BSP02 - Localization and Detection of Landmarks for Supporting UAV Navigation

Monday 21st March, 2022 - 18:00

Barbara Symeon
University of Luxembourg
Email: barbara.symeon.001@uni.lu

Dario Cazzato
University of Luxembourg
Email: Dario.Cazzato@uni.lu

Abstract

Unnamed Aerial Vehicle (UAV) have grown rapidly around the world for research. An UAV or drone is the best platform for technological development to detect the location of objects. Robot sensing and image techniques provide opportunities to develop the Artificial Intelligence (AI) field, like Computer Vision. UAV navigation supports exploration to places human cannot go and has application to improve life on earth. This is why it is important to learn robotics, sensors and computer vision. Advances in Computer Vision and Machine Learning with UAV's is a great value to add to self driving, autonomous and remote vehicles. The project goal is to find a machine learning solution to identify and localize an object of interest. The main project is distributed between students of different semester level enrolled in the Bachelor in Computer Sciences (BiCS) program at the University of Luxembourg. As part of the main project, this semester project aims to build a foundation for robot sensing and image techniques. It documents the implementation of a robot node for data visualization from sensors, using Robot Operating System (ROS), along with a scientific overview of Computer Vision and ROS tools.

1. Introduction

”Robotic sensors are nowhere near as capable as human sensors.” ... ”Human vision, on the other hand, is extremely robust. able to tolerate noise, distortion, changes in illumination, different surface reflectance functions and changes in viewing angle.” Peter K. Allen, 1987 [1]

In 2020, face recognition algorithms are better than the human abilities. This is due to advances in computer vision and machine learning. However, scientist are still far from producing intelligent system able to grasp the underlying concept of visual intelligence.

There is an increasing interest in the world towards computational vision. As a result UAV's and self driving vehicles are able to detect obstacle, generate maps and track objects with the help of sensors such as GPS/IMU, LIDAR and panoramic vision with cameras. Those system architecture work by data acquisition, first it pre-process data from camera and lasers. Then it runs processing filters involving vision algorithms. [2]

The main question of this project is to find how to use ROS to acquire data from robot sensors in the real world. Then process the data with some filters and visualize the result. The aim is to develop an understanding of computer vision. This project investigates the foundation of robot sensing, data acquisition, processing, visualization and the use of ROS tools to prepare for later projects involving more advanced concepts.

2. Project description

2.1. Domains

This project main domains fit in the field of Robotics and Artificial Intelligence. Specifically, robot sensing and design for Computer Vision, Graphics and machine Learning.

2.1.1. Scientific. The scientific domain is Computer Vision. It is a subfield of artificial intelligence and machine learning, which involves the use of general learning algorithms and specialized methods. The attempt to reproduce human vision is done with automated extraction of information. This could be images, 3D models, camera position, object detection. To solve problems there is a mix of programming, modeling, and mathematics. [3]

2.1.2. Technical. The technical domain of the project is Graphics, Image processing, unmanned autonomous vehicle sensors and Robot Operating System.

2.2. Targeted Deliverables

This project contains a scientific deliverable and a technical deliverable. The scientific deliverable is an overview of Computer Vision and Robotics for unmanned autonomous navigation. The technical deliverable is a Robot Operating System package with a focus on Image data types and messages publishing rates.

2.2.1. Scientific deliverable. The scientific deliverable is an investigation about computer vision and how it is used to extract information from images. The objective is to understand what is a digital image. This report documents how sensors are used in robotics with the Ros Operating System (ROS). There is a study on ROS architecture and the the tools used for visualisation. Then, there is some experiments on manipulating, extracting and read data from different sensors and robots.

2.2.2. Technical deliverable.

2.3. Constraints

There is some constrains before starting the project.

1) Install Linux Ubuntu 18.04 LTS.
2) Understand Linux file system.
3) Learn how to use terminal to install, edit and compile.
4) Install ROS melodic and its dependencies.
5) Learn how to deal with python's version conflicts.
6) Learn GitHub.

3. Prerequisites

There are some prerequisites for this project.

3.1. Scientific prerequisites

For the scientific part of the project, it helps to know some mathematical concepts in linear algebra, like matrices and vectors. It is essential to know Object Oriented Programming and to be familiar with different operating systems.

3.2. Technical prerequisites

For the technical part, the student must have Python experience. And, must be familiar with C++, Linux and bash.

4. A Scientific Deliverable 1

4.1. Requirements

The requirements are a study about the fundamentals of Computer Vision and ROS. This will also include an exploratory insight of data collection by sensors used in robotics for unmanned air, ground, and sea vehicles.

