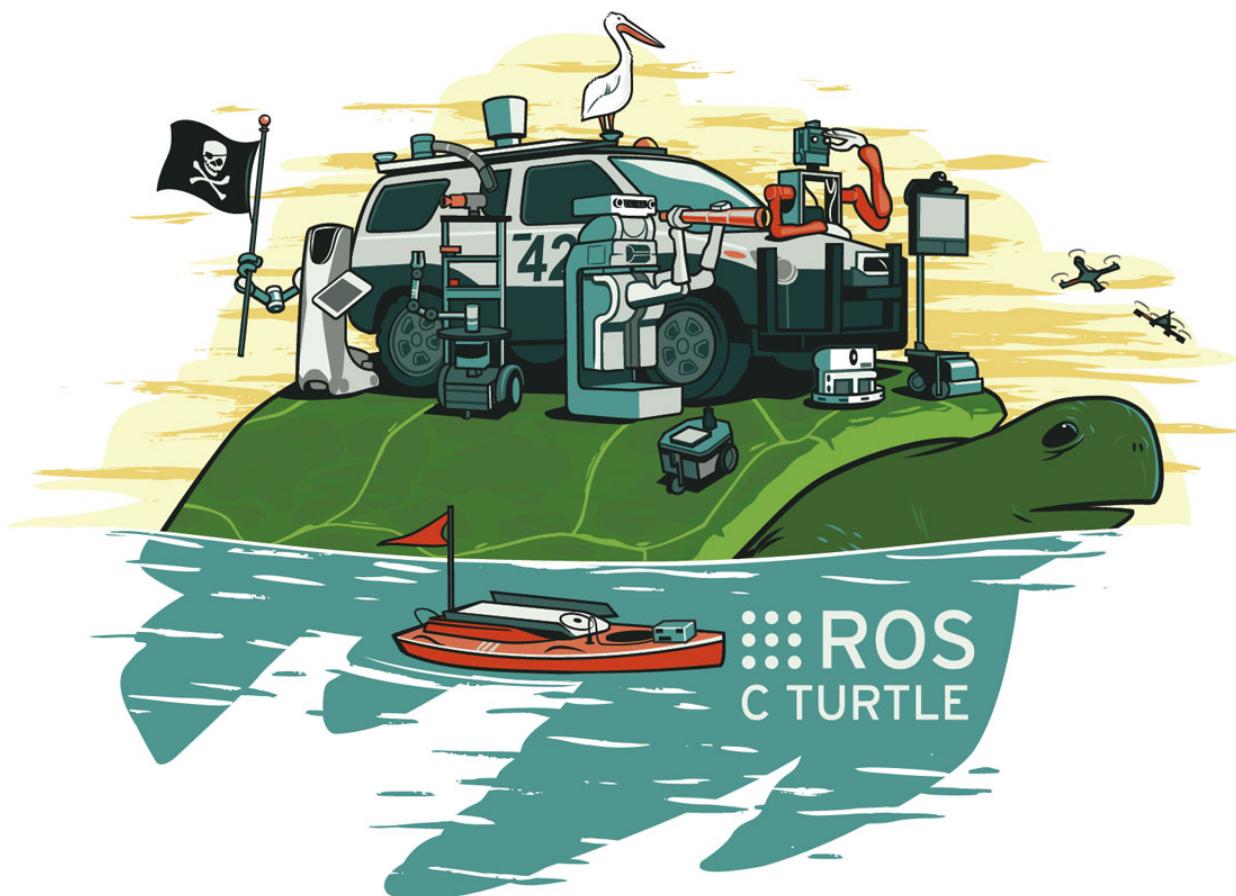


Creación de un panorama mediante características visuales



Javier Galván Martínez
48564495H

Introducción

La práctica consistirá en el desarrollo de un sistema que estudie el rendimiento de diferentes métodos de creación automática de imágenes panorámicas, mediante el uso de técnicas de visión artificial. Tomando como única información de entrada las imágenes, podremos crear un programa que encuentre automáticamente aquellas partes comunes en las imágenes.

Básicamente podemos resumir el proceso de creación de panoramas en dos pasos fundamentales. El primero de estos pasos consiste en encontrar las partes comunes que comparten las imágenes de entrada. La idea consiste en encontrar un conjunto de puntos característicos compartidos por cada par de imágenes de entrada. Una vez encontrados, estos puntos se utilizan para calcular la transformación que mejor alinea las imágenes. Existen multitud de métodos de extracción de características. Inicialmente utilizaremos varios e iremos descartando según su rendimiento. Utilizaremos *SIFT*, *SURF*, *FAST*, *MSER*, *BRIEF* y *ORB*.

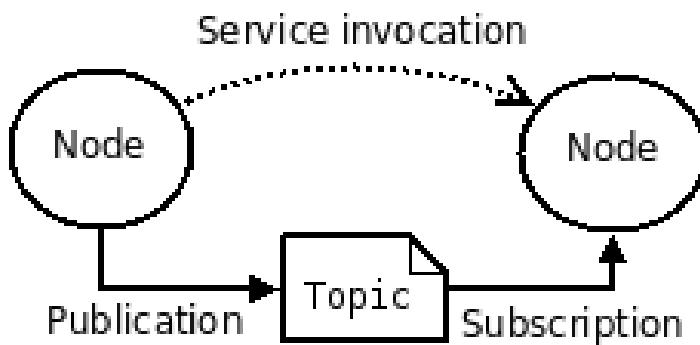
El proceso a seguir es el siguiente:

1. Llega la primera imagen. Calculamos sus características.
2. Llega la siguiente imagen. Calculamos sus características.
3. Encontramos las correspondencias entre los dos conjuntos de características. Para ello, usamos los valores de los descriptores para calcular la distancia euclídea entre cada par de características. Para cada característica de la segunda imagen, calculamos las distancias a cada una de las características de la primera imagen. Hay que tener en cuenta que siempre vamos a tener una característica cuya distancia sea la menor, pero no por eso es una correspondencia correcta. El emparejamiento será válido si:
 - a. El valor de la distancia del mejor emparejamiento es menor que el valor de la distancia del segundo mejor emparejamiento multiplicado por un coeficiente.
 - b. El valor de la distancia del mejor emparejamiento es menor que un umbral dado.
4. Usando las correspondencias encontradas, se calcula la transformación 2D que minimiza el error de los emparejamientos. Para ellos usaremos RANSAC. Esa transformación se le aplica a la segunda imagen para hacerla coincidir con la primera.
5. La segunda imagen pasa a ser la primera y empezamos desde el punto 2.

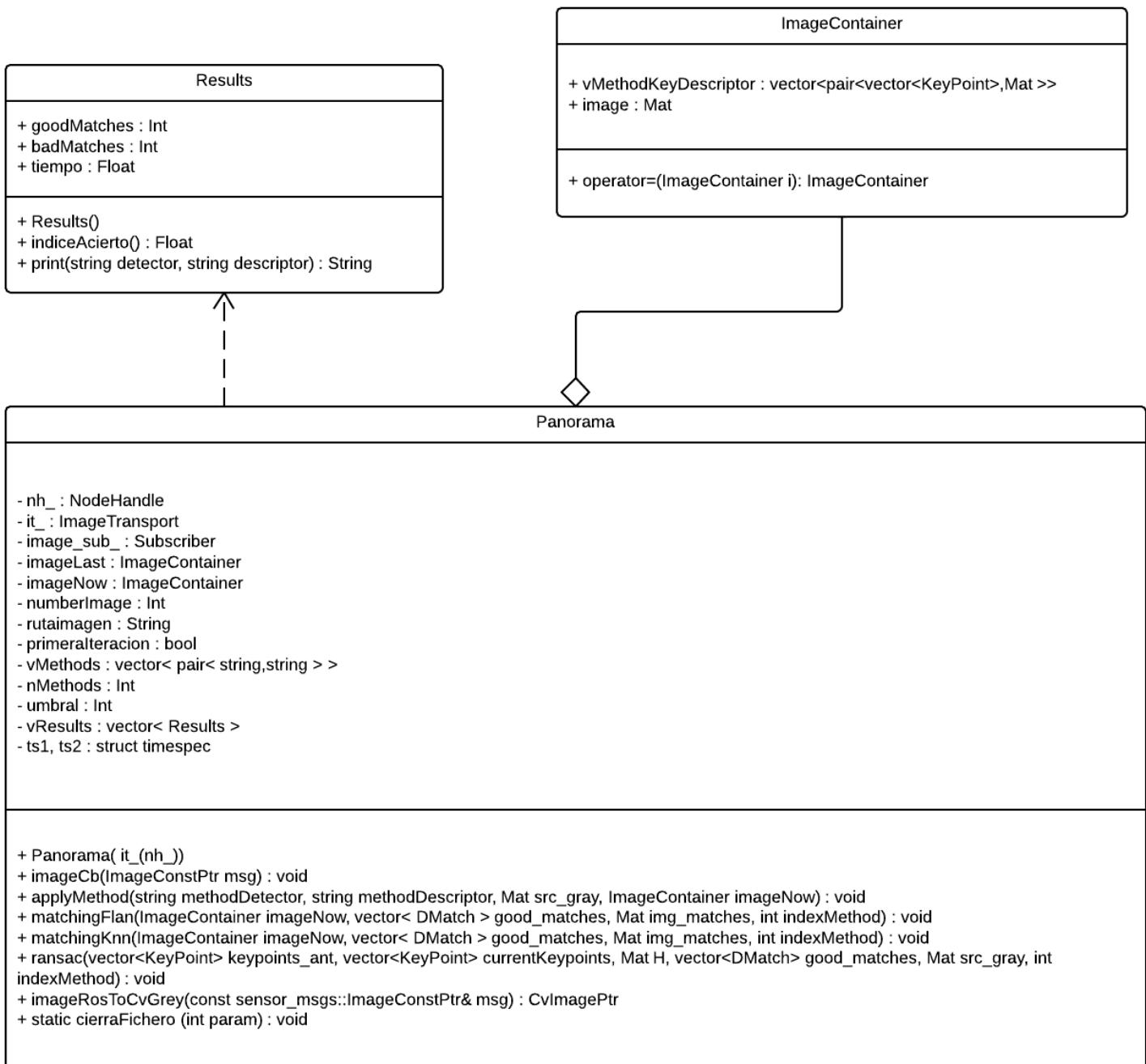
Estructura

Para la práctica utilizaremos Robot Operating System (ROS), un framework para el desarrollo de software para robots que provee los servicios estándar de un sistema operativo tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. Está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros..

En este caso tendremos un nodo de ROS (Publisher) que publique las imágenes en un topic determinado y otro nodo (Subscriber) que leerá la información.



La parte dedicada a la publicación la comentaremos más adelante(ya que no concierne al desarrollo) a la hora de poner en funcionamiento el sistema. En lo referente al *Subscriber* tenemos una división en tres clases, el suscriptor en sí, **Panorama** , que escucha el topic; **ImageContainer** que nos será de utilidad para manejar las imágenes y la información extraída de las mismas y por último **Results**, una “clase utilidades” para manejar datos de análisis sobre el sistema. A continuación vemos estas clases detalladas en un diagrama UML.



Clase ImageContainer

```
class ImageContainer{
public:
    vector< pair< vector<KeyPoint>,Mat > > vMethodKeyDescriptor;
    Mat image;
    ImageContainer & operator=(const ImageContainer &i){
        if(&i==this)
            return *this;
        else{
            vMethodKeyDescriptor.clear();
            vMethodKeyDescriptor = i.vMethodKeyDescriptor;
            image = i.image;
        }
        return *this;
    }
};
```

Como vemos esta clase cuenta con un tipo *Matriz* que contiene la imagen y una estructura para guardar sus puntos característicos *Keypoints* y los descriptores *Mat*. Se presenta en un vector de pares ya que, como veremos más adelante, **cada posición** del vector se corresponde con una **combinación de métodos**, del tipo método para extraer keypoints y método para extraer descriptores. De esta manera la posición 1 se podría corresponder con SIFT+SIFT, la segunda con SIFT+SURF, etc. así con cuantos métodos queramos probar.

Clase Results

```
class Results{
public:
    int goodMatches;
    int badMatches;
    float tiempo;

    Results(){
        goodMatches = 0;
        badMatches = 0;
        tiempo = 0;
    }
    float indiceAcierto(){
        float total = goodMatches+badMatches;
        if(total!=0)
            return (goodMatches*100)/(total);
        return 0;
    }
};
```

```

//Imprime el resultado del metodo detector y descriptor usados
string print(string detector, string descriptor){
    stringstream ss;
    float porcentaje = floorf(indiceAcierto() * 100 + 0.5) / 100;
    ss << "-----" << endl;
    ss << " | " << detector << " + " << descriptor << " | ";
    ss << "Rating: " << porcentaje/tiempo << endl;
    ss << "-----" << endl;
    ss << " | " << "GoodMatches: " << goodMatches << " | ";
    ss << "BadMatches: " << badMatches << " | ";

    ss << "Accuracy: " << porcentaje << "%" << " | ";
    ss << "Time: " << tiempo << " s" << " | " << endl << endl;

    return ss.str();
}

};


```

Se trata de una clase almacén de resultados, veremos que una vez tratados unos métodos y otros hacemos uso de sus utilidades para guardar tiempos de ejecución, buenas y malas correspondencias obtenidas con RANSAC, un porcentaje en base a esos aciertos y un “rating” que relaciona aciertos-tiempo. También construye la string que se escribirá en un fichero de resultados.

