ORB SLAM 2 with a UAV platform

Erika Cocca, Carola Motolese and Alberto Rivolta

Abstract

Localization and mapping is a fundamental aspect for autonomous UAVs operating in unknown environments. The main aim of the assignment is to allow a generic drone to map the surrounding environment by using an on-board RGB-D camera. To this aim, state-of-the art solutions for RGB-D map reconstruction will be used and integrated with existing internal odometry information based, on a Luenberger observer.

1. INTRODUCTION

Perception is an important attribute for building any sort of intelligent robot. The perception task can be split primarily into two problems: knowing the environment (area mapping) and knowing where the robot is present in the environment (robot localization).

The Simultaneous Localization and Mapping (SLAM) problem is concerned with building an intelligent robot that can identify its position when kept at an unknown location and in an unknown environment, while incrementally building a map of the environment it is placed in. SLAM has been formulated and solved by the Robotics community and several algorithms already exist. In particular, ORB-SLAM 2 is a versatile and accurate SLAM algorithm for Monocular, Stereo and RGB-D cameras, based on ORB-SLAM. It is able to compute in real-time the camera trajectory and a sparse 3D reconstruction of the scene in a wide variety of environments. The aim of this project is to run ORB-SLAM 2 on the Astec Firefly hexacopter to make it self-aware of its position and depth sensing to build a 3D map of the environment using the Microsoft Kinect sensor mounted on it.

The remainder of the report is organized as follows. Section 2 focuses on how to put the system together , its architecture and the usage. We define our tests in Section 3, and detail our results in Section 4. Finally, in Section 5 we present conclusions and possible extensions.

2. TECHNICAL MATERIALS AND METHODS

This section focus on how to put the system together. We will list the components needed for the system and walk through the setup step to get ORB SLAM 2 up and running.

2.1. System Requirements

For this system, we used the following components:

- Astec Firefly equipped with Microsoft Kinect sensor
- Laptop (Asus N552vw , i7-6700hq, 16 gb ram)

2.2. Set Up Instructions

The work environment for this project is Ubuntu 16.04. We also need the following open source projects, ROS kinetic, Pangolin, OpenCV , Eigen3 , g2o since they are dependencies of ORB SLAM 2. We will list the resources we use to set up our system in this section.

1. **Install Ubuntu 16.04**. Ubuntu is an open source operating system popular among software developers, and it is the OS we used to set up this project. Here is a link to get Ubuntu 16.04 : `http://www.ubuntu-it.org/download`
2. **Install ROS Kinetic Kame**. ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. You can find the installation instructions here: `http://wiki.ros.org/kinetic/Installation/Ubuntu`
3. **Install ORB SLAM 2 dependencies**. You can find the list of the dependencies you have to install here: `https://github.com/raulmur/ORB_SLAM2`

4. **Install ORB SLAM 2**. Now you should be able to setup and compile ORB SLAM 2, the mapping algorithm we are using. You can find the source code and instruction on `https://github.com/raulmur/ORB_SLAM2`

5. **Install any missing library**. Most of the dependencies comes with either ROS or OpenCV, but you might still need to install libraries manually. Check RGB-D SLAM compilation message to see if your system is missing any library.

In order to use this system in a simulation work environment and integrate it with ROS we need also

1. **Gazebo**. Gazebo is a 3D dynamic simulator with the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. Gazebo offers physics simulation at a much higher degree of fidelity, a suite of sensors, and interfaces for both users and programs. We used version 7.5. Here the main reference: `http://gazebosim.org/tutorials`

2. **ETHZ-Rotors Simulator**. RotorS is a UAV gazebo simulator. It provides some multirotor models such as the AscTec Hummingbird, the AscTec Pelican, or the AscTec Firefly. We worked with the last one. Here you can download the package: `https://github.com/ethz-asl/rotors_simulator/`

3. **Rviz**. ROS 3D Robot Visualizer. If you don't already have this tool, you can get it from: `http://wiki.ros.org/rviz`

2.3. System Architecture

The architecture of our system in the simulation environment is depicted in figure 1. As we can see from it, the system is based on publish/subscriber form of communication. In particular the controller node we implemented, also use this mechanism to send messages on topic */firefly/command/pose*. Moreover we extended the RGBD node in order to make the pose of the hexacopter published on a topic (*/my_pose_stamped*).