4.2. Design

The most appropriate research design for this project is an exploratory study. The scientific report will follow a research method approach involving secondary and tertiary sources.

First, there will be a literature review using books, tutorials, online lectures, scientific articles and online official documentation in the domains of study. These are found through the university library and the platform Science Direct. Non-peer reviewed data like tutorials, wiki pages, blog posts found through Google are used to provide an additional understanding which, if relevant, will be investigated.

The scientific report is written at the same time as the technical report. Images and diagrams are used to explain certain concepts. The study will be about several subjects in the following order: an introduction on the field of Computer Vision, an understanding on how sensors collect information from the real world, then image sensor are explored to understand how information is digitized. Following is a comparison between image processing and computer vision filters. Finally, ROS system and tools is used to experiment with 3D data from the real world. The goal is to find how to use ROS and sensors in computer vision for UAV's.

4.3. Production

4.3.1. Computer Vision. Computer vision is a field of computer science. It is the science and technology of making machines able to see. Scientists in computer vision deal with theory, design and implementation of algorithms to process visual data automatically. The aim is to find computational principles underlying visual intelligence to recognize objects, shapes, spatial layouts and track them. [2] Technologies in the domains of face and character recognition has surpassed human abilities. However, there is still a lot of problem to solve to produce a truly intelligent machine. [4]

Visual intelligence is more than detection, categorization, localization, segmentation and tracking of an object. Simple visual tasks for humans can involve complicated sub-tasks for computers to solve. [5] This is when new advance in machine learning take place.

The study of intelligent system has three main components:

1) Data collection. The acquisition of data with sensors.
2) Computational Vision. Interpretation and analysis of this data using learning techniques.
3) Optimal Decision Model actions. The intelligent system has assigned objectives and chooses the appropriate set of actions. [6]

Deep learning has grown image techniques for thresholding, filtering and edge detection. These techniques are applied and presented in section 5.3.7.

Image processing and computer vision are overlapping fields but there is a difference between them. Both involve doing computation on images. Image processing focus on image processing. This means that input and output are both an image. An image processing algorithm can transform images in many ways: smoothing, sharpening, highlighting the edges. In comparison, computer vision focus on making sense of what machines see. Which means that the system inputs an image and outputs a task-specific knowledge, such as object labels and coordinates.

Computer vision and image processing work together in many cases. Computer Vision systems rely on image processing algorithms. For example, a computer vision system rarely uses raw images data coming directly from a sensor. Instead the system uses images that are processed by an image signal processor (ISP). On the other hand, image processing methods have started to use computer vision systems to detect specific landmarks and tune an image locally.

Lastly, machine learning is a field of study that teaches machines how to perform a certain task given a set of examples. Computer Vision uses machine learning. As well as many advanced image processing methods for computer vision, like deep neural networks. These advanced image processing methods use machine learning models to transform images aiming to prepare them for computer vision tasks. Computer Vision system may use convolutional neural networks (CNN) for image recognition. And an internal layer of CNN can be considered as image filters with tunable parameters. So, a model such as CNN is considered adaptive image processing. Therefore, image processing and computer vision are overlapping but very distinct field.

4.3.2. Sensors. Sensors are important in robotics because it is the only way a robot can interact with the real world. Data is collected from sensors which are either passive or active. both collect radiations. Passive sensors detect radiations emitted or reflected from another source by an object. The most common radiation detected by passive sensors is sunlight. A camera is a passive sensor. See section 4.3.3 to understand how camera data becomes digital images. The second type, active sensors emit their own electromagnetic radiations by pulsating energy to illuminate the scene. Then, they collect the radiation that is reflected or backscattered. Such a sensors is Radar (Radio Detection and Ranging). Radar use a transmitter to emit electromagnetic radiations at microwave frequencies. The distance of an object is determined by a directional antenna. This receiver measures the arrival time of reflected or backscattered pulses of radiations from distant objects. For example, LiDAR an active optical sensor uses lasers to emit light pulses in the ultraviolet or near infrared spectrum and a sensitive detector to measure the reflected and backscattered light. Then it uses the speed of light to calculate the distance of the object. [7].

Inertial measurement unit (IMU) is an other type of sensor used in UAV's, but also in spacecrafts, satellites and self driving cars. It uses accelerometers and gyroscopes or even magnetometers to collect data about velocity, orientation, and gravitational forces [8]. IMU is used for manoeuvre with servo motors and vision sensors on UAV's. The set of sensors mentioned before are essential for a basic robot to move in the real world.

