Lab Course, SS15: Mobile Robots MA-INF 4310
Planning and executing flight from starting position to goal with Parrot AR.Drone type1

Linara Adilova

October 11, 2015

Instructor: Dr. Nils Goerke
University Bonn, Computer Science Department

Contents

1 Objectives

Objective is to make AR.Drone fly from the starting position to the goal position autonomously including obstacle avoidance.

First Objective
Setup AR.Drone controlling with *ardrone_autonomy* driver and *tum_ardrone* gui controller.

Second Objective
Make point cloud map of environment using *lsd_slam* library.

Third Objective
Use of point cloud map to plan movement of the vehicle with help of *MoveIt!* library.

Fourth Objective
Translate generated movement plan to commands for the robot to make it fly to the goal position.

2 Parrot AR.Drone description [1]

A quadrocopter is a helicopter, which is lifted and maneuvered by four rotors. It can be maneuvered in three-dimensional space solely by adjusting the individual engine speeds (see Figure 1): while all four rotors contribute to the upwards thrust, two opposite ones are rotating clockwise (rotors 2 and 3) while the other two (rotors 1 and 4) are rotating counter-clockwise, canceling out their respective torques. Ignoring mechanical inaccuracies and external influences, running

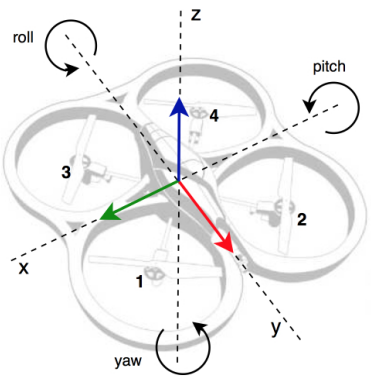

Figure 1: Schema of AR.Drone Parrot.

all engines at equal speed - precisely nullifying gravity - allows a quadrocopter to

stay in the air without moving. The following actions can be taken to maneuver the quadrocopter:

1. vertical acceleration is achieved by increasing or decreasing the speed of all four rotors equally,

2. yaw rotation can be achieved by increasing the speed of engines 1 and 4, while decreasing the speed of engines 2 and 3 (or vice-versa) - resulting in an overall clockwise (or counter-clockwise) torque, without changing overall upwards thrust or balance,

3. horizontal movement can be achieved by increasing the speed of one engine, while decreasing the speed of the opposing one, resulting in a change of the roll or pitch angle, and thereby inducing horizontal acceleration.

The fine tuning of the relative engine speeds is very sensible to small changes, making it difficult to control a quadrocopter without advanced controlling routines and accurate sensors.
The drone is equipped with an ultrasound altimeter, a 3-axis accelerometer (measuring acceleration), a 2-axis gyroscope (measuring pitch and roll angle) and a one-axis yaw precision gyroscope. The onboard controller is composed of an ARM9 468 MHz processor with 128 Mb DDR Ram, on which a BusyBox based GNU/Linux distribution is running. It has an USB service port and is controlled via wireless LAN.
The AR.Drone has two on-board cameras, one pointing forward and one pointing downward. The camera pointing forward runs at 18 fps with a resolution of 640×480 pixels, covering a field of view of $73.5° \times 58.5°$. Due to the used fish eye lens, the image is subject to significant radial distortion. Furthermore rapid drone movements produce strong motion blur, as well as linear distortion due to the camera's rolling shutter (the time between capturing the first and the last line is approximately 40 ms). The camera pointing downwards runs at 60 fps with a resolution of 176×144 pixels, covering a field of view of only $47.5° \times 36.5°$, but is afflicted only by negligible radial distortion, motion blur or rolling shutter effects. Both cameras are subject to an automatic brightness and contrast adjustment. The drone continuously transmits one video stream, which can be one of four different channels - switching between channels can be accomplished by sending a control command to the drone. AR-Drone 1 supports four modes of video streams: Front camera only, bottom camera only, front camera with bottom camera inside (picture in picture) and bottom camera with front camera inside (picture in picture). According to active configuration mode, the driver decomposes the PIP stream and publishes pure front/bottom streams to corresponding topics. The camera_info topic will include the correct image size.
The software running onboard is not accessible: while some basic communication via a *telnet* shell is possible, the control software is neither open-source nor documented in any way - while custom changes including starting additional processes are possible, this would require massive trial and error and is connected with a risk of permanently damaging the drone. Therefore it is better to

treat the drone as a black box, using only the available communication channels and interfaces to access its functionalities.
As soon as the battery is connected, the drone sets up an ad-hoc wireless LAN network to which any device may connect. Upon connect, the drone immediately starts to communicate (sending data and receiving navigational commands) on four separate channels:

1. navigation channel (UDP port 5554),
2. video channel (UDP port 5555),
3. command channel (UDP port 5556),
4. control port (TCP port 5559, optional).

