ANEXO A: CÓDIGO C++ UTILIZADO

En este anexo se procede a explicar, de forma superficial, lo que hacen cada una de las funciones que componen el algoritmo.

En primer lugar, veamos el hilo principal de ejecución. Es el que puede encontrarse en el archivo "main.cpp". Este fragmento de código sería el resultante de traducir el diagrama de flujo de la figura 3.1 a código C++. El código se muestra a continuación:

```
#include <stdio.h>
#include "functions.h"
#include <iostream>
#include <ctype.h>
#include <opencv2/opencv.hpp>
#include <cvsba/cvsba.h>
#include <string>
#include <sstream>
#include<stdlib.h>
#include <unistd.h>
#include <ctime>
#include <pcl/common/common_headers.h>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
#include <pcl/visualization/pcl_visualizer.h>
#include <Eigen/Core>
#include <Eigen/LU>
#include <Eigen/Geometry>
#include <opencv2/xfeatures2d.hpp>
#include <sys/stat.h>
#include <sys/time.h>
#include <fstream>
#include <Eigen/StdVector>
#include <opencv2/core/eigen.hpp>
#include "DBoW2.h"
#include <time.h>
#include <stdio.h>
#include <unordered_set>
#include <stdint.h>
#include <algorithm>
#include <iterator>
#include <vector>
#include "g2o/core/sparse_optimizer.h"
#include "g2o/types/icp/types_icp.h"
#include "g2o/config.h"
#include "g2o/core/block_solver.h"
#include "g2o/core/solver.h"
#include "g2o/core/robust_kernel_impl.h"
#include "g2o/core/optimization_algorithm_levenberg.h"
#include "g2o/solvers/cholmod/linear_solver_cholmod.h"
#include "g2o/solvers/dense/linear_solver_dense.h"
#include "g2o/types/sba/types_six_dof_expmap.h"
#include "g2o/solvers/structure_only/structure_only_solver.h"
#include "g2o/stuff/timeutil.h"
#include "g2o/stuff/macros.h"
#include "g2o/stuff/misc.h"
#include "g2o/config.h"
```

```cpp
int main(int argc,char ** argv)
{
    //help
    if(argc<6)
    {
        cout<< "bad input\n";
        cout << "please enter:\n";
        cout << "argv[1]= path to rgb images.
                the images must be called left_i being i the image number\n";
        cout << "argv[2]= number of images to compose the initial map\n";
        cout << "argv[3]= number of dataset images\n";
        cout<<  "argv[4]= path to vocabulary\n";
        cout << "argv[5]= flag to enable local optimization
                (1 to enable, 0 to disable)\n";
        exit(-1);
    }
    //intrinsic camera parameters
    Mat distcoef = (Mat_<float>(1, 5) << 0.2624, -0.9531, -0.0054, 0.0026, 1.1633);
    Mat distor = (Mat_<float>(5, 1) << 0.2624, -0.9531, -0.0054, 0.0026, 1.1633);
    distor.convertTo(distor, CV_64F);
    Mat intrinsic = (Mat_<float>(3, 3) << 517.3, 0., 318.6, 0., 516.5, 255.3, 0., 0., 1.);
    intrinsic.convertTo(intrinsic, CV_64F);
    distcoef.convertTo(distcoef, CV_64F);
    double focal_length=516.9;
    cv::Point2d pp=cv::Point2d(318.6,255.3);
    Eigen::Vector2d principal_point(318.6,255.3);

    //number of images to compose the initial map
    int nImages =  atoi(argv[2]);

    //number of dataset images
    int total_images;
    sscanf(argv[3],"%d",&total_images);

    //flag to enable local optimization
    int use_local_opt;
    sscanf(argv[5],"%d",&use_local_opt);

    //identifier for 3d_point
    int ident=0;

    //threshold to detect points tha are seen in more than img_threshol images
    int img_threshold=3;

    //threshold for the first filter to recject bad matches between features
    double dst_ratio=0.7;

    //variables for the RANSAC algorithm
    double confidence=0.999;
    double reproject_err=1.0;

    //g2o iterations
    int niter=50;

    /*threshold to compare the histograms of the appearance of vocabulary words in
    the image*/
    double dbow2_threshold=0.18;

    //number of keyframes to be used by the local optimization module
    int window_size=20;

    //initial depth for the three-dimensional points of the environment
    double z_plane=1.5;

    //first index of dataset
    int current_frame,last_frame;
    last_frame=0;

    //we need variables to store the last image and the last features & descriptors
    auto pt =ORB::create();
    Mat foto1_u;
    vector<KeyPoint> features1;
    Mat descriptors1;
```

```
//initialize viewer and pointcloud
pcl::visualization::PCLVisualizer viewer("Viewer");
viewer.setBackgroundColor(0.35, 0.35, 0.35);
viewer.initCameraParameters();
pcl::PointCloud<pcl::PointXYZ> cloud;

//custom class to store matching between images
tracking_store tracks;

//load first image who will be the first keyframe
foto1_u = loadImage(argv[1], current_frame, intrinsic, distcoef);
pt->detectAndCompute(foto1_u, Mat(), features1, descriptors1);


current_frame=last_frame;
vector<int> keyframes;
vector<int> valid_points;
keyframes.push_back(current_frame);
tracks.valid_frames.push_back(1);
tracks.frames_id.push_back(current_frame);
current_frame++;
//creation of two-dimensional correspondences
while(current_frame<nImages)
{
    //load new image
    Mat foto2_u = loadImage(argv[1], current_frame, intrinsic, distcoef);
    vector<KeyPoint> features2;
    Mat descriptors2;
    pt->detectAndCompute(foto2_u, Mat(), features2, descriptors2);

    //matches between images
    vector<int> left_index_matches, right_index_matches;
    matchFeatures(features1, descriptors1, features2, descriptors2, left_index_matches,
    right_index_matches,dst_ratio,confidence,reproject_err,focal_length,pp);

    //show matches for debug
    displayMatches(foto1_u, features1, left_index_matches,foto2_u, features2,
    right_index_matches);
    Mat used_features=Mat::zeros(1,int(left_index_matches.size()),CV_64F);

    if(ident>0)
    {
        add_new_projection_for_existent_point(ident,last_frame,current_frame,features1,
                                              features2,left_index_matches,right_index_matches,
                                              used_features,tracks);
    }
    add_new_points_proyections(features1,features2,left_index_matches,right_index_matches,
                               ident,last_frame,current_frame,used_features,tracks);

    //update variables for next iteration
    foto1_u=foto2_u;
    features1=features2;
    descriptors1=descriptors2;
    used_features.release();
    last_frame=current_frame;
    keyframes.push_back(current_frame);
    tracks.valid_frames.push_back(1);
    tracks.frames_id.push_back(current_frame);
    current_frame++;
}
//initial map for the visual odometry system
initial_map_generation(tracks,nImages,img_threshold,ident,focal_length,principal_point,
                       valid_points,keyframes,niter,z_plane);
int last_found=0;
double score=1;
```

```
while(current_frame<total_images || !last_found)
{
    //keyframe detection
    vector<KeyPoint> features2;
    cv::Mat descriptors2;
    cv::Mat foto2_u=loadImage(argv[1],current_frame,intrinsic,distcoef);
    pt->detectAndCompute(foto2_u,cv::Mat(),features2,descriptors2);
    vector<cv::Mat> left_descriptors,right_descriptors;
    changeStructure(descriptors1,left_descriptors);
    changeStructure(descriptors2,right_descriptors);
    score=calculate_score(last_frame,current_frame,left_descriptors,right_descriptors,argv[4]);

    if(score<= dbow2_threshold)
    {
        //keyframe found. Let's match features between the last keyframe and the current keyframe
        vector<Point2f> left_points,right_points;
        vector<int> left_idx,right_idx;
        cv::Mat mask;
        cv::Mat E;
        int res;

        //feature matching with custom filter and RANSAC filter
        matchFeatures_and_compute_essential(foto1_u,foto2_u,last_frame,current_frame,
                                            features1,features2,descriptors1,descriptors2
                                            ,left_points,right_points,left_idx,right_idx,mask,
                                            dst_ratio,E,focal_length,confidence,
                                            reproject_err,pp,tracks);

        /*motion calculation. here we have two possibilities. If the keyframe is usefull
        for the 3d reconstruction we update the camera pose and the 3D points map.
        Other way we discard the current keyframe and we search a new one*/

        double scale=1;
        vector<Point3d> triangulated_points,new_points;
        res=estimate_motion_and_calculate_3d_points(last_frame,current_frame,foto1_u,foto2_u,
                                                    left_points,right_points,left_idx,right_idx,ident,
                                                    intrinsic,E,focal_length,pp,scale,mask,triangulated_points,
                                                    tracks,valid_points,new_points);
        if(res==1)
        {
            score=1;
            last_frame=current_frame;
            foto1_u=foto2_u;
            descriptors1=descriptors2;
            features1=features2;
            last_found=1;
            current_frame++;
        }
        else
        {
            tracks.valid_frames.push_back(0);
            tracks.frames_id.push_back(current_frame);
            last_found=0;
            current_frame++;
        }

        //local optimization
        if(use_local_opt && res)
        {
            local_optimization(window_size,tracks,niter,ident,focal_length,principal_point);
        }
    }
    else
    {
        tracks.valid_frames.push_back(0);
        tracks.frames_id.push_back(current_frame);
        last_found=0;
        current_frame++;
    }
}
```

```
    //graphical representation of the trajectory and output data generation
    mkdir("./lectura_datos", 0777);
    std::ofstream file("./lectura_datos/odometry.txt");
    if (!file.is_open()) return -1;
    cv::Mat last_t,curr_t;
    for (unsigned int i = 0; i < tracks.valid_frames.size(); i++)
    {
        if(tracks.valid_frames[i]==1)
        {
            stringstream sss;
            string name;
            sss << tracks.frames_id[i];
            name = sss.str();
            Eigen::Affine3f cam_pos;
            Eigen::Matrix4d eig_cam_pos=Eigen::Matrix4d::Identity();
            Eigen::Vector3d cam_translation;
            Eigen::Matrix3d cam_rotation;
            cv::Mat cv_cam_rot=cv::Mat::zeros(3,3,CV_64F);
            cv::Mat cv_cam_tras=cv::Mat::zeros(3,1,CV_64F);
            cv_cam_rot=tracks.cam_poses[i].rowRange(0,3).colRange(0,3);
            cv_cam_tras=tracks.cam_poses[i].rowRange(0,3).col(3);
            cv2eigen(cv_cam_rot,cam_rotation);
            cv2eigen(cv_cam_tras,cam_translation);
            eig_cam_pos.block<3,3>(0,0)=cam_rotation;
            eig_cam_pos.block<3,1>(0,3) = cam_translation;
            cam_pos=eig_cam_pos.cast<float>();
            viewer.addCoordinateSystem(0.05, cam_pos, name);
            pcl::PointXYZ textPoint(cam_pos(0,3), cam_pos(1,3), cam_pos(2,3));
            viewer.addText3D(std::to_string(i), textPoint, 0.01, 1, 1, 1, "text_"+std::to_string(i));
            Eigen::Quaternionf q(cam_pos.matrix().block<3,3>(0,0));
            file << tracks.frames_id[i] << " " << cam_pos(0,3) << " " <<  cam_pos(1,3)
                << " " << cam_pos(2,3) << " " << q.x() << " " << q.y() << " " << q.z()
                << " " << q.w() << std::endl;
        }
    }
    file.close();
    for(int j=0;j<ident;j++)
    {
        if(valid_points[j]==1)
        {
            pcl::PointXYZ p(tracks.triangulated_3d_points[j].x,
                            tracks.triangulated_3d_points[j].y,
                            tracks.triangulated_3d_points[j].z);
            cloud.push_back(p);
        }
    }
    viewer.addPointCloud<pcl::PointXYZ>(cloud.makeShared(), "map");
    while (!viewer.wasStopped()) {
        viewer.spin();
    }
    return 0;
}
```