The Global Positioning System (GPS), can be used in combination with IMU to correct drift error. But it is possible to show a moving object in a 3D world without GPS. Because with IMU, there is no need to communicate with satellites and radio transponders. IMU is a smooth estimation from the starting position coordinates x_0, y_0, z_0 compared to GPS navigation.

4.3.3. Digital images. Digital images are the result of a camera pipeline, a chain of image processes. The light goes through a physical color filter array on a camera sensor with a checkerboard pattern, known as "Mosaic of Bayer." see Fig14b, p.19. Then, the image sensor captures a three-dimensional scene. The projection of the scene is a two-dimensional plane function $f(x, y)$. The location of the picture element or pixel is represented by coordinates x, y, and it contains the intensity value. The capture is defined as a digital image when the intensity and the x, y values are discrete. The image has a finite number of points (pixels), with a numerical value, represented as a matrix. [9].

There is two types of digital images: bitmaps (raster) and vectors. bitmap images are composed of pixels which are combined to make continuous-tone images. The information is stored in bits and mapped into a digital image. Vector images are geometrical object stored as mathematical formulas.

The dynamic range defines the quality of the image. It is the $Span$ of the image between the lightest and the darkest tone. Gray digital images are stored as 8-bit. The tone range has 256 values. Black has the value 0, and pure white is 255. A binary image is an MxN matrix. The intensity value is in range$[0, 1]$ with 0 for black and 1 for white.

On the other hand, Color images are a 4-bit color, consisting of at least 16 colors per pixel but are often sampled at 24-bit. The three values are red, green and blue (RGB.) [10] True-color RGB image is an MxNx3 matrix. Each pixel has a red, blue, green component. The pixel (m, n) has red = $(m, n, 1)$, green = $(m, n, 2)$, blue = $(m, n, 3)$. These RGB components are the 3 communication channels R, G, and B. Gray-scale MxN, has one channel. The 3D color space of the RGB system is represented in (Fig14a, p.19).

In short, "A digital image is made up of MxN pixels, and each of these pixels is represented by k bits. A pixel represented by k bits can have 2^k different shades of gray in a gray-scale image. These pixel values are generally integers, having values varying from 0 (black pixel) to $2^k - 1$ (white pixel). The number of pixels in an image defines resolution and determines image quality." [11].

For example, it is possible to represent images as multidimensional arrays using python libraries: OpenCV and Numpy. The array has column and rows of pixels, and each pixel has a value.

If printed on the console, we have a 3 by 3 black image. And each pixels is a single 8-bit integer. This one channel array is a grayscale image.

To get three channels there is a conversion to color there is conversion function. Each pixels has a three-element array, and each integer represent one of the three color channels: B, G and R. An illustration to this is on Fig12, p.18

The shape property returns rows, columns, and number of channels. There are other color model, like Hue Saturation and Value(HSV). The pixel value for this model is in $range[0, 180]$.[5]

In python, a digital image is simply an array of numbers defined by an integer data type.

4.3.4. Filters. There is different types of filters for tunneling images. Basic filters, like demosaicing, see Fig14b, p.19 which restructures the image, are different compared to edge detector. Demosaicing is one of the processing filter used in the camera pipeline for digital images. Which makes it an image processing filter whilst edge detection is a filter for computer vision.

The edge detector is an analysis of the image. It is a local operation. The search of image pixels where discontinuities appear on the image, often on the edge of an object. To find the edge, local variations in local intensity (discontinuity) can be quantitatively assessed. It is possible to understand the general concept by looking at graph on Fig18a, p.22 and, at the corresponding image line, where discontinuities a clearly seen on the graph.

In the technical deliverable, the edge_detection node uses a Canny edge detector and a threshold filter to experiment on the significant edges. The Canny edge detector already has two threshold filters in its algorithm so it adds a third one. A visible effect of the process can be seen on Fig18b, p.22. Where Fig18ba and Fig18bb are used to calculate Fig18bc. Then, Threshold is added to them and we have significant edges highlighted on Fig18bd. Since those figure are not using The Canny edge, here it is clear how gradients and filters work together to highlight edges.

Canny edge detection is a complex state of the art algorithm. For the technical deliverable, the OpenCV function cv2.Canny() is used. It works in a five step process [5] :

1) Gaussian filter to denoise the image.
2) Gradient intensity first derivative and direction
3) Find the best edge from a set of overlapping edges.
4) double threshold on all detected edges to prevent false positives.
5) Analyse all edges and their connection to each other to keep the real edges.

