FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Jiří Harasim

Automated Drone Boomeranging

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	prof. RNDr. Roman Barták, Ph.D.
Study program:	Informatics
Specialization:	Theoretical Computer Science

Prague 2017

Poděkování.
Děkuji Alici a rodičům za podporu. Bráchovi, že si ze mě dělal legraci. Panu prof. Bartákovi za pomoc i trpělivost. Edovi za to, jak mi vyšel vstříc. Jindrovi za spolupráci.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Automated Drone Boomeranging

Autor: Jiří Harasim

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: prof. RNDr. Roman Barták, Ph.D.

Abstrakt:
Práce navrhuje 3D navigační a plánovací systém pro autonomní vzdáleně řízenou kvadruptéru (dále dron). Řešení využívá senzorických dat drona spolu se zpracovaným obrazem čelní kamery, bez předchozí znalosti prostředí a bez použití navigačního signálu (GPS). Data z kamery jsou transformována do representace řídkým point-cloudem, ze kterého se vytváří mapa obsazenosti okolí s adaptivní velikostí buňek. Na vytvořené mapě je následně možné plánovat trasu letu s přihlédnutím k zaznamenaným překážkám. Výsledný plán je realizován jednoduchým kontrolerem.

Systém rovněž zahrnuje simulátor, na kterém je možné virtuálně provádět celý proces. Práce propojuje původně nezávislé a nekompatibilní systémy a vytváří z nich jeden funkční celek. Výsledek je demonstrován několika jednoduchými scénáři, z nichž jeden řeší problém navrácení drona na jeho počáteční pozici.

Klíčová slova: dron, navrácení zpět, kontrola letu, mapování, plánování trasy

Title: Automated Drone Boomeranging

Author: Jiří Harasim

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D.

Abstract:
The thesis proposes a 3D navigation and planning system for an autonomous remotely controlled quadcopter (drone). The solution uses the drone sensor data along with the data processed from the video camera image stream, without having any knowledge of its surroundings beforehand and without using any navigation signal (GPS). The video camera data are transformed into a sparse point-cloud representation, from it is created an occupancy map of the surrounding area with adaptive cell size. The planner can construct trajectory plans in the map, respecting the detected obstacles. The planned trajectory is executed by a simple drone controller.

The proposed system includes a simulator which enables virtual execution of the whole process. The thesis composes originally independent and incompatible subsystems into a single compactly working system. The functionality of the system is demonstrated on a few simple scenarios, one of which is the return of the drone to its starting location.

Keywords: drone, return back, control, mapping, trajectory planning

Contents

1. Introduction

There is a lot happening in the field of robotics, artificial intelligence and automation in today's world. A wide range of areas is being studied and we are trying to achieve things, that seemed hardly possible just a few decades ago. Robots come in different sizes and shapes. Various teams are trying to teach human-like robots to walk, play football or even to dance. Other projects focus on teaching swarms of small drones to cooperate and be more efficient in simple tasks. Cars are really close to be able to drive autonomously, we have automated package delivery and armies use certain types of Unmanned Aerial Vehicles (UAV) to perform difficult tasks from a military base far away to make sure that their pilots don't get hurt.

All these projects are amazing, rather costly and require larger teams of people. Human-like robots aside, they also hardly ever work really autonomously. Army UAVs are controlled from a base and autonomous cars and mail delivery systems often use some kind of a signal to locate themselves in a real world - either GPS or other sources of radio waves.

1.1 Motivation

When flying with the drone situation of a signal loss might occur or the drone might fly out of the operational range. In those cases it might be beneficial to have the drone come back to its starting location on its own, because we might be unable to operate the drone by original means. We may also consider a delivery situation of a small object to the starting location. An autonomous return of the drone in that situation will allow a person who would otherwise need to pay attention to the delivery to do something else.

For a drone to be really an autonomous and independent unit, it needs to gather all its sensor data on its own and as reliably as possible. A video camera is one of the most common sensors which can gather data about the world around the robot. It is cheap, provides huge amounts of data and recent increase of CPU and GPU computation power and advances in computer vision assures, that the robot can actually process the data provided by a video camera and use it efficiently. Furthermore, a video camera is a passive sensor, which means that more robots using it at the same time won't interfere with each other. There are millions of cheap drones with a single camera distributed among people already and more are selling every day. Therefore it makes sense to focus on developing autonomous and semi-autonomous systems with a monocular camera as their main sensor because so many devices are equipped with it.

It has been shown already in this article [3] that a precise control of a drone in a specific coordinate system is possible. However we would like to expand that idea and allow the system to build a reliable map using state-of-the art map-building technology and at the same time to provide easy access to autonomous planning framework to perform more complex and sophisticated tasks.

1.2 Thesis goals

The goal of this thesis is to implement a system for a Parrot Bebop drone, which will be able to receive high level commands and generate and execute a plan which avoids obstacles. To demonstrate the capability of the system, it will autonomously navigate back to the location where it took off (starting location). Parrot Bebop is a low-end quadcopter sold as a toy. The software will control the drone via Wi-Fi from laptop, but will use drone internal sensors only. Internal sensors are accelerometer, gyroscope, barometer, magnetometer which are considered together to be a drone Internal Measurement Unit (IMU) and a single video camera.

The system doesn't have any information about its surrounding beforehand and doesn't use any type of external navigation signal. The drone will try to reach the starting location via the shortest route possible and will try to avoid any static obstacle it encounters. It doesn't account for moving obstacles. As a fallback plan, the drone will have a simple backtracking procedure implemented which will navigate it back via original route and which will be run at demand. The whole scenario will be executable on a simulator.

1.3 Proposed solution

Returning the drone to its starting location requires an orchestration of many systems. We need a working system covering and combining sub-systems ranging from the drone driver, across the visual navigation systems and data fusion, covering map building and planning environment to the drone controller. If designed well, such system would be modular and could be easily modified to fit the needs of a specific project. It would allow for development and improvement of a single sub-system which we could swap with the old version of the sub-system and immediately compare the results with each other. The existence of system like that would limit the number of technical problems, which arise when we combine a number of originally independent systems together and which effectively delay the researchers in their intended research.

Robot Operation System (ROS) [2] is the most widely used platform for robotic research today. It has a large infrastructure behind it, which allows us to solve many tasks using previous solutions. On the other hand, it has limited support for the drone control systems. We found no over-arching project or complex solution addressing the need for a system, that would provide a researcher with a development environment which they could use as a base-line for their research. We found systems however, which provide partial solutions for the underlying problems which need to be orchestrated.

1.4 Thesis structure

We will discuss the state-of-the art in the field of drone automation in the second chapter. There are some papers about this subject with interesting ideas and results already. The system proposed by this thesis is introduced in chapter three. The fourth chapter will present the Parrot Bebop Drone in detail along with ROS which we will use as a platform to run the whole system. We will explore recent computer vision advances and discuss a few possible approaches to solve the Simultaneous Localization and Mapping (SLAM) and obstacle detection problems in the fifth chapter. The sixth chapter describes the Moveit! planning framework. The seventh chapter summarizes our implementation of the whole system. The eighth and the last chapters present experimental results and evaluation of the system.

2. Problem formulation and related works

We propose and implement a system which is able to solve the following general problem:

Given the actual position of the drone, autonomously find and execute a trajectory which leads to any target position in a specific set of coordinates, while avoiding static obstacles and respecting the drone's technical limitations. (general problem)

The problem to return the Parrot Bebop drone to the starting position is a specific case of the general formulation and thus a solution to the general problem is a solution to the problem of returning the drone to its original location as well. (the go-home problem)

2.1 Drone localization

One of the most critical tasks that each mobile robot needs to continually solve is precise localization. Without knowing its own location, the robot can move only using simple commands, such as "move forward for a while". To issue the robot with a command to move to a specific position, it must know its current coordinates and where the specified position is relative to those coordinates.

Although external localization mechanisms such as GPS are very useful, there are times where the robot cannot rely on them. Inside large buildings or underground complexes like tunnels and caves, the GPS signal is usually disturbed or not working at all. In environments like that, there is an increased need for the robot to localize itself. Devices and systems that use external navigation signals are therefore not comparable to our approach.

2.1.1 Common localization approaches

The goal of the independent localization system is to extract the path taken by the device from the sensor input stream in real time. The basic way to do that is to use the drone Internal Measurement Unit (IMU) estimations. Such measurements are limited by the sensor accuracy and are subject to an error that accumulates over time and usually does not provide information about the drone surrounding objects. Therefore we have to use other means of detecting the surrounding environment and correcting the IMU measurement over time. Video camera image stream is a possible data source to do that.

To process the data from the video camera stream into usable odometry and surrounding object information, we have to identify points at the individual images and match them across multiple images. We can directly extract the camera path from the corresponding point coordinates changes across the images, as illustrated in figure 2.1.1. As a side effect of the solution, we have the previously identified points at the images and their coordinates in 3D space. In environments where it makes sense to use a video camera as a sensor, those points are always objects to avoid and therefore we consider them obstacles.

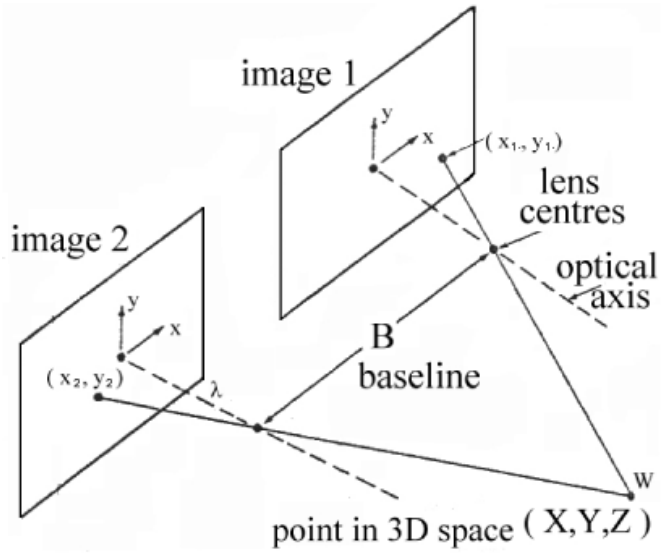


Figure 2.1.1: A schema comparing point coordinates in two different images.

The problem to solve then is to find matching points at different images and to calculate their coordinates relative to those images. The width and height of the point at any image is pretty much given. The difficult part is to get the depth of the point. The most common sensors used to solve this problem today are RGB-D camera and stereo cameras.

The RGB-D camera is an active sensor. It uses laser beams to scan the area in front of it and obtains the point depth as an integral part of its measurements. It is a very precise sensor used at multiple systems [31], [5]. The fact that it is an active sensor might disturb measurements if we use multiple devices equipped with this sensor.

In case of stereo cameras used as a main sensor [30] [8], we know the distance between the two images with the identified corresponding points, because we know their positions relative to each other. The solution to the problem then is to solve goniometric equations. From those we have point coordinates relative to the coordinates of the images and we could extract the depth of each point. Next time we encounter the same point, we know how its position changed relative to the previous encounter of the same point, which grants us the desired camera path.

In case of monocular camera though, we don't have two simultaneous images with a known distance between them. So the general idea is to take two or more images from the video stream, match the points on them and estimate the distance between those images from a mathematical model of a camera movement. If we want to apply this approach with the mobile robots, we can use their IMU estimations to provide us with some data regarding the movement between images too.

2.2 Comparable projects

2.2.1 Obstacle avoidance

DARPA is working on high speed obstacle avoidance using various technologies to detect the obstacles [5]. They used cameras, inertia measurement devices, and LIDAR and sonar sensors and successfully flew in a series of test flights in a hangar. The drone flew through an obstacle course avoiding all the obstacles and through another, not as difficult, reaching speed of 45 MPH.

Skydio is a company founded by a team of researchers from MIT and Google X's Project Wing. It developed one of the first reasonably functional real time high speed drone which flies in a real world and avoids stationary obstacles [6]. They showed a footage where the drone is following a guy on a bicycle in a park while avoiding tree branches and staying on course. They supposedly not only avoid obstacles but also fly with good path planning, so the drone is using an efficient route. Their drone is captured in the figure 2.2.1.

Figure 2.2.1: A skydio drone following a bycicle

A Chinese company DJI introduced a device "Guidance" [30] mountable on top of a drone. It uses four stereo cameras to detect obstacles all around the drone. Along with their SDK, the Guidance allows a robot to detect and react to obstacles around it.

2.2.2 Autonomous navigation

The Blue Bear company in cooperation with the university of Bristol created a system very similar to the system described in this thesis [8]. They don't rely on a GPS signal and they have created a program which they call "Smart boomerang". It uses a point cloud created from stereo cameras to navigate the drone through environment back to the starting point. The fact they employ stereo cameras as a sensor instead of a monocular camera in our system is the key difference as we discussed in the Common localization approaches subsection. The following figure 2.2.2 shows the drone used by Blue Bear.

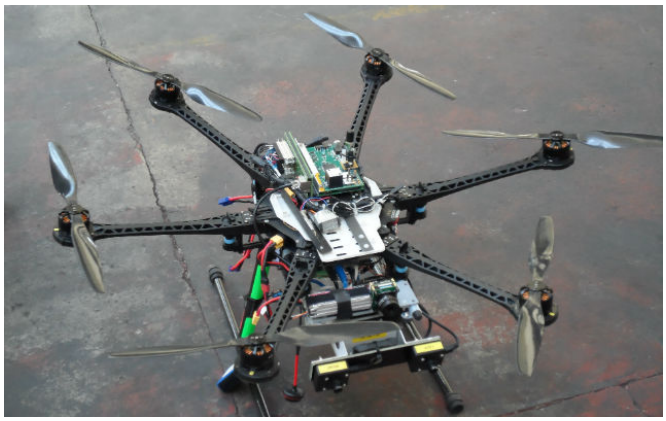


Figure 2.2.2: The Blue Bear drone

The National Institute of Astrophysics, Optics and Electronics (INAOE) in Mexico developed a vision and learning system to control and navigate the drone outdoor without relying on a GPS signal or trained personnel. [7] The system is supposedly able to account and react to air currents, while having limited computational power. One of the authors was working with the Blue Bear previously. Their system is developed to work outdoors, which is a different navigation problem due to the nature of the environment.