Clase Panorama

La clase principal encargada de escuchar el topic correspondiente y tratar las imágenes utilizando los métodos de extracción de características, matching y filtrado filtrado de correspondencias.

```

Panorama() : it_(nh_){

    // Cuando llegue una imagen al topic, llama a la funcion imgCb
    image_sub_ = it_.subscribe("/camera/rgb/image_color", 1, &Panorama::imageCb, this);

    vMethods.push_back(make_pair("SIFT", "SIFT"));
    nMethods = vMethods.size();

    //Reservamos espacio en el vector de resultados, tanto como metodos haya
    vResults.resize(nMethods);

    //Controlamos la señal de corte de ejecucion para cerrar el fichero de resultados
    void (*handler)(int);
    handler = signal(SIGINT, cierraFichero);
    FICHERO.open ("~/resultados.txt");
    RESULTADOSFIN="";

    numberImage = 0;
    primeraIteracion = true;
    umbral = 100;
}

```

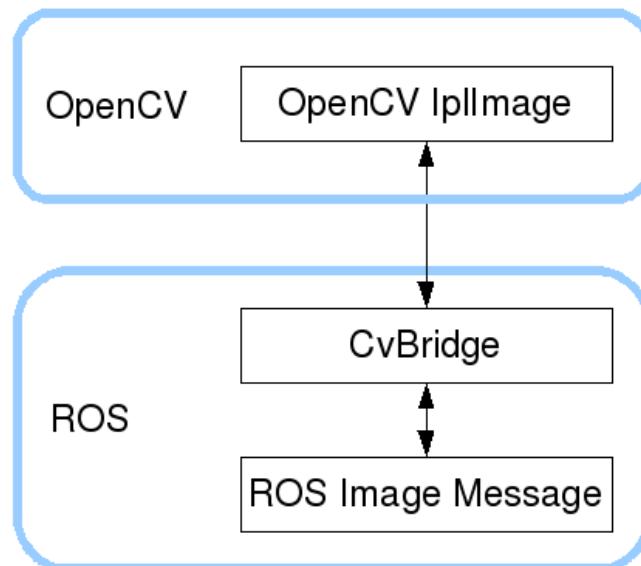
En el constructor de la clase utilizamos el tipo de dato *ImageTransport it_* y *Subscriber image_sub_* para suscribirnos al topic donde se publican las imágenes. Cuando algo se publique se llamará automáticamente a la función *imageCb* (image CallBack) que se ocupará de manejar la imagen entrante.

Añadimos los pares de métodos (detector, descriptor) al vector que se ocupa de almacenarlos *vMethods* y guardamos el número de métodos (en este caso de ejemplo es 1, pero en la experimentación usaremos varios).

El siguiente bloque de instrucciones está dedicado a la persistencia de resultados, tendremos un manejador de señales, cuando el programa se pare utilizando el comando **Ctrl+C** los resultados obtenidos hasta entonces (almacenados mediante la clase Results e impresos sobre la variable *RESULTADOSFIN*) se volcarán en el fichero “resultados.txt”. En el código se explica mediante comentarios, pero en esencia se trata de variables globales, al utilizar punteros a función para manejar señales el acceso a variables se torna un tanto enrevesado. De esta manera poco elegante podemos guardar los datos de la experimentación con relativa facilidad.

Establecemos valores iniciales para indicar el número de imágenes analizadas, si estamos en la primera iteración y el umbral del matching. Veremos la relevancia de estas variables más adelante.

A continuación debemos tratar el tipo de dato de la imagen que recibimos ya que *ROS* publica en el formato *sensor_msgs::ImageConstPtr*. Lo que haremos será una transformación a través de *CvBridge*:



```
CvImagePtr imageRosToCvGrey(const sensor_msgs::ImageConstPtr& msg){

    CvImagePtr cv_ptr;
    try{
        cv_ptr = cv_bridge::toCvCopy(msg, enc::BGR8);
    }
    catch (cv_bridge::Exception& e){
        ROS_ERROR("cv_bridge exception: %s", e.what());
        return cv_ptr;
    }

    return cv_ptr;
}
```

A continuación comentaremos la función que se invoca al recibir una imagen, ya que es bastante extensa la dividiremos en fragmentos:

```
void imageCb(const sensor_msgs::ImageConstPtr& msg){

    CvImagePtr cv_ptr = imageRosToCvGrey(msg);

    //Pasamos la imagen a escala de grises y color a tipo MAT
    Mat src_gray, imageColor;
    cvtColor( cv_ptr->image, src_gray, CV_BGR2GRAY );
    cvtColor( src_gray, imageColor, CV_GRAY2BGR );
    if(src_gray.rows==0){return;}

    /* IMAGEN ACTUAL */
    imageNow.image = imageColor;

    string mDetector;
    string mDescriptor;

    string resultadosTemp = "";

    for(int i=0; i<nMethods; ++i){

        clock_gettime( CLOCK_REALTIME, &ts1 );

        mDetector = vMethods[i].first;
        mDescriptor = vMethods[i].second;

        rutaimagen = "~/ImagenesPanorama/";

        applyMethod(mDetector,mDescriptor, src_gray, imageNow);
    }
}
```

El *CallBack* transforma la imagen *msg* que recibe, la guardamos en color y en escala de grises. Almacenamos la imagen actual en un contenedor *ImageContainer* : *ImageNow* utilizaremos esta variable para referirnos al frame actual, la imagen que analizamos en ese

momento. Declaramos una serie de variables que serán los métodos actuales utilizados y la string de resultados temporales de los mismos, utilizamos una variable temporal ya que no escribiremos estos resultados a menos que acabe el ciclo completo de extracción de características que veremos a continuación.

Comenzamos un **bucle** que se ejecutará **tantas veces** como número de **combinaciones de métodos** tengamos. Marcamos el momento de inicio de prueba del par de métodos, asignamos su nombre correspondiente al vector *vMethods* e indicamos la ruta de la imagen si quisiéramos guardar el resultado del matching, de RANSAC, la mezcla de imágenes, etc.

Llamamos a “*applyMethod(mDetector,mDescriptor, src_gray, imageNow);*” como vemos este método aplica el método de extracción de keypoints *String mDetector*, el de extracción de descriptores *String mDescriptor* sobre la imagen en escala de grises y almacenará los resultados dentro del contenedor *ImageContainer imageNow*. Así por cada iteración lo llamaremos con una combinación de métodos distintos.

Si se tratase de la primera iteración simplemente guardaríamos la información de la imagen y pondríamos el indicador de la primera iteración a *false*. Estas sentencias se indican al final del método, las veremos cuando alcancemos ese punto, de momento seguimos con el orden establecido por el número de línea.

```
if(!primeraIteracion){

    /*MATCHING*/

    vector< DMatch > good_matches;
    Mat img_matches;
    matchingFlann(imageNow, good_matches, img_matches, i);
    //matchingKnn(imageNow, good_matches, img_matches, i);

    /**PINTAMOS Keypoints y Matches*/
    vector<KeyPoint> keypointsNow = imageNow.vMethodKeyDescriptor[i].first;
    vector<KeyPoint> keypointsLast = imageLast.vMethodKeyDescriptor[i].first;

    drawMatches(imageNow.image, keypointsNow, imageLast.image,
               keypointsLast,good_matches, img_matches);

    imshow("match", img_matches);

    rutaimagen+=boost::lexical_cast<string>(numberImage);
    rutaimagen+=mDetector+"_"+mDescriptor+".jpg";
    // imwrite( rutaimagen, img_matches );
}
```

No siendo la primera iteración, suponiendo que ya tenemos una imagen anterior en memoria, procedemos a apartado de *Matching*. De esta manera obtenemos las correspondencias entre los dos conjuntos de características de la imagen anterior *imageLast* y la actual *imageNow*, eligiendo previamente una función de matching específica, en este caso nos decantamos por FLANN (Fast Library for Approximate Nearest Neighbors) o KNN (K nearest

neighbors).

```

void matchingFlann(ImageContainer &imageNow, vector< DMatch > &good_matches, Mat
&img_matches,const int &indexMethod){

    Mat descriptorNow = imageNow.vMethodKeyDescriptor[indexMethod].second;
    Mat descriptorLast = imageLast.vMethodKeyDescriptor[indexMethod].second;

    FlannBasedMatcher matcher;
    vector< DMatch > matches;
    matcher.match( descriptorNow, descriptorLast, matches );
    //double max_dist = 0;
    double min_dist = 100;

    for( int i = 0; i < descriptorNow.rows; ++i ){
        double dist = matches[i].distance;
        if( dist < min_dist ) min_dist = dist;
        //if( dist > max_dist ) max_dist = dist;
    }

    /*COMENTAR LA MAXIMA DISTANCIA Y LA MINIMA POR ESO NO USAMOS LA MEDIA*/
    //printf("-- Max dist : %f \n", max_dist );
    //printf("-- Min dist : %f \n", min_dist );

    /* Establecemos un multiplicador para ampliar el umbral basado en la minima distancia
y a parte otro arbitrario */
    //if( matches[i].distance < 2*min_dist && matches[i].distance<umbralDado )
    for( int i = 0; i < descriptorNow.rows; ++i ){
        if( matches[i].distance < 1.5*min_dist && matches[i].distance <umbral){
            good_matches.push_back( matches[i]);
        }
    }
}
}

```

Tanto en FLANN como en KNN vamos a fijarnos en cierto detalle, una vez vemos las distancias entre descriptores, nos interesa quedarnos con las de **menor valor**, de esta forma nos aseguramos una primera tanda de correspondencias correctas. Para ello guardamos el valor de la menor distancia y la multiplicamos por un **coeficiente** (1.5), así nos quedamos con aquellas que cumplan este margen. A parte aplicamos el **umbral** establecido en el constructor de la clase, como veremos este valor ya es algo más arbitrario, obtenido mediante la experimentación.

Decir también que **no** se usa la **media** ya que las distancias mayores podrían desviar en gran medida el valor de corte final. Experimentando se vieron casos con resultados máximos de 2000 y mínimos de 40, esto podría darnos una gran cantidad de malas correspondencias y por tanto queda descartado.

Con estas correspondencias y las características de las imágenes obtenidas anteriormente, podemos llamar a la función que las pinte, mostrar el resultado por pantalla, en este caso guardar la imagen obtenida en la ruta determinada...

```

/* APPLICAMOS RANSAC */

vector<Point2f> obj;
vector<Point2f> scene;
Mat H;
for( int j = 0; j < good_matches.size(); ++j )
{
    //-- Get the keypoints from the good matches
    obj.push_back(keypointsNow[ good_matches[j].queryIdx ].pt );
    scene.push_back(keypointsLast[ good_matches[j].trainIdx ].pt );
}

if(obj.size()>4 && scene.size()>4){
    Mat H = findHomography( obj, scene, CV_RANSAC );
    ransac(keypointsNow, keypointsLast, H, good_matches, src_gray, i);
}

/* GUARDAR DATOS */
clock_gettime( CLOCK_REALTIME, &ts2 );
vResults[i].tiempo +=(float) ( 1.0*(1.0*ts2.tv_nsec - ts1.tv_nsec*1.0)*1e-9+
1.0*ts2.tv_sec - 1.0*ts1.tv_sec );
resultadosTemp += vResults[i].print(mDetector, mDescriptor);

```

Una vez hemos obtenido las correspondencias, con el objetivo de afinarlas, eliminar *outliers*...aplicamos **Ransac**, para ello declaramos los tipos de datos necesarios para la transformación 2D, utilizando los keypoints de la imagen actual y la última analizada.

Como vemos le pasamos como último parámetro el índice del bucle, *i*, el cual indica en qué combinación de métodos nos encontramos; de esta manera asignamos al método utilizado, los *matchers* (buenos y malos) encontrados por ransac y los guardamos como resultado.