The concepts for cv2.Canny() algorithm are as follow: The multiple stages start with noise reduction since edge detection is sensitive to noise. Then, the intensity gradient of the image are found by calculating horizontal and vertical first derivative of the gradients G_x and G_y. These two gradient images can give the edge gradient and pixel direction for each individual pixel:

$$EdgeGradient(G) = \sqrt{G_x^2 + G_y^2}$$

$$Angle(\alpha) = tan^{-1}\frac{G_y}{G_x}$$

The direction of gradient is perpendicular to edges. It has four angles: vertical, horizontal and two diagonal directions. The magnitude is given by rounding to one of the four directions. Then unwanted pixels are remove by scanning the image. A point on an edge is compare to two other points. These points are perpendicular to the one on the edge and the point scanned is in between. If the point on the edge forms a local maximum in comparison with the two other points it is compared to, then it is set the value 1. Otherwise it is set the value 0. See on Fig18ca, p.22 the visual explanation. The result is a binary image with edges.

The last step, decides the real edges using threshold values: *minVal* and *maxVal*. Above *maxVal*, are edges. In between *minVal* and *maxVal* thresholds, if a pixel is connected to an edge above *maxVal* it is considered an edge. Other pixels are discarded regardless of the region they are in. Therefore *minVal* and *maxVal* are values to chose carefully. See the graph of *minVal* and *maxVal* with edge lines on Fig18cb, p.22. [12]

Canny edge detection is the type of computer vision filter which prepares for shape matching, such as circles or lines. Therefore, it is one of the fundamental tasks for image analysis.

4.3.5. ROS architecture and concepts. The Robot Operating System is a software framework for building robots. It is collaborative, flexible and it works across different robotic platforms widely used in the robotic community. ROS was built for collaborative group to be able to work on each other's work. This simplifies the creation of complex robust robots. As a result it is the most used software in the robotic industry [8].

ROS is compatible with most sensors and provide hardware abstraction, low-level device control, implementations of commonly used functionalities and libraries. ROS graph architecture uses nodes which are process that can read data from sensor. Messages with the data are send between nodes. *catkin* and *cmake* manage packages. For example, to navigate autonomously or, to run vision algorithms for mapping. The nodes are centralised around a *roscore*, which allows nodes to communicate messages by publishing and subscribing to topics within the graph net. Fig15b, p.20

A package is a small program inside ROS. It can have configuration files, nodes, messages, services, images, launch files and so on. The package manifest file: *package.xml* contains all the properties of the package. It allows to build the package from source. If the package dependencies are incorrect it will not work on other machines. The other important file to build the package is the *CMakeLists.txt*. It is the description of where and how to build the code, it is the input to the *CMake* build system.

The data is communicated between nodes with simplified message descriptions. Each message type has a field and a constant. For example, the std_msgs String has a field: string and a constant : data. From this, ROS can generate a source code in several programming languages.

The computation graph level is a network in which all the processes are connected. Nodes in this network can access and transmit data and interact with other nodes in it. See Fig15a, p.20. Nodes process computation, they are executable. Often one node has one functionality. These are written in c++ or python using rospy and roscpp libraries. The master register names and sets connection between nodes, it tells the node where topics are located. One master can be shared between computers but the system needs one master to function. Parameters server can store keys and change parameters of a running node. Services allows to make requests to a node and bags format to save data as messages and replay them later. Bags are essential to work with complex robots.

Data is transmitted to topics not to nodes. Therefore, production and consumption of data is decoupled. Important files of ros packages are launch files. It is a ROS feature for launching more than one node at a time. [8]

4.3.6. ROS Visualisation tool. ROS provides visualisation tools: Rviz visualizer and navigation libraries. These are high level community libraries developed in the *ros-pkg* repository. They provide powerful tools for visualization and simulation of sensors data. For example, Rqt graph shows the graph system. On it, there is interconnections between nodes, where nodes are the processes and the edges between topics are data workflow. See Fig16, p.20. Rqt_viz is a graphical interface that enable to visualize 3D data form sensors. Rviz shows the data in the correct position from multiple sensors at a time in a 3D world frame. This allows to see what the robot can see. Rviz supports data types, such as Odometry, Path, PointClouds, Images.

Simulating robots can be done using virtual world on the gazebo platform, real robots are shared by ros community. For the technical delivery, the Husky unmanned ground vehicle (UGV) [13] was used in a gazebo simulation. See Fig19a, p.23. ROS Graphical User Interface (GUI) allows to play with topics, messages and to monitor processes. Which helps to see resources that nodes use when the software run. Information about topics, publishers, subscribers, messages and rate are available on rqt topic monitor. Rqt_graph show topics and nodes connections. These are essential to inspect processes.