There is an article presenting Micro Aerial Vehicle (MAV) autonomous navigation [31] based on a RGB-D camera. They also use octrees as their map representation. They tested their approach on both simulated and real drones, one of which shows figure 2.2.3. The fact that they use a RGB-D camera as their main sensor is the key difference between our systems as discussed before.

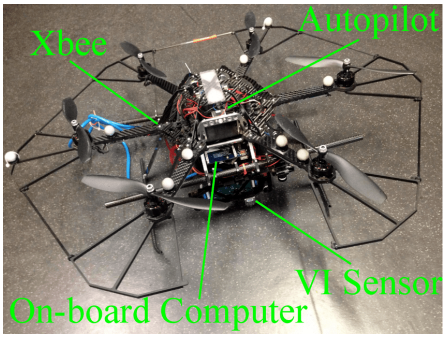


Figure 2.2.3: A custom built MAV with RGB-D camera

The article [3] shows that the monocular camera navigation of a quadcopter is feasible and working approach to autonomous navigation. The article presents sophisticated approaches to the problems, including PID Controller, Extended Kalman filter to fuse the data and an effective SLAM system. However it does not include an octomap [22] representation of the world and does not offer automated planners, neither a simple way to build a custom plan and to easily modify the planning scene.

3. Parrot Bebop

Parrot Bebop drone (drone, Bebop, shown at Figure 3.0.1) is a radio controlled flying quadcopter built by the French company Parrot. It can be controlled by a wide variety of devices, mostly smart phones and tablets. It creates a Wi-Fi hotspot to which the control device connects and sends commands. It supports iOS and Android devices by default, but there is a lot of unofficial software available for other platforms. It is used for video recording purposes, to play games with it and for relaxation mostly. Bebop is a descendant of the Parrot AR 1.0 and 2.0 models.

Figure 3.0.1: The Parrot Bebop drone

3.1 The drone technical information

The drone is constructed to be able to fly both indoors and outdoors. It is supplied with propeller protectors which are used indoors so the quadcopter would withstand a bump into an obstacle without any serious propeller damage. The protectors are usable outside as well, but the drone is then more likely to be affected by a wind or a draft. The drone battery lasts up to 11 minutes of flight.

We will fly indoors for the purposes of this thesis, so we will consider the drone with the propeller protectors always equipped. Bebop's dimensions are 330 x 380 x 36 mm. It weights 420 grams. Maximum speed is 13 m/s. Operating range of the controlling signal is 250 meters.

Bebop has P7 dual-core CPU Cortex 9 with quad core GPU. It has an 8 Gb internal flash memory used to store photos or videos. The propellers are energized with 4 brushless outrunner electromotors. The operating system runs various low level procedures to orchestrate the propellers and maintain flight stability. The drone has the following sensors:

- 3 axis accelerometer
- 3 axis gyroscope
- 3 axis magnetometer
- Optical-flow sensor: vertical stabilization camera
- Ultrasound sensors for ground altitude measurement
- Pressure sensor
- "Fish-eye" lens 180°camera, 14 Mega pixels, digital video stabilization, 1920x1080p, 30 fps

In general the drone flight feels very stable and polished. It is much more stable than its predecessors AR drone and AR drone 2.0.

3.1.1 Interface

The drone on-board processor runs the Linux operating system. It controls the propellers, all the sensors, creates the Wi-Fi hotspot, provides starting and landing procedures and flight maneuvers such as backflip. If any device connects to the Wi-Fi access point, it is then able to communicate with the drone and issue it with commands. Parrot provides an open source SDK for this.

Parrot also provides a way to control Bebop, a device called Skycontroller. It comes with virtual reality glasses or a monitor. There are other independent applications which are usable with smart phones or tablets, such as "FreeFlight" for android and iOS devices.

As the drone primary purpose is meant as a portable toy, there is no official application for PCs. We will use "bebop_autonomy" ROS package, which will handle the communication with the drone from ROS.

When Bebop bumps into anything during a flight so badly that any of its propellers slow down, the drone switches its state into emergency mode. That means that the drone stops all the propellers and lands. It is uncontrollable in this mode, as the drone basic software attempts to prevent any permanent damage taking this action and ignores all incoming commands.

3.1.2 The sensor data

Apart from the drone using the sensor data itself to regulate its flight, it also publishes an abstraction of the drone state to the channel defined in the drone communication protocol with frequency of 5 Hz. It is called "navdata" and contains estimated speed, orientation, movement and other information. We will discuss the exact form of the navdata later in this chapter.

3.1.3 The video camera data

Bebop video camera records HD video at 30 fps rate with resolution 1920 x 1080p and has a digital stabilizer in all three axes. The video is encoded in H264 format. It is capable of taking photos with resolution 4096 x 3072p in JPEG, RAW or DNG format. Both the video and the photos share 8GB internal flash memory. The "fish-eye" type camera films a panoramic image and the Bebop firmware converts it into the smaller resolution video so that the image is always in front of the drone, even if the drone is tilted. It is also possible to receive a different part of the main camera image than the forward view.

The feature to always see straight and not tilted image is very useful and it is one of the reasons why Bebop is much better than AR drone for our purposes as it allows us to detect obstacles reliably. It is possible to control which part of the fish-eye image is send from Bebop. The mechanism controlling it is called a virtual camera and the Bebop firmware always sends the front image if not told otherwise. It is possible to move the virtual camera with a message containing desired absolute tilt and pan of the camera in degrees.

3.2 Bebop interface to ROS

The drone is controlled via a laptop in this implementation. More specifically, it utilizes the "Robot Operating System" (ROS) [2]. ROS is an open source platform, used around the world to write robotic software for all kinds of robots. It has many libraries providing sensor access, logging, experiment recording, video processing and others. It is described in detail in the appendix A.

There is a "bebop_autonomy" package [4] developed at Autonomy Lab of Simon Fraser University, which provides an interface for all communication between ROS and Bebop. It sends the commands to the drone and provides the sensor and camera data from it.

3.2.1 Commands

The package comes with a launch file for teleoperating Bebop with a joystick using ROS "joy" package. It is very simple to adapt the launch file to the specific controller. We are using a X-Box One controller.

Commands are sent to Bebop using topics with messages of certain types. Basic commands such as takeoff, land and emergency are sent by publishing an empty message to the corresponding topics specified by the driver.

Piloting the drone is done by publishing a message which contains a value to move in a desired direction. The driver accepts values from $[-1, 1]$ interval in every Degree of Freedom (DOF) [1] the drone has. The meaning of the signal values is explained in the list below when you map the $[-1, 1]$ interval to it. It also lists all DOFs the drone has:

- linear.x [backward, forward]
- linear.y [right, left]
- linear.z [descend, ascend]
- angular.z [rotate clockwise, rotate counter clockwise]

3.2.2 Bebop readings (navdata)

The real time video sent from Bebop via Wi-Fi is much lower quality than the real drone capabilities as the Wi-Fi bandwidth is not able to transmit such a high video quality in real time. The quality of video stream is therefore limited to 640 x 368 at 30 fps. The field of view of this virtual camera is 80 horizontal and 50 vertical degrees. Video stream from the front camera is published via an image stream.

[1] A *Degree of Freedom (DOF)* is the number of independent parameters that define the system's configuration. A floating object in a $3D$ space has 6 DOF - translation and rotation along each of the three axis. The Bebop driver transforms this into 4 DOF as we have no direct control over the rotation along the X and Y axis.

The drone estimated position, orientation and velocity is published into the "odom" topic. It is calculated from the Bebop navdata and is limited to update rate of 5 Hz. The underlying navdata are published selectively by the Bebop firmware at the specified rate and only when the value has changed. Most of them are already processed into the odom topic, but it is possible to access other data. The interesting subtopics in the odom topic are:

- the index of a record
- the time of a record
- the current estimated drone position in space
- the current estimated drone orientation in space
- the current estimated speed of the drone split into axis components

4. Proposed Solution

To solve the general problem formulated in the chapter 2, the drone needs to localize itself on its own. We can count on it having some kind of Internal Measurement Unit (IMU) which is usually composed of accelerometer, gyroscope and a flight altitude measurement device. It might have more sensors which might improve the IMU position estimate. Bebop specifically also has a magnetometer, optical-flow stabilization sensor and barometer.

We can count on a frontal camera in case of a low cost quadcopter, but the system should be designed in a way that it could utilize dual cameras or an RGB-D sensor as well. Any of these sensors will provide us with their own odometry estimates along with the pointcloud of detected obstacles. Therefore we have two sources of the odometry data which we need to fuse.

The map has to be built based on the pointcloud and the fused odometry data and the drone needs to know its position in the map. That solves the localization part of the problem we formulated.

We need a planner which generates a trajectory with respect to the built map and with respect to the drone's technical limitations.

Finally, we need a drone controller to execute the trajectory generated by the planner.

Based on the requirements mentioned above, we propose the system architecture displayed in figure 4.0.1.

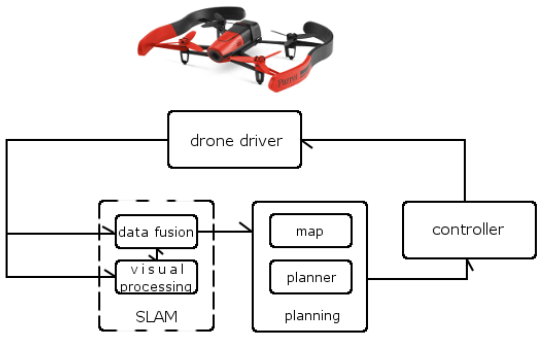


Figure 4.0.1: The proposed system schematics

The proposed architecture is composed of several sub-systems which fill their specific roles. The communication between the subsystems is mediated by the Robot Operation System (ROS) messages. Next section presents the packages we use to implement specific sub-systems. If such a need arises, individual packages are easy to replace by different packages which are able to fill the same role because they share the same interface in the form of ROS messages. More about how ROS works is at the attachment section and on the ROS web pages [2].

We didn't include a Simultaneous Localization and Mapping (SLAM) system in this thesis, which is why it is dashed in the schema. It naturally fits in as it consumes pointcloud and odometry data (fused or not fused, based on the SLAM) and produces corrected pointcloud and odometry data. We discuss the absence of the SLAM system in the Visual odometry chapter. It comes down to the great effectiveness of the used visual system.

4.1 Used sources

The "bebop_autonomy" [4] Robot Operation System (ROS) package serves as the drone driver in our implementation. We already discussed it in the previous chapter. Basically, the drone's output consists of a video stream and of an odometry estimate from the IMU mediated by the driver. The drone's input is a steering command stream with special commands to takeoff, land etc.

We use Direct Sparse Odometry (DSO) as the visual system implementation to localize the drone [1] from the camera stream and to produce the obstacle pointcloud. There are more sophisticated SLAM systems around, some of which will be discussed in the chapter about visual odometry. DSO has good runtime results and is capable of producing reasonable data even though it does not implement a long term loop closure. Figure 4.1.1 shows an image generated by the DSO system. We will look closely at it again in the visual odometry chapter.

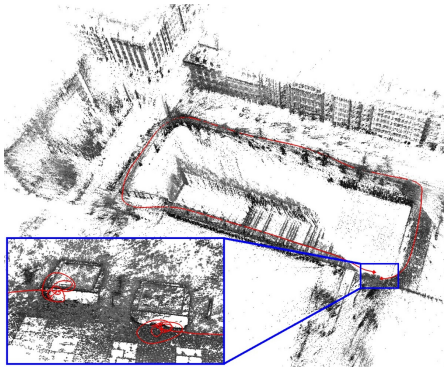


Figure 4.1.1: An image rendered using an output from the DSO system

If there arises a need for the long term loop closure [1], DSO can be swapped with a full SLAM system or the SLAM part which deals with long term loop closures could be added on top of the data produced by the DSO. The output of the DSO is the same as the one of the SLAM would be - an odometry estimation and a point cloud with detected obstacles.

The Extended Kalman filter and Unscented Kalman filter [34] are used to smooth the output of DSO and to fuse it with the drone odometry. They are implemented in robot_localization package [28]. How to configure the package is well documented in the robot_localization tutorials.

We use Moveit! (Moveit) package [16] to create a planning scene represented by the octomap [22] to store the explored map and to query it efficiently in a real time environment. Moveit! uses Open Motion Planning Library (OMPL) [23] internally to autonomously generate motion plans. We haven't found other comparable solutions both in complexity and reliability. When we were considering if we want to use Moveit planning environment, the other possibility was to implement everything ourselves. That would require extensive development and the result would implement only one of the simpler planning algorithms - probably A* or some variant of it. We decided that it would be much better to have a complex modular environment which is being developed for more than five years right now.

We use Moveit modifications and a modified controller implementation based on the work of Alessio Tonioni [21] to allow Moveit to work with a drone. Tonioni's project modifications are made for a Gazebo simulator [24]. The scene captured in figure 4.1.2 shows the simulated scene from the modified Moveit. The use of the simulator is shortly described in Appendix B. The modifications to use Moveit with drones are greatly described by Wil Selby [19] who uses it to control a drone via GPS. The simple controller used for the simulation is massively modified in this thesis to fit the real drone instead of a simulated one. We added boost mechanisms with speed limits and a simple pre-calibrated drone flight model to improve the precision of the real drone control.