The Drone is navigated by broadcasting a stream of command packages, each defining the following parameters:

1. desired roll and pitch angle, yaw rotational speed as well as vertical speed, each as fraction of the allowed maximum, i.e. as value between -1 and 1,
2. one bit switching between hover-mode (the drone tries to keep its position, ignoring any other control commands) and manual control mode,
3. one bit indicating whether the drone is supposed to enter or exit an error-state, immediately switching off all engines,
4. one bit indicating whether the drone is supposed to take off or land.

Being sent over an UDP channel, reception of any one command package cannot be guaranteed. In *tum_ardrone* package implementation the command is therefore re-sent approximately every 10 ms, allowing for smoothly controlling the drone.

3 Short introduction into ROS and packages

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. ROS is licensed under an open source, BSD license.
Ardrone_autonomy is a ROS driver for Parrot AR-Drone 1.0 and 2.0 quadrocopter. This driver is based on official AR-Drone SDK version 2.0.1. ardrone_autonomy is a fork of AR-Drone Brown driver.
LSD-SLAM is a novel approach to real-time monocular SLAM. It is fully direct (i.e. does not use keypoints / features) and creates large-scale, semi-dense maps in real-time on a laptop. Lsd_slam package implements this approach.
MoveIt! is state of the art software for mobile manipulation, incorporating the latest advances in motion planning, manipulation, 3D perception, kinematics, control and navigation. It provides an easy-to-use platform for developing

advanced robotics applications, evaluating new robot designs and building integrated robotics products for industrial, commercial and other domains.
RViZ is 3D visualization tool for ROS.

4 Working with AR.Drone

In order to start working one have first to charge the battery and plug it into the drone's battery-place. After that vehicle will set up hot spot wifi and PC has to be connected into that network. The network will be called "ardrone_CODE", where CODE is some numerical sequence. Now driver [2] for the vehicle can be started. First run *roscore* and then run command in order to create a connection to the drone from the *ROS*:
rosrun ardrone_autonomy ardrone_driver
One can now start trying to control the vehicle and all of its parts. For example, to watch the video from the cameras of the drone *image_view* package [3] can be used. The driver will create three topics for drone's cameras: *ardrone/image_ra*, *ardrone/front/image_raw* and *ardrone/bottom/image_raw*. Each of these three are standard *ROS* camera interface and publish messages of type *image transport*. Run command as following:
rosrun image_view image_view image:=ardrone/front/image_raw
This will display frontal camera video. In order to change channel *service call* can be used:
rosservice call /ardrone/togglecam
The driver is also a standard *ROS* camera driver, therefore if camera calibration information is provided either as a set of ROS parameters or through ardrone_front.yaml and/or ardrone_bottom.yaml files, calibration information will be also published via camera_info topics. In order to calibrate cameras just use *ros_calibration* package.
Next step is to start *tum_ardrone* package [4] in order to control the vehicle. Running the command:
roslaunch tum_ardrone tum_ardrone.launch
will show three windows - video image, map image and gui for controlling vehicle. All the information about working with it can be found on ros wiki page for the package [4].

5 Making a point cloud map

In order to make a point cloud map *lsd_slam* package [9] is used. One can just control the drone, using *tum_ardrone* package and start *lsd_slam* with command:
rosrun lsd_slam_core live_slam image:=/ardrone/front/image_rect
_calib:=/src/lsd_slam/calibration_front.cfg
Where file calibration_front.cfg is a text file that contains data on the front camera of the vehicle. I used this way of starting *lsd_slam* as the info from the /ardrone/front topic did not work for the camera parameter of the package (just