Vamos a describir ahora las funciones y estructuras más relevantes para comprender adecuadamente como funciona nuestro código.

Comencemos por la clase de C++ a la que denominamos *tracking_store*. En esta clase se almacenan las estructuras de datos a las que denominamos “Correspondencias 2D-2D”, “Mapa tridimensional entorno” y “Posiciones del robot” en la figura 3.1. Además, también se utiliza para almacenar los índices de las imágenes que han resultado ser *keyframes*.

Para trabajar con nuestra clase personalizada *tracking_store* definimos un objeto al que denominamos *tracks*.

Esta clase se muestra a continuación:

```
class tracking_store
{

    public:
    //3D points projections
    unordered_map<int,vector<Point2f>> pt_2d;

    //3D points visibility along images;
    unordered_map<int,vector<int>> img_index;

    //match index
    unordered_map<int,vector<int>> match_index;

    //3D points initialization
    unordered_map<int,Eigen::Vector3d> init_guess;

    //map for correspondences between vertex identifier in g2o and point identifier
    unordered_map<int,int> map_to_custom_struct;

    //3D points
    unordered_map<int,cv::Point3d> triangulated_3d_points;

    //valid_frames_for_triangulation
    vector<int> frames_id;
    vector<int> valid_frames;

    //camera poses
    unordered_map<int,cv::Mat> cam_poses;

    ~tracking_store(){}
};
```

Donde:

- *pt_2d* se encarga de almacenar las correspondencias 2D-2D.
- *img_index* almacena para cada característica visual los índices de las imágenes en las que ha sido detectada.
- *match_index* almacena para cada característica visual el índice de la componente del vector descriptor encargado de describir dicha característica visual en cada una de las imágenes.
- *init_guess* se utiliza para almacenar el valor inicial de las coordenadas de los puntos 3D del entorno.
- frames_id almacena los índices de las imágenes que han resultado ser *keyframes*.
- *valid_frames* es un vector que indica si una determinada imagen es *keyframe*. Si la componente i del vector es 1, la imagen i es un *keyframe*, si por el contrario dicha componente es nula, la imagen i no sería un fotograma clave.
- El mapa *triangulated_3d_points* se utiliza para almacenar los puntos tridimensionales del entorno.
- *cam_poses* almacena cada una de las posicione del robot estimadas por nuestro algoritmo.
- *map_to_custom_struct* se utiliza para establecer correspondencias entre el identificador de cada uno de nuestros puntos tridimensionales en las estructuras *pt_2d*, *triangulated_3d_points* e *init_guess* y el identificador que reciben dichos puntos tridimensionales en el grafo utilizado por g2o.

Para la lectura de imágenes utilizamos la siguiente función a la que denominamos *loadImage:*

```
cv::Mat loadImage(std::string _folder, int _number, cv::Mat &_intrinsics, cv::Mat &_coeffs) {
    stringstream ss;
    ss << _folder << "/left_" << _number << ".png";
    std::cout << "Loading image: " << ss.str() << std::endl;
    Mat image = imread(ss.str(), CV_LOAD_IMAGE_COLOR);
    cvtColor(image, image, COLOR_BGR2GRAY);
    cv::Mat image_u;
    undistort(image, image_u, _intrinsics, _coeffs);
    return image_u;
}
```

Esta función recibe como parámetros de entrada la ruta de la carpeta en la que se encuentra almacenada la secuencia de imágenes, el índice de la imagen a leer, los parámetros intrínsecos de la cámara y los coeficientes de distorsión correspondientes al modelo matemático descrito en el apartado 2.3.2.

En primer lugar, la función lee la imagen con la función *imread*, como la imagen es a color se utiliza el *flag CV_LOAD_IMAGE_COLOR.*

Como no nos interesa trabajar constantemente con los tres canales que constituyen la imagen a color, pasamos la imagen a escala de grises combinando los tres canales de color con la función cvtColor de la librería OpenCV.

Acto seguido se utiliza la función undistort de la librería OpenCV. Esta función se encarga de rectificar la imagen eliminando la distorsión existente.

Para la detección y descripción de puntos de interés se han utilizado una serie de funciones de la librería OpenCV. La secuencia de instrucciones que tenemos que utilizar cada vez que queramos obtener características visuales de una imagen sería la siguiente:

- Definir un puntero a la función *create* de la clase *ORB*. Esto nos proporcionaría un detector y descriptor de características visuales de tipo ORB con los parámetros por defecto. Estos parámetros serían los siguientes:
    1. Retener como máximo 500 características visuales.
    2. Una pirámide de imágenes de 8 niveles.
    3. El número de puntos implicados para generar una componente del vector descriptor es 2.
    4. Se utiliza la puntuación Harris para clasificar las características visuales.
- Invocar la función *detectAndCompute*. Esta función nos devolvería un vector de tipo *cv :: Keypoint* en el que se almacenan las características visuales y una matriz en la que una determinada fila i almacena el vector descriptor de la característica visual correspondiente a la componente i del vector citado anteriormente.

Expliquemos ahora como se crean las correspondencias entre características visuales de imágenes diferentes. Para ello hacemos uso de la función a la que hemos denominado *matchFeatures*. Esta función recibe los siguientes parámetros de entrada:

- Los vectores que almacenan las características visuales de ambas imágenes.
- Las matrices que almacenan los vectores descriptores de las características visuales de ambas imágenes.