[1] *Loop closure* is the problem of encountering and recognizing previously-visited location and updating it accordingly. It might happen by visiting the same location for the second time or by wrongly assigning an odometry update to a wrong location and visiting that location later.

We use a ros package Rviz [29] to view the scene with the drone. Moveit introduces a plugin to Rviz which allows for easy use of planning environment.

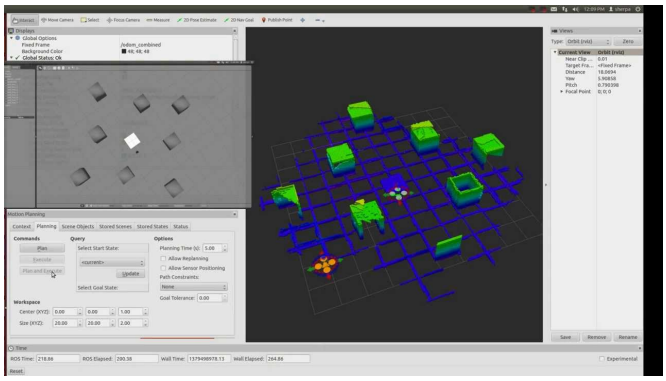


Figure 4.1.2: An image of Moveit! with a simulated quadrotor and scene in Rviz

4.2 The go-home problem solution

The system described earlier in this chapter allows us to automatically generate plans with obstacles included in the planning scene and to execute the plans at demand. We will take two approaches to the problem. The trajectory generated by both of the approaches will be visualized in Rviz. The plan will be executable by the controller on demand.

4.2.1 Backtracker

First one will be a backtracker. We will record the odometry messages created by the odometry fusion system and remember the positions in them. The records will be gather with the same frequency the drone publishes its odometry data, which is 5 Hz. We will slightly filter the messages, so that when the drone stays in one place longer or moves very slowly, we do not have a lot of the same or very similar messages in a row recorded. When demanded, we will create a plan from the path taken backwards.

4.2.2 Boomerang

The second approach will be called a boomerang. It prompts a planner, which is automatically assigned by Moveit from the pool of planners, to generate a plan from actual drone location to the starting location, specified by coordinates $(0, 0, 1)$. The planner is set to optimize for the shortest route and respects the obstacles recorded in the octomap. We will talk more about the specifics of the planning process in chapter 6.

5. Visual Odometry

The visual odometry is the most important part of navigating the robot as it provides it with the data necessary to avoid obstacles. There are many systems that provide monocular visual odometry or monocular Simultaneous Localization And Mapping (SLAM) system. With recent development in this area, the monocular systems could become viable alternative to RGB-D vision or stereo cameras.

In this chapter, we will describe the Direct Sparse Odometry (DSO) [1] and compare it with other monocular localization systems. We will also present our integration of DSO into ROS.

5.1 Monocular odometry

In the past, the problem was mostly approached from the indirect perspective. Such cases were monocular SLAM [9], PTAM [10] or ORB-SLAM [11]. An indirect approach first extracts features (often landmarks) from raw images and then finds corresponding features on other images. Second, it interprets those features as measurement from the camera and estimates the camera path and the positions of surrounding objects.

In contrast, the direct approach uses points in raw images without any preprocessing and attempts to match them directly onto each other to estimate the path and the surrounding world. [12] [1]

Another classification of approaches to solve the problem is how they treat the relations between points or features identified directly or indirectly. The dense approach processes the whole image from the camera and the semi-dense approach processes only parts of it, but in both of them the identified points have defined neighbors. The sparse approach only uses an independent subset of points, where no neighborhood is defined. According to the DSO authors, there was one work trying to do the direct sparse approach before, but it didn't use any deeper camera model and only tried to employ Extended Kalman Filter which didn't prove competitive with other approaches at the time.

5.2 Direct Sparse Odometry

The authors state and demonstrate in their paper [1] that this system outperforms its current competitors in SLAMs mentioned before. Validated by real performance of their system in this implementation, DSO is capable of stable and sufficient run-time performance. With some minor changes to the interface and adjustment of the bebop behavior, we were able to successfully use DSO in real world implementation.

DSO operates with key-frames. The expectations on key-frames are that there are sufficient differences between them with respect to the optical flow and the distance between the key-frames themselves. DSO forces a new key-frame when the optical environment changes significantly. In every key-frame, the DSO in our implementation has 1200 candidate points. It tracks them, but doesn't use all of them in a model. The candidate points are chosen in a way, that covers the whole image and that the parts of the image with the highest gradient are covered the most. The algorithm splits the image into regions, evaluates the gradients in them and picks the points with the highest gradients in every region. It also tracks all of candidate points.

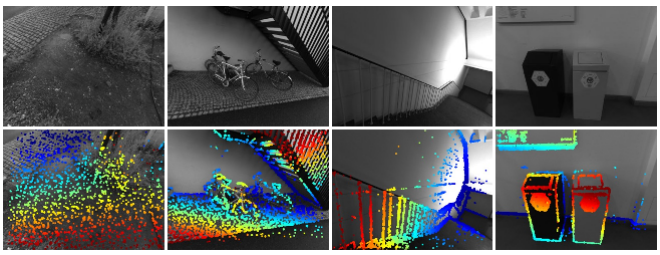


Figure 5.2.1: DSO images with calculated depth of tracked points.

DSO decides which points should be activated and used in a model based on uniform spatial distribution across the image and to be equally distributed across all key-frames. Figure 5.2.1 shows an example of the active points in various frames. Each key-frame also remembers inverse depth of its activated points. It has 1200 active points across all key-frames which is the same number as candidate points in each key-frame. It marginalizes key-frames that cover less then 5% of visible points and those that have the highest distance score when DSO has too many key-frames. When a key-frame is marginalized, all points on it are threw away.

The model is based on optimizing the photometric error with respect to the camera calibration model for image formation. The model can account for geometric and photometric camera calibration and its vignette for every specific device. The DSO has a complete tutorials on how to perform such calibrations. The model is described in detail in the DSO paper [1] and is quite technical.

5.3 Integration into our system

The integration with our system was bumpy, as the ROS wrapper interface had a bug in it. The method "publishCamPose" published corrupt data without knowing it. We reported the issue to the author and introduced a workaround, which solved the problem.

The wrapper right now publishes camera pose update to the tf tree and actual camera pose to the camera odometry topic. It also publishes the pointcloud. As the DSO odometry measurements have a different coordinate base than the drone odometry measurements, it is necessary to transform the data to the drone coordinate base. We worked on a solution to this issue, which computes the transformation matrix from the eigen library [14] based on the Least-Square Estimation [13]. The solution was somewhat working, but was less reliable than simple constant transformation based on experimental data. There were two possible explanations for it. The time-stamp of the messages might had been off, so the system matched messages from the different real world positions. Or the problem could have been that the odometry detected by the DSO system was not a smooth line, but rather really curve one. It is possible, that messages took from the DSO system were the ones that strayed from the real path and noised the conversion.

We stopped the work on this problem so that we could finish the whole system rather than have one sophisticated part in an incomplete system. So right now, the system uses a fixed transformation along with the Unscented Kalman Filter (UKF) to fuse both the odometry data into the bebop original coordinate system. That specifically means, that the starting position we refer to is framed in the drone natural coordinate system. The transformation constant is parametrized from the ROS parameter server. If these parameters are not set, no transformation happens as a default behavior.

The DSO behavior is relatively stable in our implemented setup, but we still wanted to add some robustness for occasional shutdowns. So we created a node responsible for monitoring the DSO system and when DSO crashes, the node restarts the whole system. If that happens, there will obviously be a hole in the pointcloud and odometry data from the DSO, but the new pointcloud is located correctly and the Kalman Filter merging dso odometry and the drone odometry handles the merge well too.

5.3.1 Parametrization

To setup the dso to drone base transformation data, parameters should be added to the launch file or to the ROS parameter server directly. They have a type of double and their names are dso_base_tr_[xyz]_rotation, dso_base_tr_[xyz]_scale and dso_base_tr_[xyz]_translation. The rotation is in radians and is applied first, scale is a simple multiplication and is applied second and translation is in meters and applied last. Future modification to update these parameters dynamically which could be based on a more complicated model should be a simple matter of one update procedure.

To stabilize the output of the DSO, we use speed limiter for our flight. The speed limits should be a value between 0 and 1 as those are the acceptable values for the drone command as specified in the driver documentation. The limits are loaded from the ROS parameter server from the parameters cmd_vel_limit for a translation limts and cmd_vel_rotation_limit for a rotation limit, both of type double. We use value 0.3 in our implementation for both of these limits.

6. Moveit!

Moveit! [16] is an open source planning library, developed primarily to operate robotic hands, but it has been used with some mobile ground robots too [15]. It incorporates motion planning, 3D perception, kinematics, control interface, execution monitoring, and navigation in 3D space. It comes with an easy to use rviz plugin, which presents the robotic scene with a control panel. The planning scene is internally represented as an octomap. Moveit is a modular system and already has a lot of automatic planners, a few kinematic solvers, octomap updaters, and other components implemented. A new component can be implemented easily thanks to the Moveit plug-in architecture. It is also very straightforward to create and execute a concrete scenario from code. The whole system can be used with the Gazebo simulator [24], which gives us a possibility to test and refine plans or new planners in a simulation first.

ROS has a navigation stack [17], which does the same thing as Moveit does, but only in 2D for mobile land robots. There is a project for navigation in 3D on ROS wiki [18]. But in fact, it plans in what we call 2.5D as it introduces multiple 2D layers. It is meant for the mobile ground robots as the original navigation stack is. I have not found a pure 3D navigation or planning library in ROS other than Moveit.

6.1 Moveit architecture and concepts

The core object of the Moveit architecture is the move_group object. Figure 6.1.1 captures the moveit architecture. Move_group gets updates from the robot, its sensors, robot controllers or from user interface. The robotic data are loaded from unified robot description format (urdf) and semantic robot description format (srdf) files. Those files among other information contain the description of robot parts as links, how they are joined together with joints and which ones we can move and with which limitations.

More data are loaded from configuration, which specify planners, kinematics, sensors, controller interfaces, robot and joint state topics and other. Moveit builds its own octomap [22] representation from the incoming pointcloud and can also accept other objects which it incorporates into its planning scene. It communicates with its surrounding systems utilizing ROS messages and services. The generated trajectory is using the ROS action library [20] to provide an action for controller to execute.

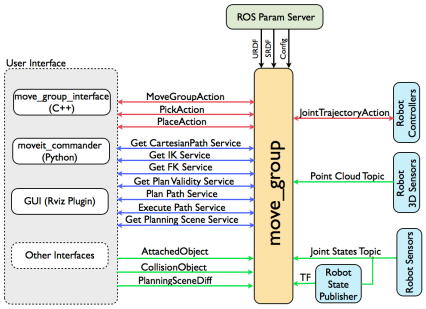


Figure 6.1.1: Moveit architecture schematics

6.1.1 Adaptation to drones

It was shown and tested in a different project [19] that it is possible to setup the Moveit system to operate a quad-copter. We specify the drone as a virtual floating joint to the world and allow planning for the drone base. Moveit provides a Setup Assistant and following the instructions [19], it is possible to setup Moveit for the quadcopter easily.

Moveit doesn't have an action to follow the generated trajectory with a floating joint. Therefore we need to define our own action and modify the action manager the system uses to be able to process this new action. Figure 6.1.2 shows the action server / action client interaction. When an action for a floating joint is generated, the action server announces the action to be handled. Moveit controller manager makes sure, that it is delivered to a controller with appropriate action handle.

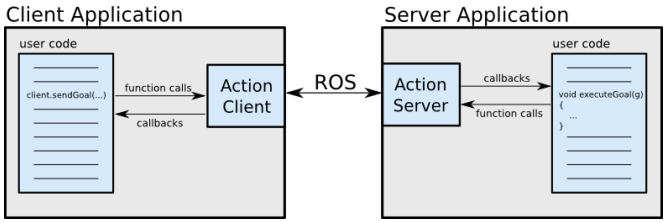


Figure 6.1.2: The schema of how the action is generated and processed using the action library.

6.2 Octomap

The planner needs a planning scene which we can update in real time and we need to be able to quickly answer traversability queries to it. There are many different possibilities how to represent a map, but a state-of-the-art map representation for this job is the octomap. Figure 6.2.1 shows an example of an octree scaling on a car model. Moveit uses its own encapsulation of the octomap.

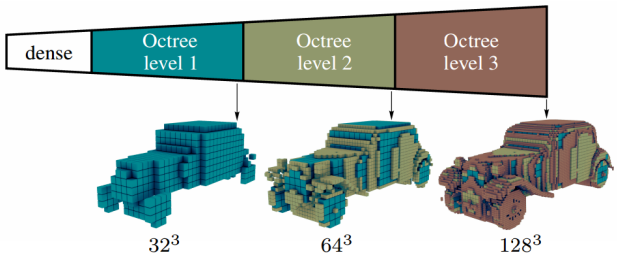


Figure 6.2.1: An example of a octree scaling.

The octomap uses an octree structure to store information in cells with dynamic edge sizes. Every node in the tree has none or eight children which represent the node with eight subnodes. Every child has the edge length half of the original node. An octree specifically remembers free space, occupied space and unexplored space. This representation saves a lot of memory for big chunks of the same type of space. An encoded octree is displayed in figure 6.2.2 It is a bit slower for the traversability checks though, because neighboring nodes are not close to each other in memory. Those checks are still fast enough to be usable in real time though.

