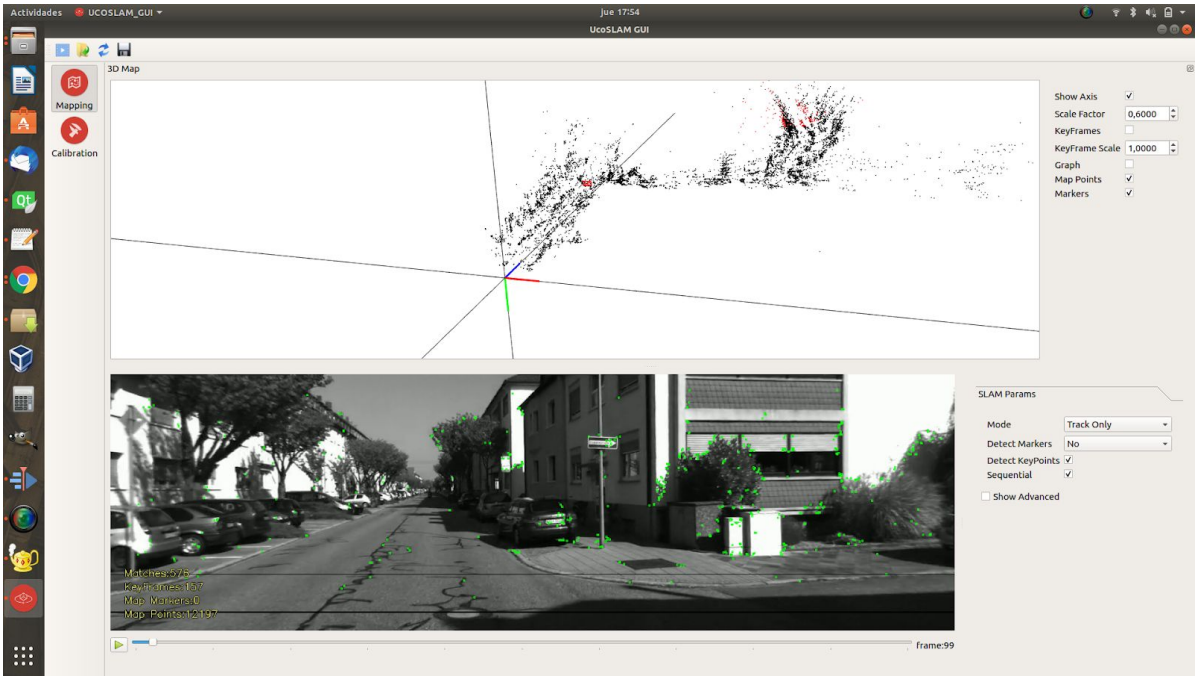
UcoSLAM: A fast and flexible library for SLAM in real applications



The technical document under submission: [Read it here]

Author: Rafael Muñoz Salinas
email:rmsalinas@uco.es

License and how to cite

The library is released under GPLv3 license. For a closed-source version of the library for commercial purposes, please contact rmsalinas@uco.es. Besides, the library depends on OpenCV, Libg2o, and Eigen3. Please read their licenses.

If you use the library or any of its associated projects you must the most relevant publication associated by visiting the *Citing* Section at ucoslam.com.

The technical document describing the system is currently under review in a Journal. However, it can be read the preprint version in this link.

Getting Support

If you need help with the library, please send an email to Rafael Muñoz Salinas: rmsalinas@uco.es

Introduction

This document describes the library UcoSLAM. It is a library for Simultaneous Localization and Mapping designed to be used in real applications. The library allows creating maps of the environment, save them to file, and use them later for localization purposes. Besides, UcoSLAM can handle squared fiducial markers in the environment. The markers can be placed in strategic locations in your environment for several purposes: ease map initialization, obtaining the real map scale even with monocular cameras, setting long-term visual references that will not change over time.

The library is a complete recodification of the popular ORBSLAM2 adding some critical features to make it useful in realistic scenarios. The tests performed in the most popular datasets such as Kitti, Euroc-MAV, etc, prove that UcoSLAM outperforms ORBSLAM2 when monocular cameras are employed.

Main Features

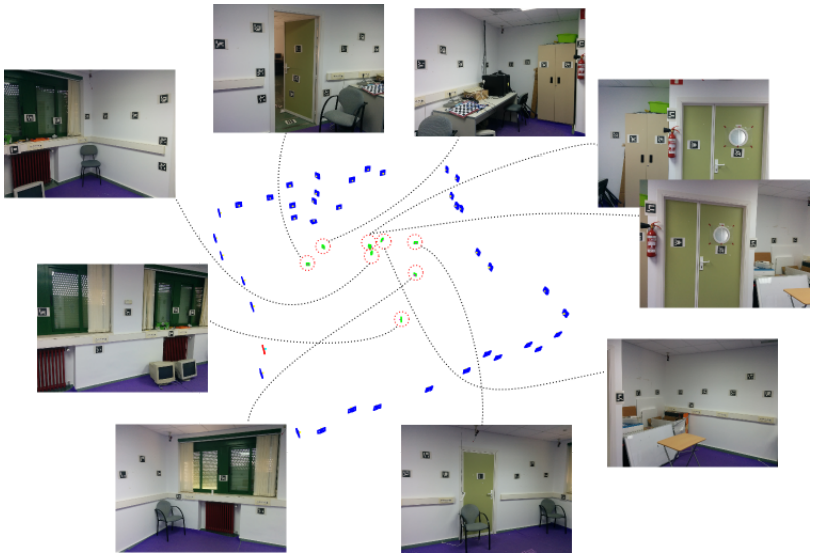
The main improvements of UcoSLAM over ORBSLAM2 are:

1. Better performance.
2. Ability to save/load maps generated.
3. Ability to use markers to enhance, initialization, tracking, and long-term relocalization. Also, markers allow estimating the real map scale from monocular cameras. It is integrated with the ArUco library.
4. Support for sequential mapping, ensuring no frames are dropped in the process. Also, the sequential mode guarantees a deterministic behaviour. In other words, processing the same video sequence twice produces the same results.
5. Parallelization of the KeyPoint detector makes the tracking much faster
6. Only one external dependency, OpenCV. The rest of the required packages are in the library. The compilation is straightforward.
7. Multiplatform, ready to be compiled in Windows, Linux and Android systems
8. An easy-to-use graphical user interface to process your videos, save and visualize your maps, calibrating your camera, etc.
9. PPA repository for Ubuntu and ready-to-use packages for Windows. You do not need to be a developer to use it.