- Dos vectores de tipo entero que se utilizan para almacenar las componentes de los vectores de tipo *cv :: Keypoint* que identifican a las características visuales que son emparejadas y que además superan los filtros propuestos en el apartado 3.3.
- El umbral para el filtro propuesto en la ecuación 3.1.
- La probabilidad de éxito que se le exige al algoritmo RANSAC encargado de calcular la matriz Esencial.

- La distancia mínima usada por el algoritmo RANSAC para determinar si un punto es parte del modelo estimado.
- Los parámetros intrínsecos de la cámara.

La función es la que se muestra a continuación:

```
void matchFeatures( vector<KeyPoint> &_features1, cv::Mat &_desc1,
                    vector<KeyPoint> &_features2, cv::Mat &_desc2,
                    vector<int> &_ifKeypoints, vector<int> &_jfKeypoints,double dst_ratio,double confidence,double reproject_err,
                    double focal_lenght,cv::Point2d pp){

    Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create("BruteForce-Hamming");
    vector<vector<DMatch>> matches;
    vector<Point2d> source, destination;
    vector<uchar> mask;
    vector<int> i_keypoint, j_keypoint;
    cv::Mat E;
    matcher->knnMatch(_desc1, _desc2, matches, 2);
    for (unsigned int k = 0; k < matches.size(); k++)
    {
        if (matches[k][0].distance < dst_ratio * matches[k][1].distance)
        {
            source.push_back(_features1[matches[k][0].queryIdx].pt);
            destination.push_back(_features2[matches[k][0].trainIdx].pt);
            i_keypoint.push_back(matches[k][0].queryIdx);
            j_keypoint.push_back(matches[k][0].trainIdx);
        }
    }
    E=findEssentialMat(source,destination,focal_lenght,pp,RANSAC,confidence,reproject_err,mask);
    for (unsigned int m = 0; m < mask.size(); m++)
    {
        if (mask[m])
        {
            _ifKeypoints.push_back(i_keypoint[m]);
            _jfKeypoints.push_back(j_keypoint[m]);
        }
    }
}
```

Esta función es bastante fácil de seguir:

En primer lugar, se utilizan las librerías de OpenCV para crear un emparejador de tipo Fuerza Bruta basado en la norma Hamming. A través de la función *knnMatch* se le indica a este emparejador que debe buscar las dos mejores coincidencias para cada característica visual de la primera imagen en la segunda imagen.

Acto seguido se implementa el filtro de la ecuación 3.1.

Con las características visuales que han superado este primer filtro, almacenadas en los vectores *source* y *destination*, se procede a calcular la matriz Esencial a través del algoritmo RANSAC con la ayuda

de la función *findEssentialMat* de la librería OpenCV. A través del vector mask se indican los pares de puntos que se utilizaron para calcular la matriz Esencial o, dicho de otra forma, se indican las características visuales que superan el segundo filtro propuesto en el apartado apartado 3.3.

Cada vez que se genera una nueva correspondencia entre características visuales tenemos que actualizar nuestro objeto *tracks.*

Si se trata de una característica visual cuya primera aparición fue en las imágenes que se consideran en la iteración actual del algoritmo, se utiliza la función *add_new_points_proyections* que se muestra a continuación:

```
void add_new_points_proyections(vector<KeyPoint> _features1,vector<KeyPoint> _features2,
                                vector<int> _left_matches,vector<int> _right_matches,
                                int &_point_identifier,int last_frame,int current_frame,
                                Mat used_features,tracking_store &obj)
{
    for (unsigned int i=0;i<_left_matches.size();i++)
    {
        if(used_features.at<double>(i)==0)
        {
            obj.pt_2d[_point_identifier]=vector<Point2f>();
            obj.pt_2d[_point_identifier].push_back(_features1[_left_matches[i]].pt);
            obj.img_index[_point_identifier]=vector<int>();
            obj.img_index[_point_identifier].push_back(last_frame);
            obj.match_index[_point_identifier]=vector<int>();
            obj.match_index[_point_identifier].push_back(_left_matches[i]);
            obj.pt_2d[_point_identifier].push_back(_features2[_right_matches[i]].pt);
            obj.img_index[_point_identifier].push_back(current_frame);
            obj.match_index[_point_identifier].push_back(_right_matches[i]);
            _point_identifier++;
        }
    }
}
```

Esta función recibe los siguientes parámetros de entrada:

- Las características visuales de dos imágenes consecutivas.
- Dos vectores de tipo entero que se utilizan para almacenar las componentes de los vectores de tipo *cv :: Keypoint* que identifican a las características visuales que son emparejadas y que además superan los filtros propuestos en el apartado 3.3.
- El identificador que recibe la última característica visual detectada en nuestras estructuras de datos personalizadas almacenadas en el objeto *tracks*,
- Los índices de las imágenes en las que se han detectado las características visuales.
- Un vector, al que denominamos *used_features*, que indica que características visuales se han detectado por primera vez. Este vector funciona en paralelo con los vectores de tipo entero mencionados anteriormente. Si consideramos la componente i de cada uno de los vectores enteros, tendremos acceso a un par de características visuales que han superado el emparejamiento robusto. Si esa misma componente i del vector *used_features* es nula, entonces habremos detectado un par de características visuales nuevas.

Para cada característica visual nueva generamos una nueva entrada para cada uno de los *unordered_map* de nuestro objeto *tracks*. En estas nuevas entradas añadiríamos la información que identifica a dicha característica visual en cada una de las imágenes que se proporcionan como entrada a la función. Para despejar dudas, se recomienda al lector de este documento que vea nuevamente el ejemplo simplificado que se propone en la figura 3.1.1. En este ejemplo se mostraba como se van llenando las estructuras de datos de nuestro algoritmo para dos hipotéticas características visuales.

En el caso de que la característica visual no sea nueva, tendríamos que actualizar entradas ya existentes

de los *unordered_map* de nuestro objeto *tracks*. En este caso utilizamos la función a la que hemos denominado *add_new_projection_for_existent_point*. La misión de esta rutina es la misma que la de la función anterior, actualizar nuestras estructuras de datos.

En este caso la función recorre cada uno de los *unordered_map* de nuestro objeto *tracks* y busca características visuales ya existentes. Para acometer esta tarea utiliza la información de cada una de las características visuales de la imagen que es considerada como referencia para la iteración actual, y la compara con la información existente en nuestra estructura de datos. Si encontramos algún identificador en nuestra estructura de datos que almacene tanto las mismas coordenadas 2D que la característica visual considerada, así como el mismo índice de imagen, habremos encontrado una característica visual que superó el emparejamiento robusto en una iteración anterior del algoritmo, es decir, una característica visual ya existente.

Con este identificador accedemos a los *unordered_maps* que almacenan la información de esta característica visual y añadimos la información correspondiente a la nueva imagen que se considera en la iteración actual del algoritmo.

En definitiva, lo que estamos haciendo es considerar la información correspondiente a dos iteraciones consecutivas del algoritmo. Es decir, se consideran tres imágenes consecutivas y se analiza la información obtenida del emparejamiento de características visuales entre las imágenes I-2 e I-1 y entre las imágenes I-1 e I, siendo I la imagen para la iteración actual del algoritmo e I-1 e I-2 las dos imágenes anteriores de nuestro *dataset*. Las características visuales ya existentes serían aquellas que han superado el emparejamiento robusto para las imágenes I-2 e I-1 y que lo han vuelto a superar para las imágenes I-1 e I.

A continuación, se muestra la función descrita en los párrafos anteriores:

```
void add_new_projection_for_existent_point(int _point_ident,int last_frame,int current_frame,
                                           vector<KeyPoint> _features1,vector<KeyPoint> _features2,
                                           vector<int> _left_matches,vector<int> _right_matches,
                                           Mat &used_points,tracking_store &obj)
{
    for(int j=0;j<_point_ident;j++)
    {
        auto search_match=obj.match_index.find(j);
        auto search_img=obj.img_index.find(j);
        if(search_match!=obj.match_index.end() && search_img!=obj.img_index.end())
        {
            auto it_match=search_match->second.end();
            it_match--;
            auto it_img=search_img->second.end();
            it_img--;
            int last_match=*it_match;
            int last_img=*it_img;
            int flag=0;
            for(unsigned int k=0;k<_left_matches.size() && !flag;k++)
            {
                if(_left_matches[k]==last_match && last_img==last_frame)
                {
                    //we add the new projection for the same 3d point
                    obj.pt_2d[j].push_back(_features2[_right_matches[k]].pt);
                    obj.img_index[j].push_back(current_frame);
                    obj.match_index[j].push_back(_right_matches[k]);
                    used_points.at<double>(k)=1;
                    flag=1;
                }
            }
        }
    }
}
```

Ya hemos descrito las herramientas necesarias para obtener un conjunto de correspondencias 2D-2D. Situémonos ahora en el primer bucle while de nuestro hilo principal de ejecución. En este bucle se utilizan las funciones descritas anteriormente para crear un conjunto de correspondencias 2D-2D para

un conjunto de *nImages* imágenes, siendo *nImages* el número de imágenes utilizadas para la creación del mapa inicial del entorno.