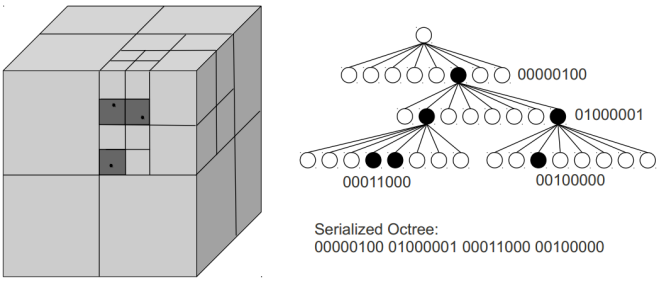


Figure 6.2.2: Octree schema with encoding

The implementation used in Moveit allows to limit the minimum node edge size and limit how big part of the environment around the robot should be remembered. This is very nice to save memory, so if we knew in advance the size of the relevant map and if we could determine reasonable resolution of the map we need to avoid obstacles, we could set up the octomap to effectively save memory. Also the traversability queries will be faster, because the depth of the tree will be limited. Based on the Bebop size and the fact that we need to have some distance from obstacles, we decided to use the cell limit size of $10cm$. We also use map size constraints of 7.5 to save memory. That means that the map parts that are more than $7.5m$ away from the actual drone position are forgotten. The

areas outside of the map limit are not forgotten right away, but only when the drone moves in the certain direction closer to the recent edge of the map. Both the cell size and the map limit are loaded from the configuration file specified in Moveit tutorials and could be changed easily.

Octomap uses a probabilistic model of a world. Every observation that certain node is occupied or that it is free changes the probability of occupation in that node. If that probability breaks a treshold, the node is either considered occupied if it was free before or it is considered free if it was occupied. This probability is internally represented in log scales to speed up computing. Furthermore, the probabilities are clamped on each side with a clamping limit, which enables the nodes to be more flexible and adapt to the change in the environment fast enough. More information about octomaps could be found at its presenting paper [22].

6.2.1 Map updates

The map is updated every time our DSO wrapper publishes new pointcloud message. It uses an original Moveit updater without a robot model consideration, because it is not needed for the drone - none of the drone parts could possibly be in front of the camera at any time - and because the drone model was disrupting the measurements. It is furthermore modified in a way that allows the editation of the underlying probabilistic octomap sensor model with a configuration file.

The original sensor model values were used with expectation of a dense pointcloud input from the RGB-D camera. That doesn't work well enough with our sparse pointcloud input. The values used are determined by experimenting with the settings and the output of a map on some test runs. The ability of our map to reliably detect an obstacle is experimentally verified in the Evaluation chapter.

The map update itself utilizes a ray-casting routine, which casts rays from every point in the pointcloud to the position of the camera. It increases the probability of the octomap nodes in the ray path to be free space and it increases the probability of the target point in the cloud to be occupied by the detection probability specified at the sensor model.

The modified configuration file is the one described in Moveit in 3D perception update tutorials as a sensor yaml configuration file. The new possible fields are:

- *occupancy_tresh* sets the octomap occupancy treshold for a map node to the specified value
- *prob_hit* and *prob_miss* which set the value of the new observation probability chance in a node
- *clamping_thres_min* and *clamping_thres_max* which set the value of the clamping values for free and occupied space

- *disable_model_filtering* which skips the filtering of the robot model

With an exception in the model disabling parameter, all the other values are doubles and if not set, the default values from the octomap [22] are used in the sensor model. The limitations and usage of these values are closely discussed in the octomap paper. Values used in our project are well tested and if you have a need to change them, read the octomap documentation first.

The parameter which disables model is an integer and accepts 0 as false and 1 as true. The model filtering is very handy when working with robotic hands, as parts of the hand might sometimes be in front of the 3D sensors. It disturbs the updates of our system though and the model filtering is not needed when working with a small quadcopter with a camera at the front. We therefore disable the model filtering for our purposes.

6.3 Controller

The new action defined when we bended Moveit system for the drone usage requires a controller which can handle that action. We will be using a modified controller from the Autonomous-flight-ROS project [21] which uses Moveit to fly with a simulated drone. Because the original controller was used only in simulations with the same drone model all the time, we need to modify it to work well with the Bebop in real time. The result is still a simple controller without a control loop, but it is a more precise one. Our controller is tested in chapter 8.

The modifications to the controller include a bug fix, which caused a drone to have different real behavior when going left and forward than right and backward. Furthermore, we modify the blocking time of the controller after each command was issued with a new value, which is loaded from the ROS parameter server. The value from the original project is the default value if the parameter is not set. The parameter names are "simple_controller_rotation" and "simple_controller_movement" of a type double and specify how many seconds the controller thread waits after issuing a command in a movement case and a sleep modifier for the rotation case as the drone rotation has different dynamics than the drone movement has.

The output of the original controller is good enough to run a simulation. With the real drone however, the effect is not sufficient. Especially if the drone was moving before, it takes too long for the new command to have a desired effect. We therefore include a drone flight model and furthermore modify the output of a controller with a boost technique. The flight model describes the behavior of the specific drone and is set up with parameters from the ROS parameter server. The boost technique increases the intensity at the start and at the end of the signal to move in a certain direction.

The values used in the model are based on the specific drone flight dynamics and are obtained experimentally with individual drone. An ideal solution to this problem would be a control loop which would monitor how the drone behavior reflects the commands it receives and the controller would then be able to adapt to the situation which would not only reflect the drone specific flight dynamics, but also would be able to correct for detected drafts and air currents. We had plans to implement a control loop at the beginnings of the project, however numerous more serious technical issues prevented the control loop to be implemented.

The parameters loaded from the ROS parameter server are of a type double and compensate the output of the cmd_vel topic for the third of every second. Their names are "compensation_x", "compensation_y", "compensation_z" and "compensation_rot". Their default values are 0.0 and they don't modify the output with default values.

The boost technique provides significant boost to the intensity of the drone control message whenever the drone starts a movement in a new direction. There is also a short boost in the opposite direction of the movement when the movement in a certain direction is finished. The boost in the opposite direction serves as a break, to stop the movement of the drone. The boost mechanism takes into account the flight dynamics parameters so it is larger on the weaker directions of the drone movement and is smaller on the more dominant directions of the drone movement. The boost makes big difference for the good control of a real drone as documented in the Evaluations chapter.

6.4 Planning

Moveit uses Open motion planning library (OMPL) [23] by default. It has more planning libraries integrated and it is possible to write custom planner if the need for it arises. It is also very straightforward to create custom scenarios, such as flying in a square or flying a few meters back and forth which are used in the evaluation section.

The OMPL library planners are totally abstract and operate without any knowledge of a robot. That also means the library itself doesn't implement any viewers and collision checking. The concrete planning algorithms are described and documented on the project pages. They can be split to geometric planners which take into account kinematics and geometry of the scene and control-based planners which also use differential constraints to limit speed at certain areas.

Moveit prompts OMPL to generate a few plans at the same time, checks for collisions and then picks one of the valid plans. OMPL uses RRT and KPIECE algorithms by default, because they provide good real time real world results [23]. It is very simple to pick a different planning algorithm if there is a need for that. We will use the RRT planner for purposes of this implementation as we have good experimental results with it.

Moveit includes an infrastructure for the control loop of the planning execution called an Execution monitor. It monitors the plan execution and in case something goes wrong or something unexpected happens, the monitor can respond to the new situation. This feature is supported in all elements of Moveit which means, that even though the monitor is not implemented in this thesis, it could be added naturally to the system. That will allow us to react to unexpected obstacles in the planned trajectory during the execution of the plan and re-plan when that happens. The execution monitor differs from the controller control loop, as the control loop in controller only assures precise execution of the plan. The execution monitor assured that the system can react to new situations accordingly.

6.4.1 RRT

Rapidly-exploring random trees [33] is a tree based motion planning single query algorithm which samples random state rs in a state space. Then it finds already seen state ss which is the closest tree vertex to rs with respect to a given distance metric. It expands towards rs until it finds a middle state ms which satisfies given geometric and kinematic constraints. It then adds ms to the tree, if the edge between ss and ms satisfies the constraints as well. The edge sampling in OMPL implementation is performed randomly for a given number of states on the edge. When the new state is added to the tree, additional constraints are added for the states reachable from the new tree vertex, respecting the kinodynamics of the whole system.

The algorithm ends if it finds a valid trajectory which satisfies kinematic and geometric constraints. It is terminated if it did not found a solution after a specified time or when the tree reaches a specific size. The default time constraint in our implementation is $0.08s$ as described in the Moveit documentation and we have not found any vertex limit.

The state space is considered to be bounded and if not said otherwise, the default value of the boundary is a cube 10x10x10 around the starting point. The state space itself is not limited to poses in 3D space and obstacles (geometric constraints), but also includes speed and acceleration limits (kinematic constraints). The limits are defined in a Moveit configuration file. The RRT random state generator has a massive bias towards the unexplored areas of the state space which leads to a rapid exploration of the whole state space area.

6.5 Custom plans

It is inteded to use the system this thesis proposes for goals other then to solve the "go-home" problem. To do that, we need to provide a simple way to specify navigation and planning goals from the code. We call that simple way a custom plan. A more complex behavior of the drone would be created for example by combining multiple custom plan nodes into a decision tree.

Moveit provides a way of creating a custom plan right of the box. How to do that is well documented in Moveit tutorials. However with the modifications for the use of drones we made, ways described in the Moveit tutorials do not work. This section describes two types of plans we can create with our system and provide examples of how to do that.

We call the two types "plans with specific planning" and "plans with automatic planning". We will now describe the general properties those two plan categories share. The description of individual categories follows in next two subsections. This thesis provides an implementation of both of these categories, which might serve as a template to create more complicated plans for specific tasks.

A custom plan is instantiated as a ROS node and it publishes the plan it generates to a topic. The plan is then displayed in Rviz and can be executed. The custom plan node can subscribe and publish to topics, it can call ROS services etc.

A basic way to interact with the planning node is with Rviz visual tools plugin which is a part of Moveit. We run the custom plan node and when we want it to do an action (plan, display trajectory, execute a plan etc), we press the button in Rviz and the node does its job. If we wanted to prompt the node to perform an action by other means, we could also do that as a service call or by notifying the node on a topic it listens on.

All our implemented planning nodes wait on their launch to be prompted. First prompt makes them to generate a plan and second prompt makes them to execute the generated plan. Third prompt shuts the node down.

6.5.1 Specific planning

A plan with a specific planning is a ROS node, which allows us to create a motion plan based on the list of poses. For example, we want the drone to move in a square trajectory no matter what the environment around it looks like. We have to specify all the poses we specifically want the drone to reach and their correct order in the list. The specific planner converts that list of poses straight into a valid trajectory plan. That plan can be viewed in Rviz as a trajectory to execute or it can be directly executed.

All the custom scenarios in the Evaluation chapter and the backtracker node are specific planning nodes.

We have created a ROS service called get_robot_trajectory_from_path, which takes a message of a type nav_msgs/Path and creates a moveit_msgs/RobotTrajectory type message, which is directly accepted in Rviz as a planned trajectory and can be directly executed by the move_group. The nav_msgs/Path message is basically a list of poses in space. The specific planner nodes all use this service. They create a path message and convert it into the plan with this node.

6.5.2 Automatic planning

A plan with an automatic planning is a ROS node, which allows us to create a motion plan based on a starting position and a list of targets. The planning node then uses the OMPL planning library to automatically generate a valid motion plan. The plan can be then viewed in Rviz or directly executed.

Our "boomerang" automatic planning node implementation is an example of how to utilize Moveit planners from the code. Moveit tutorials don't cover this part, as we use modified Moveit actions and a floating virtual joint to control the drone - not at all the same as with the robotic hands and ground mobile robots, which are covered in Moveit tutorials well.

The move_group instance has a wide range of different settings which can be specified. They include custom distance metrics to optimize, workspace, type of a planner to use, constraints on poses, speed etc. All the setting is available in Moveit documentation.

Our "boomerang" implementation uses the same prompting rules as are the rules in the specific planning nodes and could be easily changed as well.

7. Implementation

We started the work on this system at the start of year 2014, originally with the Parrot AR Drone 2.0 and with a different system than ROS. After creating a solution that somewhat worked, the struggle with the system and the clunky AR 2.0 was too much to handle. The key decisions were to use ROS for the project and to change from AR 2.0 to Bebop platform. Bebop is much more stable and agile and its fish-eye lens allows the visual odometry to actually detect obstacle during a flight. That was difficult with the old AR 2.0, because the camera didn't maintain the line of sight. It produced a lot of floor pictures when it was flying forward which didn't allow for stable obstacle detection.

We started the work with ROS and Bebop in september of 2016 and compared to what we were able to achieve with the old system, this went much faster. ROS is a wide spread system with big community and has a lot of already completed packages. That simplifies the development significantly. We are using bebop_autonomy [4] as a Bebop driver, robot_localization [28] provide both Extended and Unscented Kalman filters configurable implementations and Moveit! [16] as a planning framework which uses Rviz [29] as visualization tool, octomap [22] to represent a scene and OMPL [23] to generate plans. We use Gazebo [24] to simulate scenarios and try things in the simulator before attempting them in reality.

Along with that, we employ Eigen [14] for vector calculations, pcl [26] to operate easily with the pointcloud, part of the library [25] for mutex to stop XBox controller overriding the commands the automated controller sends to it and tf [27] for easy frame switching. We use a few more simple helper nodes which are considered part of the ROS system and we do not list them specifically.