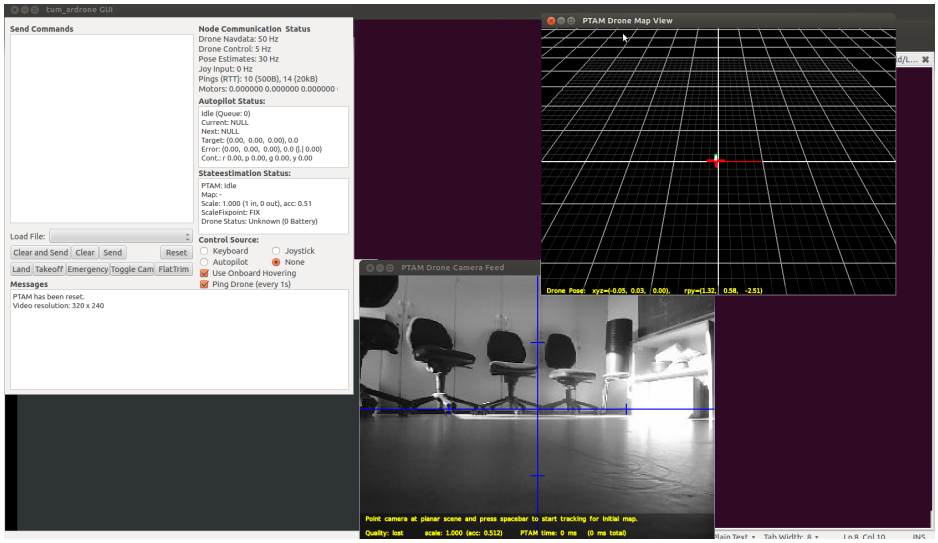

Figure 2: Tum AR.Drone gui: three windows - video image, map image and gui for controlling the vehicle.

hanging with 'waiting for ROS calibration' message, although it was already done).
In order to view currently recorded point cloud one can use *lsd_slam_viewer*:
rosrun lsd_slam_viewer viewer
Key moment - perfect calibration of the camera and right camera configuration file. In order to calibrate camera of the vehicle *camera_calibration* package was used [5]. After finishing the process described in tutorial for calibration one will get camera matrix [6]. For example I got data:

```
...
width
320
height
240
camera matrix
226.867688 0.000000 198.383827
0.000000 225.882639 114.079573
0.000000 0.000000 1.000000
...
```

There are several formats of configuration file for the camera described in the *lsd_slam* package description, I used the one for pre-rectified image [7]. From the camera-matrix I calculated next data for configuration file:

```
0.70896153 0.94117766 0.61994946 0.47533155 0
```

Figure 3: Picture of the scenery when the example testing was made.

320 240
none
320 240

In order to get pre-rectified image from the camera of the vehicle *image_proc* package [8] was used and topic *image_rect* was passed as input for *live_slam* node.
ROS_NAMESPACE=ardrone/front rosrun image_proc image_proc
Map can be built using some flight plan, sent on the vehicle with *tum_ardrone*, by manually moving drone's camera around, or by manually controlling it through *tum_ardrone*. I preferred the second option as the most safe. After some time of controlling rather satisfying cloud map can be seen in lsd_slam_viewer. Current

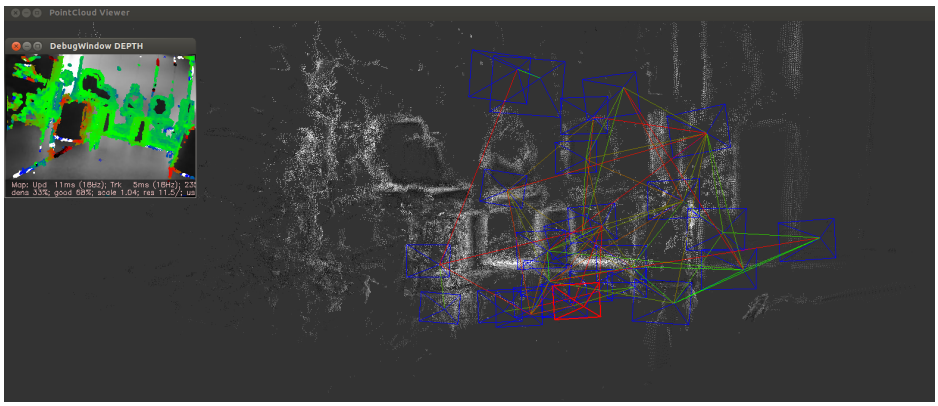

Figure 4: Registered point cloud for the testing scenery, see Figure 3.

point cloud can be saved from *lsd_slam_viewer* by typing 'p' while focusing on the window. This will create a file named 'pc.ply' in *lsd_slam_viewer* package directory. It is a file of 'ply' format that was designed for storing 3-dimensional data from 3D scanners [10].