Al salir de este bucle ya hemos generado este conjunto de correspondencias bidimensionales y estamos en disposición de acceder a la rutina *initial_map_generation* encargada de crear el mencionado mapa tridimensional del entorno.

Esta rutina se muestra a continuación:

```
void initial_map_generation(tracking_store &obj,int nImages,int img_threshold,int &ident,double focal_length,
                            Eigen::Vector2d principal_point,vector<int> &valid_points,vector<int> &keyframes,
                            int niter,double z_plane)
{
    g2o::SparseOptimizer optimizer;
    optimizer.setVerbose(false);

    std::unique_ptr<g2o::BlockSolver_6_3::LinearSolverType> linearSolver;

    linearSolver = g2o::make_unique<g2o::LinearSolverCholmod<g2o::BlockSolver_6_3::PoseMatrixType>>();

    g2o::OptimizationAlgorithmLevenberg* solver =
    new g2o::OptimizationAlgorithmLevenberg(g2o::make_unique<g2o::BlockSolver_6_3>(std::move(linearSolver)));

    optimizer.setAlgorithm(solver);
    //filter to reject points that are not visible in more than 3 images
    int remaining_points=0;
    delete_invalid_points(img_threshold,ident,remaining_points,valid_points,obj);
    //we add camera vertices to optimizer
    vector<g2o::SE3Quat,Eigen::aligned_allocator<g2o::SE3Quat> > camera_poses;
    g2o::CameraParameters * cam_params = new g2o::CameraParameters (focal_length, principal_point, 0.);
    cam_params->setId(0);
    optimizer.addParameter(cam_params);
    int vertex_id=0;
    for(int i=0;i<nImages;i++)
    {
        g2o::VertexSE3Expmap * v_se3=new g2o::VertexSE3Expmap();
        g2o::SE3Quat pose;
        if(i==0)
        {
            Eigen::Matrix4d poseMatrix=Eigen::Matrix4d::Identity();
            Eigen::Quaterniond qb;
            Eigen::Vector3d tb;
            get_rt_from_t(qb,tb,poseMatrix);
            pose=g2o::SE3Quat(qb,tb);
            v_se3->setId(vertex_id);
            v_se3->setEstimate(pose);
            v_se3->setFixed(true);
            optimizer.addVertex(v_se3);
            camera_poses.push_back(pose);
            vertex_id++;
        }
        else
        {
            pose=camera_poses[0];
            v_se3->setId(vertex_id);
            v_se3->setEstimate(pose);
            optimizer.addVertex(v_se3);
            camera_poses.push_back(pose);
            vertex_id++;
        }
    }
```

```
    //initial guess for 3d points
    int point_id=vertex_id;
    initial_guess_for_3d_points(ident,valid_points,principal_point,focal_length,obj,z_plane);
    //we update g2o's variables

    for(int j=0;j<ident;j++)
    {
        if(valid_points[j]==1)
        {
            Eigen::Vector3d init_guess;
            extract_values(j,init_guess,obj);
            set_correspondence(point_id,j,obj);
            g2o::VertexSBAPointXYZ * v_p= new g2o::VertexSBAPointXYZ();
            v_p->setId(point_id);
            v_p->setMarginalized(true);
            v_p->setEstimate(init_guess);
            optimizer.addVertex(v_p);
            vector<Point2f> aux_pt;
            vector<int> aux_im;
            extract_values(j,aux_pt,aux_im,obj);
            //we search point j on image i
            for(unsigned int p=0;p<aux_im.size();p++)
            {
                //we add the edge connecting the vertex of camera position and the vertex point
                Eigen::Vector2d measurement(aux_pt[p].x,aux_pt[p].y);
                g2o::EdgeProjectXYZ2UV * e= new g2o::EdgeProjectXYZ2UV();
                e->setVertex(0, dynamic_cast<g2o::OptimizableGraph::Vertex*>(v_p));
                for (unsigned int k=0;k<keyframes.size();k++)
                {
                    if(aux_im[p]==keyframes[k])
                    {
                        e->setVertex(1,
                        dynamic_cast<g2o::OptimizableGraph::Vertex*>(optimizer.vertices().find(k)->second));
                    }
                }
                e->setMeasurement(measurement);
                e->information() = Eigen::Matrix2d::Identity();
                e->setParameterId(0, 0);
                optimizer.addEdge(e);
            }
            point_id++;
        }
    }
    optimizer.initializeOptimization();
    optimizer.setVerbose(true);
    optimizer.optimize(niter);
    optimizer.save("test.g2o");

    for (unsigned int i = 0; i < keyframes.size(); i++)
    {
        Eigen::Affine3f cam_pos;
        g2o::SE3Quat updated_pose;
        Eigen::Matrix4f eig_cam_pos=Eigen::Matrix4f::Identity();
        Eigen::Quaterniond cam_quat;
        Eigen::Vector3d cam_translation;
        g2o::HyperGraph::VertexIDMap::iterator pose_it= optimizer.vertices().find(i);
        g2o::VertexSE3Expmap * v_se3= dynamic_cast< g2o::VertexSE3Expmap * >(pose_it->second);
        updated_pose=v_se3->estimate();
        cam_translation=updated_pose.translation();
        cam_quat=updated_pose.rotation();
        eig_cam_pos.block<3,3>(0,0) = cam_quat.matrix().cast<float>();
        eig_cam_pos.block<3,1>(0,3) = cam_translation.cast<float>();
        cam_pos=eig_cam_pos.inverse();
        Eigen::Quaternionf q(cam_pos.matrix().block<3,3>(0,0));
        Eigen::Matrix4d eig_inv=eig_cam_pos.inverse().cast<double>();
        cv::Mat cam_pos_cv=eigentocv(eig_inv);
        obj.cam_poses[i]=cam_pos_cv;
    }
    for(int j=0;j<remaining_points;j++)
    {
        g2o::HyperGraph::VertexIDMap::iterator point_it= optimizer.vertices().find(vertex_id+j);
        g2o::VertexSBAPointXYZ * v_p= dynamic_cast< g2o::VertexSBAPointXYZ * > (point_it->second);
        Eigen::Vector3d p_aux=v_p->estimate();
        cv::Point3d point_cv;
        point_cv.x=p_aux[0];
        point_cv.y=p_aux[1];
        point_cv.z=p_aux[2];
        fill_new_3d_points(vertex_id+j,point_cv,obj);
    }
    optimizer.clear();
}
```

Vayamos paso a paso. En primer lugar, se utilizan una serie de funciones de la librería g2o. No aporta información útil que hablemos sobre ellas. Lo único que es interesante saber es que definen el tipo de optimizador que vamos a utilizar para la creación de nuestro mapa inicial. Este optimizador trata de minimizar una función de coste como la que se describe en la ecuación 3.5.

A continuación, se utiliza la función *delete_invalid_points* que nos permite identificar qué características visuales de las que tenemos almacenadas en nuestra estructura de datos son visibles en al menos tres imágenes. Esta función presenta la siguiente forma:

```
void delete_invalid_points(int img_threshold,int total_points,int &remaining_points,vector<int> &valid_points,tracking_store &obj)
{
    for(int i=0;i<total_points;i++)
    {
        auto search_point=obj.pt_2d.find(i);
        if(search_point!=obj.pt_2d.end())
        {
            int dimension= search_point->second.size();
            if(dimension>=img_threshold)
            {
                valid_points.push_back(1);
                remaining_points++;
            }
            else
            {
                valid_points.push_back(0);
            }
        }
    }
}
```

Donde *img_threshold* sería el número mínimo de imágenes en las que debe estar presente una determinada característica visual, que en nuestro caso es 3, y *total_points* es el número total de características visuales diferentes que tenemos almacenadas en nuestra estructura de datos. En el vector *valid_points* almacenamos los identificadores de las características visuales de nuestra estructura de datos que han superado este criterio.

Una vez realizada esta tarea se empieza a añadir la información al optimizador tal y como se explicó en el apartado 3.1.

Primero se inicializan todos los vértices correspondientes a la posición del robot móvil en la misma posición. Estos vértices son los que aparecen bajo el nombre *VertexSE3Expmap*. A estos vértices hay que proporcionarles como valor una posición. Para expresar estas posiciones la librería g2o utiliza la clase *SE3Quat*. Esta clase está constituida por dos componentes. La primera es un cuaternión que representa la orientación del robot móvil y la segunda es un vector de tres componentes que contiene las coordenadas 3D de dicha posición.