As part of our work, we have also prepared a repository with the most famous SLAM datasets, such as Kitti, Euroc-MAV, TUM, etc. The repository contains all the datasets with ground truth using the same naming convention. It makes extremely easy to compare the performance of different SLAM methods. The repository is publicly available and will be used along this documentation to guide the examples.

Use Case 1:

Image the following scenario, you have a robot (drone, car, or whatever) and you want to create a map of the environment to safely navigate in it. You can solve the task with UcoSLAM as follows. First, record a video footage of the environment using your camera. Then, use UcoSLAM to generate the map of the environment (SLAM). You can configure UcoSLAM in sequential mode so that no frames are skipped during the mapping process. When the



whole sequence is processed, you save your map to a file. Then, you can use the map to localize your robot in the recorded environment.

Use Case 2:

You want to create a map of fiducial markers to cover a large area. However, you can not put as many markers as required for SPM-SLAM and MarkerMapper to work. These two projects requires a lot of markers. You can use UcoSLAM to create the marker map.

Download

The source and precompiled binaries for Windows can be downloaded in https://sourceforge.net/projects/ucoslam/.

For Ubuntu users, you can add the ucoslam PPA repository which has the library and its headers, and also the graphical user interface program UcoSLAM_GUI

Build from source

Watch the Video tutorial

Ubuntu >= 16.04

```
sudo apt-get install cmake libopencv-dev qtbase5-dev libqt5opengl5-dev libopenni2-dev
```

Note: qtbase5-dev libqt5opengl-dev are optional and only required for the graphical user interface.
Note 2: libopenni2-dev is optional and only if you plan to use OpenNI2 devices.

Download source package file from https://sourceforge.net/projects/ucoslam/and uncompress. Then:

```
cd dir_uncompressed
mkdir build
cd build
cmake ../ -DBUILD_GUI=ON
make -j4
[sudo make install ] //will install it on your system
```

Note: the option -DBUILD_GUI=ON is to build the graphical user interface. You can skip it if you do not need it.

Video tutorials

We have prepared a set of video tutorials:

Build from source (Linux)
Create a File with Tracking Parameters
SLAM with a monocular camera (and save to file)
SLAM with a stereo camera and then with a monocular
GUI: Overview
GUI: Camera Calibration
GUI: Create a Map
GUI: Map Update

Library programs

You can watch the Video tutorials describing the process below

The library comes with a set of command line programs already prepared for testing the System and to process your videos. Apart from that, we have created a GUI that is even easier to use. But now, let's go with the command line programs. In the source, you find them in the folder utils. In there, the programs have the naming convention *ucoslam_xxx.*

ucoslam_createinitialparamfile

As explained below, the system has a long list of parameters that can be tuned. You can use a configuration file with all your parameters. This application creates a text file with the default parameters. Then, you can edit the parameters and use them for later.

Example of usage:

```
> ucoslam_createinitialparamfile myparams.yml
```

The program will generate a file called *myparams.yml* that looks like this

```
%YAML:1.0
---
kpDescriptorType: orb
KPNonMaximaSuppresion: 0
targetFocus: -1.
minKFConfidence: 6.0000002384185791e-01
maxNewPoints: 350
targetWidth: 1.
maxFeatures: 4000
```

```
nthreads_feature_detector: 2
nOctaveLevels: 8
scaleFactor: 1.2000000476837158e+00
runSequential: 0
....
```

Edit the file so that:

```
runSequential: 1
detectMarkers: 0
```

then, save the file. We are using it for the next program.

ucoslam_monocular

This program allows you to process a monocular video sequence. It is an extended version of the first example explained below. You can run the program without arguments to see its many options. As an example, download the Sequence 00 of Kitti of our repository, the orb vocabulary and do:

```
>ucoslam_monocular cam0.mp4 cam0.yml -voc orb.fbow -sequential
```

If you want, we can use the parameter file as:

```
>ucoslam_monocular cam0.mp4 cam0.yml -voc orb.fbow -params myparams.yml
```

The program will show the basic graphical user interface that lets you start/stop video pressing ‘s’, move the 3D map with the mouse, and change the view pressing ‘m’. Let the program run until frame 100 or so, then press ESC to finished. As you see, the program finishes saving the generated map in *world.map*.

You can now rerun the program to use the generated map with the right camera:

```
>ucoslam_monocular cam1.mp4 cam1.yml -map world.map  -params myparams.yml
```

In that occasion, we do not need to specify the vocabulary again since it is already in the vocabulary. As we see, the program this time loads the map previously created and do SLAM using it.

ucoslam_stereo

As the name suggests, this program is like the previous one, but with stereo images. Try running it as

```
>ucoslam_stereo cam0.mp4 cam1.mp4 stereo.yml -params myparams.yml -voc orb.fbow
```

Let it run for 100 frames. As the name suggests, this program is like the previous one, but with stereo images. Again, the map is saved in *world.map*. Let's again rerun, but now with a monocular sequence in the generated map.

```
>ucoslam_monocular cam1.mp4 cam1.yml -map world.map  -params myparams.yml
```

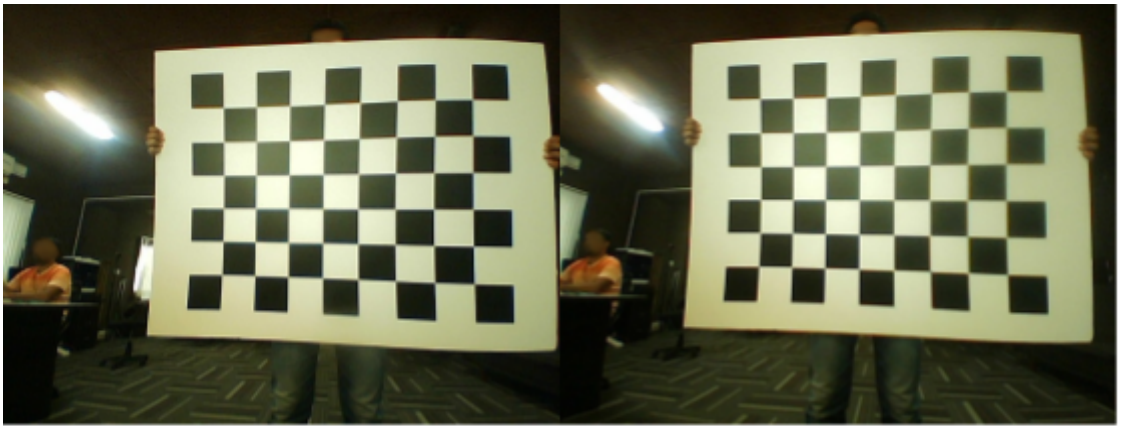
It works!!!!

ucoslam_stereocalibrate

This program lets you calibrate your camera and generate the stereo.yml file you need to run ucoslam with your own stereo camera.