In the case of the real environment the architecture is depicted in figure 2. Note that the Kinect node is absent because we used a rosbag to create this picture.

The highlighted components are the ones we modified.

2.4. System Usage

In this section, we will cover how to use the system.

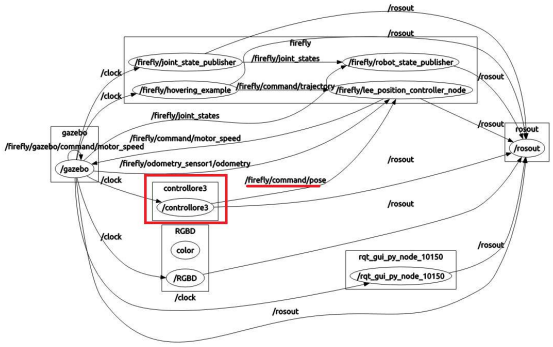


Figure 1. Simulation components graph

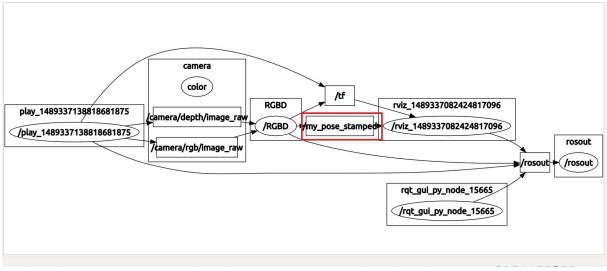


Figure 2. Real system component graph

2.4.1. Simulation. At first we must have a running roscore in order for ROS nodes to communicate. It is launched using the roscore command.

```
$ roscore
```

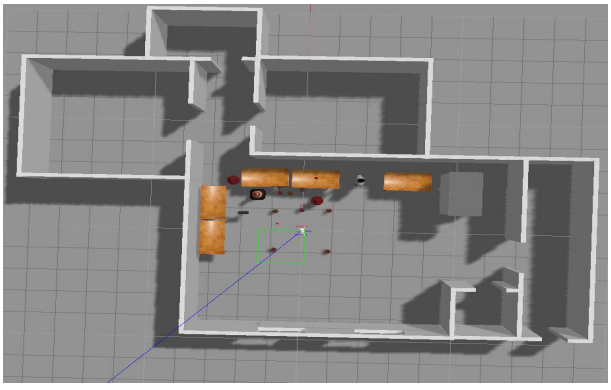


Figure 3. Elab2 world view

Then we have to launch gazebo with the chosen mav model (in our case, Firefly already equipped with the vi sensor to acquire images) and the world model. RotorS package provides some world model, we tried *basic* model (an empty world), *house* model and *test_city* model. We have also created another world

model (elab.world and elab2.world (3)) from scratch because we needed an environment with many objects.

```
$ roslaunch rotors_gazebo
    mav_hovering_example_with_vi_sensor
    .launch mav_name:=firefly
    world_name:=elab2
```

Now we can visualize the hexacopter flying in the world. To move it through the environment, instead of using manual commands directly sent to the hexacopter (and this can be done with the following command

```
$ rostopic pub /firefly/command/
    motor_speed mav_msgs/Actuators '{
    angular_velocities: [100, 100,
    100, 100, 100, 100]}') ,
```

we decided to implement a keyboard controller node. It publishes messages of type geometry_msgs/PoseStamped on topic /firefly/command/pose. The WASD keys are used for movement in the xy plane, QE keys for movement in the z-axis (up and down) and ZX keys to control the orientation and yaw of the drone (counterclockwise / clockwise).

```
\$ rosrun crm_ros controllore3
```