En la función RANSAC

```

//Guardamos los matches buenos y malos correspondientes al metodo
vResults[indexMethod].badMatches += contdrawmalos;
vResults[indexMethod].goodMatches += contdrawbuenos;

```

```

/* GUARDAR DATOS */
clock_gettime( CLOCK_REALTIME, &ts2 );
vResults[i].tiempo +=(float) ( 1.0*(1.0*ts2.tv_nsec - ts1.tv_nsec*1.0)*1e-9+
1.0*ts2.tv_sec - 1.0*ts1.tv_sec );
resultadosTemp += vResults[i].print(mDetector, mDescriptor);
}
}
RESULTADOSFIN = resultadosTemp;
imageLast = imageNow;
imageNow.vMethodKeyDescriptor.clear();
numberImage++;

```

```
primeraIteracion = false;
cv::waitKey(3);
```

Para terminar, medimos el tiempo, guardamos los resultados que correspondan a los métodos usados y actualizamos la imagen, el número de la imagen, etc.

Este método ***imageCb*** se llamará por cada frame que se publique en el topic, realizará el bucle con las sentencias señaladas tantas veces como métodos queramos probar y así hará hasta que recibe una señal Ctrl+C de corte de ejecución, en ese momento se volcarán los resultados sobre un fichero y se cerrará el programa.

Experimentación

Inicialmente probamos una batería de combinaciones de métodos bastante amplia para determinar cuales son los más eficientes, así realizar las pruebas siguientes con los más prometedores en entornos de mayor problemática. La experimentación se realizará sobre varios conjuntos de imágenes, *datasets*, dos de ellos dados con anterioridad y el último de ellos obtenido mediante la webcam del propio ordenador.

```
vMethods.push_back(make_pair("SIFT", "SIFT"));
vMethods.push_back(make_pair("SIFT", "BRIEF"));
vMethods.push_back(make_pair("SIFT", "SURF"));
//vMethods.push_back(make_pair("SIFT", "ORB"));
vMethods.push_back(make_pair("SURF", "SIFT"));
vMethods.push_back(make_pair("SURF", "SURF"));
vMethods.push_back(make_pair("SURF", "BRIEF"));
//vMethods.push_back(make_pair("SURF", "ORB"));
vMethods.push_back(make_pair("FAST", "SURF"));
vMethods.push_back(make_pair("FAST", "SIFT"));
vMethods.push_back(make_pair("FAST", "BRIEF"));
//ORB ya usa FAST y BRIEF con lo cual no probamos esa combinacion
vMethods.push_back(make_pair("ORB", "SIFT"));
vMethods.push_back(make_pair("ORB", "SURF"));
vMethods.push_back(make_pair("ORB", "ORB"));
vMethods.push_back(make_pair("MSER", "SIFT"));
vMethods.push_back(make_pair("MSER", "SURF"));
vMethods.push_back(make_pair("MSER", "ORB"));
vMethods.push_back(make_pair("MSER", "BRIEF"));
```

Para lanzar la práctica debemos **compilar** con *catkin_make*, lanzar *roscore* para poder ejecutar las **funciones** de **ROS**, lanzar nuestro nodo ***Subscriber-Panorama*** con el comando ¹*optirun rosrun practica1 panorama* y la entrada que leeremos del topic vendrá dada por el **dataset** “*rgbd_dataset_freiburg3_nostructure_texture_far.bag*” para lanzar esta consecución de imágenes ejecutamos la orden

```
rosbag play -l -r 0.05 rgbd_dataset_freiburg3_nostructure_texture_far.bag
```

Al aplicar tantos métodos por cada frame indicamos una publicación de bastante más lenta con el comando **-r 0.05**.

¹Nota: Ejecutamos el comando optirun para utilizar la gráfica Nvidia mediante los drivers no oficiales Bumblebee que dan soporte a estas GPUs sobre Linux.

```

javi@ultra1 ~ $ cd catkin_ws/
javi@ultra1 ~/catkin_ws $ catkin_make

javi@ultra1 ~/catkin_ws/src/practical $ rosbag play -l -r 0.05
CMakeLists.txt
include/
package.xml
javi@ultra1 ~/catkin_ws/src/practical $ rosbag play -l -r 0.05 rgbd_dataset_freiburg3_nostructure_texture_far.bag
g

javi@ultra1 ~ $ roscore
javi@ultra1 ~ $ rosrun practical panorama

```

Haremos dos pasadas, una utilizando FLANN y otra KNN, ambas pruebas rondan los 5 minutos de duración. Resaltamos los métodos con mejor resultado fijándonos en el **rating**, una medida de precisión con respecto al tiempo, siendo la precisión la relación existente entre aciertos buenos y malos.

Feature Matching with FLANN:

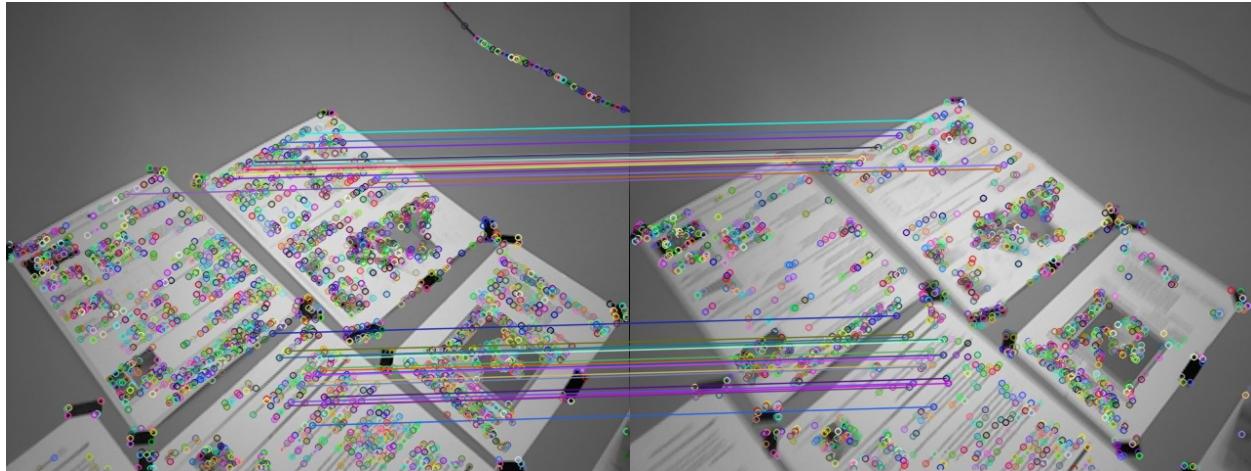
Métodos	Rating	GoodMatches	BadMatches	Accuracy	Time
SIFT + SIFT	2,34344	345	261	56.93%	24.2933 s
SIFT + BRIEF	6,80739	31	8	79.49%	11.677 s
SIFT + SURF	2,92611	274	313	46.68%	15.9529 s
SURF + SIFT	1,44767	1072	4	99.63%	68.821 s
SURF + SURF	2,89588	476	97	83.07%	28.6855 s
SURF + BRIEF	7,50089	26	1	96.3%	12.8385 s
FAST + SURF	6,03449	250	209	54.47%	9.02644 s

<i>FAST + SIFT</i>	4,98024	1404	27	98.11%	19.6999 s
<i>FAST + BRIEF</i>	34,552	5	0	100%	2.89419 s
<i>ORB + SIFT</i>	3,37943	602	26	95.86%	28.3657 s
<i>ORB + SURF</i>	4,73023	361	8	97.83%	20.6819 s
<i>ORB + ORB</i>	52,3116	34	2	94.44%	1.80534 s
<i>MSER + SIFT</i>	3,19912	674	20	97.12%	30.3584 s
<i>MSER + SURF</i>	13,3521	190	6	96.94%	7.26027 s
<i>MSER + ORB</i>	13,8157	42	38	52.5%	3.80003 s
<i>MSER + BRIEF</i>	13,043	27	28	49.09%	3.7637 s

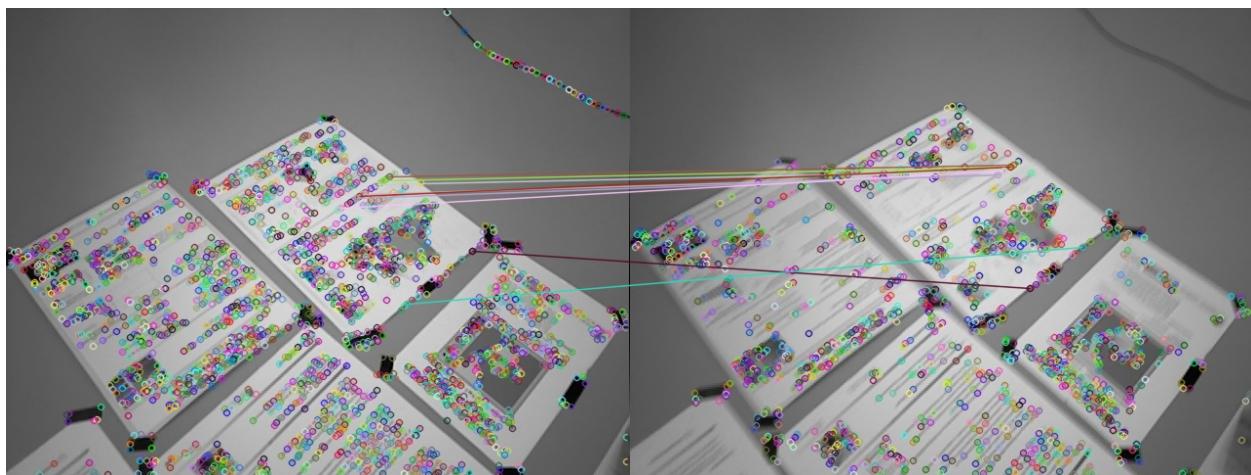
Feature Matching with KNN:

Métodos	Rating	GoodMatches	BadMatches	Accuracy	Time
<i>SIFT + SIFT</i>	3,39473	12903	1550	89.28%	26.2996 s
<i>SIFT + BRIEF</i>	7,36443	4849	289	94.38%	12.8157 s
<i>SIFT + SURF</i>	2,29687	5660	7654	42.51%	18.5078 s
<i>SURF + SIFT</i>	1,38587	26020	337	98.72%	71.2334 s
<i>SURF + SURF</i>	3,00447	29699	2571	92.03%	30.631 s
<i>SURF + BRIEF</i>	6,74107	8194	499	94.26%	13.9829 s
<i>FAST + SURF</i>	5,25021	13080	9063	59.07%	11.251 s
<i>FAST + SIFT</i>	4,66336	45889	267	99.42%	21.3194 s
<i>FAST + BRIEF</i>	28,3019	9380	299	96.91%	3.42415 s
<i>ORB + SIFT</i>	3,19155	13520	761	94.67%	29.6627 s
<i>ORB + SURF</i>	4,45116	18178	629	96.66%	21.7136 s
<i>ORB + ORB</i>	50,56115	929	2	99.79%	1.97363 s
<i>MSER + SIFT</i>	2,90992	5013	105	97.95%	33.6607 s
<i>MSER + SURF</i>	10,6525	4847	598	89.02%	8.3567 s
<i>MSER + ORB</i>	24,9711	637	9	98.61%	3.94896 s
<i>MSER + BRIEF</i>	24,8204	1114	18	98.41%	3.96488 s

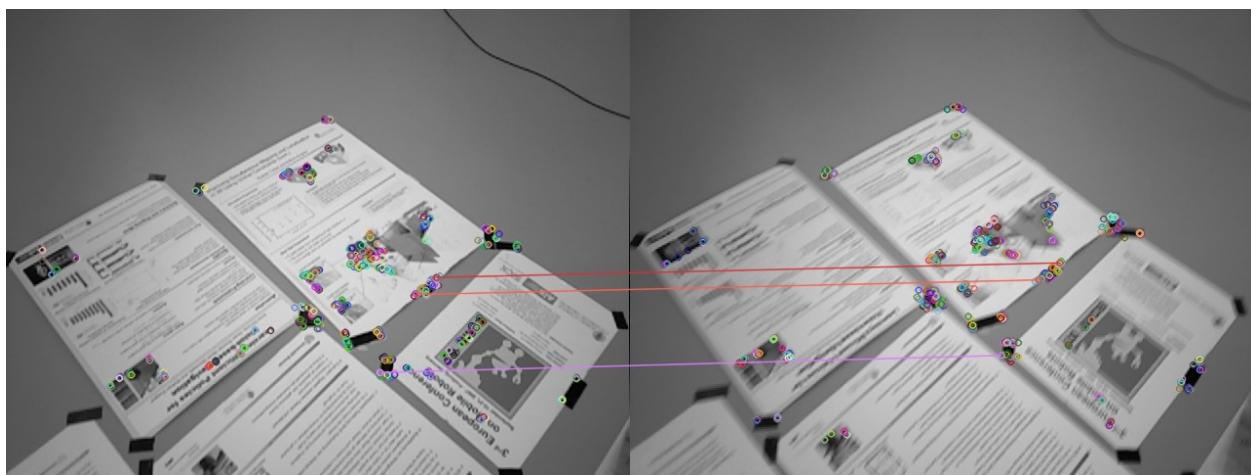
FAST+SIFT: Vemos una gran cantidad de coincidencias pero falla a menudo.