Acto seguido se inicializan los vértices correspondientes a los puntos 3D del entorno. Para ello se utiliza la función *initial_guess_for_3d_points*. La tarea de esta rutina es utilizar el conjunto de ecuaciones 3.6 para inicializar los puntos tridimensionales del entorno. Los valores obtenidos se almacenan en el *unordered_map init_guess* que utilizaremos para rellenar adecuadamente los vértices del optimizador correspondientes a los puntos 3D del entorno.

Para que esta función actúe adecuadamente se le deben proporcionar los parámetros intrínsecos de la cámara y la profundidad del plano x-y en el que se pretenden inicializar todos los puntos 3D del entorno. Como siempre también hay que proporcionarle acceso a nuestro objeto *tracks* para que pueda almacenar los valores obtenidos en el *unordered_map* correspondiente.

Esta rutina presenta las siguientes líneas de código:

```
void initial_guess_for_3d_points(int total_points,vector<int> valid_points,Eigen::Vector2d principal_point,
                                 double focal_length,tracking_store &obj,double z_plane)
{
    for(int i=0;i<total_points;i++)
    {
        auto search_point=obj.pt_2d.find(i);
        if(search_point!=obj.pt_2d.end())
        {
            if(valid_points[i]==1)
            {
                vector<Point2f> aux=obj.pt_2d[i];
                double dimension=aux.size();
                double  z=z_plane; //initial z
                Eigen::Vector3d init_guess_aux;
                init_guess_aux << 0.,0.,0.;
                Eigen::Vector3d value;
                for (unsigned int j=0;j<aux.size();j++)
                {
                    init_guess_aux[0]+=(((double)search_point->second[j].x - principal_point[0])/focal_length)*z;
                    init_guess_aux[1]+=(((double)search_point->second[j].y - principal_point[1])/focal_length)*z;
                    init_guess_aux[2]+=z;
                }
                value[0]=init_guess_aux[0]/dimension;
                value[1]=init_guess_aux[1]/dimension;
                value[2]=init_guess_aux[2]/dimension;
                obj.init_guess[i]=value;
            }
        }
    }
}
```

Una vez inicializados los puntos 3D del entorno y las posiciones de las cámaras tan solo tenemos que definir las aristas o ejes que unen estos dos tipos de vértices. Tras la rutina *initial_guess_for_3d_points* se proponen una serie de bucles *for* anidados un tanto sofisticados. Veamos un poco el funcionamiento.

Para cada punto tridimensional inicializado a través del *tracking* de una determinada característica visual que superó el criterio de ser detectada en al menos tres imágenes hacemos lo siguiente:

- Recuperamos el valor de la inicialización de los puntos 3D del mapa *initial_guess_for_3d_points* a través de la siguiente función:

```
int extract_values(int ident,Eigen::Vector3d &xyz_coordinates,tracking_store &obj)
{
    auto search_value=obj.init_guess.find(ident);
    if(search_value !=obj.init_guess.end())
    {
        xyz_coordinates=obj.init_guess[ident];
        return 1;
    }
    else
    {
        return 0;
    }
}
```

  Agregamos este valor al optimizador. Los puntos 3D del entorno son vértices que en nuestro optimizador se almacenan en una clase de C++ que se conoce como *VertexSBAPointXYZ*.

- Establecemos una correspondencia entre el identificador que tiene este punto 3D en nuestro mapa y el identificador que tendrá en el optimizador a través de la siguiente rutina:

```
void set_correspondence(int map_ident,int structure_ident,tracking_store &obj)
{
    obj.map_to_custom_struct[map_ident]=structure_ident;
}
```

- Buscamos en que imágenes es visible este punto 3D. Por cada imagen en la que este punto sea visible se genera un eje de tipo *EdgeProjectXYZ2UV* para nuestro optimizador. Esta arista relaciona la posición de nuestro punto 3D con la posición en la que estaba el robot cuando capturó la imagen en la que el punto 3D es observable.

  Recordemos que cada punto 3D tiene el mismo identificador que el vector de coordenadas 2D que representa a este punto 3D en cada una de las imágenes en las que es visible. Por tanto, solo tenemos que recuperar este vector del *unordered_map* pt_2d a través de la siguiente rutina y establecer un nuevo eje por cada componente que tenga dicho vector:

```
int extract_values(int ident,vector<Point2f> &projections,vector<int> &imgs,tracking_store &obj)
{
    auto search_value=obj.pt_2d.find(ident);
    if(search_value !=obj.pt_2d.end())
    {
        projections=obj.pt_2d[ident];
        imgs=obj.img_index[ident];
        return 1;
    }
    else
    {
        return 0;
    }

}
```

En este punto ya hemos salido de los bucles *for* anidados y procedemos a ejecutar el optimizador con la función de g2o *optimize* a la que se le proporcionan el número de iteraciones a realizar por el optimizador.

El resto de las líneas de código de la rutina *initial_map_generation* se emplean para actualizar las estructuras de datos en las que se almacenan las posiciones del robot y las posiciones de los puntos 3D del entorno, es decir, los *unordered_maps cam_poses* y *triangulated_3d_points.* Aparecen dos funciones personalizadas que se muestran a continuación. La primera de ellas se encarga de almacenar los puntos 3D en el objeto *tracks.* La segunda se utiliza para expresar matrices de la librería Eigen en términos de matrices de la librería OpenCV.

```
void fill_new_3d_points(int map_vertex,cv::Point3d pt,tracking_store &obj)
{
    int custom_structure_id;
    custom_structure_id=obj.map_to_custom_struct[map_vertex];
    obj.triangulated_3d_points[custom_structure_id]=pt;

}
```

```
cv::Mat eigentocv(Eigen::Matrix4d &mat_eigen)
{
    cv::Mat inter=cv::Mat::zeros(4,4,CV_64F);
    inter.at<double>(0,0)=mat_eigen(0,0);
    inter.at<double>(0,1)=mat_eigen(0,1);
    inter.at<double>(0,2)=mat_eigen(0,2);
    inter.at<double>(0,3)=mat_eigen(0,3);
    inter.at<double>(1,0)=mat_eigen(1,0);
    inter.at<double>(1,1)=mat_eigen(1,1);
    inter.at<double>(1,2)=mat_eigen(1,2);
    inter.at<double>(1,3)=mat_eigen(1,3);
    inter.at<double>(2,0)=mat_eigen(2,0);
    inter.at<double>(2,1)=mat_eigen(2,1);
    inter.at<double>(2,2)=mat_eigen(2,2);
    inter.at<double>(2,3)=mat_eigen(2,3);
    inter.at<double>(3,0)=mat_eigen(3,0);
    inter.at<double>(3,1)=mat_eigen(3,1);
    inter.at<double>(3,2)=mat_eigen(3,2);
    inter.at<double>(3,3)=mat_eigen(3,3);
    return inter;
}
```

Una vez creado el mapa inicial del entorno volvemos nuevamente al hilo principal de ejecución de nuestro algoritmo. Nos encontramos en la posición en la que el robot ha capturado la última imagen que se ha utilizado para la creación del mapa inicial y tomamos esta posición como referencia, puesto que ya es conocida tras la ejecución del optimizador.

La tarea ahora consiste en buscar un nuevo *keyframe* que podamos utilizar para estimar el movimiento relativo. Entramos en un nuevo bucle *while* que se repetirá mientras el algoritmo siga recibiendo imágenes del dataset. En primer lugar, calculamos las características visuales y los descriptores de la imagen que se recibe en la iteración actual. Esta tarea se realiza a través de la librería OpenCV tal y como se ha descrito anteriormente.

Posteriormente accedemos al diccionario de palabras visuales que se proporciona en el repositorio mencionado al comienzo del anexo. Este diccionario recibe el nombre de *small_voc.yml.gz* en dicho repositorio. Con la siguiente rutina, constituida principalmente por funciones de la librería de C++ DBoW2, obtenemos una puntuación de comparar el histograma de aparición de palabras del vocabulario en la imagen de referencia con el histograma de aparición de palabras del vocabulario en la imagen candidata a ser *keyframe*:

```
double calculate_score(int last_frame,int current_frame,const vector<cv::Mat> &descriptors_left,
                       const vector<cv::Mat> &descriptors_right,string path)
{
    // load the vocabulary from disk
    OrbVocabulary voc(path);
    cout << "Matching images against themselves (0 low, 1 high): " << endl;
    BowVector v1, v2;
    voc.transform(descriptors_left, v1);
    voc.transform(descriptors_right, v2);
    double score = voc.score(v1, v2);
    cout << "Image " << last_frame << " vs Image " << current_frame << ": " << score << endl;
    return score;
}
```

Si la puntuación es inferior a un determinado umbral consideramos que la imagen que se recibe en la iteración actual del algoritmo es un fotograma clave y continuamos con esta iteración. En caso contrario pasamos a la siguiente iteración con la idea de volver a buscar un *keyframe*.