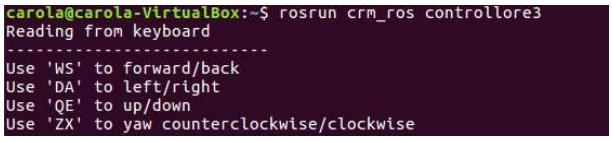


Figure 4. Keyboard controller

At this point we are able to navigate into the world as we want and we can start to map the environment and to localize the drone with the proposed algorithm. For an RGB-D input from topics */camera/rgb/image_raw* and */camera/depth/image_raw*, run node ORB_SLAM2/RGBD. You will need to provide the vocabulary file and a settings file. To launch ORB SLAM 2 with default parameters, according to the instruction on Github you have to type:

```
$ rosrun ORB_SLAM2 RGBD
    PATH_TO_VOCABULARY
    PATH_TO_SETTINGS_FILE
```

In our case we used the vocabulary provided with the ORB_SLAM 2 package but instead of using the Asus (Asus.yaml) settings file we created a new setting file (kinect.yaml) to tune parameters according to our needs. Moreover before launching ORB SLAM we have to remap the topics used by the algorithm (the code can be found in ros_rgbd.cc file). In the simulation we used */firefly/vi_sensor/left/image_raw* and */firefly/vi_sensor/camera_depth/camera/image_raw* topics provided by Gazebo and RotorS package.

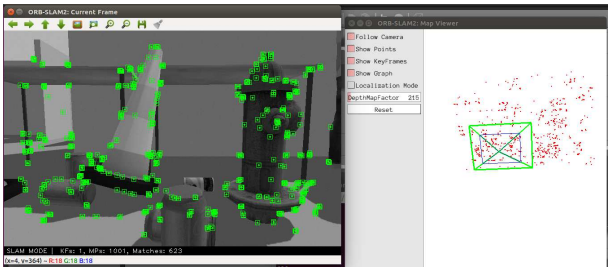


Figure 5. Orb Slam view

```
rosrun ORB_SLAM2 RGBD /home/carola/
    catkin_ws/src/ORB_SLAM2/
    Vocabulary/ORBvoc.txt /home/
    carola/catkin_ws/src/ORB_SLAM2/
    Examples/ROS/ORB_SLAM2/kinect.
    yaml
```

Finally, we can visualize all the topics described in this section also in Rviz (especially the estimated pose published by the RGBD node) typing the following command to launch it:

```
$ rosrun rviz rviz
```

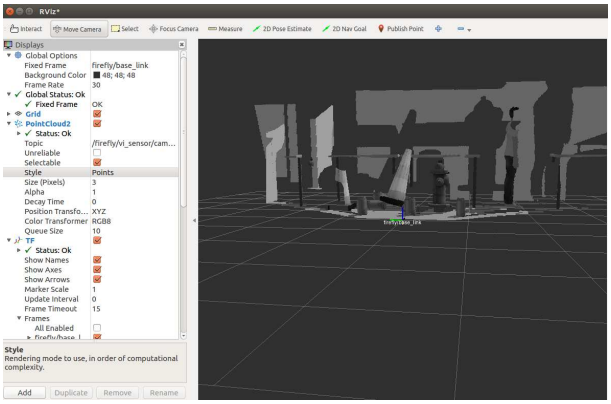


Figure 6. Rviz view

2.4.2. Real environment. To run experiments in the real world we firstly have to connect us to a ROS Network in which a roscore is running. We run roscore directly from the drone. Then we have to type, in our machine, the following command containing the IP address of the machine with roscore running and the port number:

```
$ export ROS_MASTER_URI=http
    ://130.251.13.167:11311
```

From the drone we also run Kinect node to start acquiring information from the camera (if you want to start the Kinect node on your laptop you have to install freenect package).

Then we can check the connection , typing in our machine

```
$ rostopic list
```

If all had been successful, the above command will show all the active topics of the system.