6 Setting up MoveIt! for AR.Drone

In order to start working with path planner from *MoveIt!* package [11], the setup of the robot (quadrocopter in our case) has to be done at first. The setup assistant of the package helps to do this almost at once. The only thing, that has to be done manually is preparing the urdf file [12] for the robot.

```
< robot name="AR.Drone">
  <link name="drone">
    <visual>
      <geometry>
        <sphere radius="0.2" />
      </geometry>
    </visual>
  </link>
</robot>
```

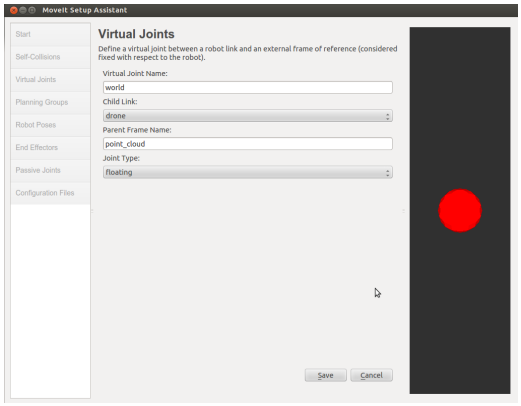

Figure 5: Configuration of joints for moving.

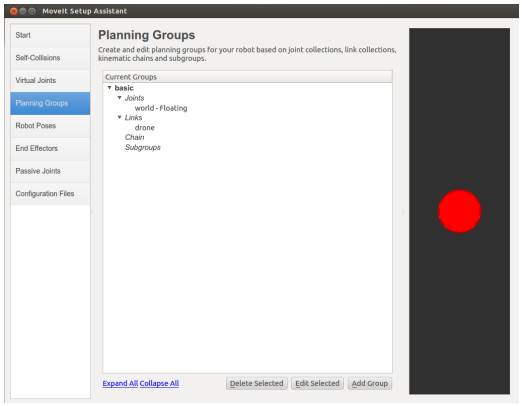

Figure 6: Configuration of planning groups.

Use this file in setup assistant, that can be started by:
roslaunch moveit_setup_assistant setup_assistant.launch
and add one virtual joint to floor with floating joint type (this means that link or drone will move in all 6 directions) and one planning group, consisting of this joint and the link of drone.
Then save configuration files - actually it will create a package for controlling exactly this robot. *MoveIt!* planner can be started with command (if package