Our installation of ROS runs on the VirtualBox Ubuntu 16.04 on the machine with Windows 10. The possibility to take snapshots of the system and return to them according to the need proved invaluable, however I would strongly advise against this approach for future work. The reason for that is that both Rviz visualization and Gazebo simulator use 3D rendering and 3D computations. In VirtualBox, all of those calculations are performed on the CPU and that is much slower than with the GPU. It is possible to install "Guest additions for VirtualBox" to mediate usage of the real GPU on the simulated machine, but that makes both Gazebo and Rviz crash on their startup. According to their documentation, neither of them is indeed compatible with this approach.

7.1 Proposed system implementation

This section will cover our actual realization of the proposed system with an overall review of its capabilities and its shortcomings.

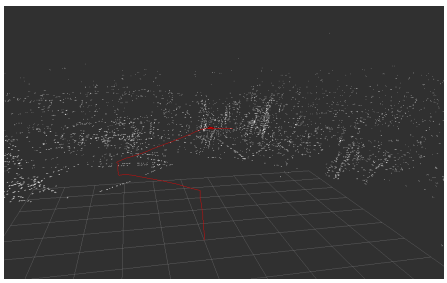
As a driver, we are using a bebop_autonomy package. It provides us with the odometry and visual stream data and mediates a steering signal, landing, takeoff and emergency landing commands for the drone.

We employ DSO to process the visual data and create a sparse point-cloud and odometry messages independent of the drone odometry.

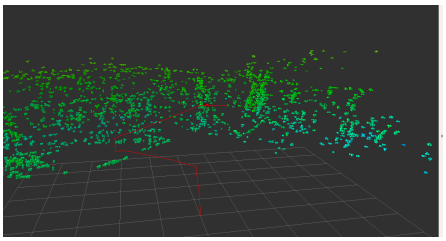
We use configured values acquired experimentally and UKF to fuse the data from the DSO and drone IMU. Specifically, we successfully transform the point-cloud data generated by the DSO to the drone base frame.

The Moveit! planning framework is used for following tasks:

- We use octomap with modified underlying sensor model to update the map from our sparse point-cloud. Figure 7.1.1 shows a point-cloud and the octomap created from that point-cloud
- The OMPL is used to automatically generate plans.
- We create custom scenarios and custom plans with the standard Moveit workflow.
- We use the Moveit Rviz plugin to monitor and view actual planning scene.



(a) Pointcloud received by the octomap



(b) Octomap representation of the scene

Figure 7.1.1: A comparison of a point-cloud received from DSO and the octomap representation of the scene, based on the received point-cloud.

We use a simple controller with limited and boosted output adjusted by an experimentally acquired flight model to control the drone automatically.

We are using an X-Box One controller to fly with the drone manually.

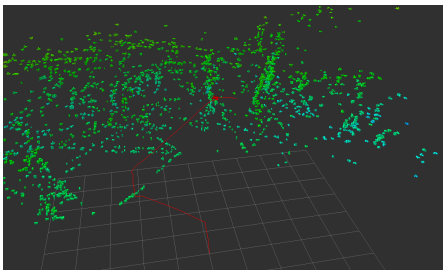
A mutual exclusivity between the X-Box controller and the automatic controller is ensured by a third party mutex node.

7.1.1 Summary

The following list summarizes the capabilities of our current realization of the proposed system:

- The system actual implementation is capable of manually flying with Bebop using an X-Box controller and not interfering with the automatic controller output.
- The DSO consumes the drone visual stream and extracts the drone's surroundings into a point-cloud.
- The point cloud is fused with the drone odometry to create a map of the environment the drone is in. The map has an edge $7.5m$ distant from the actual drone position. Figure 7.1.2 shows a real scene and its octomap representation.
- On demand, we can automatically generate motion plans respecting the obstacles in our map to reach the desired position.
- The simple controller we created can execute the motion plans on demand.
- We are able to easily generate custom scenarios.
- We can view the whole scene and send commands to the drone with only one interface.
- We can run the whole system on a simulator to test the scenario before we try it in reality.

(a) A photograph of the scene



(b) The system's perception of the scene

Figure 7.1.2: A figure comparing the real scene and the scene perceived by the system. The structure of the wall, the flower, the first table and the green barrier are identifiable in the sparse map our system created. The floor has some perceived triangles on it as well.

This list summarizes the shortcomings of the actual realization of the system:

- The odometry data fusion is heavily weighted towards the bebop data. It causes that the drift is not accounted for and therefore cannot be regulated.
- The use of a simple controller with no control loop limits the execution precision of the planned trajectory and does not allow the controller to modify its output.
- The lack of an execution monitor does not allow the planner to re-plan after encountering an unexpected obstacle in the planned path during the plan execution.

7.2 The go-home problem implementation

In this section, we will talk about the specific implementation of two solutions to the go-home problem proposed in chapter 4. Both of those solutions are specific plans in the broader proposed system. That means that they share the actual realization of specific sub-systems along with their current limitations described in the previous section.

For the general idea behind the implementation of both nodes, refer to the chapter 4, section 2.

7.2.1 Backtracker

The mechanism behind specific plan implementation was described at the end of the previous chapter. The backtracker node utilizes that mechanism.

We realize the remembering of the odometry messages with a ROS node "Frodopathy". The node expects two parameters, one specifying the odometry topic and the other specifying the output topic to which the result should be published. The node creates a "nav_msgs/Path" structure and adds the most recent odometry message at the end of the path if its position differs from the last message at least by certain value. This value is specified on the ROS parameter in parameter of type double called "frodo_limit". The default value of this parameter is 0.05 which corresponds to $5cm$ in the drone coordinate system. Our implementation uses the default value.

The backtracker node subscribes to the path topic specified in the launch file and remembers the latest path published. When prompted for the plan, it unsubscribes from the topic and sends a request to the node which offers the service "get_robot_trajectory_from_path" described in the previous chapter. The plan created from the path by the service is then published to Rviz for visualization and to the move_group object to be executed on demand.

7.2.2 Boomerang

The mechanism behind automatic plan implementation, which this node utilizes, was described at the end of the previous chapter. It specifies the maximum planning time to be $2s$ and limits the planner workspace to an area $10m$ in x and y directions for the drone and the flight altitude to be between $0.7m$ and $1.7m$. Then we specify our target position to be 0 in x and y coordinates and 1 in z coordinate and let the planner to generate a plan automatically with respect to the obstacles recorded in our octomap.

The result is published to Rviz and is viewed as a trajectory. It can be then executed on demand.

7.3 System usage

This section presents how to use the system to navigate and create plans for a quadcopter. It is assumed that a user uses a quadcopter operated from ROS. A ROS description with more details is in the attachments chapter of this thesis. Note that as a broader robotic community solution, ROS is well maintained and documented. Almost every package or maintainer provides easy to use interface and/or is well documented and has tutorials.

We recommend the "Parrot Bebop" described in chapter 3 for its affordability and flight stability, however a comparable drone is certainly usable. Make sure that different drone publishes its IMU data on the ROS topic. The system should be also usable with different main sensors, such as with stereo cameras or with the RGB-D camera with small modifications.

The whole system is started with three ROS launch files. They could be started all at once, but because a few systems write their runtime output to the console, we prefer that they are started in different terminals for readability and flight monitoring. The first launch file starts a world representation which is either a real drone, a bag file or the Gazebo simulator. The second file starts a controller. The third file starts the Moveit planner and Rviz visualization.

There are many parameters to set before the full system works well. It is necessary to provide DSO with a camera calibration data as described in the documentation [1]. The Moveit planner needs to be setup by following the Setup Assistant tutorial on the Moveit site [16] with the modifications described in this project [19] to allow for the quadcopter control. The octomap probability tresholds should be edited according to your sensor model in a Moveit sensor file. Your drone flight model should be updated as a series of ROS parameters. Which parameters and configuration files to use for those purposes was described in the previous chapters.

When all is correctly set up, a user is able to query Moveit from Rviz for plans with a starting and target positions. Rviz then visualizes the planned path with respect to the obstacles the octomap represents and executes it either on demand or straight away. If the user wants to execute more sophisticated plans, they can use some of our evaluation node plans or the backtracking node as a starting point and modify them according to their needs.

7.4 Development

This implementation is developed on Ubuntu 16.04 with *ROSKinetic* and *C++11*. The developed code and the modified projects (DSO, Moveit, Controller) are attached on the flash drive and everything is also shared on github. The code is compiled with CMake in case of DSO and with catkin in case of all the ROS packages. The catkin is a well documented ROS tool, which is able to locate and build all the necessary packages based on each package dependency specifications.

The code is mostly object oriented with a few simple node exceptions. It is written with the good practices in mind and it aims to be easily readable and self-explanatory. The configuration files are documented either in the Moveit documentation, in previous chapters or in the configuration files with commentaries. The class description of our nodes follows:

- *dso_node* encapsulates the ros_output_wrapper node. It takes care of correct initializations of the DSO system and accepts the calibration files as parameters. It resets the system in case of a failure. It is possible to activate the original dso viewer "Pangolin" in the launch file. It consumes the drone *image_raw* topic for DSO.
- *ros_output_wrapper* produces two output topics from the DSO into ROS. The topic *dso_odom* publishes the odometry estimated by the DSO system. The topic *pcl* contains a pointcloud message based on the the DSO odometry transformed into the base_link frame. The transformation values are loaded from the ROS parameter server.
- Modified DSO *Output3DWrapper* interface and its implementations to workaround a bug of the wrong memory address for the camera pose output.
- Modified Moveit! *pointcloud_octomap_updater* to allow probabilistic sensor model modifications and to allow disabling robot model filtering. If the configuration file doesn't specify any model numbers, it still uses the original octomap values.
- Modified controller *action_controller* from the action_controller package from the Autonomous-flight-ROS project [21]. Modifications allow to specify the controller blocking times after the command is sent. The new values are loaded from the ROS parameter server and have their original values as

default. The original implementation was sufficient for the Gazebo simulator control but the real world scenarios require finer control over the drone. Also fixed a serious bug when the drone behavior was different when flying forward or left than when it was flying backward or right.

- *cmdvelrepeater* is a node which takes the command from the controller and republishes it with desired frequency. The original node implementation only sent a command once which was good enough for the simulation but the bebop driver expects different input [4].

- *cmdveladjustor* is a manager node that covers the limiter, compensator and booster in itself. It is technically a part of a controller and modifies the output of the action_controller node which is based on the controller for the simulator.

- *cmdvellimiter* is a manager node that covers the limiter. It is applied in case when we are controlling the drone directly to ensure that the flight is smooth and slow enough for the DSO to be stable.

- *cmdlimiter* modifies the absolute maximum value of the command sent to the drone. The limit is loaded from the ROS parameter server. The official absolute maximum according to bebop driver [4] had a value of 1. This number is lowered with the limiter to improve DSO behavior and to allow the boost mechanism to take place. It limits both the translation and the rotation commands and can accept different values for the two.

- *cmdcompensator* compensates the output of the drone according to specific flight constants if they were provided. It applies the compensation for a third of every second.

- *cmdbooster* adds a boost to the command signal when it receives a new direction or a counter-boost when the movement in the direction is finished. In a case when the new direction is opposite to the previous one, the boost is significantly longer. When provided, the booster takes into account unique flight constants of a drone and applies corrections according to the constants. These constants also modify the boost behavior so that it is shorter in dominant directions and longer in weak directions of a specific drone.

- *transodom* transforms the odometry of the DSO into the drone frame. It accepts the same parameters from the ROS parameter server as the transformation of the pointcloud.

- *frodopathy* node transforms the received odometry to the path message. It is used to display the path the drone took and serves as an input to the backtracking node.

- *robottrajectoryfrompahtservice* transforms the path message to the trajectory message used by Moveit to display and execute the trajectory. It is a service which is called on demand from planning nodes we use as our custom plans and scenarios.

- *backtracker* is a custom planner node which plans the return of the drone to the starting position by backtracking on demand. It accepts the path message from the frodopathy node, turns the path around and creates a trajectory to execute from the path.
- *boomerang* is an automatic planner node which plans the trajectory for the drone to return to the starting position by shortest route possible on demand. It accepts the path message from the frodopathy node, turns the path around and creates a trajectory to execute from the path.
- *square* and *backandforth* are our custom scenarios for the performance evaluation. They script the desired path, transform it into trajectory and send it to the action server for the controller to execute the trajectory.

8. Evaluation

To evaluate the results of our implementation, several experiments were designed to observe some of the sub-systems performances. It is very straightforward to create a planned trajectory by hand, as some of these experiments demonstrate to test the controller reliability. We provide some photographic documentation as well as trajectory perceived by the system. Most of the experiments are recorded into ROS-bags and can be downloaded at the project site [37]. Their are too big to include them as attachment of this thesis.

The bags allow the experiment to be replayed using a script which is a part of the thesis attachments. The bags contain all the odometry data, visual data, perceived point-clouds and maps generated from them, the paths generated by either automatic planner or designed as a part of the scenario and the commands which the drone receives either from the automatic controller or our X-Box controller.

The experiments are presented in following sections and subsections. Each section describes the scenario, expected behavior of the system or a subsystem, actual behavior observed, photographic documentation of some of the experiment instances and statistics based on a number of individual experiments executed. If an experiment doesn't go as planned, we examine the results for that and propose a possible solution.

In general we expected that because of the simple controller the flight will not be too precise, but should be reasonable. Because of the drift, caused by the air currents from drone propellers near obstacles, and the inability of the current implementation of the designed system to detect drift, there will be inaccuracies when the drone is flying too close to the objects. Also some natural drone drift happens when the drone stays still. It has been shown [3] that with a more advanced controller, the drifts are detectable and the drone can counter them properly.

The experiments were executed in the large hall which floor is structured into colored right triangles. For reference, their legs are $200cm$ and $75cm$. The photographs of the hall are presented in the figure 8.0.1 and more of them are attached in the "Attachment 1". The bag files recorded during these experiments are available at the project site [37] for examination, as stated in the first paragraph of this chapter.