Now, as in simulation, we can launch ORB SLAM node, but we have to remap topics again first. This time, we receive messages from the kinect node and in particular from */camera/rgb/image_raw* and */camera/depth/image_raw* topics.

```
$ rosrun ORB_SLAM2 RGBD /home/
  carola/catkin_ws/src/ORB_SLAM2/
  Vocabulary/ORBvoc.txt /home/
  carola/catkin_ws/src/ORB_SLAM2/
  Examples/ROS/ORB_SLAM2/kinect.
  yaml
```

2.4.3. Shortcut Usage. To tweak experience, we created a custom launch file for the simulation environment. In this way, we could launch the system with a simple command:

```
$ roslaunch ORB_SLAM2 sim.launch
```

3. TEST

3.0.1. Testing system in simulation. After the system setup phase we have run a complete simulation test. Once we have activated all needed nodes, as described in the previous section, we have computed a trajectory using our keyboard controller. We checked in real time if the hexacopter was able to map the environment and if it was able to localize himself and draw an acceptable trajectory. The video showing this test is available here. The result was quite good as we can see from the Fig.7.

3.0.2. Orb slam parameters. Before trying the system in real environment we collected data from the Kinect of the drone but without let it fly with the rosbag utility. Thanks to this we noticed that parameters used in simulation don't fit very well for the real environment. In particular the *depthMapFactor* parameter of the file setting has to be modified according to the specific environment. The depthMap parameter is a scale factor that multiplies the input depthmap (if needed, see Fig.8).

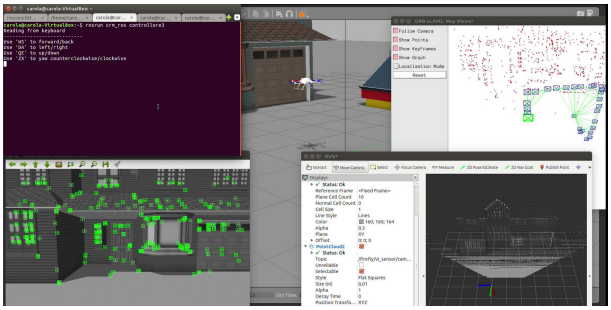


Figure 7. Trajectory view

```
if((fabs(mDepthMapFactor-1.0f)>1e-5) || imDepth.type()!=CV_32F)
    imDepth.convertTo(imDepth,CV_32F,mDepthMapFactor);
```

Figure 8. Usage of depthMapFactor

The tuning of this parameter can optimize the algorithm for a specific setting. Here there is a short video which highlights what happens by changing this parameter in simulation. In order to better understand the influence of this factor we run many tests with different values of it. To make this analysis easier and to avoid to stop the system and change manually the value, we added to the viewer a slider which allows to change the factor at run time, as depicted in Fig.9.

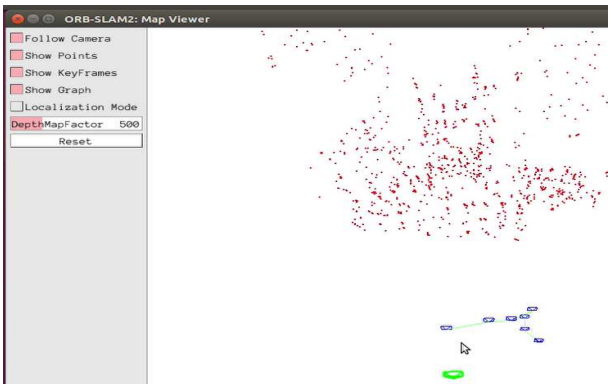


Figure 9. Modified viewer

We list all the tests we have carried out in the table in table 1.

3.0.3. Testing system in real world. Once performed all tests we have set the factor to 1000 and we validated the algorithm with the real robot in the EMARO arena. Here the video showing it. To see if the result is good, the idea was to compare the pose estimated by ORB SLAM with the real pose given by the Mocap system. The first problem we have faced is the different reference frame between the Mocap system and the ORB

DepthMapFactor	Video
1	simulation
1000	simulation
5000	simulation
500	real
900	real
1000	real
1500	real
5000	real