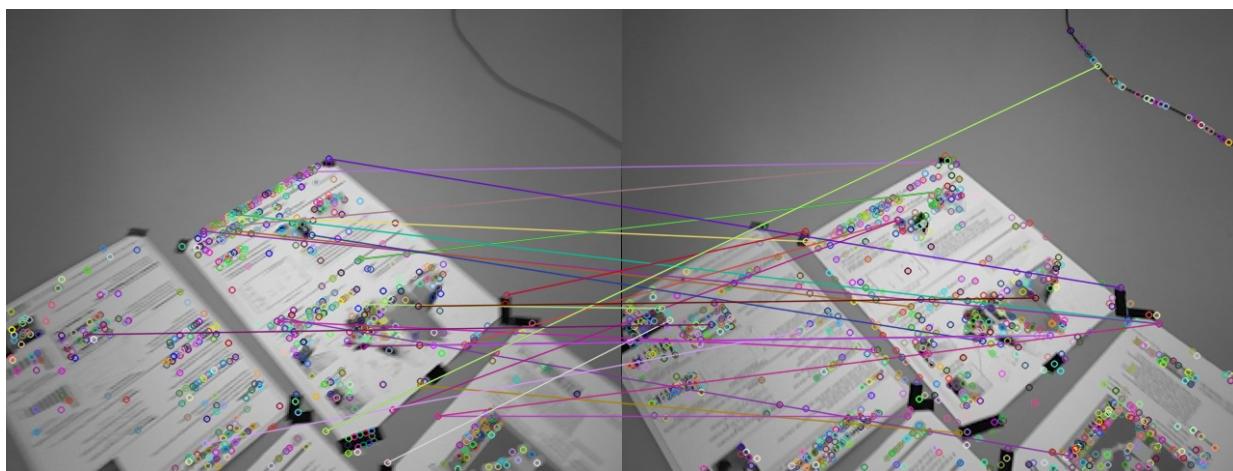
FAST+SURF: Encontramos un escenario parecido en esta combinación, FAST encuentra una gran cantidad de keypoints pero el resultado final deja mucho que desear.



ORB+ORB: Nos da unas coincidencias muy limitadas pero casi siempre correctas



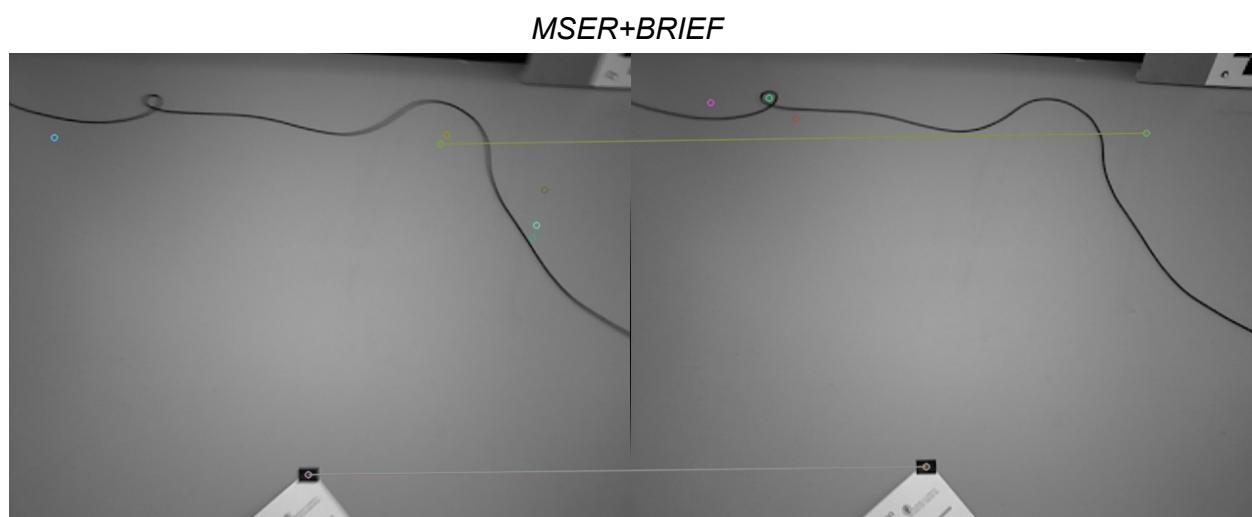
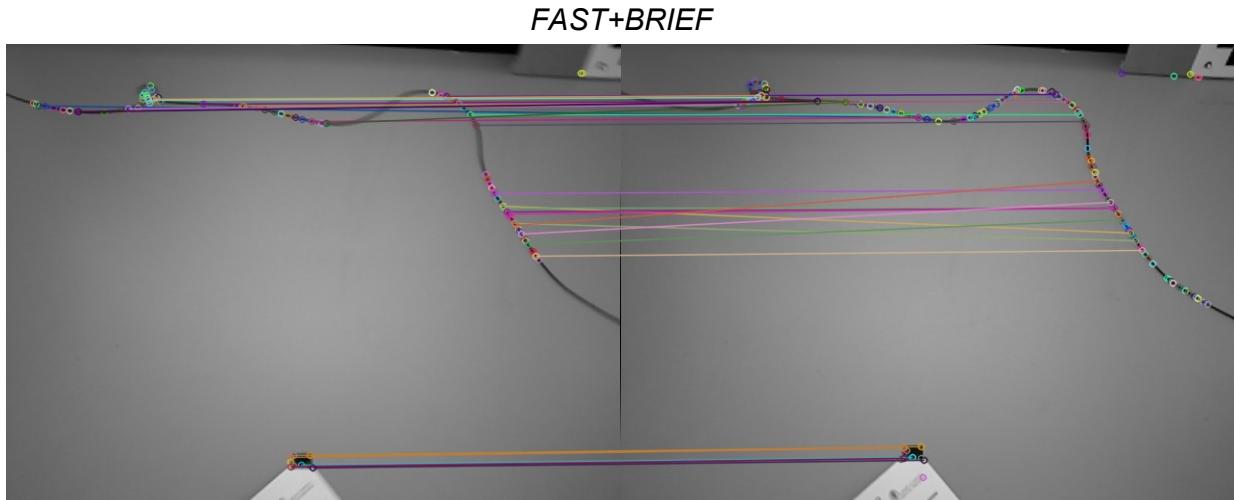
SIFT+SURF: Da resultados desastrosos en ciertas situaciones.



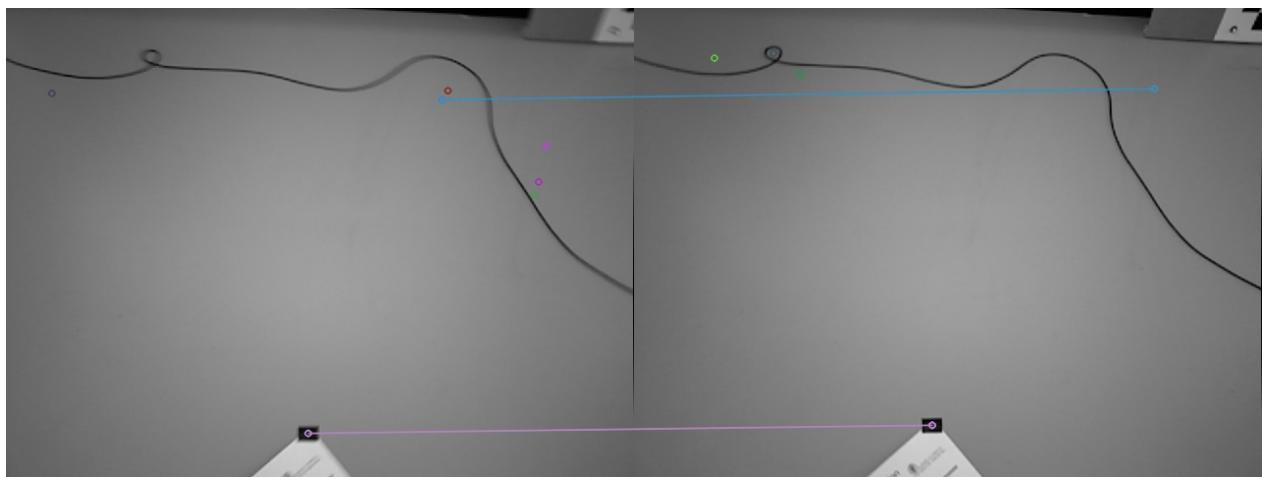
Como podemos ver FLANN nos da menos precisión que KNN, mientras que KNN es ligeramente más lento. La medida temporal de KNN con respecto a FLANN apenas se aprecia mientras que en precisión se ve claramente la superioridad la KNN.

Una vez tenemos los métodos más efectivos, probamos las partes más conflictivas de este primer dataset, los “cables” que aparecen a partir del segundo 10. También ampliamos el umbral a 200 para lograr una mayor cantidad de puntos significativos. Para acotar el *rosbag* utilizamos el argumento **-s 12** con una velocidad de frame ya más elevada:

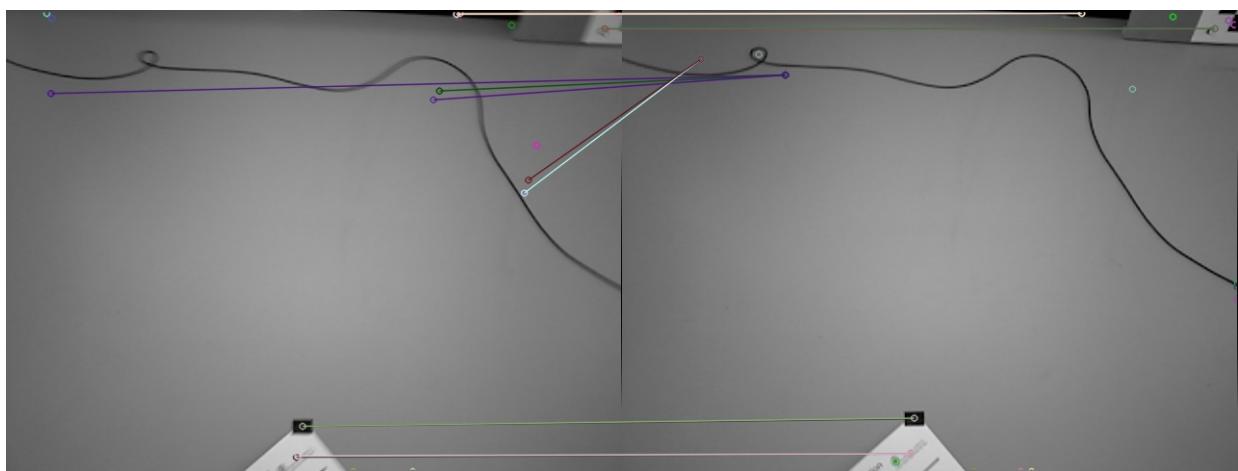
```
rosbag play -l -r 0.1 -s 12 rgbd_dataset_freiburg3_nostructure_texture_far.bag
```



