aceptables.

En la imagen 21 del afiche de la RoboCup se obtiene una proyección mucho menos ajustada que RANSAC esto se explica porque fue calculada con muchos menos DMatch (8 en el caso de Hough). A diferencia de la de RANSAC que fue calculado con como mínimo 30 DMatch

En la imagen 22 map de kansai y en la imagen 15 de las páginas amarillas, en ambas habían pocos DMatch Sin embargo, Hough logra llegar a un modelo bastante aceptable.

Por último en las imágenes totalmente diferentes, teóricamente, no se debería encontrar una transformación, y lo que se confirma porque no se logra encontrar y resulta la imagen tal como entró.

### III-L. Variación de hiperparámetros de Hough

Al aumentar la mínima cantidad de votos para aceptar el modelo (*umbralvotos* = 5) al doble, es decir, a 10. Sucede que el Hough no llega a un modelo aceptable. Porque para ningún par de imágenes se obtiene un Dmatch con más de 10 votos. Al disminuirlo a 2 votos en las 3 primeras imágenes debería siempre llegar a una transformación buena o mala. Sorprendentemente,

se llegan a transformaciones bastante parecidas a las óptimas solo que mas holgadas. Finalmente si se pone en 0 llega a proyecciones que no tienen sentido, ni siquiera son romboídes.

Al achicar los hiperparámetros que definen los intervalos en la tabla acumuladora de votos lo que pasa es que nunca se llegan a modelos (bajo umbral de votos = 5) porque los votos se reparten entre muchas casillas así que ninguna supera el umbral. Por el contrario, al agrandarlos, siempre se llega a un modelo cuyos romboídes son menos ajustados.

## IV. CONCLUSIÓN

Finalizada la experiencia se logra verificar que los resultados obtenidos son los que se esperan en función de lo que indica la teoría con respecto a que aplicando las funciones implementadas, que funcionando correctamente, se logra llegar a una transformación afín que gráficamente demuestra ser correcta al verse bien proyectadas las imágenes. Lo importante es que los resultados son consistentes, es decir, para imágenes del afiche de robocop, las páginas amarillas y el afiche de "map of kansai" lo esperado era que el método funcionara bien y funcione bien. Para el par de imágenes 101 del afiche de "giorama retro base" y el dibujo de un monito a palitos, lo esperado era que el método no encontrara una proyección y el método fue consistente en no encontrarla; porque ni RANSAC ni Hough logran encontrar un conjunto de matches lo suficientemente grande y que fuera un candidato aceptable para generar una transformación afín. Lo destacable es que se comprueba que el método implementado es robusto traslaciones, escalamientos y rotaciones en los planos paralelo y perpendicular a las imágenes.



Figura 20. Calces Hough imágenes totalmente diferentes.



Figura 21. Proyección Hough imágenes Robocop

Lo realizado en esta tarea fue calcular los puntos de interés DoG y sus los descriptores SIFT. Se calculan los calces iniciales minimizando la distancia entre descriptores. Se implementa la función que calcula la transformación de semejanza a partir de un calce. La función que calcula el consenso de dicha transformación, es decir, la cantidad de calces cuyo error de proyección es menor a un umbral. Se programan las funciones que realizan RANSAC y Hough. La función que calcula una transformación afín por mínimos cuadrados con base en los DMatches filtrados por RANSAC y Hough; y la función que dibuja un romboide que representa la imagen de prueba proyectada en la imagen de referencia. Todas las imágenes generadas se guardan automáticamente. Se varían los hiperparámetros de los métodos y se presentan y analizan esos resultados. Se programa un script que ejecuta automáticamente todas estas funciones en todas las imágenes entregadas. Llegando finalmente a un script que se ejecuta una sola vez y genera todas las imágenes requeridas por esta tarea.

Sobre indicar cual método RANSAC u Hough se comporta mejor en las imágenes, esto depende de la imagen. Como se menciona anteriormente, ambos métodos funcionaron en las tres imágenes que tenían que funcionar y no funcionaron en el par de imágenes donde no tenían que funcionar. Sobre cual funciona mejor



Figura 22. Proyección Hough imágenes maps of kansay



Figura 23. Proyección Hough imágenes páginas amarillas

depende del par de imágenes, más específicamente de la cantidad de outliers, que se expresa intuitivamente como que tan diferente son cada par de imágenes o si una está enteramente contenida en la otra.

Para las imágenes del afiche de la robocup y del afiche del mapa de kansai, el afiche está enteramente contenido en la imagen de la pared, así que hay menos outliers comparado con las otras imágenes; entonces, en este caso RANSAC funciona mejor, obteniendo una proyección más ajustada. Para la imagen de las páginas amarillas, una imagen no estaba totalmente contenida en la otra y se tenían una cantidad más bien reducida de outliers. En este caso Hough funcionó mucho mejor logrando un rombo más ajustado. Se comprueba entonces, que en general RANSAC presenta problemas cuando hay una reducida cantidad de inliers y Hough es más robusto a outliers pero funciona más lento.



Figura 24. Proyección Hough imágenes totalmente diferentes

Los aprendizajes obtenidos al realizar esta tarea son profundizar la programación en C++ y el trabajo con la distribución de Linux Ubuntu. Lo más importante es que se aprendió a trabajar con objetos para labores específicas para estas técnicas como *DMatch*, *KeyPoint*, *SparseMat*. También, se aumentan los conocimientos en temas operativos muy útiles como concatenar strings, guardar imágenes e importar librerías desde archivos como la que calcula los descriptores SIFT. Finalmente, se aprende a escribir scripts de rápida ejecución en la maquina por ser de C++ y que procesan todas las imágenes de una carpeta fuente sin necesidad de compilar/correr el script varias veces.

Las dificultades que se encontraron durante el desarrollo de la experiencia fue que se debía aprender C++ que es un lenguaje más alejado del usuario, es decir, los códigos son menos intuitivos. La principal dificultad fue el hacer referencia a atributos de distintos objetos de clases *DMatch*, *KeyPoint*, *SparseMat* para asignarles un valor y luego printearlos para saber si se habían asignado correctamente era una dificultad porque la consola imprimir la dirección de memoria ram en que se guardaba el objeto. La programación y el debug se hacen más complejos porque se compila y debugea desde la consola con lo que fue difícil identificar los errores. También, se presentaron dificultades con invocar la librería para SIFT desde archivos. Finalmente, las dificultades se solucionaron preguntando a los compañeros de clase, investigando en la ficha de cada clase en la documentación de OpenCV y leyendo dudas parecidas en Stackoverflow.

## REFERENCIAS

- [1] OpenCV Documentation KeyPoint Angle, disponible:[https://docs.opencv.org/3.4/d2/d29/classcv\\_1\\_1Key.html](https://docs.opencv.org/3.4/d2/d29/classcv_1_1Key.html)

*Point.html#a4484e94502486930e94e7391adf9d215*, version september 2, 2019.

- [2] OpenCV Documentation line Drawing Function, disponible:[https://docs.opencv.org/2.4/modules/core/doc/drawing\\_functions.html#line](https://docs.opencv.org/2.4/modules/core/doc/drawing_functions.html#line), version september 2, 2019.