Table 1. Parameter testing

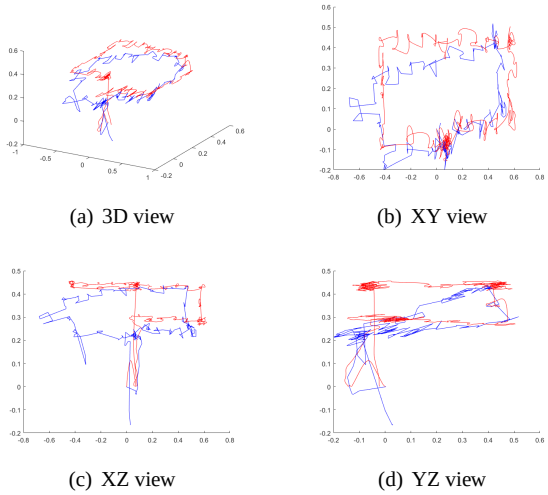
algorithm, so we changed the rotation matrix contained in the ros_rgbd.cc file to get them coincide. Then, due to the fact that the trajectory was not available at run time we added a piece of code to publish the estimated pose on a topic (/my_pose_stamped).

4. TEST RESULTS

To go further in the analysis of the results, we wrote a Matlab function to visualize the estimated trajectory and the real trajectory and to compare them.

```
function plotTrajectory( filename_bag, topic_real, topic_estimated )
%plotTrajectory plot a trajectory
%   Plot in the same figure the real trajectory (red line)
%   and the estimaded one (blue line).
%   plotTrajectory( filename_bag, topic_real, topic_estimated ) where
%       filename_bag is the path of your file bag
%       topic_real is the name of the topic publishing real pose
%       topic estimated is the name of the topic publishing ORB_SLAM2
        estimated pose
% USAGE
%       plotTrajectory('C:/Users/Carola/Dropbox/ROS
        /2017-03-09-18-36-40-900.bag','/Robot_1/pose','/my_pose_stamped' );

% Set path of your bag file
filePath = fullfile(filename_bag);
bagselect = rosbag(filePath);

% See avaiable topic recorder in rosbag
% bagselect.AvailableTopics;

% Select data from a specific topic stored in the bag file
%bagselect1 = select(bagselect, 'Topic', '/firefly/ground_truth/pose');
bagselect1 = select(bagselect, 'Topic', topic_real);
bagselect2 = select(bagselect, 'Topic', topic_estimated);

% Create a time series
ts_real = timeseries(bagselect1, 'Pose.Position.X', 'Pose.Position.Y', '
      Pose.Position.Z');
ts_estimated = timeseries(bagselect2, 'Pose.Position.X', 'Pose.Position.Y
      ', 'Pose.Position.Z');

% Visualize time series data
% ts.Data;

% Convert to matrix
m_real = ts_real.Data;
m_estimated = ts_estimated.Data;

% Offset
offz = m_real(1,3);
offy = m_real(1,2);
offx = m_real(1,1);

% Plot 3D data in the same figure
figure
plot3(m_real(:,1)- offx ,m_real(:,2)- offy ,m_real(:,3)- offz,'-r');
hold on
plot3(m_estimated(:,1),m_estimated(:,2),m_estimated(:,3),'-b');
hold off

end
```

The figure 4 shows the result of the final test with a depthMap factor set to 1000. It seems to be a quite good result. The fact that there is an error on the z-axiz on the left is probably due to the lack of objects in front of the drone. In fact, objects in our environment were concentrated in the center of the image. Consequently, at the center, the number of matches was very high, while moving the drone left or right it looses many matches generating an error in the estimated pose. Anyway the localization works with at least 30 matches. Under this lower bound it looses his trace and tries to reset the system.



(a) 3D view (b) XY view

(c) XZ view (d) YZ view

Figure 10. Trajectories comparison: the red line is the real trajectory, while the blue line is the estimated one. The axes scale is in meters.

5. CONCLUSION

With this project we proved that an implementation of ORB-SLAM non ROS-oriented could be integrated in a system working with it. We also demonstrated its performance and efficiency relative to different settings of parameters and environments. A possible extension of this project is to integrate a module for path planning and obstacle avoidance.

References

[1] Martinez, Aaron and Fernandez, Enrique, Learning ROS for Robotics Programming, 2013, Packt Publishing.

[2] Gautham Ponnu, Jacob George, REAL-TIME ROS-BERRY PI SLAM ROBOT, 2016.