The KITTI Vision Benchmark Suite is a project of Karlsruhe Institute of Technology and Toyota Technological Institute at Chicago. the project provides data sets collected from an autonomous driving platform: Annieway. It is a station wagon car equipped with two high-resolution color and grayscale video camera, a Velodyne laser scanner and a GPS localization system. See the car sensors on Fig17a, p.21. The data sets are captured by driving around the city of Karlsruhe. [14] The data was converted to a rosbag to use with ROS. The rosbag file contains all the messages published on topics during an autonomous car session. The record timestamp and the timestamp in the header of published messages allows to play back everything with different rates. Topics and messages types can be inspected from the rosbag file too, see Fig17c, p.21. All the information about the data set is detailed.

Visualizing data from sensors in Rviz is in relation with frames. The transformation frame coordinates all the frames. LiDAR PointClouds have a location in the world and we can see how frames move in relation to the transformation frame. The robots handles calculation and knows where components are. This allows a robot to move in a simulation like with the Husky robot. [12] For example, the Odometry is the distance relative to a point. So, it is the distance between a point in frame Odom and base_link. The transformation tree for the Annieway driving platform can be seen on (Fig17b, p.21). For Anniway sensor frames are relative to IMU, whilst for Husky frames are relative to base_link.

The ROS Point Cloud Library (PCL) has data types and data structures to help represent points of a sampled space in 3D. There is Point Clouds types to manipulate information from 3D sensors in rviz, like LiDAR. 3D sensors sample a finite set points in a 3D frame of reference. Then there is state of the art algorithms for 3D data processing. ROS has functions to convert between messages and point clouds data structures. This combination makes surface reconstruction, filtering and model estimation possible. [8] There was experimenting with point clouds and laser sensors with the Anniway platform rosbag. Point cloud are important for computer vision.

4.4. Assessment

The objective of the scientific deliverable was to give a general overview of Computer Vision using Robot Operating System.

The field of Computer Vision was introduced and the literature review was conducted to review the main concepts of Computer Vision in relation to Robotics. The concepts related to Graphics, like image processing and basic filters were accessible from the start.

The research provided an intermediate understanding of the ROS architecture and how it works. Concepts, tools and application were experimented. An excellent overview was achieved on the project to understand abstract concepts in robotics. Many tools of the rqt interface were learned. Specifically to develop and inspect nodes, topics and messages. For then visualize, evaluate and validate results. This includes complex 3D data type with rviz and image view and the inspection of graphs from different systems. As well as the conversion of raw data to rosbags and the interaction with real world data from sensors.

Since every aspect of the research was new, the difficult parts was to apply basic concepts, solving bugs and errors for basic tasks related to ROS. This means interacting with topics and messages. Or, running visualizing tools. Converting data to rosbag was difficult, since the kitty to rosbag package needs a dozen of other packages to work properly. Which is not obvious at the start. But later, with experiments on different robot systems it was easier to understand how robots processes relate to each other, and how it works.

These experiments with ROS tools are a good base for later robotics projects. Computer vision concepts is only an introduction. Therefore, there needs to be more research on machine and deep learning algorithms as the student advances in the program.

5. A Technical Deliverable 1

5.1. Requirements

Requirement are a guide of features and performance factor that enables a software to serve its purpose. Every requirements implies that there is a level of effort needed to find a solution, like data gathering or automation of tasks. [15]. Functional requirements are about the behavior of the software. Non functional requirements are the quality attributes.

Functional requirements:

FR1: A node streams Image data type with FPS control on a slider.

FR2: A node applies image processing filter.

FR3: A node applies a computer vision filter.

FR4: A node tracks the path of a simulated robot using Odom data type.

FR5: The main launch file opens configured rviz tool window and image windows.

Non-functional requirements:

NFR1: There is a *catkin* compliant *package.xml* file which provides meta information about itself.

NFR2: There is a *CMakeLists.txt* which uses *catkin*.

NFR3: The package must have its own folder.

NFR4: Processes have distinct executable nodes.

NFR5: There is configuration files for ROS visualisation tools.

NFR6: Each node has an individual launch file.

NFR7: There is a main launch file for each type of data source (Image from webcam, video file and rosbag; Odometry from gazebo simulation).