Disponemos de la siguiente función para adaptar los vectores descriptores de características visuales a un formato adecuado para trabajar con la librería DBoW2:

```
void changeStructure(const cv::Mat &plain, vector<cv::Mat> &out)
{
  out.resize(plain.rows);

  for(int i = 0; i < plain.rows; ++i)
  {
    out[i] = plain.row(i);
  }
}
```

Para seguir con la explicación del algoritmo supongamos que la imagen candidata ha resultado ser un fotograma clave y que, por tanto, seguimos en la misma iteración. La siguiente tarea sería realizar el emparejamiento robusto de características visuales entre la imagen de referencia y la imagen actual y calcular la matriz Esencial. Para ello utilizamos la siguiente rutina:

```
void matchFeatures_and_compute_essential(cv::Mat &img1,cv::Mat &img2, int last_frame,int current_frame,
                                         vector<KeyPoint> _features1,vector<KeyPoint> _features2,
                                         cv::Mat _desc1,cv::Mat _desc2,
                                         vector<Point2f> &corresponding_left,vector<Point2f> &corresponding_right,
                                         vector<int> &left_index,vector<int> &right_index,
                                         cv::Mat &mask,double dst_ratio,cv::Mat &E,
                                         double focal_lenght,double confidence,double reproject_err,Point2d pp,tracking_store &obj)
{
    Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create("BruteForce-Hamming");
    vector<vector<DMatch>> matches;
    matcher->knnMatch( _desc1, _desc2, matches, 2);
    displayMatches(img1, _features2,img2, _features2);
    for (unsigned int k = 0; k < matches.size(); k++)
    {
        if (matches[k][0].distance < dst_ratio * matches[k][1].distance)
        {
            corresponding_left.push_back( _features1[matches[k][0].queryIdx].pt);
            corresponding_right.push_back( _features2[matches[k][0].trainIdx].pt);
            left_index.push_back(matches[k][0].queryIdx);
            right_index.push_back(matches[k][0].trainIdx);
        }
    }
    E=findEssentialMat(corresponding_left,corresponding_right,focal_lenght,pp,RANSAC,confidence,reproject_err,mask);
}
```

Esta rutina es muy parecida a la función *matchFeatures* ya descrita anteriormente, así que no entraremos en más detalles sobre ella.

Una vez realizado el emparejamiento de características visuales y obtenida la matriz Esencial, accedemos a la rutina *estimate_motion_and_calculate_3d_points* que se muestra a continuación:

```cpp
int estimate_motion_and_calculate_3d_points(int last_frame,int current_frame,cv::Mat &img1,cv::Mat &img2,
                                            vector<Point2f> points_left,vector<Point2f> points_right,
                                            vector<int> index_left,vector<int> index_right,int &ident,
                                            cv::Mat &intrinsic,cv::Mat &E,double focal_lenght,Point2d pp,
                                            double scale,cv::Mat &mask,vector<Point3d> &pts3d,tracking_store &obj,vector<int> &valid_points,
                                            vector<Point3d> &new_points)
{
    vector<Point2d> triangulation_points_left,triangulation_points_right;
    vector<int> inliers_recover_left,inliers_recover_right;
    Mat R,t;
    if((int)points_left.size() < 8)
    {
        return 0;
    }
    recoverPose(E,points_left,points_right,R,t,focal_lenght,pp,mask);
    if(E.cols!=3 || E.rows!=3)
    {
        return 0;
    }
    for(int i=0;i<mask.rows;i++)
    {
        if(mask.at<unsigned char>(i))
        {
            triangulation_points_left.push_back(Point2d((double)points_left[i].x,(double)points_left[i].y));
            triangulation_points_right.push_back(Point2d((double)points_right[i].x,(double)points_right[i].y));
            inliers_recover_left.push_back(index_left[i]);
            inliers_recover_right.push_back(index_right[i]);
        }
    }
    displayMatches(img1,triangulation_points_left,img2,triangulation_points_right);
    //do tracking
    cv::Mat last_pose=obj.cam_poses[last_frame];
    cv::Mat curr_rel_motion=generate_4x4_transformation(R,t);
    cv::Mat new_camera_pose=last_pose*(curr_rel_motion.inv());
    relative_triangulation(triangulation_points_left,triangulation_points_right,intrinsic,R,t,pts3d);
    //map update && store camera_pose
    vector<int> identifiers,used_features;
    search_existent_points(ident,last_frame,inliers_recover_left,used_features,identifiers,obj);
    vector<Point3d> existent_3d,corresponding_3d;
    for(unsigned int i=0;i<used_features.size();i++)
    {
        if(used_features[i]==1)
        {
            existent_3d.push_back(obj.triangulated_3d_points[identifiers[i]]);
            corresponding_3d.push_back(pts3d[i]);
        }
    }
    update_scale(existent_3d,corresponding_3d,scale);
    t*=scale;
    relative_triangulation(triangulation_points_left,triangulation_points_right,intrinsic,R,t,pts3d);
    vector<Point3d> pts3d_in_world,new_points_for_cloud;
    for(unsigned int i=0;i<pts3d.size();i++)
    {
        cv::Point3d aux_point=change_points_to_other_ref_system(pts3d[i],last_pose);
        pts3d_in_world.push_back(aux_point);
    }
    curr_rel_motion=generate_4x4_transformation(R,t);
    new_camera_pose=last_pose*(curr_rel_motion.inv());
    obj.cam_poses[current_frame]=new_camera_pose;
    for(unsigned int i=0;i<used_features.size();i++)
    {
        if(used_features[i]==0)
        {
            add_new_3d_point(last_frame,current_frame,pts3d_in_world[i],inliers_recover_left[i],inliers_recover_right[i],
            triangulation_points_left[i],triangulation_points_right[i],ident,obj);
            valid_points.push_back(1);
            new_points_for_cloud.push_back(pts3d_in_world[i]);
        }
        else
        {
            update_3d_point(obj,identifiers[i],inliers_recover_right[i],current_frame,triangulation_points_right[i],pts3d_in_world[i]);
        }
    }
    obj.valid_frames.push_back(1);
    obj.frames_id.push_back(current_frame);
    return 1;
}
```

Vayamos paso a paso. La función recibe como parámetros de entrada los índices de los *keyframes* que van a intervenir en el proceso de estimación de la trayectoria, las parejas de características visuales que han superado el emparejamiento robusto, la matriz Esencial y los parámetros intrínsecos de la cámara. Además, también se le proporciona acceso al objeto *tracks* para que añada la nueva posición del robot y las nuevas correspondencias 2D-2D.

En primer lugar, se comprueba que haya suficientes puntos para obtener la rotación y traslación relativas entre los *keyframes* actual y de referencia. Si este número es inferior a 8 se descarta el *keyframe*, la función devuelve un 0 y se pasa a la siguiente iteración del algoritmo.

El siguiente paso es obtener las mencionadas rotación y traslación relativas a través de la función *recoverPose* de la librería OpenCV. La rotación se almacena en la matriz a la que denominamos R y el vector de traslación en la variable t.

Esta última función nos devuelve una máscara que indica las parejas de puntos que se han utilizado para obtener la rotación y la traslación. Utilizamos esta máscara para construir los vectores *triangulation_points_left* y *triangulation_points_right* en los que almacenamos los emparejamientos útiles.

Por motivos de depuración se muestran por pantalla dichos emparejamientos. Para ello utilizamos la siguiente función:

```
void displayMatches(cv::Mat &_img1, std::vector<cv::Point2d> &_features1,cv::Mat &_img2, std::vector<cv::Point2d> &_features2)
{
    cv::Mat display;
    cv::hconcat(_img1, _img2, display);
    cv::cvtColor(display, display, CV_GRAY2BGR);
    for(unsigned int i = 0; i < _features1.size(); i++)
    {
        auto p1 = _features1[i];
        auto p2 = _features2[i] + cv::Point2d(_img1.cols, 0);
        cv::circle(display, p1, 2, cv::Scalar(0,255,0),2);
        cv::circle(display, p2, 2, cv::Scalar(0,255,0),2);
        cv::line(display,p1, p2, cv::Scalar(0,255,0),1);
    }
    cv::imshow("display", display);
    cv::waitKey(3);
}
```