First, take a chessboard calibration pattern (typical in OpenCv), take pictures of it with your left and right cameras and to a folder and save them to a folder. The names of the images must be : `left0.jpg right0.jpg left1.jpg right1.jpg ...`
The idea is that you have <position><number>.<extension>. The extension can be jpg and png.

Below, you can see the left and right pictures taken with our camera.



Example of chessboard pattern of 8x6 corners captured with stereo camera

Then, just execute the command

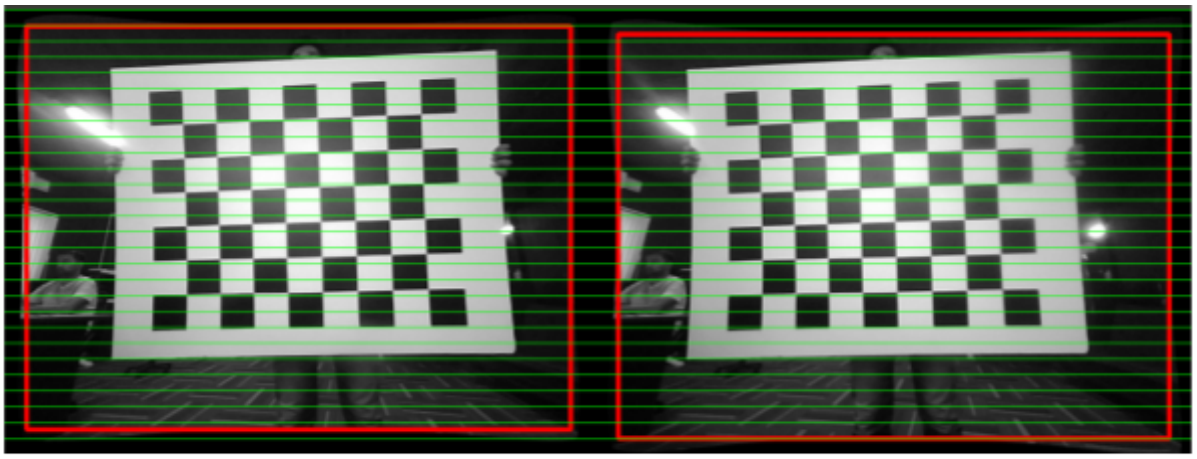
```
./ucoslam_stereocalibrate <pathToDirWithImages> <out.yml> -w <int > -h <int> -s <float>
```

The first parameter is the path to the dir with the images.

The second is the output file that will be generated.
-w is the number of corners of the chessboard in the horizontal direction.
-h is the number of corners of the chessboard in the vertical direction.
-s is the size of each square in the chessboard pattern.

In our case, we run
`./ucoslam_stereocalibrate <pathToDirWithImages> stereo.yml -w 8 -h 6 -s 0.05`

The program will process all images, saving the stereo file to stereo.yml. At the end, the program will show you the rectification results. If everything is correct, scene points should be in the same horizontal line in both images.



Rectified results after calibration

ucoslam_mapviewer

This program will let you visualize a map.

```
>ucoslam_mapviewer world.map
```

ucoslam_map_removeunusedkeypoint

Maps have many keypoints that are never used. This program will remove the unused keypoints saving space.

```
>ucoslam_map_removeunusedkeypoint world.map world2.map
```

Then, visualize.

```
>ucoslam_mapviewer world2.map
```

Since the map contains the vocabulary, which can be big, the reduction can not be very noticeable in maps with few KeyFrames.
Also note that the reduction will not affect to relocalization, since the keypoints are not removed from the BagOfWord database. However, it may affect a little if you try to update the map since the removed keypoints are not present. So, I recommend to use this tool once the map is completed.

ucoslam_rgbd

Note: This is only compiled if the OpenNI2 development library is in the system (for Ubuntu).

Note2: In Windows, the precompiled binary does not have it included (TODO).

This program lets you use a RGB-D device such as Kinect or Asus Xition. The program is already prepared with the camera parameters of such devices. However, adapting it to any other device is straightforward. So, if you have Asus Xition plug it and type

```
>ucoslam_rgbd live
```

You can provide as first parameter a oni video file as well.

UcoSLAM_GUI

This program is the GUI that is explained below. You can find it in the build path gui/UcoSLAM_GUI

Quick tutorial for developers

The github repository ucoslam_samples has a set of examples to learn using the library.

First example

Note: We are assuming in this example that you are using the sequence 00 of the Kitti dataset that we provide in our dataset.

We'll start by the simplest mocular example. To compile the code, you must have the UcoSLAM library installed in your system. Run the program with the following arguments:

```
>ucoslam_test_monocular cam0.mp4 cam0.yml orb.fbow kitti00_cam0.map
```

Let's see the code. As you can observe, at the beginning we declare the main classes.

```
cv::VideoCapture VideoIn;//video capturer
ucoslam::UcoSlam SLAM;//The main class
ucoslam::Params UcoSlamParams;//processing parameters
ucoslam::ImageParams cameraParams;//camera parameters
ucoslam::MapViewer MapViwer;//Viewer to see the 3D map and the input images
```

The ucoslam::*UcoSlam* class is the one that makes the hard work. It receives the input images, creates and updates the map. The *ucoslam::Params* is a critical class because it determines how the images are processed, how often keyframes are added, etc.

The `ucoslam::ImageParams` has the internal parameters of the camera being used. These are called intrinsic parameters and we follow the OpenCV format. They are stored in a YAML format.

The `ucoslam::MapViewer` class is a very convenient class for showing the generated 3D map and the input images with the keypoints being employed. The class renders in 3D using the SGL library specifically developed for this project. It does not require OpenGL and run in any platform. You'll be able to rotate and translate your 3D model in a OpenCV window.

Finally, the `ucoslam::Map` class represents the map created in the SLAM process. It is created as a smart pointer since it is internally referred by several classes working in parallel.

```
UcoSlamParams.runSequential=true;//run in sequential mode to avoid skipping
frames
UcoSlamParams.detectMarkers=false;//no markers in this example.
```

These lines of code are employed to configure the way UcoSLAM will process the input frames. The system has a long list of parmeters, most of which are not important. However, `UcoSlamParams.runSequential` indicates to the system that all frames must be processed without dropping any. The flag `UcoSlamParams.detectMarkers` is deactivated when your environment does not have fiducial markers.