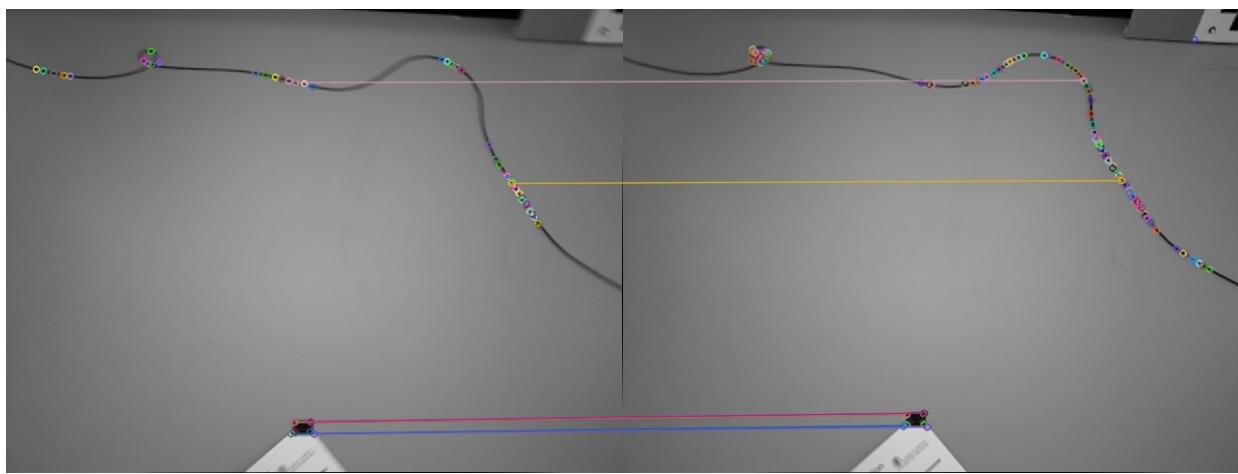
MSER+ORB

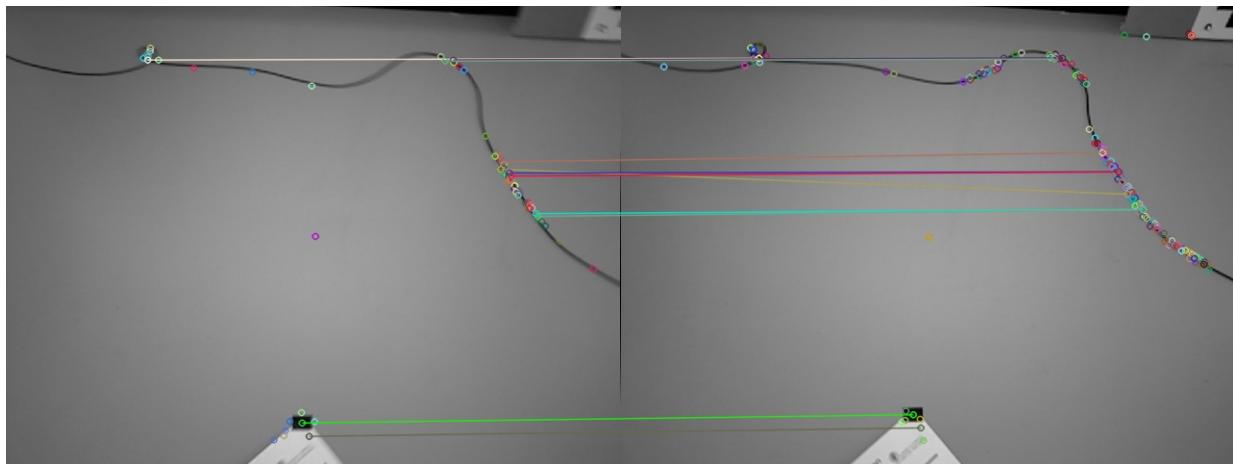
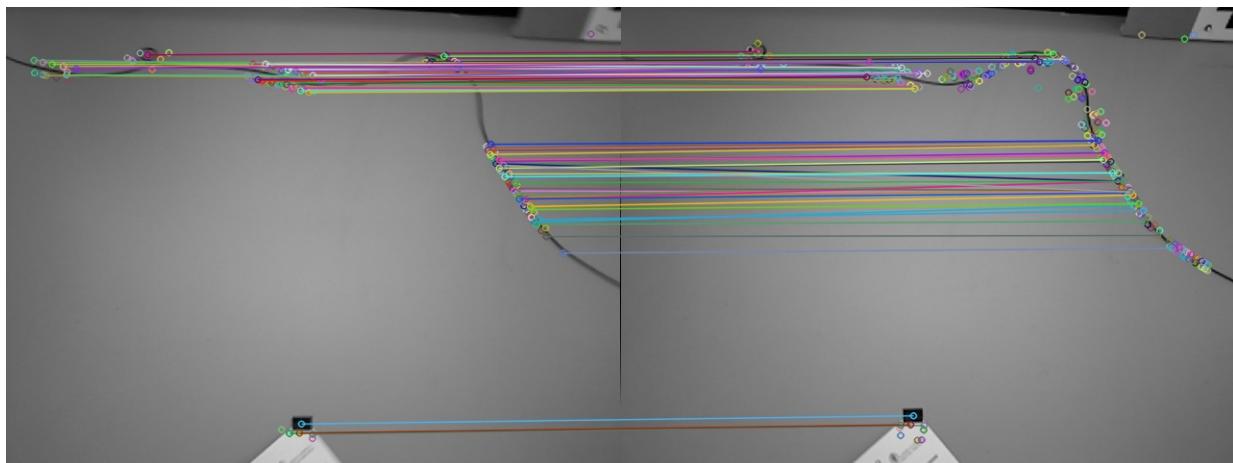


MSER+SURF



ORB+ORB



SIFT+BRIEF**SURF+BRIEF****Entorno conflictivo 1 Cables (KNN):**

Tiempo total de ejecución: 29.9173s

Métodos	Rating	GoodMatches	BadMatches	Accuracy	Time
<i>SIFT + BRIEF</i>	9,99197	1469	311	82.53%	8.25964 s
<i>SURF + BRIEF</i>	11,4173	4014	610	86.81%	7.60334 s
<i>FAST + BRIEF</i>	80,8445	3337	408	89.11%	1.10224 s
<i>ORB + ORB</i>	70,545	1714	57	96.78%	1.37189 s
<i>MSER + SURF</i>	15,2267	603	226	72.74%	4.77712 s
<i>MSER + ORB</i>	39,6515	146	17	89.57%	2.25893 s
<i>MSER + BRIEF</i>	43,2175	243	5	97.98%	2.26714 s

Se aprecia claramente como unos métodos dan una gran cantidad de *matches* con respecto a otros, pero de nada nos sirve si la cantidad de malas correspondencias aumenta también. Por tanto podríamos decir que ***ORB*** es el que mejor resultado obtiene incluso en este escenario ya que alcanza la mayor cantidad de aciertos con respecto a errores en un tiempo bastante ajustado. Por supuesto ***FAST+BRIEF*** también sería una buena elección pero estos ya se encuentran integrados en ORB (Oriented ***FAST*** and Rotated ***BRIEF***).

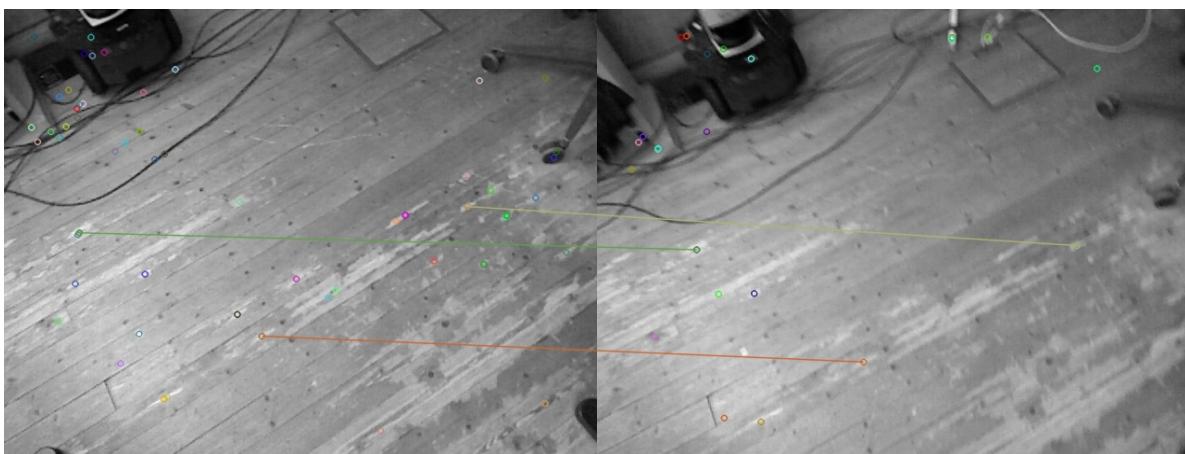
Para continuar probaremos el *dataset_freiburg1_360*, en este video encontramos un movimiento más brusco y por tanto las imágenes aparecen de una forma menos clara. De nuevo daremos un tiempo lo suficientemente largo como para encontrar lecturas concluyentes (unos 5 minutos). Para ello el comando será:

```
rosbag play -L -r 0.2 rgbd_dataset_freiburg1_360.bag
```