(a) A view into the hall

(b) Different view into the hall

Figure 8.0.1: Some of the photographs of the hall in which the experiments took place. Other photographs are included as an attachment.

8.1 Obstacle detections

To test the ability of the octomap updates to create a map which captures obstacles in front of the drone, we designed a simple experiment. We flew with a drone left and right approximately one meter a few times with the camera aiming into the room. The drone natural flight altitude is $1m$. There were three situations:

- an empty room
- a board $2.2m$ high and $3m$ wide in front of the drone in $3m$ distance
- a chair $1m$ high in front of the drone in $3m$ distance

We offer photos of the scene and screen-shots of the octomap created by the system for evaluation in the figure 8.1.3. The detection of both, the board and the chair, is visible in the images.

Figure 8.1.1: A photograph of a room used to conclude obstacle perception experiments and the octomap representation of that room, created by the system's current implementation.

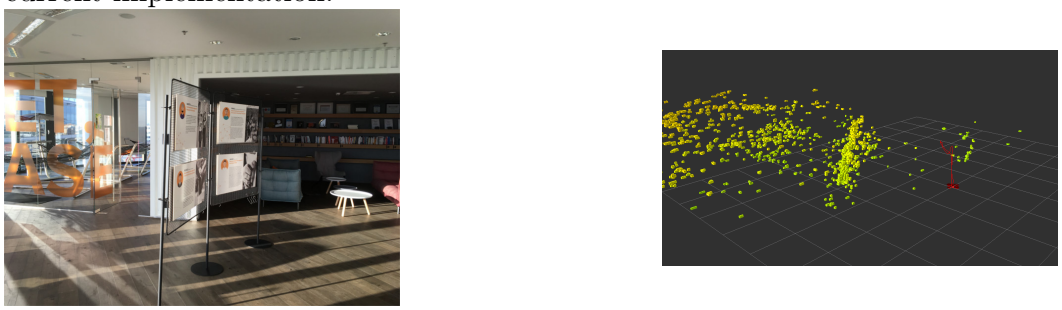


Figure 8.1.2: A comparison between the photograph of a scene with a board and the octomap representation of the scene with the board. The board is clearly visible to the drone when compared to the octomap created with an empty room.

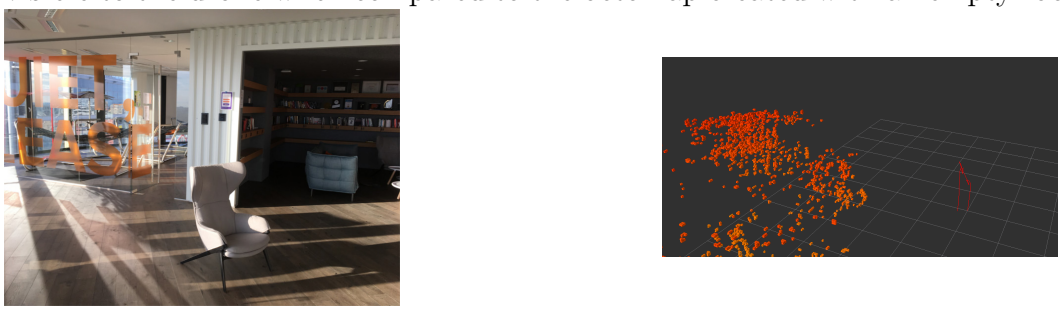


Figure 8.1.3: A comparison between the photograph of a scene with a chair and the octomap representation of the scene with the chair. The chair is also clearly detectable by our perception system.

8.2 Controller performance

To test the controller ability to follow the trajectory generated by the planner or by hand, we designed two experiments.

First we created a scenario where the drone objective was to fly back and forth $3m$ in a line several times and measured the distance between the starting and finishing positions.

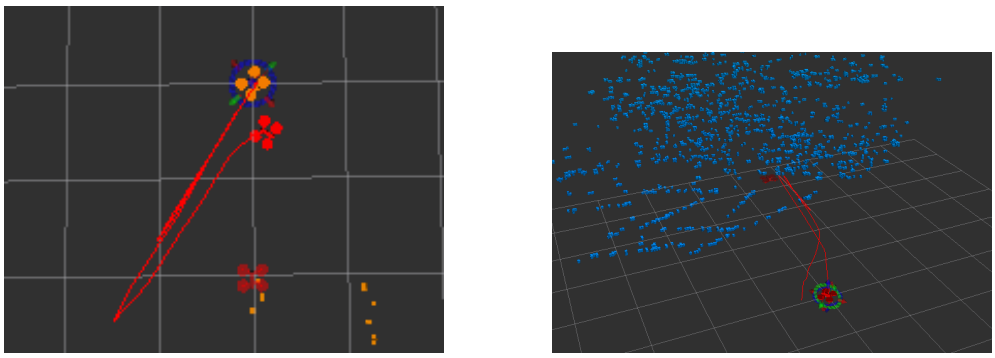
Second scenario was a square with an edge of $3m$ and the drone had to fly in the square. Again, we measured the distance between the starting and finishing positions. This scenario should evaluate the drone behavior in execution of more complex plan.

The original simple controller we took from the simulator was failing at these scenarios miserably. The back and forth flight would usually result in a $6m$ distance. The time the drone was supposed to fly in the opposite direction than the one it started with was used only to stop the previous movement. The square flight would be even worse. The drone would fly in the direction of the first two commands and the second two commands would only stop the previous movement. The drone would usually end up being around $5m$ away in both of the first two movement directions. Unfortunately, we do not have records of the flights we concluded with the unmodified controller. However they are replicable by disabling the cmdveladjustor node in the launch file.

8.2.1 Back and forth flight

The planning node executing this scenario accepts a parameter "plan_variant" with integer value between 1 and 4 which specifies which direction the drone should move first. It transforms one to forward, two to back, three to left and four to right. We realized in our previous experiments, that the drone construction makes sure the drone stays stable when flying back and forth and it is unstable when flying left and right meaning, that the orientation of the drone slightly changes at the second case. It is caused by the fact, that both propellers on the left rotate clockwise and both propellers on the right rotate counter-clockwise. Therefore we only include the back and forth variants of this experiment. The screenshots and photographs of the left and right variants are documented at "Attachment 1".

The drone flew the line four times in each experiment, two times in one direction and two times in the other. We expected the distance to be close to the $3m$ every time and the resulting position being reasonably close to the starting position. Figure 8.2.1 shows the trajectories the drone took when executing two of our experiments.



(a) First experiment (b) Second experiment

Figure 8.2.1: The red line represents a trajectory the drone took when executing the experiments based on the fused odometry. The drone with the arrows around it represents the starting location. The ghost drone is just a drone representation from Rviz, ignore it.

We executed a set of twelve flights. Six of them were forward first and six of them were backward first. The deviations from the starting position when flying forward first shows the table 8.1 and deviations when flying back first shows the table 8.2. An average deviation from all twelve measurements is rounded to $66cm$.

Table 8.1: The table presents deviation from the starting position when flying forward first.

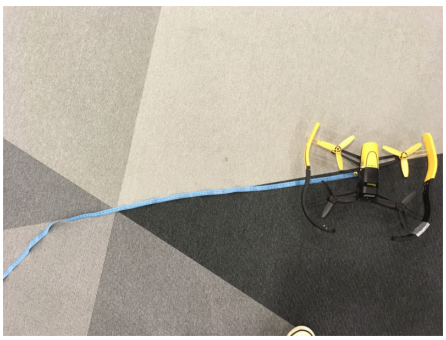
measurement no.	1	2	3	4	5	6
deviation from the start (cm)	30	25	90	80	45	50

Table 8.2: The table presents deviation from the starting position when flying backward first.

measurement no.	1	2	3	4	5	6
deviation from the start (cm)	20	15	145	80	130	75

Photographs presented at figure 8.2.2 capture one of the experiment's starting and finishing positions. There are more photographs documenting the experiments attached to the thesis in the Attachment 1. It is clear from the photographs that the deviation happened mostly in the left to right axis. That indicates that the deviation is probably caused by the drift.

(a) Starting position



(b) Finishing position

Figure 8.2.2: A deviation from the starting position after executing one of the experiments.

8.2.2 Square flight

The planning node again accepts a parameter "plan_variant". This time with integer value 1 or 2, where 1 specifies a counter-clockwise square with a start forward and 2 specifies a clockwise square with a start backward. The drone does not change the orientation in the corners, it only changes direction of the flight.

We expected the drone to be relatively more deviated when compared to the first experimental scenario, because the fact that we only have a simple controller without a control loop will have higher impact on the execution of a more complex plan. However our expectation was that the drone will execute a reasonably rectangular-like trajectory, thanks to our booster technique which should handle the changes of the directions reasonably well. We didn't expect any real differences between the two variants of this experiment. Two of the trajectories are displayed in the figure 8.2.3

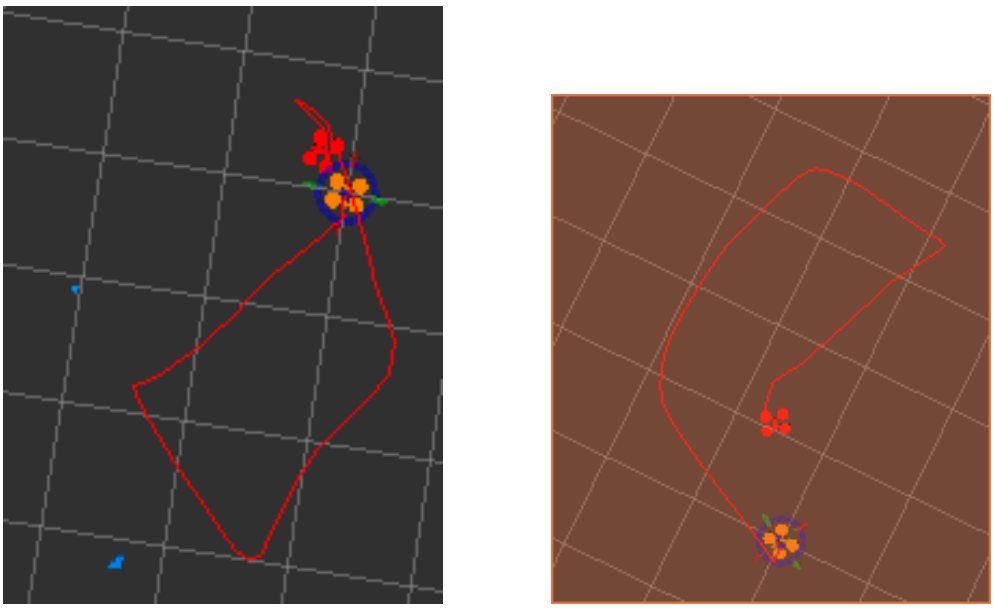
We executed a set of eleven flights. Six of them were counter-clockwise and six of them were clockwise. The deviations from the starting position when flying counter-clockwise shows the table 8.3 and deviations when flying clockwise shows the table 8.4. An average deviation from ten measurements is rounded to $146cm$. The eleventh measurement did not finish due to the drift and is not accounted for in the deviation average.

Table 8.3: The table presents deviation from the starting position when flying counter-clockwise.

measurement no.	1	2	3	4	5	6
deviation from the start (cm)	130	50	140	75	DNF	∼200

Table 8.4: The table presents deviation from the starting position when flying clockwise.

measurement no.	1	2	3	4	5
deviation from the start (cm)	∼200	150	160	150	∼200



(a) First experiment (b) Second experiment

Figure 8.2.3: As with the back and forth experiments, the red line represents a trajectory the drone took when executing the experiments based on the fused odometry. The orange drone represents the starting position and the red drone the finishing position.

The figure 8.2.4 shows the difference between the starting and finishing positions in one of the experiments. We expected the results to be worse than with the back and forth flight which they are. The trajectory seems to be mostly rectangular which indicates that the booster works as intended and our only problem seems to be the controller precision.

(a) Starting position (b) Finishing position

Figure 8.2.4: A deviation from the starting position after executing one of the square experiments.

8.3 The go-home problem

It was the thesis objective to solve this problem. We have two ways of returning the drone to its starting position.

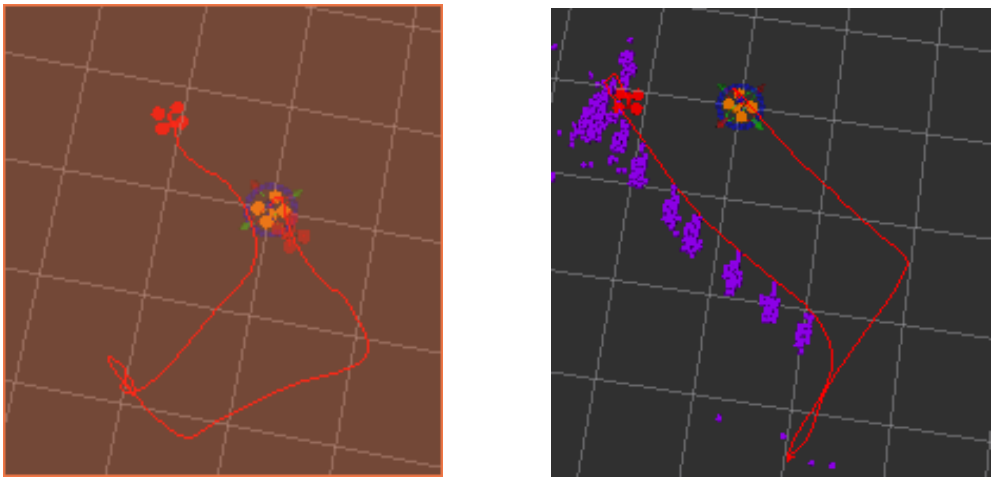
The first is a simple backtracker. It remembers the flight path detected by the drone and plans the trajectory following the exact same trajectory back. In fact, it does not use obstacle detection at all.