Then, we call

```
SLAM.setParams(map,UcoSlamParams,argv[3]);
```

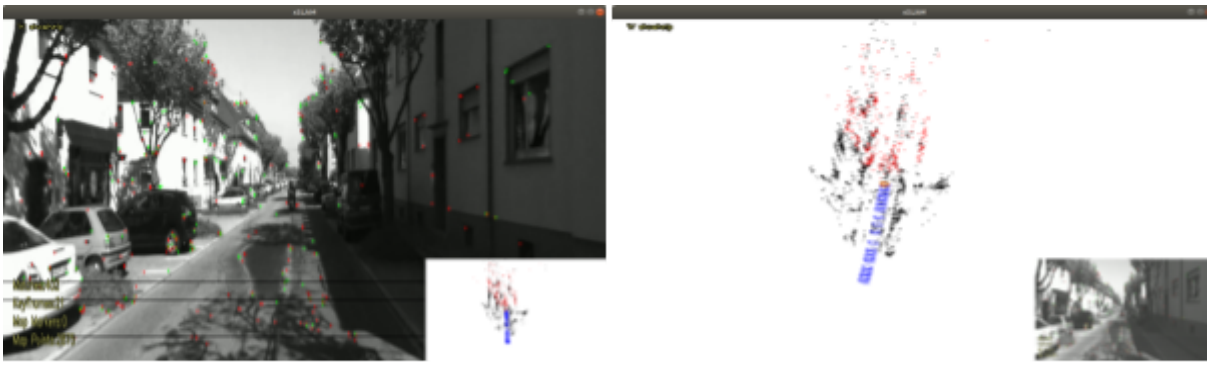
to initialize the system. The last parameter is the path to the vocabulary file. What is the vocabulary file? UcoSLAM a visual Bag-Of-Words approach for camera relozation. It is, when the system looses track of the camera position, if you go back to a known position,

the system may be able to recognize that location and find the location again. To do this, the system creates a DataBase of visited locations using a Bag-Of-Words. To speed up the search, you must provide a vocabulary with the valid words. For the ORB keypoint descriptor, we have already a vocabulary created that you can download with the library (*orb.fbow*). Alternatively, you can download it from HERE. In any case, it is not mandatory to indicate a vocabulary to operate the system. You can leave the 3rd parameter empty. The only problem is that you'll not be able to do relocalization with keypoints. However, you'll always be able to do relocalization from the fiducial markers if you use them.

Below in the code we see:

```
cv::Mat posef2g= SLAM.process(inputImage,cameraParams,frameNumber);
if(posef2g.empty()){
	std::cerr<<"Frame "<<frameNumber<<" pose not found"<<std::endl;
}
Else
	std::cerr<<"Frame "<<frameNumber<<" pose "<<posef2g<<std::endl;
//draw a mininimal interface in an opencv window
keyPressed=MapViwer.show(map,inputImage,posef2g);
```

The call to `ucoslam::UcoSLAM::process` is the main entry to the system. You must provide the input image, the camera parameters, and as a final value, the frame counter. It should be a value that increases at each frame. Its only purpose is to know when you read the map, from which image a keyframe was created. The function returns a 4x4 matrix expressing the SE3 transform that moves points from the global reference system of the map to the current camera pose. Please notice that along the code we employ the following naming convention C2G to indicate that a matrix moves from the Global to the Camera (C<-G). We use right-to-left naming convention because of the homogeneous transformation matrices work in that direction.
The returned matrix can be empty, meaning that the system is unable to locate the camera location.



Simple graphical interface shown in the example. Left image shows the input image, and keypoints superimposes. Also the 3D map is small in a subwindow. In the right image, the GUI shows the 3D map bigger.

After processing the image, we call the show method of the MapViewer that will update the 3D view of the map and the image processed. This will open a window showing the 3D map generated and the image processed. You can switch from one view to another pressing the key ‘m’. You can get help about the manipulation of this simple GUI pressing ‘h’. In short, you can rotate and translate the viewpoint using your left mouse button and a combination of the keys SHIFT and CTRL. Also, you can move to the next frame by just pressing any key, or pressing *space* to start/stop the video. Finally, press ESC to end.

Finally, the program saves the map to a file.

```
map->saveToFile(argv[4]);
```

The saved map can be loaded and passed to the system to update it or just doing tracking. You’ll only have to do

```
map->loadFromFile(pathToMap);
SLAM.setParams(map,UcoSlamParams,argv[3]);
```

See the tutorial below for more info.

You can work with the map to do other tasks. There an application provided with the library (*utils/ucoslam_mapviewer*) that can be used for visualization purposes of the map only.

Library Description

UcoSLAM is the result of a three years effort to create a state-of-the-art library for SLAM that can be used in realistic applications. Most of the code used by the library has been implemented from scratch creating a set of auxiliary projects that can also be used in a wide range of computer vision tasks.

KeyPoint-Based SLAM

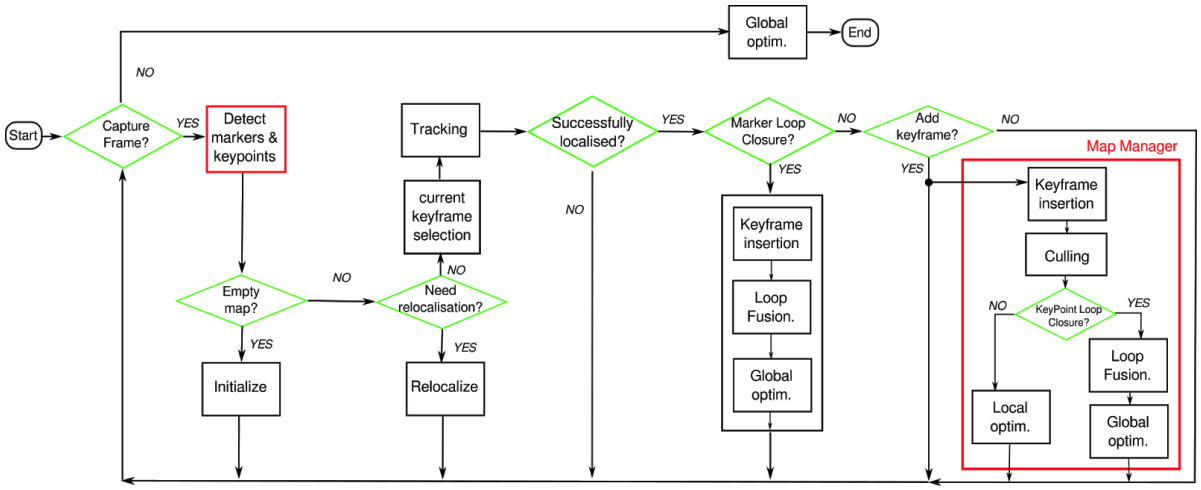
UcoSLAM employs a spare keypoint approach for simultaneously creating a map from a sequence of images captured with a camera. Like other libraries, our system can work with monocular, stereo and rgbd cameras. Indeed, the last ones are much more precise than a monocular camera. However, UcoSLAM also allows employing fiducial markers. Although the use of fiducial markers is not mandatory, they have several advantages. First, they help with map initialization when monocular cameras are employed. Second, they allow knowing the real map dimensions when monocular cameras are employed. Also, they help with tracking and relocalization.