5.2. Design

The ros-visualize package has multiple nodes for data visualization. The technical deliverable sets the foundation for later projects. It aims to teach how to use and process the data collected from sensors with the robot operating system. First, some time is dedicated to fulfill the constrains listed in section 2.3 to ensure the project can take place. Then, Package nodes are implemented with Python 3.7, on a Linux machine with ros melodic installed. The Python client library for ROS used for the implementation is rospy[12]. The advantage of rospy over The C++ client library roscpp, is developer time over run time speed, and the student is more familiar with Python.

The first part of the project consists of building a package with a simple node. Then, the scientific review introduces ROS tools with the Kitty data set.[14] Which is used to Inspect nodes topics and messages. Then, the implementation of image processing nodes starts while the scientific review focuses on understanding digital images.

Once, image topics are understood, Python libraries for Computer Vision are tested on various small programs using OpenCV 3.4 and Numpy. These small programs are later implemented as ros nodes. The most important node is the frame rate control slider. It is ros node that combines image topics with fps algorithm, and it is tunable with an Open CV slider.

The last, part of the project is to use all the knowledge acquired to implement a node with a new type of message. The Odometry message is used to draw a path behind a moving robot.

The intention of the package is to show different type of data using ROS framework and tools. This semester project is focused on learning more than it is on producing an end product. As such, most programs produced during implementation phases are not going to be discussed or shared, only the four most interesting nodes are given together in one package with launch and configuration files.

5.3. Production

5.3.1. Simple node. A node is an executable file in a ros package. Nodes need a Master to run to be able to find topics to subscribe and publish to. When the master is running it registers addresses and unique name of nodes, topics, to form a network. Nodes connect to the network to publish a message to a topic. Now other nodes can subscribe to this topic and read message data.

To make sure a ros node is executed with Python the first line is always the Python shebang. Then rospy and the type of message used by the node is also imported. The function Publisher() communicate the topics, the message type and a queue size.

Following, the rospy.init_node function, gives a name to the node and enables communication with the Master. The parameter anonymous is set to "True" to gives a unique name to the node and avoid conflicts between nodes.

A rate object is created to work with the sleep() method. This ensure the node publishes at the desired rate. If the rate is set to 10, there is going to be 10 loops per second. The loop has check is_shutdown() flag, to know if it need to stop. Then, inside is the process. For a simple node a string is declared and published.

The subscriber is similar to the publisher node. It starts with the Python shebang. Then libraries and message types are imported. The only difference is a callback function for subscribing to topic messages. When new messages are received, callback is invoked. Rospy.spin() is keeping the node running until the node is shutdown. See Fig5, p.12. The node is functional when *catkin_make* is run in a catkin workspace. [12]

5.3.2. Ros package. The ROS package deliverable file system tree can be seen on Fig1b, p.10, with the command to install the package on Fig2, p.10. The command makes the catkin workspace and source directories, then builds the executable in the build space directory configured.

The folder which contains all packages is the workspace. See on Fig1a, p.10, each folder role is as follow: [8]

1) The source space: contains the project packages. It will invoke the *src* folder on the *CMakeLists.txt* with *CMake* to configure the package.
2) The build space: contains all the intermediate files for *CMake* and *catkin*.
3) The devel space: keeps compiled programs for testing.

In the *package.xml* file Fig3, p.11, all dependencies are specified. In the *CMakeLists.txt*, tags <build_depend> is for packages that must be installed before installing this package and tags <exec_depend> is for packages needed to be able to run the package.

The package has launch, configuration and compilation files. External packages *usb_cam* [16] and *vision_opencv* [17]. The processes are shared between nodes, each nodes has a launch files with tunable parameters.

5.3.3. Kitti to rosbag. The KITTI data set raw data [14] was converted to a ROS bag with the *Kitti_to_rosbag* tool and the following packages from ETHZ repository on GitHub: catkin_boost_, python_buildtool, catkin_simple, cmake-build-debug, eigen_catkin, eigen_checks, gflags_catkin, glog_catkin, minkindr, minkindr_ros, numpy_eigen.

The rosbag can be inspected with ROS tools. First, the rosbag is played and examined, by running rosbag info after playing the bag file. See Fig17c, p.21. After running Rqt and Rviz tools, there is a lot of message types to inspect. Like Point Cloud, Image, Imu, Tf.

5.3.4. Image. Three of the nodes produced are processing Image type messages. These type of nodes subscribes to an Image *sensor_msgs* topics from a camera device, for example *image_raw*. Then to process the data, the Open Source Computer Vision Library (OpenCV) is used. As well as Python libraries Numpy, to provide computing functionalities and efficient arrays. [5]

The OpenCV Library is an infrastructure for computer vision applications. The library has optimized algorithms and contains classic and State Of The Art computer vision and machine learning algorithms. " These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality, etc." [18].