FAST + BRIEF



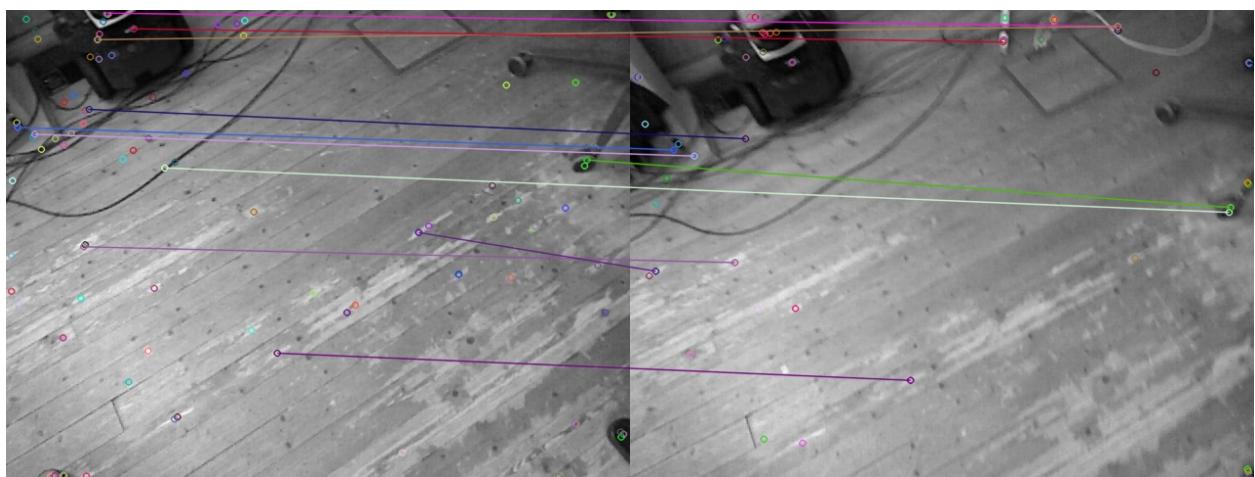
MSER+BRIEF



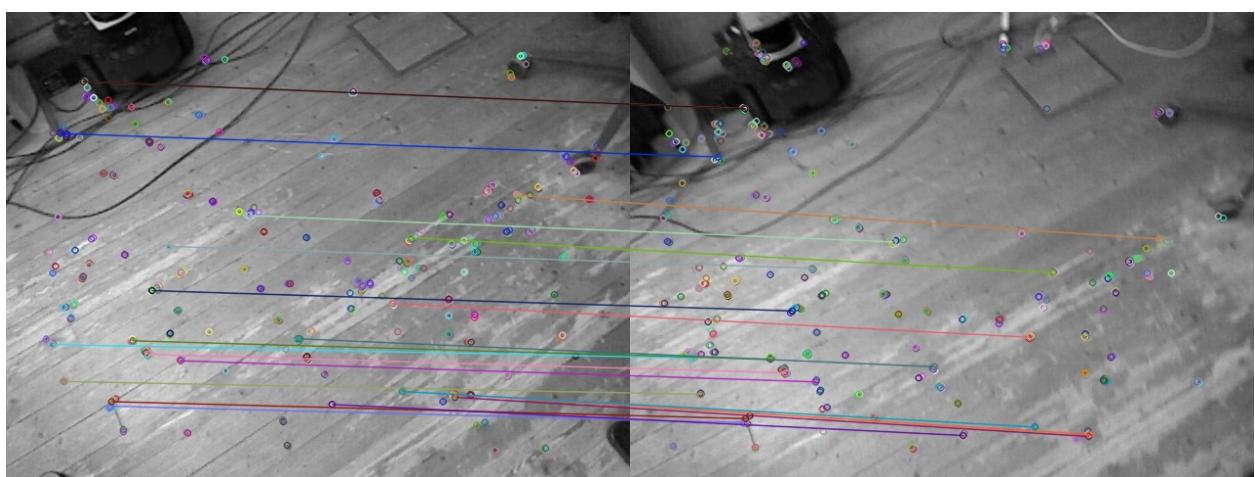
MSER+ORB



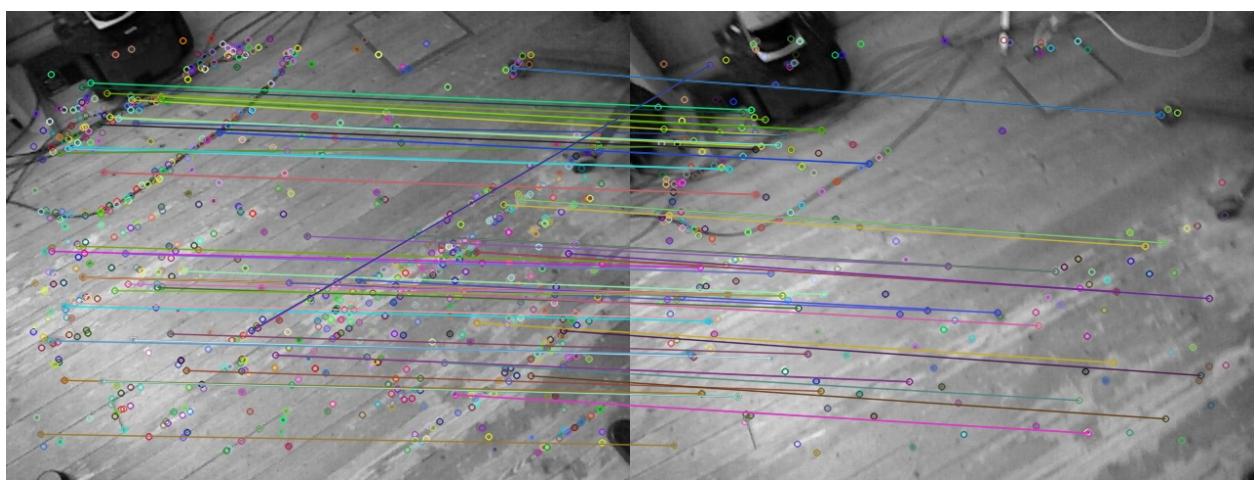
MSER+SURF



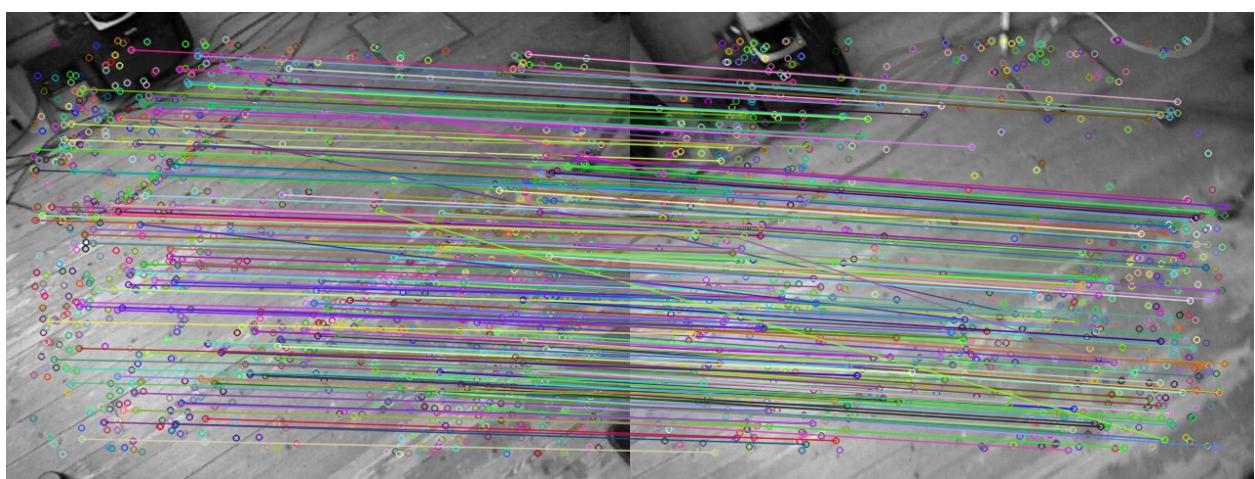
ORB+ORB



SIFT+BRIEF



SURF+BRIEF



FAST + BRIEF



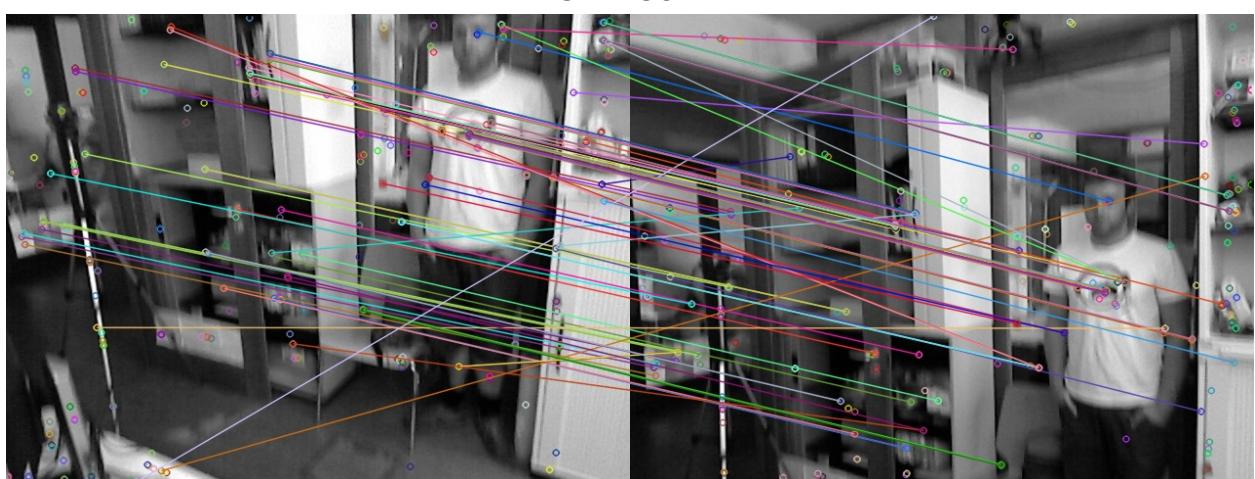
MSER+BRIEF



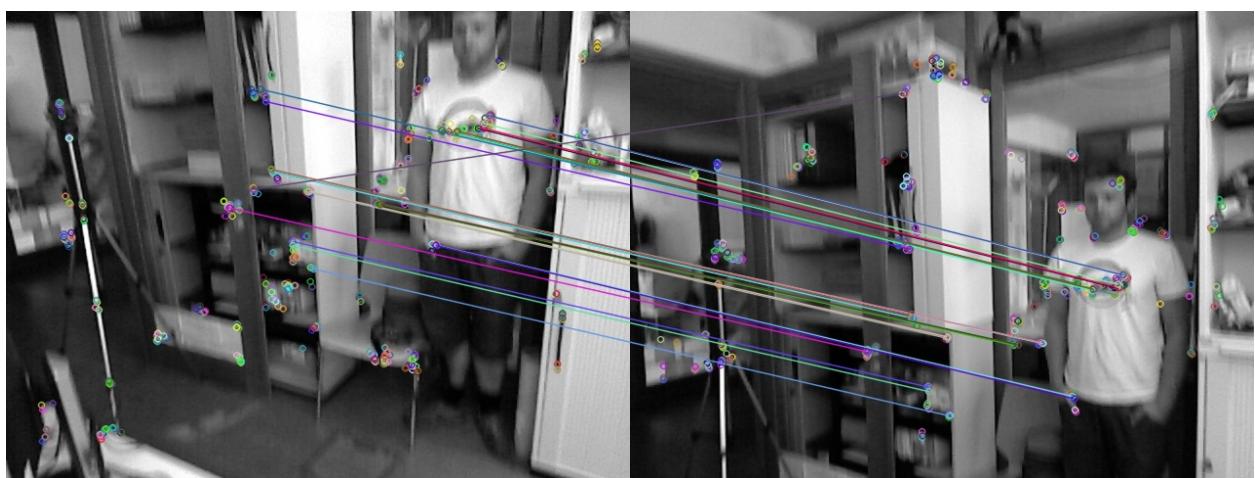
MSER+ORB



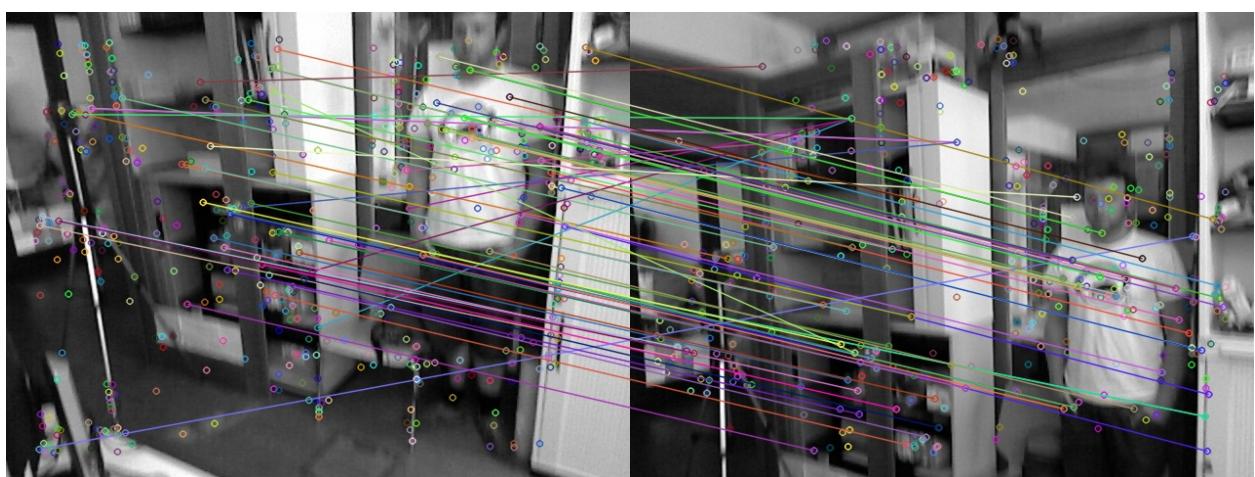
MSER+SURF



ORB+ORB



SIFT+BRIEF



SURF+BRIEF



Entorno conflictivo 2 *freiburg1_360 -r 0.2 (KNN)*:

Tiempo total de ejecucion: 315.87 s

Métodos	Rating	GoodMatches	BadMatches	Accuracy	Time
<i>SIFT + BRIEF</i>	1,4905	18705	1670	91.8%	61.5899 s
<i>SURF + BRIEF</i>	1,16314	63945	13136	82.96%	71.3243 s
<i>FAST + BRIEF</i>	4,85865	68177	11282	85.8%	17.6592 s
<i>ORB + ORB</i>	8,85433	15737	684	95.83%	10.823 s
<i>MSER + SURF</i>	1,87442	9448	2855	76.79%	40.9672 s
<i>MSER + ORB</i>	3,37984	2716	362	88.24%	26.1077 s
<i>MSER + BRIEF</i>	3,58874	4182	284	93.64%	26.0927 s

Entorno conflictivo 2 *freiburg1_360 -r 0.2 (FLANN)*:

Tiempo total de ejecucion: 317.63 s

Métodos	Rating	GoodMatches	BadMatches	Accuracy	Time
<i>SIFT + BRIEF</i>	1,32031	243	42	85.26%	64.5758 s
<i>SURF + BRIEF</i>	0,99498	53	22	70.67%	71.0266 s
<i>FAST + BRIEF</i>	5,64543	249	102	70.94%	12.5659 s
<i>ORB + ORB</i>	9,35237	689	39	94.64%	10.1194 s
<i>MSER + SURF</i>	1,95465	1045	205	83.6%	42.7699 s
<i>MSER + ORB</i>	2,20994	208	128	61.9%	28.0098 s
<i>MSER + BRIEF</i>	1,88302	163	148	52.41%	27.833 s

Ninguno se salva de tener alguna falla pero de nuevo **ORB** arroja los mejores resultados más equilibrados.

Experimentación usando la cámara

Tras todas estas pruebas podemos encaminarnos a el test con la **webcam**, escogiendo ORB+ORB como métodos para extraer los descriptores y KNN para el filtrado de correspondencias.