The map created is comprised of a set of 3D points, markers, and keyframes. The 3D points represent regions of the environment that have enough visual contrast to be distinguished from the rest. Their location is obtained by triangulating their observations from multiple frames (viewpoints). To do triangulation, the system has to keep a set of keyframes. They are processed frames that are employed to triangulation purposes, but will also be used for relocalization. Finally, our map also has markers. As indicated, they can be used for long-term relocalization.

Operational Pipeline

The pipeline of UcoSLAM is summarized in the following chart, which is the most common approach in most visual SLAM approaches. The main difference in our case is the combined use of keypoints and squared planar markers.

Our system maintains a map of the environment that is created and updated every time a new frame is acquired. In the beginning, the map is empty and initialization is required. The map can be initialized from the homography or the fundamental matrices (using keypoints), and also using one or several markers.



If the map is already initialized, the system goes either into tracking or relocalization mode. If the camera pose was adequately determined in the last frame, then, the system tries to estimate the current location using the last one as a starting point. Tracking is performed by jointly optimizing the reprojection errors of keypoints and marker corners. But before tracking, the *reference keyframe* is selected. In essence, this is the most similar keyframe of the database to the current frame acquired by the camera and plays an essential role in the rest of the process.

After successfully tracking, the system must search for loop closures caused by markers. In contrast to keypoint-based loop closure detection, when using markers, this process must be performed in advance thus preventing its use in the subsequent tracking iteration without proper drift correction. If loop closure is detected, then, the two sides of the map must be appropriately connected. Then, the current frame is inserted as a new keyframe, and the drift accumulated removed and a global optimization process is applied to correct the whole map.

If no loop closure is detected, the system must decide whether the current frame will be inserted as a new keyframe. This is a process run in an independent thread named *Map Manager*. If the keyframe is inserted, the map is updated and a culling process is run. Culling allows removing redundant information to keep the map manageable, following a similar approach to ORBSLAM2.

Afterward, the system must detect if there is a loop closure by using only keypoints. If not, a local optimization is applied to integrate the new information. As in ORBSLAM2, only the keyframes directly connected to the inserted keyframe are employed in the local optimization. If loop closure is detected, the two sides of the loop are fused and a global optimization performed.

If tracking failed in the last frame, then, the system enters in relocalization mode.

Relocalisation is done looking first for markers already registered on the map. If no known markers are visible, or they are not reliably detected, relocalization using a bag-of-words approach is attempted.

Sequential vs Real-Time Operation

In theory SLAM is a process that is run on board of our robot. Then, we try to process as many images as possible, as they are captured. Thus, we need our system to be fast and SLAM uses multiple threads to parallelize the process. In general, the mapping process is slower than the tracking thread. As a consequence, there are frames that can not be processed and must be dropped from processing. As long as the distance travelled between two frames processed is not long, the SLAM system usually work fine.

Nevertheless, in most of the cases, we do not let our expensive robot/drone/car to walk around discovering its environment. Instead, we prefer to have a map, carefully double checked by a human, and then, use it with our robot. Thus, we usually operate by first recording the environment, then using the video sequence to create the map, and finally using map in our robot. In that case, we want our map to be as good as possible.

This approach has a problem. When reading images from a video, if our system is very fast, it can feed the system with too many frames per second, so that the mapping thread is not able to process them at the correct pace. In that case, it is better to avoid using independent threads for mapping. In our case, we have implemented a very efficient method for processing keypoints, that is able to work at 100fps in 640x480 images. Thus, processing a video at this speed is equivalent to move the robot at three-times the real speed the video was recorded. The mapping thread will not be able to run that fast, and you'll skip frames, and possibly you will lose track. Please notice that it is not a problem for tracking, which you want to be as efficient as possible. But, the problem is with mapping if it must be in parallel with tracking.

A second problem in real-time SLAM when processing a video sequence is that two executions will not produce the same results.

To avoid these problems, UcoSLAM can do the mapping process in sequential mode. In that case, all frames are analyzed. The mapping thread works in sequential mode, processing all frames, and having enough time to decide which are added as keyframes.

Please notice than in real-time mode there are several threads running in parallel. If you want to stop them you can call ucoslam::UcoSlam::waitForFinished().

Library Dependencies

As part of the development of the library, a set of libraries have been. The main goal was to avoid others dependencies as wells as to provide state-of-the-art libraries to solve the different parts involved in SLAM.

In particular, the following libraries have been created as part of the project:

- fbow: (Fast Bag-of-Word) is a library created for relocalization. Its main advantage is that it is hugely optimized using MMX, SSE, and AVX vectorized instructions. *fbow* is ~80x faster than DBOW2 in loading vocabularies. When transforming an image into a bag of words using on machines with AVX instructions, it is ~6.4x faster.
- **Xflann** (Extremely Fast Library for Approximate Nearest Neighbors) is complete re-implementation of the FLANN library that uses MMX, SSE, and AVX instructions to speed up the search. The performance of the library has been compared to the original flann, opencv flann and vlfeat libraries. Our library ranks best if most of the tests performed.
- picoflann: is a header-only library for exact kdtree search. It is created as an easier-to-use alternative to nanoflann. The tests performed to prove that the library is faster than OpenCV, nanoflann, and flann.
- SGL (Simplest Graphic Library): is header only a library for 3D rendering of points and lines. It has been developed to show the 3D map generated in any platform without requiring OpenGL.

If you use any of these libraries in your academic work, you must include the following cite, cite the most relevant related paper by visiting the *Citing* Section at ucoslam.com.

The only part of the library we are still depending on is LibG2O, used for graph-optimization. Although we already have an initial approach to replace this library, our current implementation is not yet ready. Nevertheless, this external dependency is appropriately integrated into the project so that you do not need to download it.