In Ros, an Image message is an uncompressed image matrix with the point (0, 0) on the top left corner. The full message raw and compact definition are in the ros documentation [12]. Inside the nodes we use the matrix *data* which is an uint8 array of size (step x rows).

To use OpenCV interface with ROS, the ROS images are converted to OpenCV images with *cv_bridge*. Then once the image is ready to be published by the node onto a topic, the image is converted back to image ROS format. [12] For the implementation of the nodes, functions on Fig6, p.13 were used in all nodes involving Image data type. The conversion is wrapped to catch conversion errors. Dependencies need to be added in the manifest *CMakeLists.txt* Fig4, p.11. These are: sensor_msgs, opencv2, cv_bridge, rospy, std_msgs.

The image message is the input for the conversion. The encoding of the Image message is set to "passthrough" to stay unchanged. The encoding is a CV::Mat image. Mat is C++ class with two data parts: a matrix header and a pointer to the matrix containing pixel values. the header has a constant size, since it contains a fix number of information about the matrix but the matrix size varies. [19]

Cvbridge colors pixels, the encoding illustrates very well the different colors channels and bit sizes for processing images. See the encoding on Fig13, p.18.

5.3.5. Filters. Two nodes uses filters and all of them uses OpenCV sliders as a switch or to tune the filter values. For the trackbar high-level GUI functions are necessary: [19]

1) cv2.namedWindow(): generates a window used as a placeholder for trackbars and images. To close the window and stop memory usage cv2.destroyAllWindows is called.
2) cv2.createTrackbar(): Creates a trackbar attached to the specified window. Name and range control are the first two paremeter, then the range value is synchronized with a callback function called on position change.
3) cv2.getTrackbarPos(): returns the trackbar position

In the filter node, a gray switch conversion and Threshold filter with range control were implemented. See Fig6, p.13 for an example using a switch and a range along with OpenCV image filter function grayscale and threshold. In the fps control node, the fps slider is a simple range from 0 to 30. Since the minimum value for any slider is always 0, for the Fps slider, the value for 0 is changed to 1 automatically with an *if* statement to avoid a *ZeroDivisionError*. See the node update() function on Fig8, p.15.

For the filter node, the threshold sliders controls the value for a truncate threshold. A binary threshold sets the image value to either black or white, whilst the truncate threshold give an intensity value. In the scientific section we studied that, for a pixel point src(*x, y*), the canny edge algorithm compares the point to the maxVal, then set the value to 1 if it is bigger than maxVal, and to 0 otherwise. For the truncate enumerator, if the pixel point src(*x, y*) is bigger, it sets the value to threshold, otherwise the pixel keeps its value.

Therefore the *thresh_trunc* enumerator in a threshold function does not return a binary image. See the filter implementation on Fig6, p.13. Binary thresholds are used in the canny edge detector node. The sliders control the *minVal* and *maxVal*. See the edge detector algorithm and trackbar implementation on Fig9, p.16. A combination of all the filters and parameters is possible when the nodes are connected in a pipeline style, see nodes graph systems on Fig16, p.20.

5.3.6. frame rate control. The fps controller is in two nodes, one that captures frames from a video file with the cv2.VideoCapture() function, and one that captures frames by subscribing to a camera device topic. There is two ways to control the frame rate with a slider. The easy one is to use an external library. See how it works on Fig7, p.14. Here, the *Imutils* library was uses with the cv2.VideoCapture function. [20]

The other node controls the frame publication on other topics with a simple algorithm. this second second node is using incoming Image data from a camera topic. See Fig8, p.15 for the algorithm. The frame control algorithm for this node works in steps. First, the node initialises and gets an instance of the class Fps. This generates a trackbar for frame control. The controller starts with the fps value retrieved from the launch file parameters. Then the node subscribes to the Image topic using the rospy.Subscriber() function. The subscriber function calls the callback function for every frame. The address is set with rospy.Publisher() for the topic to which the node will publish each frame. Then, the function update() retrieves the trackbar position and checks for *ZeroDivisionError*, and returns 1 if the value is 0 for the variable fps. Then, the current frame time for ros is set with rospy.Duration(1/fps), this gives an approximate duration. If the fps parameter is set to 30 for the first loop, this duration will be a 30^{Th} of a second. Now, that we have a duration, a current time is allocated to the variable begin_time with rospy.Time.now().