En primer lugar debemos instalar la herramienta *gscam*, para ello ejecutamos:

```
sudo apt-get install ros-hydro-gscam
```

Y nos aseguramos de añadir al archivo .bashrc la siguiente directiva:

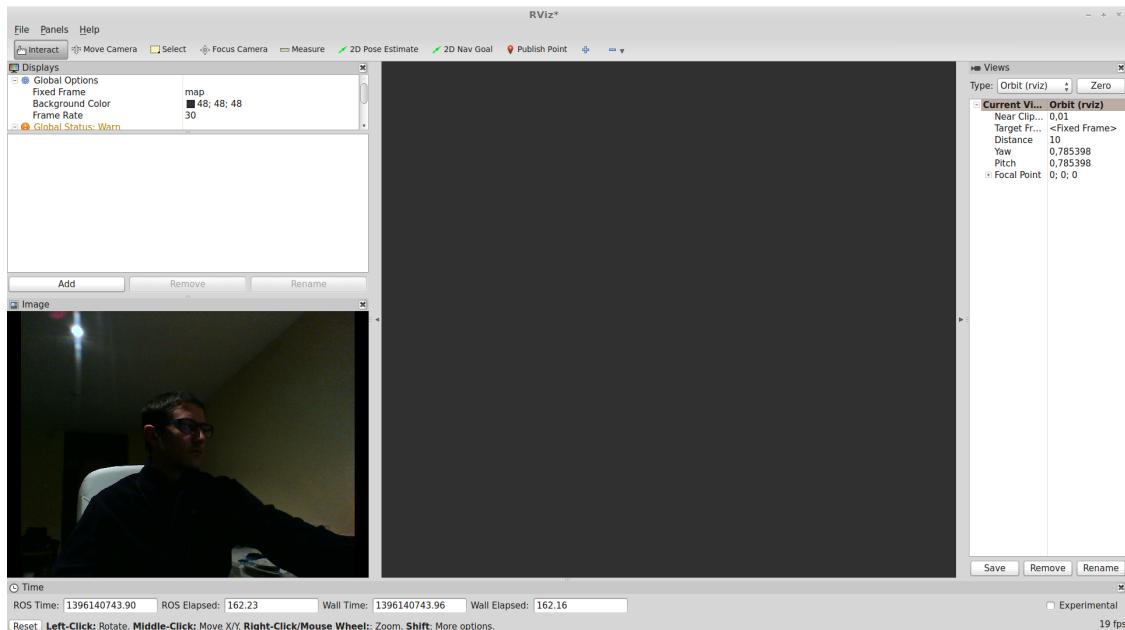
```
export GSCAM_CONFIG=
"v4l2src device=/dev/video0 ! video/x-raw-rgb,framerate=30/1 ! ffmpegcolorspace"
```

Ya podemos ejecutar los comandos necesarios para lanzar la publicación de imágenes por medio de la cámara, y abrimos la interfaz gráfica rviz junto con el topic pertinente para comprobar su funcionamiento.

```
javi@ultral ~ $ rosrun gscam gscam
[ INFO] [1396140560.308855572]: Using gstreamer config from env: "v4l2src device=/dev/video0 ! video/x-raw-rgb,framerate=30/1 ! ffmpegcolorspace"
[ INFO] [1396140560.317243945]: using default calibration URL
[ INFO] [1396140560.317464264]: camera calibration URL: file:///home/javi/.ros/camera_info/camera.yaml
[ INFO] [1396140560.317681161]: Unable to open camera calibration file [/home/javi/.ros/camera_info/camera.yaml]
[ WARN] [1396140560.317800136]: Camera calibration file /home/javi/.ros/camera_info/camera.yaml not found.
[ INFO] [1396140560.317920150]: Loaded camera calibration from
[ INFO] [1396140560.317957969]: No camera frame_id set, using frame "/camera_frame".
[ INFO] [1396140560.476736738]: Time offset: 1396118563,162
[ INFO] [1396140561.472478823]: Publishing stream...
[ INFO] [1396140561.472763454]: Started stream.
```

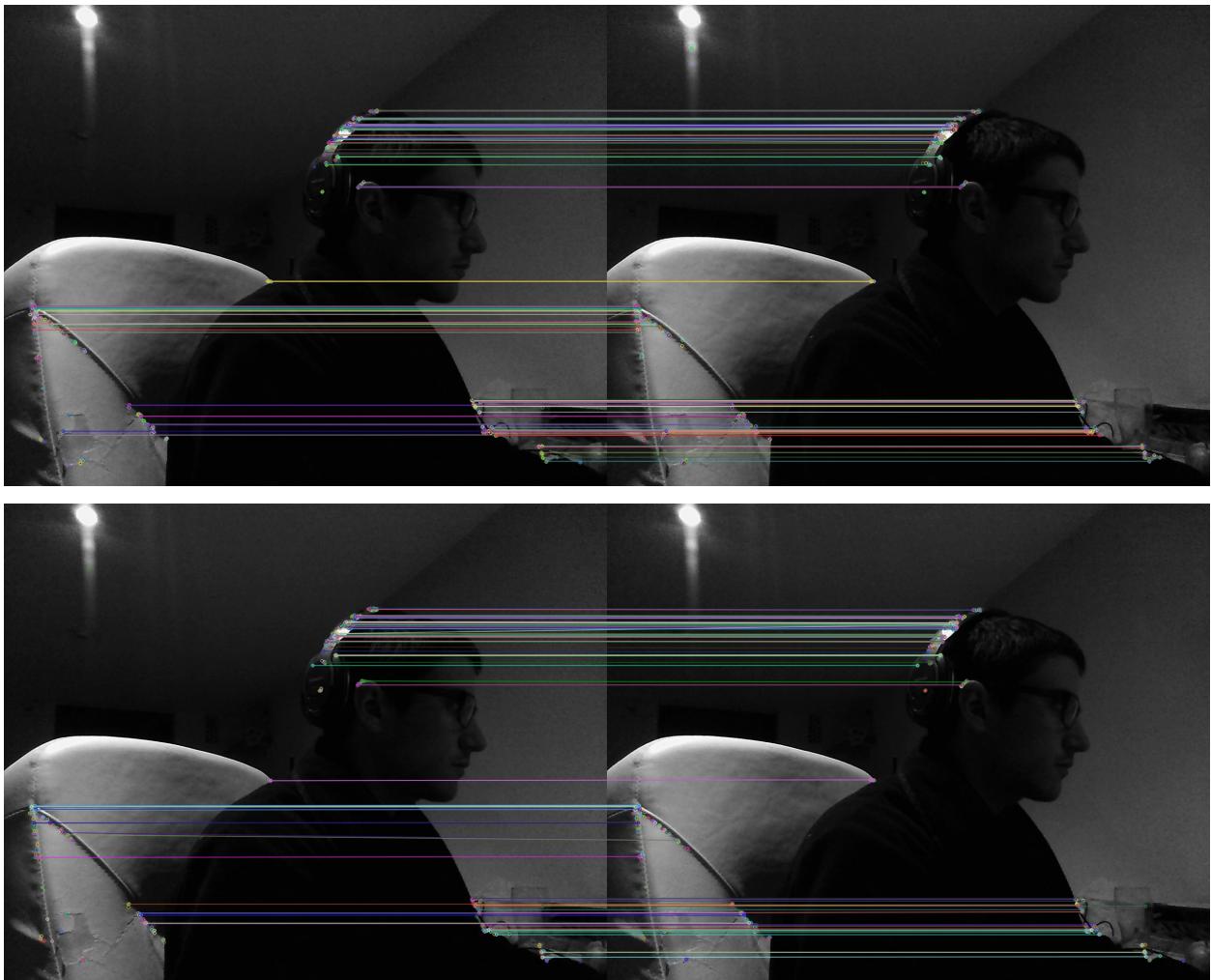


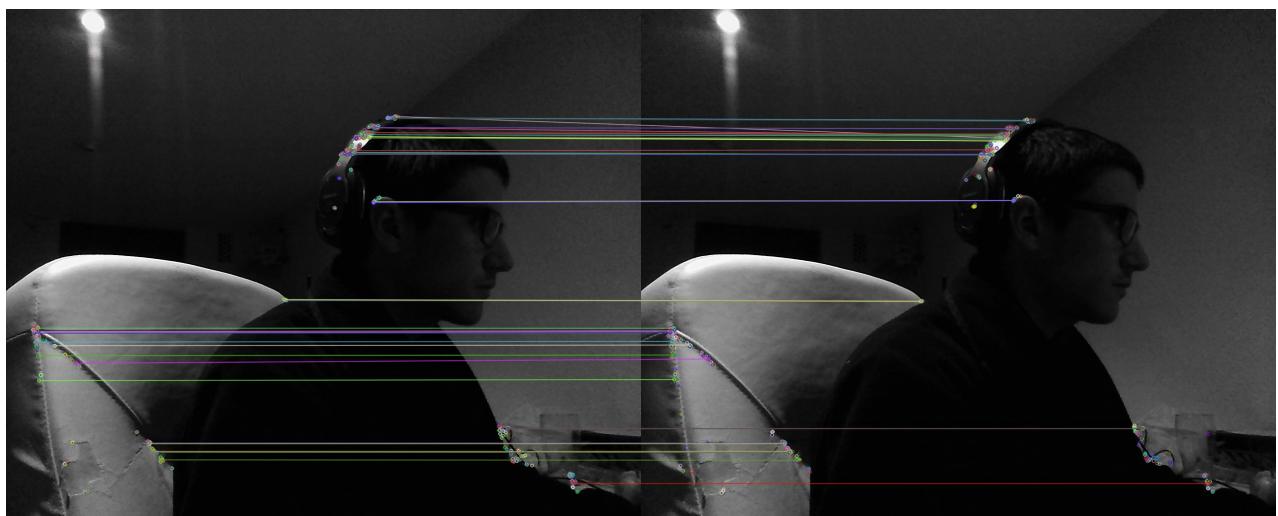
```
javi@ultral ~ $ optirun rosrun rviz rviz
[rosrun] Couldn't find executable named rviz below /opt/ros/hydro/share/rviz
javi@ultral ~ $ optirun rosrun rviz rviz
[ INFO] [1396140578.894073265]: rviz version 1.10.11
[ INFO] [1396140578.894197826]: compiled against OGRE version 1.7.4 (Cthugha)
[ INFO] [1396140580.129991937]: OpenGL version: 4.2 (GLSL 4.2).
```



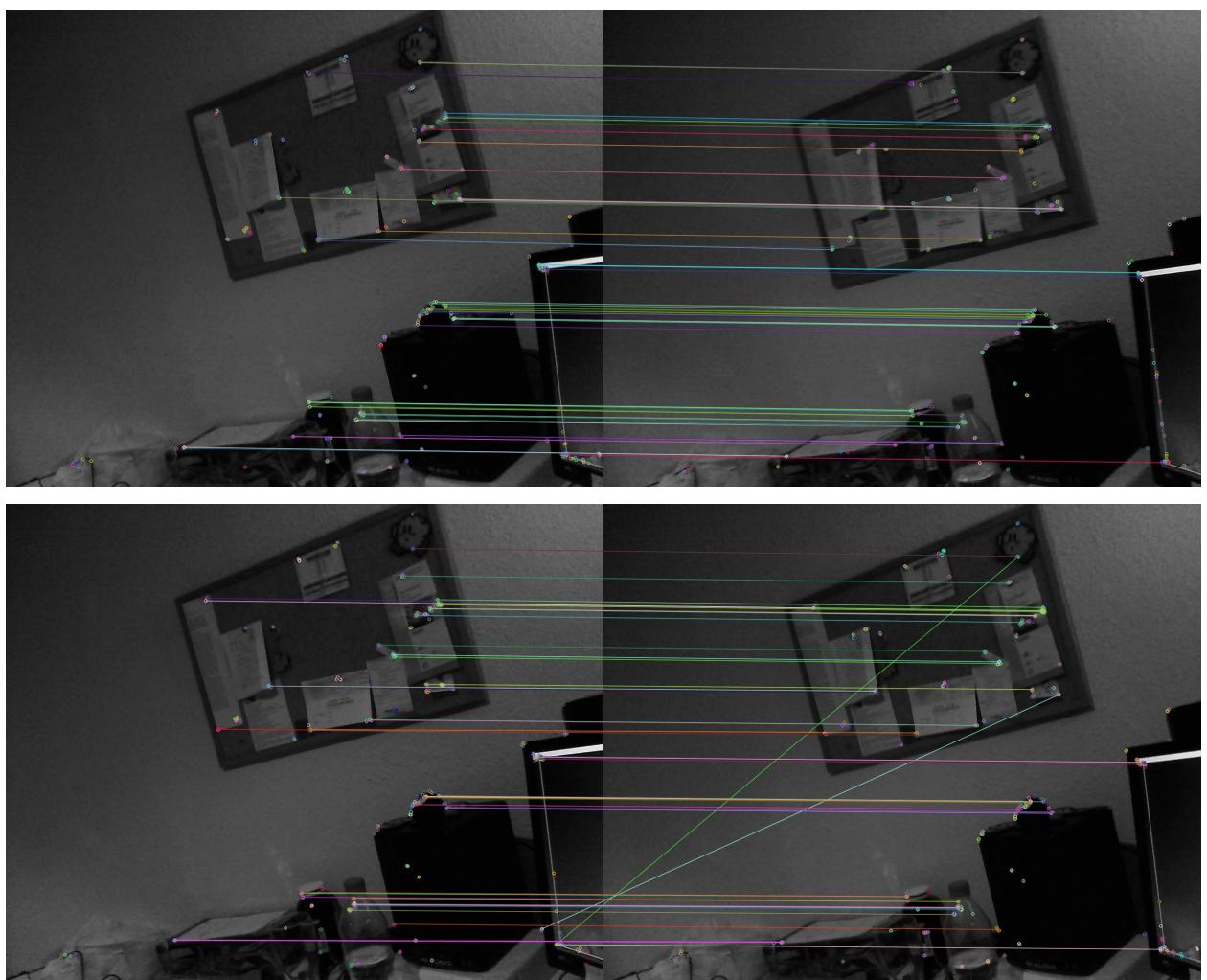
Una vez comprobamos que todo funciona correctamente debemos dejar de usar el rosbag y comenzar a trabajar con la cámara, para cambiar el topic que escucha el Subscriber podemos indicarlo por consola:

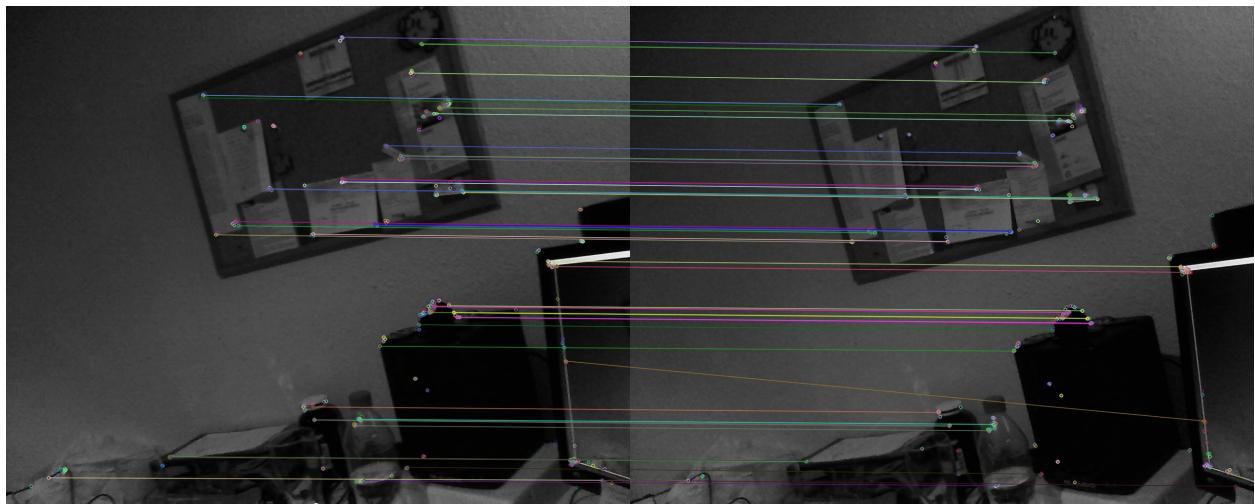
```
optirun rosrun practica1 panorama /camera/rgb/image_color:=/camera/image_raw
```



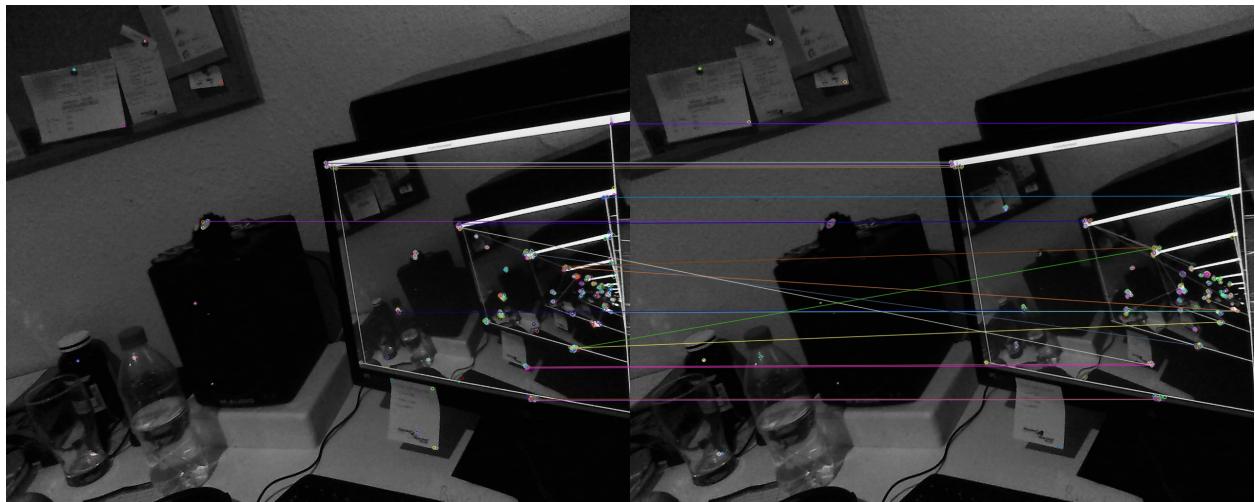


Como vemos recoge las correspondencias sin error aparente

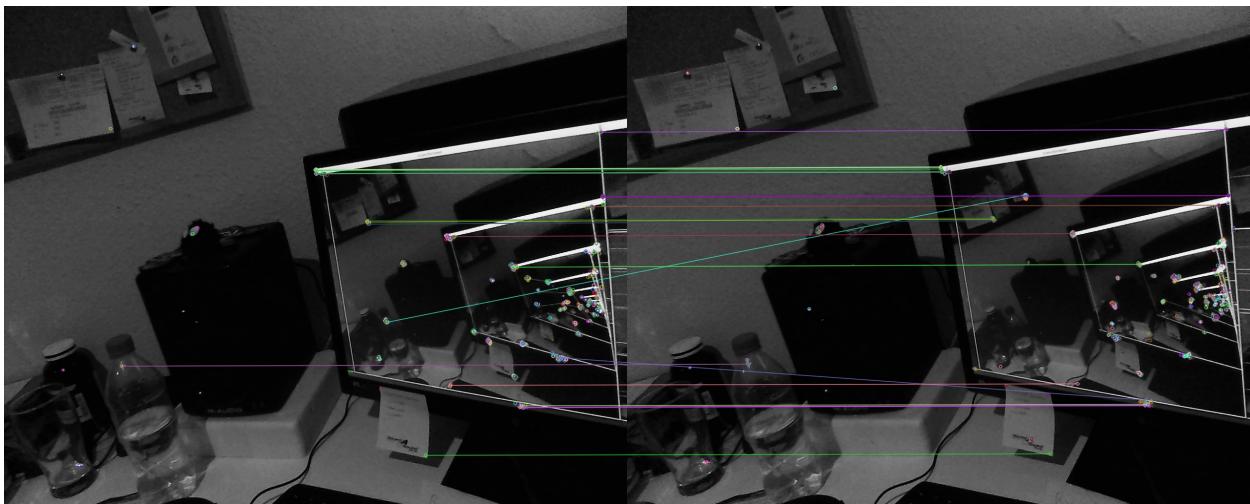




Al complicar el escenario se pueden apreciar ciertos errores. Para comprobar hasta que punto es robusto nuestro sistema enfocamos a la propia pantalla que muestra las correspondencias.





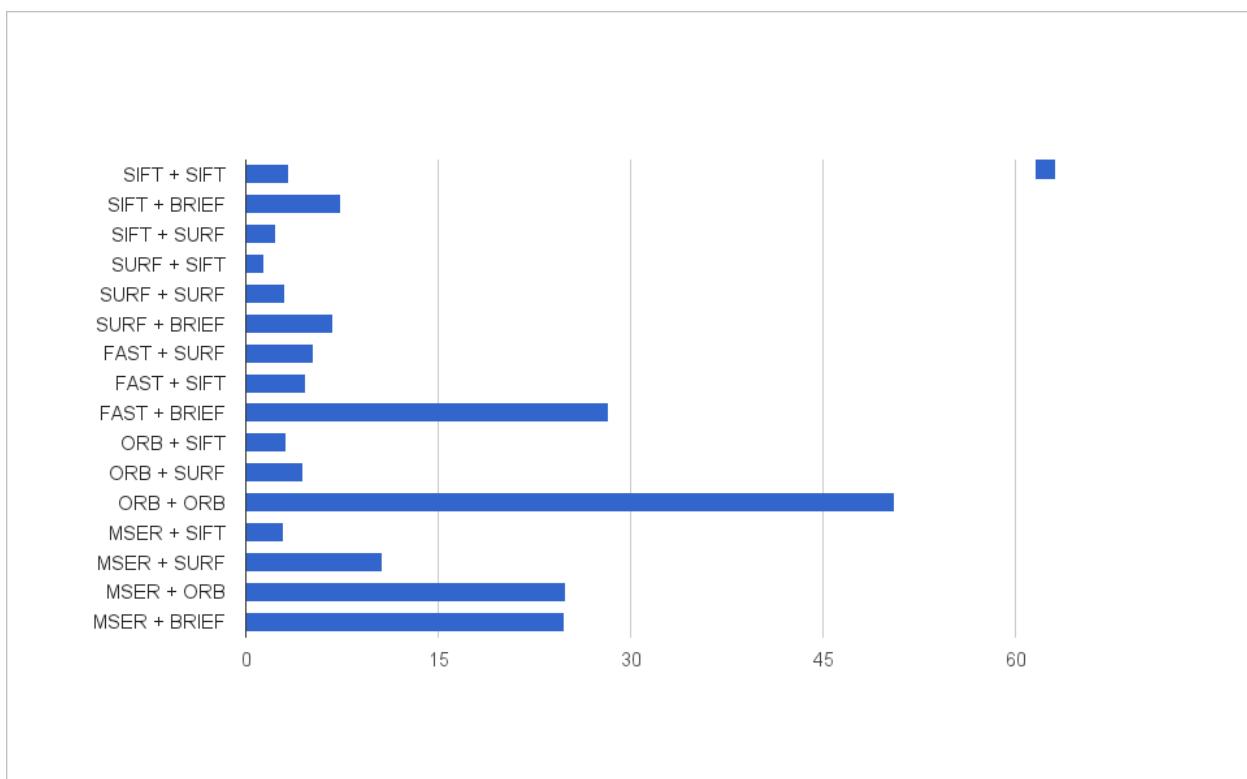


Aún en estas condiciones los resultados son bastante acertados.

Conclusiones y problemas encontrados

Tras la experimentación realizada hemos visto varias diferencias que existen entre los métodos de extracción de características visuales y su posterior búsqueda de correspondencias. Algunos son capaces de encontrar una gran cantidad de *keypoints* y *descriptores* pero no dan buen resultado cuando se trata de hallar las correspondencias entre dos conjuntos de características. Otros pueden darnos buenos resultados en este campo pero a la vez tardan demasiado, lo cual no es viable si pretendemos realizar una mezcla de imágenes en tiempo real o alguna otra tarea que requiera velocidad.

Métodos privativos famosas como pueda ser *SIFT* no dan buenos resultados y *SURF* ha obtenido unas puntuaciones notoriamente bajas. Si debemos elegir, *ORB* se erige como el mejor entre los probados a lo largo de la práctica, su eficiencia y efectividad se ven claramente. Para cerrar retomamos un gráfico dado por una de las comparativas anteriores bastante representativa (*bag3+KNN*), donde se aprecian las diferencias que se han mantenido entre métodos a lo largo de la experimentación.



En cuanto a los problemas encontrados a lo largo del desarrollo:

- Difícil instalación de la herramienta *ROS*, ya que esta solo se encuentra disponible a nivel binario para ciertas distribuciones de Ubuntu, lo cual supuso horas tratando de recompilar el sistema sobre otras distribuciones linux sin ningún resultado satisfactorio. Posteriormente se optó por instalar un sistema operativo Linux apto para *ROS*.
- A nivel de drivers también hubo diversos problemas, para lograr hacer funcionar la GPU Nvidia se deben instalar controladores no oficiales *Bumblebee* los cuales daban problemas sobre ciertas distribuciones antiguas de Ubuntu. Posteriormente se optó por instalar un sistema operativo Linux apto para *ROS* y *Bumblebee*.
- Inicialmente la mera comprensión del código y las sentencias de *ROS* y *OpenCv* ha requerido horas de dedicación y estudio paso a paso desde abajo.
- La falta de un IDE de desarrollo ha influido notablemente en el ritmo productivo ya que herramientas como el sugerir funciones de las librerías incluidas, siendo estas desconocidas, resulta de gran ayuda.
- Los errores arrojados por ciertas funciones de *ROS* y *OpenCv* generalmente hacían referencia a parámetros del core interno o excepciones con las que no estamos familiarizados.
- Como ejemplo de error, algunas combinaciones de métodos daban problemas.

```
vMethods.push_back(make_pair("SIFT", "ORB"));
vMethods.push_back(make_pair("SURF", "ORB"));
```

Salida:

```
OpenCV Error: Assertion failed (dsiz.area() || (inv_scale_x > 0 && inv_scale_y > 0))
in resize, file
/tmp/buildd/ros-hydro-opencv2-2.4.6-3raring-20140130-1757/modules/imgproc/src/imgwarp.c
pp, line 1724 terminate called after throwing an instance of 'cv::Exception' what():
/tmp/buildd/ros-hydro-opencv2-2.4.6-3raring-20140130-1757/modules/imgproc/src/imgwarp.c
pp:1724: error: (-215) dsiz.area() || (inv_scale_x > 0 && inv_scale_y > 0) in function
resize
```