Main Classes

The code is made to be both flexible, easy to use. Most of the implementation is hidden to the user. The main classes the user has to know are:

- UcoSLAM (ucoslam.h): this is the class that does most work. Youll provides images to this class to create the map and to compute the location of your camera.
- ImageParams (imageparams.h): this class defines the parameters of the camera employed. We use OpenCV camera model and its YAML files. The graphical user

interface (UcoSLAM_GUI) has a tool will help you to calibrate your camera with writing code :).

- Params (ucoslamytypes.h): This class defines the operational parameters for SLAM, e.g., enable/disable detection markers or not, choose between sequential or real-time SLAM, etc.
- Map (map.h): represents the map created by UcoSLAM. A map is comprised by :
    - ;apPoints: represent the set of 3D points detected in the map.
    - KeFframes: represent locations at which the camera has been during the mapping process. They are used for mappoint triangulation and relocalization purposes.
    - Markers (optional): represent the markers placed in the environment.

Below, we provide a detailed description of each class

ucoslam::UcoSlam

This is the main class of the library. Receives as input the images captured and creates or updates the map. Before using the class, you must set the required parameters via

```
void setParams(std::shared_ptr<Map> map, const ucoslam::Params &params, const std::string &vocabulary="");
```

You must pass a shared pointer to a map class, the operational parameters and optionally the path to the vocabulary file.

The map you pass can be either en empty map, or a map already created. In the first case, the system will have to initialize the map. Initialization using monocular cameras requires to find a pair of good images from which to do triangulation. If you user markers, the process is generally simpler. If the map is already created, then the system will not be able to start until the camera is at a known location. In other words, the first task in a non-empty map is relocalization.
Please notice, saved maps includes the vocabulary employed. So, if you create a map indicating a vocabulary and you save it, the vocabulary is already in the map. As a consequence, you'll do not have to specify the vocabulary file the second time in the call to setParams().

The main function to feed the system with images is processXX(). Depending on the type of sensor you use, you'll have to choose the appropriate one:

```
cv::Mat process( cv::Mat &in_image,const ImageParams &ip,uint32_t frameseq_idx);
cv::Mat processStereo( cv::Mat &left_image,const cv::Mat &right_image,const
                       ImageParams &ip,uint32_t frameseq_idx);
cv::Mat processRGBD( cv::Mat &in_image,const cv::Mat & depth,const ImageParams
                     &ip,uint32_t frameseq_idx);
```

Please notice that for stereo images, they must be properly rectified.

The function returns the pose of the camera, in case the pose can be estimated. It is returned as a 4x4 SE3 matrix. If the pose can not be estimated, then an empty matrix is returned. The matrix passed is the one moving points from the global reference system to the current camera.

The global reference system is established as the first keyframe of the map, i.e., which was used for initialization.

Running modes

The function setMode(MODES mode); is employed to set the running mode. There are to modes :MODE_SLAM and MODE_LOCALIZATION. The first one is used to create and update maps. In that mode, the MapManager thread Is activated to decide when keyframes are added, etc.

If you only need to do tracking, you can use the localization mode that will not update the map. This is much faster and specially appropriated once your map is stable and you only need to do tracking.

ucoslam::Params

The class ucoslam::Params is the one that tells ucoslam::UcoSLAM how it must operate. It has many parameters, but only few of them are relevant for most users. Below we provide a description of them:

- bool detectMarkers: Default value: true
Enables and disables marker detection. Marker detection takes some time to be computed. If you do not use markers in your environment, you can save that time.

- float aruco_markerSize: range (0,inf). Default value:1
Indicates the physical size of the markers of the environment. All markers are assumed to be of the same size. The map is scaled according to this parameter if markers are detected.

- bool detectKeyPoints: enables/disables keypoint detection. If you are only interested in creating map of markers, you can disable keypoints. If you do that, you are using our previous work SPM-SLAM.
- DescriptorTypes::Type kpDescriptorType: UcoSLAM can handle any type of keypoint descriptor (orb,brief,SIFT,SUR,etc). By default, ORB is employed since it provides good results.
- float KFMinConfidence: range (0,inf) . Default value:0.6

This value regulates when a keyframe must be added. If you increase the value, keyframes will be added more often. This has the drawback of making optimization slower. On the other hand, you reduce the chances of losing track. We recommend values in range [0.6-1].

- float KFCulling: range [0,1]. Default value: 0.8

Culling is the process of removing redundant keyframes. A keyframe is redundant if it does not add relevant information for the optimization process. In general, this decision is taken based on the number of keypoints of a keyframe that are seen in other keyframes. KFCulling indicates the percentage of redudant points seen in other keyframes. Values near 0 will remove more keyframes, values near 1 will hardly remove any keyframe.

- bool KPNonMaximaSuppresion: Default value :false

In many cases when detecting keypoints, several keypoints appear very near to each other. Activating this flag will remove non-maxima ones thus reducing the total number of map points. This has an impact in speed and memory. However, it could affect to tracking robustness. PLease notice that this non-maxima suppression is on the top of the one performed by the FAST detector.

- float aruco_markerSize: Default value:1

Expresses the size of the fiducial markers employed in the environment. This value can be used to have the map in the real scale.

- float kptImageScaleFactor. Range (0,1). Default value:1

In many cases, you may decide to reduce the size of the input image to achieve the desired speed. For instance, your camera records at 1920x1080, but you work fine at 800x600. Marker detection is a process much faster than keypoint detection, reaching 1000fps in 4K images. Thus, it would be preferable to detect markers at the original resolution.

UcoSLAM can handle this situation by appropriately setting the kptImageScaleFactor value. When, it is 1, the input image will not be altered. If , for instance you set this to 0.5, the original image is employed for marker detection, but an image of half size will be employed for keypoint detection.

- int nthreads_feature_detector. Range (1,inf). Default value 2.

UcoSLAM has a parallel implementation of the keypoint detector that is up to several times faster than the one in OpenCV. We achieve this performance by carefully parallelizing the work in several threads. We have predefined profiles for 1,2,3 and 4 threads. However, our tests suggests that the best trade-off is obtained for 2.

- bool autoAdjustKpSensitivity. Default value: false

KeyPoint detectors normally have a threshold that adapts their sensitivity. For instance, ORB has a threshold to decide the strength of the gradient necessary to trigger a detection. The value can not fit all possible illuminations. Thus, our system auto-adapts its sensitivity aiming at obtaining the number of features desired (see maxFeatures below)

- std::string aruco_Dictionary="ARUCO_MIP_36h12";
- std::string aruco_DetectionMode="DM_NORMAL";
- std::string aruco_CornerRefimentMethod="CORNER_SUBPIX";
- float aruco_minMarkerSize=1.0;

Parameters employed for ArUco marker detection.