The second way we implemented is a boomerang. It automatically plans a trajectory respecting the detected obstacles using the Moveit planners.

8.3.1 Backtracker

The flights in this experiment consisted of two or three significant changes of direction and a rotation. We didn't fly more than eight meters away in one direction and we also tried to fly seven to ten meters in total.

We expected that this experiment will have comparable results to the square experiment, as it is executing relatively comparable plan with two or three changes of directions and in addition to that there are some rotations. We expected that the planned path will be the same as the path of the original drone path, only backwards. We expected that the path executed will have a shape similar to the path the drone took on the way there. Figure 8.3.1 captures two of the experimental trajectories.



(a) First experiment (b) Second experiment

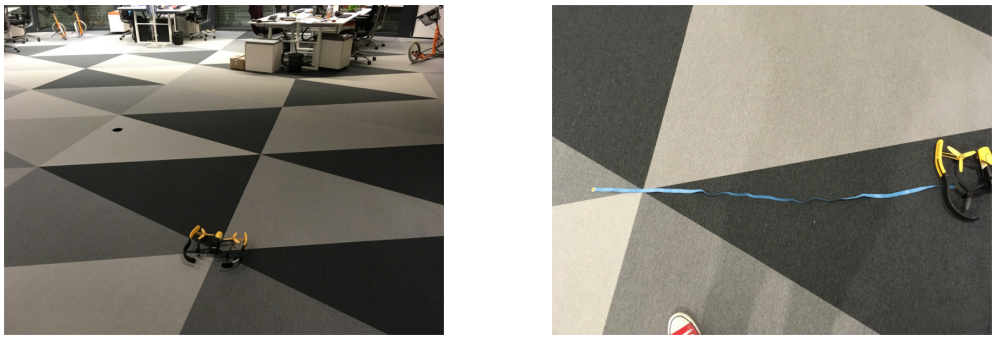
Figure 8.3.1: The executed backtracker trajectories have similar shape to the original trajectories.

We executed a set of six flights. The deviations from the starting position shows the table 8.5. An average deviation from five of the measurements is rounded to $166cm$. The sixth measurement did not finish due to the drift.

Table 8.5: The table presents deviation from the starting position when returning to the starting position by backtracking.

measurement no.	1	2	3	4	5	6
deviation from the start (cm)	180	120	DNF	∼200	∼200	130

The figure 8.3.2 presents photographs of starting and finishing positions of one experiment.



(a) Starting position (b) Finishing position

Figure 8.3.2: The deviation from the starting position is higher than the square experiment.

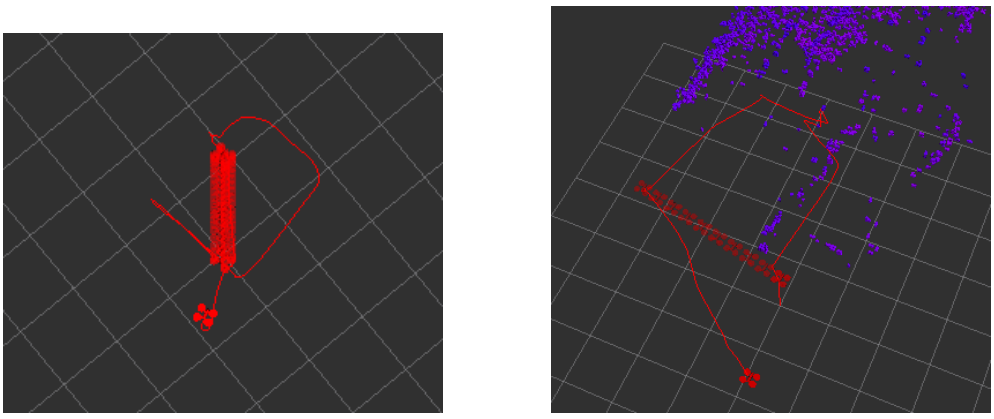
We see that the overall results are slightly worse than the results with the square experiment. The path taken by the drone seems to have a similar shape to the original path. That indicates that a better controller would increase the measured precision of the experiment.

8.3.2 Boomerang

The expectations from this experiment were that they will be similar to those of the backtracking experiment. We expected that the system would detect some noise obstacles in the general area of the experiment and that it would plan to avoid them, therefore resulting in the plan in the same level of difficulty as the backtracker.

The flights we took with the boomerang also did not fly more than seven to eight meters away in one direction, but we didn't have any restriction regarding the flight trajectory. We basically flew for a while, changing directions very often, sometimes rotating the drone. The trajectory of the flight was usually around twenty meters long.

We executed a set of ten flights. Four of them were recorded to a bagfile and six of them were not, as we ran out of disk space. The deviations from the starting position of recorded flights shows the table 8.6 and deviations from not recorded flights shows the table 8.7. An average deviation from ten measurements is rounded to $88cm$.



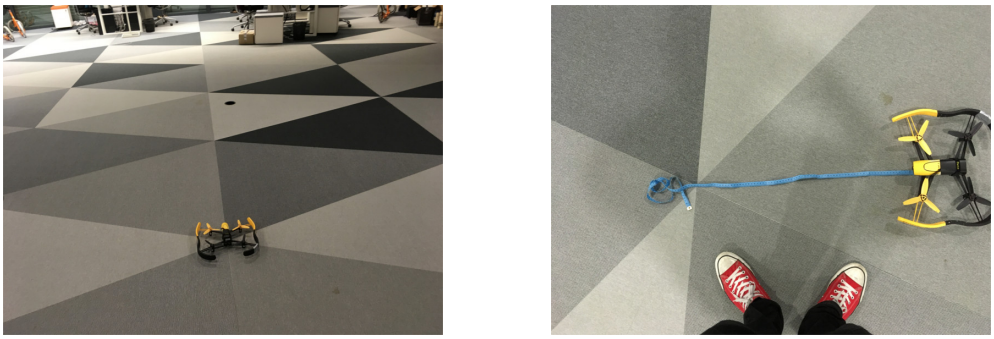
(a) First experiment (b) Second experiment

Figure 8.3.3: The perception system did not register many noise obstacles. The planned trajectories were therefore simple and could be executed relatively precisely even with our simple controller.

Table 8.6: The table presents deviation from the starting position when returning to the starting position with recorded boomerang.

measurement no.	1	2	3	4
deviation from the start (cm)	75	∼200	20	120

The figure 8.3.4 presents photographs of starting and finishing positions of one experiment.



(a) Starting position (b) Finishing position

Figure 8.3.4: The results were quite close with the boomerang automatic planner.

As we can see, the results are a lot better then with the backtracking node. The reason for it is probably that the noise obstacle detection rate was very low and the planner planned simple trajectories to return to the original location. We took more flights with the boomerang planner off the record as our hard drive space was running low and each record of the longer flight took well over 3GB. Those are the deviations in the brackets. We could see that the results were on average below one meter of deviation from the original position for flights that took longer time.

Table 8.7: The table presents deviation from the starting position when returning to the starting position with not recorded boomerang.

measurement no.	1	2	3	4	5	6
deviation from the start (cm)	60	120	35	15	80	150

8.3.3 Boomerang with obstacles

Next experiment with the boomerang planner was to fly around a board and position the drone in a way that the board is in between the starting and actual location when starting the automatic planner. We expected drift problems, but we were hoping that some experiments would be successful.

- Plan not found due to the noise obstacle in the exact place of the starting location,
- DNF - the plan created a valid trajectory to the starting point, but crashed due to the drift
- DNF - the plan created a valid trajectory to the starting point, but crashed due to the drift

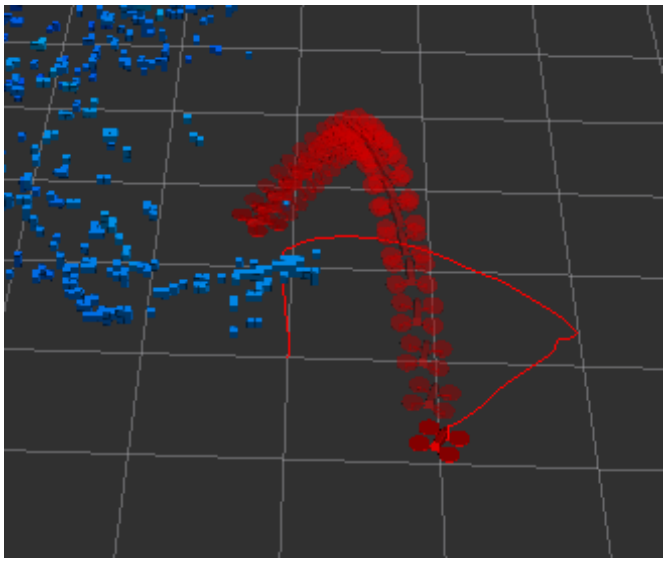


Figure 8.3.5: The obstacle detected between the actual position and the desired drone destination. The trajectory is planned around the obstacle.

As we see in the figure 8.3.5, the planner was able to generate a valid trajectory but limitations of our system's current implementation prevented us from reaching the desired position. Supposedly, both lack of the drift detection and a simplicity of our controller contributed to the failure of the generated plan execution.

We wanted to test the hypothesis that the planner generates correct plans and only our current controller implementation fails at executing them at the

simulator. We created the exact same situation ten times and the whole scenario was successfully completed every time. So the planner is able to generate correct plan for specific situation and further improvement of our current implementation is required in order to successfully complete this experiment.

8.4 Discussion

The experiments with the boomerang planner show, that the overall plan generation is good and we are able to generate trajectories which are among the shortest possible and avoid obstacles at the same time.

The octomap detects the obstacles and allows the Moveit planners to generate plans which avoid detected obstacles. We see that it also generates some noise, meaning some small obstacles which are not real. We could eliminate some of those at the cost of lowering the detection rate of smaller obstacles by modifying the underlying octomap sensor model.

We see from the results that the controller is able to follow the trajectory, but is not very accurate at it. That is what we expected when we designed the controller in the first place, as it is a simple controller without a control loop. A way to improve its behavior is to implement a control loop, which would give the controller real time feedback about how well it sticks to the planned trajectory. The controller would then adjust its output which would regulate the difference between the expected and observed behavior.

Another problem is a drift detection. That should be resolved by finishing the data fusion from the DSO and the drone odometry as the DSO odometry detects the drift correctly. When we detect that drift and we have a controller with a control loop, the system should be able to eliminate it easily [3]. The current fusion is described in chapter

9. Conclusion

When we were looking for a complex control system in which we would like to solve the drone "go-home" problem, we realized there is no such system currently available in Robot Operation System platform. We present the system we designed for that purpose in this thesis along with two solutions to the "go-home" problem within the proposed system.

The first proposed solution is a simple backtracking mechanism. It remembers the path of the drone when flying somewhere. When prompted, this system creates a reverse trajectory of the drone and attempts to return to the starting position by backtracking.
The second solution we propose is an automatically generated plan. Its starting point is the current drone location and it generates a plan to the original drone coordinates. The generated plan is executed when prompted.

The proposed system aims to be as modular as possible and it is designed to solve broader navigation and planning problems in real time. It covers a drone driver, a visual perception system, data fusion, map building and maintaining, automated planning and drone control.

The specific subsystems used in this thesis reflect the needs of a Parrot Bebop which was the drone we were developing our system for. We use a visual processor designed for a monocular camera. The data from the visual processing system are used to update a map. The planning system then generates a trajectory with respect to the created map, leading from the drone's actual location to a desired position. In a case of the "go-home" problem, the desired position coordinates in the drone coordinate system are $(0, 0, 1)$. The controller then executes the generated trajectory.

The current system implementation has its problems. We are not able to detect and regulate the drift of the drone, the controller we are using is not precise and we are not able to detect new obstacles when we are already executing the planned trajectory. However, the proposed system is designed to be modular and allows individual subsystem replacement. Our proposal how to fix those problems is in the next section.

The system's overall functionality and the proposed solutions to the "go-home" problem were evaluated by the series of experiments. We concluded that the overall system design worked well as it detected the obstacles and was able to create plans which avoid the detected obstacles. We were able to solve the "go-home" problem with no obstacles. The boomerang solution had a deviation under $1m$ and the backtracking solution's deviation was under $1.7m$.
When the system encountered an obstacle in a more complicated scenario, the obstacle was detected and the boomerang planner planned a trajectory around it successfully. However the drift of the drone caused by the obstacle prevented the controller to execute the trajectory correctly. The backtracker had a similar

problem, as it also had a valid trajectory, but the drift had the same effect and prevented the controller from the correct execution.

9.1 Future work

We encountered many technical problems when we were working on the system this thesis presents. Those problems delayed the implementation of more sophisticated subsystems. We solved some of the problems only sufficiently enough for the overall system to work, but there are theoretical results already which should allow a far better solution. The problems which remain to be solved are described here in a descending perceived importance. Other problems we solved resulted in a working implementation of the proposed system and solutions to some of the more technical ones are described on github of this project [37].

The plan execution monitoring adds a new capability to the system - to react to new obstacles when executing a trajectory. The controller improvement and better odometry data fusion should improve the precision of the system greatly and should allow more complex scenarios to be successfully completed. We would consider implementations of these three features as major improvements to the overall system.

When all of those three systems are implemented, we would be able to create a complex solution to the go-home problem. It would be a single program, which would generate a plan to the starting position. During the execution of that plan, it would monitor the planned trajectory for obstacles. If it would have found a new obstacle, it would be able to re-plan the solution and start the execution of a new plan. The program would be also able to start the backtracking option at any time. A good time would be if the battery was running low or if the execution was taking too long. If the program was not successful in its execution, it could find a safe place to land.