La siguiente tarea es realizar la 1º triangulación para obtener puntos 3D. Esto último nos proporcionaría unos puntos 3D sujetos al problema de escala. Para ello utilizamos la siguiente rutina:

```
void relative_triangulation(vector<Point2d> triang_left,vector<Point2d> triang_right,cv::Mat intrinsic,
cv::Mat R_2_to_1,cv::Mat t_2_to_1,vector<Point3d> &pts3d)
{
    vector<Point2d> normalized_left,normalized_right;
    for(unsigned int i=0;i<triang_left.size();i++)
    {
        cv::Point2d left,right;
        pixel_to_cam_plane(triang_left[i],left,intrinsic);
        normalized_left.push_back(left);
        pixel_to_cam_plane(triang_right[i],right,intrinsic);
        normalized_right.push_back(right);
    }
    cv::Mat left_project_mat=cv::Mat::eye(3,4,CV_64F);
    cv::Mat right_project_mat=generate_projection_matrix(R_2_to_1,t_2_to_1);
    cv::Mat point3d_homo;
    triangulatePoints(left_project_mat,right_project_mat,normalized_left,normalized_right,point3d_homo);
    pts3d.clear();
    for( int i=0;i<point3d_homo.cols;i++)
    {
        Point3d aux;
        aux.x=(point3d_homo.col(i).at<double>(0)/point3d_homo.col(i).at<double>(3));
        aux.y=(point3d_homo.col(i).at<double>(1)/point3d_homo.col(i).at<double>(3));
        aux.z=(point3d_homo.col(i).at<double>(2)/point3d_homo.col(i).at<double>(3));
        pts3d.push_back(aux);
    }
}
```

La función recibe los emparejamientos que fueron utilizados para calcular la rotación y la traslación relativas, las propias rotación y traslación relativas y los parámetros intrínsecos de la cámara. Como resultado nos devuelve los puntos 3D del entorno sujetos al problema de escala.

Lo primero que hace esta función es utilizar los parámetros intrínsecos de la cámara para pasar las coordenadas de las características visuales emparejadas a un plano de cámara normalizado. Para ello se apoya en la función auxiliar *pixel_to_cam_plane* que se muestra a continuación:

```
void pixel_to_cam_plane(Point2d &pixel_plane,Point2d &cam_plane,cv::Mat &intrinsic)
{
    cam_plane.x=(pixel_plane.x-intrinsic.at<double>(0,2))/intrinsic.at<double>(0,0);
    cam_plane.y=(pixel_plane.y-intrinsic.at<double>(1,2))/intrinsic.at<double>(1,1);
}
```

En estas condiciones la matriz de proyección de la cámara que se sitúa en la posición del *keyframe* actual vendría dada por la matriz 3x4 que se construye con la traslación y rotación relativas que ya conocemos. Dicho esto, utilizamos la función *generate_projection_matrix* que se muestra a continuación para construir dicha matriz 3x4:

```
cv::Mat generate_projection_matrix(cv::Mat &R,cv::Mat &t)
{
    cv::Mat inter;
    hconcat(R,t,inter);
    return inter;
}
```

Al estar realizando una triangulación relativa los puntos 3D estarían expresados con respecto al sistema de coordenadas de la posición del *keyframe* de referencia. La matriz de proyección de este *keyframe* estaría constituida por una matriz de rotación 3x3 igual a la matriz identidad y un vector de traslación 3x1 cuyas componentes son nulas.

Una vez conocemos las matrices de proyección de las cámaras y las parejas de características visuales que se van a utilizar para la reconstrucción de puntos 3D, tan solo tenemos que invocar la función *triangulatePoints* de la librería OpenCV. Esta función nos devuelve los puntos 3D en coordenadas homogéneas. Una vez pasamos de coordenadas homogéneas a coordenadas normales la función encargada de realizar esta primera triangulación llega a su fin.

El siguiente paso para estimar la nueva posición es calcular el factor de escala relativa. Para ello se empieza por generar una serie de correspondencias 3D-3D entre los puntos del mapa del entorno almacenados en nuestro objeto *tracks* y los puntos 3D sujetos al problema de escala que acabamos de obtener. Esta tarea es llevada a cabo por la siguiente rutina:

```
void search_existent_points(int ident,int last_frame,vector<int> ref_match_idx,vector<int> &used_points,
                            vector<int> &identifiers,tracking_store obj)
{
    for(unsigned int i=0;i<ref_match_idx.size();i++)
    {
        int stop_flag=0;
        for(int j=0;j<ident && !stop_flag;j++)
        {
            auto search_match=obj.match_index.find(j);
            auto search_img=obj.img_index.find(j);
            if(search_match!=obj.match_index.end() && search_img!=obj.img_index.end())
            {
                auto it_match=search_match->second.end();
                it_match--;
                auto it_img=search_img->second.end();
                it_img--;
                int last_match=*it_match;
                int last_img=*it_img;
                if(ref_match_idx[i]==last_match && last_img==last_frame)
                {
                    used_points.push_back(1);
                    identifiers.push_back(j);
                    stop_flag=1;
                }
            }
        }
        if(!stop_flag)
        {
            used_points.push_back(0);
            identifiers.push_back(0);
        }
    }
}
```

Una vez generadas las correspondencias 3D-3D invocamos a la función *update_scale* para calcular el factor de escala relativa:

```
void update_scale(vector<Point3d> existing,vector<Point3d> corresponding,double &scale)
{
    vector<double> scales;
    for (size_t j=0; j < existing.size()-1; j++)
    {
      for (size_t k=j+1; k< existing.size(); k++)
      {
        double s = norm(existing[j] - existing[k]) / norm(corresponding[j] - corresponding[k]);
        scales.push_back(s);
      }
    }
    sort(scales.begin(),scales.end());
    int n=scales.size();
    if (n % 2 != 0) scale=scales[n/2];
    else scale=(scales[(n-1)/2] + scales[n/2])/2.0;
}
```

Esta función realiza las operaciones descritas en el algoritmo 3.6.1.

En este punto ya tenemos el factor de escala relativa. Nos encontramos nuevamente en la rutina *estimate_motion_and_calculate_3d_points*. La tarea que sigue consiste en reescalar la traslación utilizando este factor de escala relativa.

Una vez realizada esta operación, aplicamos nuevamente el método de triangulación volviendo a invocar la rutina *relative_triangulation*. De aquí obtendríamos una serie de puntos 3D que ya serían aptos para ser incorporados a nuestro mapa del entorno. No obstante, para que todos los puntos estén expresados con respecto al mismo sistema de referencia, tenemos que trasladar los puntos 3D obtenidos al sistema de referencia de la posición en la que el robot inició el movimiento. Para ello utilizamos la función de C++ denominada *change_points_to_other_ref_system.* Esta rutina es demasiado elemental y no se ha considerado oportuno mostrarla.

Ya tan solo tenemos que actualizar las variables antes de abandonar el módulo encargado de obtener la nueva posición del robot. Para acometer esta tarea nos apoyamos en las siguientes funciones auxiliares:

```
void add_new_3d_point(int last_frame,int current_frame,cv::Point3d pt,int last_match,int curr_match,
                      cv::Point2d left_project,cv::Point2d right_project,int &ident,tracking_store &obj)
{

    obj.triangulated_3d_points[ident]=pt;
    obj.img_index[ident].push_back(last_frame);
    obj.img_index[ident].push_back(current_frame);
    obj.pt_2d[ident].push_back(cv::Point2f((float)left_project.x,(float)left_project.y));
    obj.pt_2d[ident].push_back(cv::Point2f((float)right_project.x,(float)right_project.y));
    obj.match_index[ident].push_back(last_match);
    obj.match_index[ident].push_back(curr_match);
    ident++;
}
```

```
void update_3d_point(tracking_store &obj,int identifier,int match_id,int current_frame,cv::Point2d projection,cv::Point3d pt)
{
    obj.triangulated_3d_points[identifier]=pt;
    obj.match_index[identifier].push_back(match_id);
    obj.pt_2d[identifier].push_back(cv::Point2f((float)projection.x,(float)projection.y));
    obj.img_index[identifier].push_back(current_frame);
}
```

Utilizaremos una u otra en función de si el punto 3D obtenido es nuevo o no.

En último lugar, como conocemos la posición del robot en el *keyframe* de referencia y ya tenemos una rotación y traslación relativas que hemos reescalado adecuadamente, estamos en disposición de calcular la posición del robot aplicando la ecuación 3.17

Si hemos alcanzado este punto, el módulo encargado de calcular la posición del robot en el *keyframe* actual ha cumplido su objetivo correctamente y devuelve un 1 al hilo de ejecución principal.

Por último, antes de finalizar la iteración, se ejecuta un módulo de optimización local para refinar las coordenadas de la posición obtenida.