- int maxFeatures;//number of features to be detected in the image
- int nOctaveLevels;//number of octaves for keypoint detection
- float scaleFactor;//scale factor employed to create the octaves

These are the parameters that determines the operation of the keypoint detector.

- void saveToYMLFile(const std::string &path);
- void readFromYMLFile(const std::string &path);

Convenient functions to load from file and read from file the parameters.

ucoslam::Map

The map of the environment is comprised by 3D points, markers and keyframes. These are the main elements of a map:

- map_points: container with the MapPoints.
- keyframes: container with the KeyFrames.
- map_makers: container with the markers of the map.

MapPoints, KeyFrames and Markers have a unique id. You can access to a any of these elements by its id using the operator []. (see example for more info). Both for map_points and key_frames, the container is of type ReusableContainer (more info).

The map can be saved to file and read from file.

```
void saveToFile(string fpath);
void readFromFile(std::string  fpath);
```

You can visualize a map using the tool provided with the library utils/*ucoslam_mapviewer* or using the graphical user interface. Additionally, you can save it to a .pcd file and visualized with pcl_viewer.

```
void  saveToPcd(std::string filepath,float pointViewDirectionDist=0)const;
```

Additionally, you can save only the map of markers and use it with ArUco.

```
void saveToMarkerMap(std::string filepath)const ;
```

If the map was created using a vocabulary, it will be saved with it. So, you don't need to set it in the call to *ucoslam::UcoSlam::setParameters* again (see example)

The map have functions to remove unused keypoints.

```
void removeUnUsedKeyPoints();
```

Many of the keypoints detected in the images are never matched. This gives an idea that these are not important keypoints. Thus we can remove them and save space. In general, nearly 80% of the keypoints detected are not used. In any case, this operation should be done once the whole map has been created, just in case.

Finally, the map has a function to apply a transform to all its elements. Imagine you want to move the map to a custom reference system.

```
void applyTransform(cv::Mat m4x4);
```

Also. you can set the center of a markers as the center of the global reference system:

```
bool centerRefSystemInMarker(uint32_t markerId);
```

ucoslam::Frame

A Frame is an image captured by the camera. Some of the images will be kept in the map as KeyFrames in order to do triangulation and relocalization. When a image is passed to the system, it creates a Frame. The main elements of a Frame are:

- idx: Unique number that identifies the frame
- und_kpts: vector with keypoints detected. The distortion has been removed to speed up optimization.
- desc: matrix with the descriptors. Each row is a descriptor corresponding to the keypoint in the und_kpts vector.
- ids: This is probably the most important vector in the Frame. The ids vector has as many elements are keypoints. Their values indicate the id of the MapPoint it represents.
  For instance, if `ids[0]==23`. It means that the first keypoint of the Frame, `und_kpts[0]` with descriptor `desc.row(0)`, is an observation of the MapPoint with id 23.

If `ids[0]==std::numeric_limits<uint32_t>::max()`, it means that the keypoint does not correspond to any MapPoint. It is an unused keypoint. This is the default value of the elements of ids when the Frame is created.

- fseq_idx: the third parameter passed to ucoslam::UcoSlam::process(). It can be used to match a KeyFrame with its image in a video sequence.
- pose_f2g: transform that moves points from the global reference system to the this Frame.
- imageParams: image parameters. Second parameter passed to ucoslam::UcoSlam::process(). It may not be exactly the same if the image is rescaled
- markers: vector of markers observed in this Frame.

ucoslam::MapPoint

A MapPoint represents a 3D point in the environment. The location of a map point is obtained triangulating its observation from multiple KeyFrames.

For each point, the following information is kept:

- id: a unique value that identifies the point amongst the rest of mappoints.
- Its 3D coordinates: cv::Point3f getCoordinates()
  This is the 3D position of the point in the global reference system of the map.
- Its normal direction: cv::Point3f getNormal()
  It is the average direction of the point to all the KeyFrames it is observed from.
- std::vector<std::pair<uint32_t,uint32_t> > getObservingFrames()const;
  This is one of the most relevant functions to know where the 3D point is being observed from. This function returns a vector with pairs of <idx,i>, where idx is a frame identifier, and i is the position in the Frame vector und_kpts (see example on how to iterate through mappoints)

ucoslam::Marker

A marker represents a physical marker placed in the environment. They can be used to scale the map when using monocular cameras, help with map initialization and relocalization. So far, we assume that all markers have the same size that is determined by the parameter ucoslam::Params::aruco_minMarkerSize. So far, when the System detects markers, it is assumed that they all have the same size. However, this feature could be changed in the future to adapt other situations.

The elements of a marker are:

- uint32_t id: Unique identifier of the marker.

- Se3Transform pose_g2m: 4x4 matrix transform moving points from the center of the marker to the global map references system.

  Please notice that you can obtain the transform from the marker to the camera as:

  Se3Transform f2m= keyframe.pose_f2g*marker.pose_g2m;

- float size=0: size of the marker. Takes the value from ucoslam::Params::aruco_markerSize
- std::set<uint32_t> frames: The set of frames that sees this marker
- std::string dict_info: Dictionary the marker belongs to.

It is important to remark that a marker could be added in the map, but without knowing its pose. In this case, we say that the marker is invalid. You can check by calling:

```
If (!pose_g2m.isValid())
```

A marker is added as soon as it is seen. A Keyframe is added along with the markers. The pose of marker can sometimes be detected by a single location. But in other occasions, there is an ambiguity that makes it impossible. In that case, the System waits until the pose can estimated from other viewpoints or by triangulation.

ucoslam::ReusableContainer

Our system is concerned with long-term map update and conceived to be used in real applications. For that reason, a very important aspect is the possibility to store the generated map for later uses. The main problem with serialization comes from the following requisites. First, it is desired to store the map elements (points, markers, and keyframes) in a data structure that allows random access, such as a vector, so that tracking and mapping could be as fast as possible. Second, as will be explained later, many of the created points will be removed due to lack of temporal consistency in a culling process. Third, the number of map elements is not known in advance.

A regular vector is not a good option for storing the data for the following reason. Since the number of elements is not known in advance, the vector size will have to be eventually increased, and its data copied from the original to the new vector, which is a time-consuming blocking process, especially as the size of the map increases. In ORB-SLAM, the problem is solved by dynamically creating/deleting the map elements, and using their memory references along the code. However, that approach has the problem of making it very complicated to serialize the map. As a consequence, their work does not

include any serialization method, and thus, its usage in some applications is limited. Using a list or a tree-indexed structure is not fast enough (as we have already tested).