9.1.1 Problems to solve

Work on the data fusion in a way that the visual odometry plays more important role and helps to correct drift mistakes [3]. The drone does not detect drift properly without it. We did quite an advancement when working on the solution, but we were forced to move forward in order to create a working system without finishing the work on this problem. The fused data with update to the tf tree should greatly improve behavior of the system. The code covering our advancement in the odometry data fusion is one of the attachments of this thesis.

- Add a control loop to the controller or write a full PID controller. That will improve the plan execution precision.

- Moveit supports a control loop over a plan execution. Use it to re-plan when encountered an obstacle in a planned trajectory. It will add a new and desired capability to the system.
- Create a model which estimates the point-cloud scale instead of using fixed configuration all the time.
- Include the drone IMU in the DSO initialization process to get faster and more robust initialization. It might help with the point-cloud scale and odometry data fusion too.
- A long term loop closure would help for longer flights and with some of the map updates. DSO is relatively good right now, but it is always possible to improve sub-systems.

Appendices

A. Robot Operating System

ROS is a meta-operating system, providing services including hardware abstraction, low-level device control, message passing between processes, package management, common-used libraries and tools to write and run own code even across multiple computers or devices. It runs on Unix-based platforms. It is primarily developed for Ubuntu and Mac OS X systems. ROS is highly modular and very handy for reusing code.

ROS doesn't guarantee real time processing, mainly because some messages between processes might be delayed for a few milliseconds. For our purposes it isn't a huge problem, as the message transfer from and to Bebop is delayed anyways with higher delay than within ROS processes. There is a documented way to integrate real time systems in ROS if there is a need for that. There is also a branch of ROS in development which would guarantee real time message delivery for its processes.

A.1 ROS file system

File system mostly covers resources found on a hard drive.

Package is the basic ROS unit for organizing software. A package could contain a runtime process(node), dependency library, data structures etc. It is the most granular thing one could release in ROS.

Package manifest (package.xml) provides metadata about a package such as name, description, author, dependencies etc.

Repository is a collection of packages which are logically bound together.

Message types are message descriptions stored in "*path*/package/msg/MessageType.msg" which define the data structures for specific messages.

Service types are service descriptions stored in "*path*package/srv/ServiceType.srv" which define the data structures for specific services.

A.1.1 ROS Computation graph

The Computation graph is a peer-to-peer network of objects which perform a task together.

Nodes are processes that perform computation and other tasks. A robot con-

trol system is usually composed of many nodes. Some of them provide device interface, some of them provide storage access, some execute algorithms etc. A node is written using a ROS client library, such as roscpp or rospy.

Master provides a name registration and lookup for other parts of the system. It is the core part of ROS.

Parameters server allows data to be stored by key in a central location.

Message is a data structure used for node communication. It supports primitive data types, arrays and nested structures.

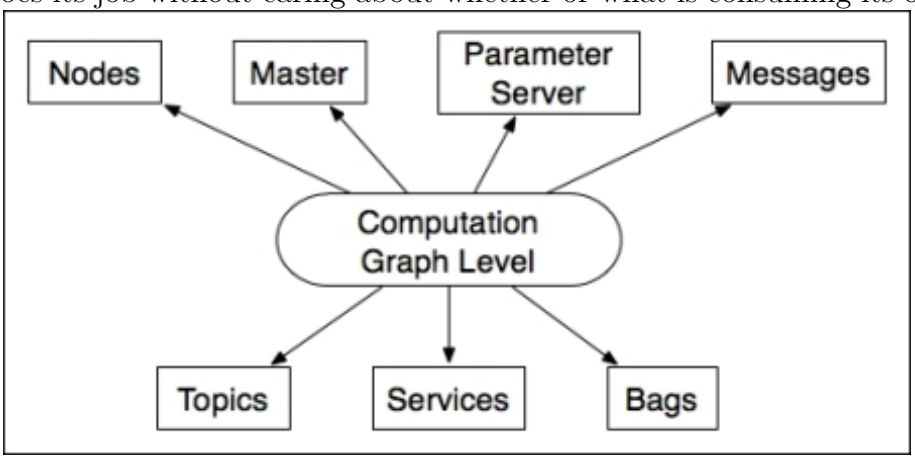
Topic is a system for a node to publish messages with publisher/subscriber semantics. A node publishes messages to a topic and other nodes may subscribe to that topic and consume messages there. Any number of nodes may publish to a specific topic and a node can subscribe to any number of topics as long as they are of the correct type. Publishers and subscribers are not aware of each others existence.

Services are used for request/reply node interaction. A node providing a service usually offers service by its name and input/output data types and a client sends a request and awaits its result.

Bags are a format for saving and playing back ROS messages. They are very useful for storing an experiment data and playing the experiment over and over again to test and develop algorithms.

Master provides connection data to nodes in a way DNS server provides addresses. Nodes register their services and topics directly to master. It also provides any updates and changes which my occur at runtime, such as that some node started providing new service, the topic changed its type etc.

After nodes get their information from master, they communicate directly with each other or directly with the topics. The communication is very similar to TCP/IP protocol in its socket usage and is called a TCPROS protocol. This allows the communication and computation to be decoupled in a way that each node does its job without caring about whether or what is consuming its output.



A.2 Important packages

We described the most important ROS principals. What makes ROS such a useful tool though is what is already done and usable inside it. We will present few basic or low level ROS packages here.

Starting multiple nodes in the right order might get complicated. *Roslaunch* is a tool which takes care of that task. It launches ROS nodes locally or remotely as well as sets up values on parameter server. Everything is specified in a XML file.

Rosbag is a tool for recording topics and replaying them later. It avoids serialization and deserialization of messages. A bag has an option to run a simulated clock which corresponds to the time the data was recorded.

Roswtf is a tool for diagnosing a running ROS system.

Tf2 is the second generation of transformation library. It keeps track of multiple coordinate frames over time. It maintains a tree structure of the frames in time and lets user quickly get a transformation from any frame to another one at any time. We will be using tf, the older library. It was rewritten in tf2 though and only provides the same interface as tf did.

Joy package is a driver for generic joystick or gamepad device. It provides a joy_node which provides a "joy" message, containing state of all buttons and axis on the joystick.

B. Simulator

Gazebo [24] simulation of the whole system is available. We can fly the drone around and create motion plans for it. The simulator is based on the work of Alessio Tonioni [21].

To run the simulation, go to the BebopAutoFly project to the sh_files directory and open three command windows. Although the whole system could be run from just one command window, three of them will make sure we do not get confused with the outputs.

- In the first command window, run the bash script 1-simulator. That runs Gazebo and spawns a drone in it.
- In the second command window, run the bash script 2-controller. It runs the controller and it is usefull to have it in a separate window to see what the controller is doing.
- The third command window should run the bash script 3-planner. It runs the Moveit system and Rviz.

You can now control the drone by the means you are used to. It is now possible to use the command window in Rviz to create and execute plans. All created planner nodes as described in chapter 6 of this thesis, are usable in the simulator as well. The simulator is very useful in custom plan development as we don't need to fly the real drone in order to test new planners.

Bibliography

[1] 2016 arXiv:1607.02565 J. Engel and V. Koltun and D. Cremers *"Direct Sparse Odometry"*

[2] http://wiki.ros.org ROS *"Robot Operation System"* `http://wiki.ros.org`

[3] 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) J. Engel, J. Sturm, D. Cremers. *"Camera-based navigation of a low-cost quadrocopter"* pp. 2815-2821

[4] https://github.com Mani Monajjemi (Autonomy Lab, Simon Fraser University) *"Bebop autonomy"* `https://github.com/AutonomyLab/bebop_autonomy`

[5] http://www.theverge.com Nick Statt *"Watch this DARPA drone speed around a warehouse at 45 mph"* `http://www.theverge.com/2016/2/12/10981740/darpa-drone-autonomous-flight-fla-program`

[6] http://spectrum.ieee.org Evan Ackerman *"Skydio's Camera Drone Finally Delivers on Autonomous Flying Promises"* `http://spectrum.ieee.org/automaton/robotics/drones/skydio-camera-drone-autonomous-flying`

[7] http://spectrum.ieee.org ScienceDaily, 26 May 2015 *"Scientist created drones that fly autonomously and learn new routes"* `www.sciencedaily.com/releases/2015/05/150526085134.htm`

[8] https://www.flightglobal.com Beth Stevenson *"Blue Bear furthers UAV automation research"* `https://www.flightglobal.com/news/articles/blue-bear-furthers-uav-automation-research-403815`

[9] 2007 IEEE VOL. 29, NO. 6 Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, Olivier Stasse, 26 May 2015 *"MonoSLAM: Real-Time Single Camera SLAM"*

[10] http://www.robots.ox.ac.uk/ Georg Klein, David Murray, 2007 *"Parallel Tracking and Mapping for Small AR Workspaces"* `http://www.robots.ox.ac.uk/~gk/publications/KleinMurray2007ISMAR.pdf`

[11] IEEE Transactions on Robotics, vol. 31, no. 5 R. Mur-Artal, J. M. M. Montiel and J. D. Tardós 2015 *"ORB-SLAM: A Versatile and Accurate Monocular SLAM System"*

[12] International Conference on Intelligent Robots and Systems (IROS) David Caruso and Jakob Engel and Daniel Cremers 2015 *"Large-Scale Direct SLAM for Omnidirectional Cameras"*

[13] IEEE VOL 13. NO 4. Umeyama 1991 *"Least-Squares Estimation of Transformation Parameters Between Two Point Patterns"*

[14] http://eigen.tuxfamily.org R. Mur-Artal, J. M. M. Montiel and J. D. Tardós 2015 *"Eigen v3"* https://eigen.tuxfamily.org/dox/group__Geometry__Module.html

[15] http://moveit.ros.org Moveit! *"Moveit! using robots"* http://moveit.ros.org/robots/

[16] http://moveit.ros.org Moveit! *"Moveit! home page"* http://moveit.ros.org/

[17] http://wiki.ros.org ROS *"ROS wiki navigation"* http://wiki.ros.org/navigation

[18] http://wiki.ros.org ROS *"ROS wiki 3d navigation"* http://wiki.ros.org/3d_navigation

[19] https://www.wilselby.com Wil Selby *"UAV ROS integration"* https://www.wilselby.com/research/ros-integration/3d-mapping-navigation/

[20] http://wiki.ros.org ROS *"ROS actionlib"* http://wiki.ros.org/actionlib

[21] https://github.com Github *"ROS Autonomous Flight Simulation"* https://github.com/AlessioTonioni/Autonomous-Flight-ROS

[22] Autonomous Robots ISSN 1573-7527 Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, Wolfram Burgard 2013 *"OctoMap: an efficient probabilistic 3D mapping framework based on octrees"*

[23] IEEE Robotics & Automation Magazine, 19(4):72–82 Ioan A. Şucan, Mark Moll, Lydia E. Kavraki 2012 *"ROS Autonomous Flight Simulation"* http://ompl.kavrakilab.org

[24] http://wiki.ros.org ROS *"Gazebo"* http://wiki.ros.org/gazebo

[25] http://wiki.ros.org ROS *"Yujin ocs"* http://wiki.ros.org/yujin_ocs

[26] http://wiki.ros.org ROS *"Pointcloud library"* http://wiki.ros.org/pcl

[27] http://wiki.ros.org ROS *"TF"* http://wiki.ros.org/tf

[28] http://wiki.ros.org ROS *"Robot localization"* http://wiki.ros.org/robot_localization

[29] http://wiki.ros.org ROS *"Rviz"* http://wiki.ros.org/rviz

[30] https://www.dji.com/ DJI *"Dji"* https://www.dji.com/guidance

[31] IROS, 2015 IEEE/RSJ International Conference Shengdong Xu, Dominik Honegger, Marc Pollefeys, Lionel Heng *"Real-time 3d navigation for autonomous vision-guided mavs"*

[32] HTTP://WIKI.ROS.ORG ROS *"Catkin"* `http://wiki.ros.org/catkin`

[33] Steven M. Lavalle 1998 *"Rapidly-Exploring Random Trees: A New Tool for Path Planning"*

[34] PROCEEDINGS OF THE IEEE, VOL. 92 S. J. Julier, J. K. Uhlmann 2004 *"Unscented filtering and nonlinear estimation"*

[35] IEEE TRANSACTIONS ON ROBOTICS, VOL. 28 I. A. Sucan, L. E. Kavraki 2012 *"A Sampling-Based Tree Planner for Systems With Complex Dynamics"*

[36] HTTP://WIKI.ROS.ORG ROS *"Cmd vel mux"* `http://wiki.ros.org/cmd_vel_mux`

[37] HTTPS://GITHUB.COM github *"This project on github"* `https://github.com/JirkaHarasim/BebopAutoFly`

List of Abbreviations

- UAV - Unmanned aerial vehicle
- MAV - Micro aerial vehicle
- DOF - Degree of freedom
- ROS - Robot operating system
- SLAM - Simultaneous Localization and Mapping
- DSO - Direct sparse odometry
- GPS - Global positioning system
- IMU - Internal measurement unit
- RGB-D - Red, green, blue - depth camera, a camera with enhanced information about the depth of the pixels
- RRT - Rapidly-exploring random trees

List of Attachments

There is an USB flash drive attached to this work, referred to as "Attachment 1". It contains following items:

- A source code of the implementation along with the modified files from other projects
- All the launch files necessary
- A documentation of how to run the whole project in a readme.md file used on github
- An electronic version of this thesis in a pdf format
- A list of encountered technical problems not mentioned here and how to solve them
- The photographic documentation of the experiments
- Screen-shots of the system perception when running the experiments