Para ello se propone la rutina *local_optimization* que se muestra a continuación:

```
void local_optimization(int window_size,tracking_store &obj,int niter,int ident,double focal_length,Eigen::Vector2d principal_point)
{
    g2o::SparseOptimizer opt;
    opt.setVerbose(false);
    unordered_map<int,int> opt_to_custom;
    std::unique_ptr<g2o::BlockSolver_6_3::LinearSolverType> linearSolver;
    linearSolver = g2o::make_unique<g2o::LinearSolverCholmod<g2o::BlockSolver_6_3::PoseMatrixType>>();

    g2o::OptimizationAlgorithmLevenberg* solver =
    new g2o::OptimizationAlgorithmLevenberg(g2o::make_unique<g2o::BlockSolver_6_3>(std::move(linearSolver)));
    opt.setAlgorithm(solver);

    vector<g2o::SE3Quat,Eigen::aligned_allocator<g2o::SE3Quat> > camera_poses;
    g2o::CameraParameters * cam_params = new g2o::CameraParameters (focal_length, principal_point, 0.);
    cam_params->setId(0);
    opt.addParameter(cam_params);
    // we search the "window_size" last valid_frames
    int dim=0;
    vector<Eigen::Matrix4d> inter_cam_poses;
    vector<int> identifiers;
    for(unsigned int j=(obj.valid_frames.size())-1;j>=0 && dim<window_size;j--)
    {
        if(obj.valid_frames[j]==1)
        {
            cv::Mat pos_cv=obj.cam_poses[obj.frames_id[j]];
            dim++;
            Eigen::Matrix4d eig_pos;
            cv2eigen(pos_cv,eig_pos);
            inter_cam_poses.push_back(eig_pos);
            identifiers.push_back(obj.frames_id[j]);
        }
    }
    int vertex_id=0;
    for(int i=1;i<=window_size;i++)
    {
        g2o::VertexSE3Expmap * v_se3=new g2o::VertexSE3Expmap();
        g2o::SE3Quat pose;
        Eigen::Matrix4d poseMatrix=inter_cam_poses[window_size-i];
        if(i!=window_size)
        {
            Eigen::Quaterniond qb;
            Eigen::Vector3d tb;
            get_rt_from_t(qb,tb,poseMatrix);
            pose=g2o::SE3Quat(qb,tb);
            v_se3->setId(vertex_id);
            v_se3->setEstimate(pose);
            v_se3->setFixed(true);
            opt.addVertex(v_se3);
            camera_poses.push_back(pose);
            vertex_id++;
        }
        else
        {
            Eigen::Quaterniond qb;
            Eigen::Vector3d tb;
            get_rt_from_t(qb,tb,poseMatrix);
            pose=g2o::SE3Quat(qb,tb);
            v_se3->setId(vertex_id);
            v_se3->setEstimate(pose);
            opt.addVertex(v_se3);
            camera_poses.push_back(pose);
            vertex_id++;
        }
    }
    //initial guess for 3d points
    int point_id=vertex_id;
```

```
    //we update g2o's variables
    for(int j=0;j<ident;j++)
    {
        int not_yet=0;
        g2o::VertexSBAPointXYZ * v_p= new g2o::VertexSBAPointXYZ();
        for(int p=1;p<=window_size;p++)
        {
            int img=identifiers[window_size-p];
            vector<int> point_vis=obj.img_index[j];
            vector<Point2f> projections=obj.pt_2d[j];
            int stop_flag=0;
            for(unsigned int k=0;k<point_vis.size() && !stop_flag;k++)
            {
                if(point_vis[k]==img)
                {
                    if(!not_yet)
                    {
                        cv::Point3d actual_value=obj.triangulated_3d_points[j];
                        Eigen::Vector3d init_guess;
                        init_guess[0]=actual_value.x;
                        init_guess[1]=actual_value.y;
                        init_guess[2]=actual_value.z;
                        opt_to_custom[point_id]=j;
                        v_p->setId(point_id);
                        v_p->setMarginalized(true);
                        v_p->setEstimate(init_guess);
                        opt.addVertex(v_p);
                        not_yet=1;
                        point_id++;
                    }
                    //we add the edge connecting the vertex of camera position and the vertex point
                    Eigen::Vector2d measurement(projections[k].x,projections[k].y);
                    g2o::EdgeProjectXYZ2UV * e= new g2o::EdgeProjectXYZ2UV();
                    e->setVertex(0, dynamic_cast<g2o::OptimizableGraph::Vertex*>(v_p));
                    e->setVertex(1, dynamic_cast<g2o::OptimizableGraph::Vertex*>(opt.vertices().find(p-1)->second));
                    e->setMeasurement(measurement);
                    e->information() = Eigen::Matrix2d::Identity();
                    e->setParameterId(0, 0);
                    opt.addEdge(e);
                    stop_flag=1;
                }
            }
        }
    }
    opt.initializeOptimization();
    opt.setVerbose(true);
    opt.optimize(niter);

    for (int i = 1; i <= window_size; i++)
    {
        Eigen::Affine3f cam_pos;
        g2o::SE3Quat updated_pose;
        Eigen::Matrix4f eig_cam_pos=Eigen::Matrix4f::Identity();
        Eigen::Quaterniond cam_quat;
        Eigen::Vector3d cam_translation;
        g2o::HyperGraph::VertexIDMap::iterator pose_it= opt.vertices().find(i-1);
        g2o::VertexSE3Expmap * v_se3= dynamic_cast< g2o::VertexSE3Expmap * >(pose_it->second);
        updated_pose=v_se3->estimate();
        cam_translation=updated_pose.translation();
        cam_quat=updated_pose.rotation();
        eig_cam_pos.block<3,3>(0,0) = cam_quat.matrix().cast<float>();
        eig_cam_pos.block<3,1>(0,3) = cam_translation.cast<float>();
        cam_pos=eig_cam_pos.inverse();
        Eigen::Quaternionf q(cam_pos.matrix().block<3,3>(0,0));
        Eigen::Matrix4d eig_inv=eig_cam_pos.inverse().cast<double>();
        cv::Mat cam_pos_cv=eigentocv(eig_inv);
        obj.cam_poses[identifiers[window_size-i]]=cam_pos_cv;
    }
    for(int j=vertex_id;j<point_id;j++)
    {
        g2o::HyperGraph::VertexIDMap::iterator point_it= opt.vertices().find(j);
        g2o::VertexSBAPointXYZ * v_p= dynamic_cast< g2o::VertexSBAPointXYZ * > (point_it->second);
        Eigen::Vector3d p_aux=v_p->estimate();
        cv::Point3d point_cv;
        point_cv.x=p_aux[0];
        point_cv.y=p_aux[1];
        point_cv.z=p_aux[2];
        obj.triangulated_3d_points[opt_to_custom[j]]=point_cv;
    }
}
```

Esta rutina es muy similar a la utilizada para la creación del mapa inicial por lo que se omite repetir nuevamente que significan algunas de sus variables. No obstante, haremos hincapié en algunos detalles que las diferencian. En este caso el optimizador se está utilizando para refinar la posición del robot y no para obtener un mapa del entorno.

En esta situación se utilizan las posiciones de la cámara correspondientes a un número determinado de *keyframes* anteriores al *keyframe* de la iteración actual. Si considerásemos todos los *keyframes* de la secuencia el algoritmo sería mucho más lento y no podría utilizarse para aplicaciones de tiempo real.

Los vértices que representan a los puntos 3D del entorno serían todos aquellos puntos de nuestra estructura de datos que son observables desde los *keyframes* considerados para este proceso de optimización local.

También tendríamos que obtener los ejes que relacionan los vértices que representan a las posiciones de las cámaras con los vértices que representan a los puntos del entorno, tal y como se hizo en la rutina encargada de crear el mapa inicial.

Cuando creamos el mapa inicial, tan solo se fijó la posición inicial de la cámara y se dio al optimizador libertad para poder optimizar todas las demás posiciones. En este caso se asume que las posiciones de los *keyframes* anteriores ya están bastante optimizadas, de forma que, aunque se considera la información correspondiente a una serie de *keyframes* de la secuencia para construir el optimizador, tan solo se le da libertad a este para optimizar la posición actual de la cámara.

Una vez construido el grafo, invocamos a la función *optimize* de g2o, esperamos a que esta finalice su ejecución y actualizamos las posiciones del robot y los puntos 3D de nuestra estructura de datos. Estos pasos ya se explicaron detenidamente en la creación del mapa inicial, así que no perderemos más tiempo con ellos.

Al finalizar el proceso de optimización local la iteración finaliza y el algoritmo empieza a buscar un nuevo *keyframe.*

Todos los pasos descritos anteriormente se repiten hasta que se agotan las imágenes del *dataset.*

Al agotar todas las imágenes el algoritmo abandona el segundo bucle while del hilo principal de ejecución y procede a representar gráficamente la trayectoria recorrida por el robot móvil y una nube de dispersa de puntos 3D del entorno.

Para esta misión se utilizan funciones de la librería *Point Cloud Library* bastante simples. Se invita al lector de este documento que, tras instalar las dependencias necesarias descritas en el archivo README.md del repositorio, ejecute el algoritmo para apreciar los resultados.