Our work uses an efficient mixed data structure to solve the above-mentioned problems. In the beginning, a vector with a fixed size is created. When a new element is created, it is added in the first free vector position, which is employed to identify and access to the element. Whenever an element is removed, the vector position is marked as free and annotated in a list of free positions so that subsequent insertions are placed in the free positions. When the vector is full, a new vector is created for storing the new elements. With this scheme, accessing to an element requires calculating first the vector in which the element is, and then its position into the vector. The approach allows accessing the elements in constant time

The class ReusableContainer implements the above ideas. The operator[], allows you to access the ith element of the container. However, some elements may be invalid. Accessing to an invalid element is dangerous. So, you should call the member function `bool is(uint32_t index)const` to know is the element is valid.

Graphical User Interface

With the aim of giving access to UcoSLAM to a wider range of users, we have created an easy-to-use GUI. The name of the program is UcoSLAM_GUI and it has been created using the Qt5 libraries.

The program lets you calibrate your camera, creating maps, visualize them, etc.

In order to show you how to use it, we have created a series of Video Tutorials in YouTube.

Tutorials

Map reuse

This tutorial will show you how to create a map with the left image of the Kitti 00 dataset, save it to a file, and then load it to track using the right image. We will base our explanation in the test ucoslam_test_left_right. We assume that you have downloaded the folder with the Sequence 00 of Kittti and the orb vocabulary.

```
createMapWithLeft(argv[1]+string("/cam0.mp4"),argv[1]+string("/cam0.yml"),argv[2],mapName)
```

At first, we call the function that will prepare the environment for creating a map from scratch using the left video stream of the kitti sequence and will save it to a file. It is worth noticing that we call

```
//let us compress the map removing unused key points [optional]
map->removeUnUsedKeyPoints();
```

This is a member function that will remove the unused keypoints from the frames. Many of the keypoints detected in the images are never matched. This gives an idea that these are not important keypoints. If that have not been matched properly up to this point, this may mean that they are not good ones. So, we can remove them. In general, nearly 80% of the keypoints detected are not used. In any case, this operation should be done once the whole map has been created, just in case.

After the map is saved, the main function continues preparing the classes to do tracking in the map just created. This time, however, we are using the information from the right camera. It is worth considering the following lines of code:

```
//sets the operating parameters
//we do not have to specify the vocabulary since is already in the map
SLAM.setParams(map,ucoParams);
SLAM.setMode(ucoslam::MODE_LOCALIZATION);//localization only, no map update
```

When we call setParams, we are not setting the path to the vocabulary. This is because we know that the vocabulary is already saved with the map. Then, we set the localization mode. In that case, the map will not be updated. Only tracking will be done.

Then the program goes into a while-loop until esc is pressed or the video sequence finishes. Pressing the keys ‘u’ and ‘i’ you can move backwards and forward in the sequence 10 frames.

Map Iterator

The code explained here correspond is the ucoslam_test_mapiterator.cpp that shows how to iterate through the map points.

```
cout<<"Number of MapPoints= "<<Map->map_points.size()<<endl;
cout<<"Number of KeyFrames= "<<Map->keyframes.size()<<endl;
cout<<"Number of Markers= "<<Map->map_markers.size()<<endl;
```

In the code above, we access to the different containers and print how many of them are there. Then, we iterate along the map points.

```
//move along all map points
for(const ucoslam::MapPoint &point:Map->map_points){
```

```
            cout<<"Map Point:"<<point.id<<" pos="<<point.getCoordinates()<<endl;
            cout<<" Is observed in:"<<endl;
            for(auto _pair:point.getObservingFrames()){
                cout<<"     Frame idx="<<_pair.first<<" keypoint: "<<_pair.second;
                //print the keypoint 2d position
                cv::Point2f pixel=Map->keyframes[ _pair.first ].und_kpts[ _pair.second].pt;
               cout<<" at the pixel:"<<pixel<<endl;
            }
        }
```

We are using the C++11 iterator style. Please consider that some of the elements in the `map_points` container are invalid. So, traversing the elements in a more C++98 fashion would be something like this:

```
 //the same in a C++98 style
        for(uint32_t i=0;i< Map->map_points.capacity();i++){
            if( Map->map_points.is(i)){//check if valid
                const ucoslam::MapPoint  &point=Map->map_points[i];
                cout<<"Map Point:"<<point.id<<" pos="<<point.getCoordinates()<<endl;
                cout<<" Is observed in:"<<endl;
                for(auto _pair:point.getObservingFrames()){
                    cout<<"\tFrame idx="<<_pair.first<<" keypoint: "<<_pair.second;
                    //print the keypoint 2d position
                cv::Point2f pixel=Map->keyframes[ _pair.first ].und_kpts[ _pair.second].pt;
                 cout<<" at the pixel:"<<pixel<<endl;
                }
            }
        }
```

In this case, `Map->map_points.capacity()` indicates how many elements the container has. However, some of them correspond to invalid elements. So, you must call `Map->map_points.is(i)` to know if the i-th element is valid. Please notice the difference with `Map->map_points.size()` that tells you how many valid elements the container has.

Finally, we show how to do the process the other way around. Iterate in KeyFrames, and printing all the MapPoints observed from each KeyFrame:

```
for(const auto &Kframe:Map->keyframes){
 cout<<"Frame "<<Kframe.idx<<" is at "<< Kframe.getCameraCenter()<< endl;
 cout<<"The frame observes the following MapPoints:"<<endl;
 //iterate through the ids.
 for(size_t i=0;i<Kframe.ids.size();i++){
        if(  Kframe.ids[i]!=std::numeric_limits<uint32_t>::max()){
                uint32_t mapId=Kframe.ids[i];
                cout<<"\tMapPoint.id="<< mapId<<" 3D
                        position="<<Map->map_points[mapId].getCoordinates()<<endl;
        }
  }
}
```

FAQ

1. Why the image shows that ugly horizontal black lines?
It seems to be a bug in OpenCV function to write tex (cv::putText()). If is corrected in OpenCv 3.4.3 at least.

2. Why is part of code obfuscated?
Parts of the code have been obfuscated to encourage private companies that wants to use UcoSLAM for commercial purposes to contact us for a commercial license. Additionally, we have published yet the corresponding paper and we aim to avoid possible modifications of the software until then.

3. How can I get a commercial license.
You must contact rmsalinas@uco.es