was saved in folder ardrone_moveit):
roslaunch ardrone_moveit demo.launch

7 Loading point cloud to RViZ of MoveIt! and planning path

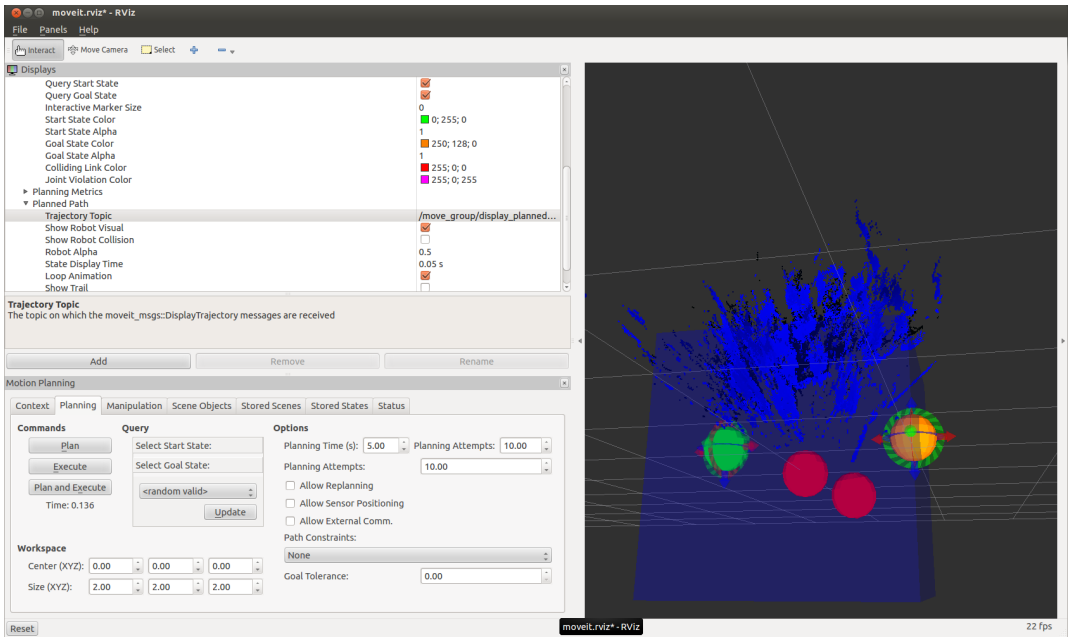

Figure 7: Point cloud in MoveIt!.

After creating pc.ply file with *lsd_slam* it can be loaded into *RViZ* for *MoveIt!* as map. The problem is that it is not quite in the needed format [13]. In order to read it with standard library, that reads *.ply files I had to import it into meshlab [14] and export again. After this I was able to read and publish it as *ROS* topic with *ply_publisher* package [15] in order to use in *RViZ MoveIt!* planner interface. So, if one will just run:
rosrun ply_publisher ply_publisher _file_path:=/pc.ply _topic:=/point_cloud _frame:=/point_cloud _rate:=1.0
topic will be created, that can be used as topic at yaml configuration of sensors for getting point cloud in environment. Note, that the frame name has to be the same as was used in configuring the MoveIt! frame. Config for the sensors looked like:

sensors:

```
- sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater
  point_cloud_topic: point_cloud
  max_range: 8.0
  point_subsample: 1
  padding_offset: 0.1
  padding_scale: 1.0
  filtered_cloud_topic: point_cloud
```

Sensor plugin is standard library in *MoveIt!* for loading point cloud sensors from robot. And the topic passed the name of topic from *ply_publisher*. Also the launch file moveit_sensor_manager.launch has to be updated by adding a line for loading previously created file

```
<rosparam command="load"
file="$(find ardrone_moveit)/config/ardrone_sensors.yaml" />
```

and updating the line with the name of the frame:

```
<param name="octomap_frame" type="string" value="point_cloud" />
```

After reloading the demo.launch point-cloud will be in the planning scene.
The plan of the movement can be generated just by dragging the start position and the goal position and clicking Plan button in planning tab. It can be noticed, that the trajectory will "see" the point-cloud and move in order not to make a collision. The planned trajectory will be available as moveit_msgs::MotionPlanResponse [16] and can be either displayed (as done for *RViZ* window) or used for controlling the vehicle.

References

[1] Master's Thesis in Informatik *Autonomous Camera-Based Navigation of a Quadrocopter* Jakob Julian Engel, DER TECHNISCHEN UNIVERSITAT MUNCHEN, 2011

[2] AR.Drone driver, *http://wiki.ros.org/ardrone_autonomy*

[3] image_view package, *http://wiki.ros.org/image_view*

[4] tum_ardrone package, *http://wiki.ros.org/tum_ardrone*

[5] Tutorial for calibration of the camera, *http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration*

[6] Camera matrix description, *http://docs.opencv.org/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html*

[7] Format of configuration file for pre-rectified images, *https://github.com/tum-vision/lsd_slam#calibration-file-for-pre-rectified-images*

[8] image_proc package, *http://wiki.ros.org/image_proc*

[9] tum-vision's lsd_slam package, *https://github.com/tum-vision/lsd_slam*

[10] PLY format, *https://en.wikipedia.org/wiki/PLY_(file_format)*

[11] MoveIt! platform, *http://moveit.ros.org/*

[12] Tutorial for building URDF description, *http://wiki.ros.org/urdf/Tutorials/Building a Visual Robot Model with URDF from Scratch*

[13] Issue with wrong header format in lsd_slam point cloud file, *https://github.com/tum-vision/lsd_slam/issues/22*

[14] Meshlab, *http://meshlab.sourceforge.net/*

[15] ply_publisher package, *https://github.com/ethz-asl/ply_publisher*

[16] Motion plan response messages, *http://docs.ros.org/hydro/api/moveit_msgs/html/msg/MotionPlanResponse.html*