By adding these two variables together, we have set a time value in the future at a 30^{Th} of a second later. This is used to return a Boolean value to enter the while loop until a 30^{Th} of second has passed since the callback function was called. The while loop publishes a frame. Then the rospy.sleep() function keeps the loop on pause for the frame rate duration set earlier. For this example, the callback function takes exactly a 30^{Th} of second to publish one frame. Therefore every time the value changes on the trackbar, its inverse values sets the time it takes to publish one frame with the callback function. This allows to control the frame rate per second from an interactive slider.

5.3.7. Robot path with Odometry. The last node is focused on the Odometry message type. Odometry is navigation message. See Fig19b, p.23 for a visual explanation. The message is an estimation of the position and velocity of an object in the six direction of freedom in a 3D space. The Inertial Movement Unit or the simulation with gazebo gives the Odometry twist and pose. The position is given by a covariance estimation with twist and a covariance estimation with pose.

To draw the path, we need the coordinate frame from the odometry header of the robot. To do this we need to import Path and Odometry nav_msgs, and the PoseStamped from geometry_msgs. Then When the node is initiated, it subscribe to the Odometry topic. In the subscriber callback the variable path header is set from the Odometry header. Then the pose header and pose are set from the Odometry header and pose, which is the odometry pose, the frame it is from and a timestamp. The pose and header are appended to the Path message type. The path is now an array of poses which can be represented as a path in a 3D frame. The path is published on a topic. See in rviz on See Fig19c, p.23Finally, the process starts again when Odometry is updated. [12]. See Fig10, p.17.

The path can be visualized with rviz by launching a simulation robot and by subscribing the path node to the robot odometry topic. See Fig11, p.17. Inside the comments are two husky simulation and a world. The launch file launches multiple nodes and configuration files. An empty world is simulated with the husky robot, the ROS tool rviz is launched configured for navigation. The path node is launch, subscribing to an Odometry topic and publishing on the topic *path*. Finally, the launch file runs a .perspective file for rqt tool to analyse the tf graph, the node graph and to check that everything works.

5.4. Assessment

The technical part of the the bachelor semester project had a couple of stages. The first stage was to learn how to make a simple ROS node. This was technically hard for student because the terminal interactions, compiling problems with Python versions and ros in general was new. It was hard to differentiate between system, Python version, compiling, control flow or algorithm bugs. It took approximately two weeks to have a running system with ros melodic with all the packages and dependencies. It was easy to implement a simple node but it took an extended amount of time to understand how to implement an Image node properly. ROS is complex at first.

However, once the student understood how to think about ROS, how it is used and what it can do, implementing nodes and exploring message types became more natural.

The second stage, after implementing simple node was to convert a data set to rosbag. It is simple to understand rosbag but it is only when using ros was more natural that the student was able to use the data set for basic visualization in rviz. Finding the right packages on GitHub to make the conversion work made the student understand deeply how dependencies and packages function. This is true for ROS but also for Linux in general. A lot of technical skills were learned, in terms of operating systems, source files, build with *CMake*, and so on. See Fig16c, p.20 for the system node graph for this deliverable.

There was two error that the student could not solve. In a ROS node that uses the OpenCV VideoCapture() function, streamer.py, an error escalates to *SIGKILL* when terminating the roscore. Also, I noticed that the video node, streamer.py, is only processing a few frame per second. The reason may be that because the video compression from the drone used is 4k. Due to the size and decoding time, it seems that the video processing pipeline is only processing a few frame per second. This happens inside a ROS node, it is not a problem when using OpenCV VideoCapture() in a basic Python program. See Fig16b, p.20 for the system node graph for this deliverable.

The last unsolved issue is in relation to the path.py node which is publishing messages too fast. This overwhelms subscribers to the */path* topic. The student understands how to fix it but did not take the time to solve it. Instead, the student spent time to explore and understand the concept underlying the message type Path and the other messages it is dependent from. See Fig19c, p.23 for the path draw in rviz for this deliverable.

Lastly, the system from a camera device work perfectly, See Fig16a, p.20 for the system node graph for this deliverable.

Acknowledgment

The author is grateful for the help and guidance given by her supervisor Dr. Dario Cazzato during the course of this project. She also would like to thank the University of Luxembourg and the director of the Bachelor in Computer Sciences program, Dr Nicolas Guelfi, and its team, for the project, guidance, a reference document[21] and a template[22]. She would like to thanks her family and friends for their support.

5.5. Conclusion

The bachelor semester project has a scientific and a technical part. The scientific part gave a deep understanding of the concepts and steps needed to visualize data on a computer. From, the nature of the information itself, that is radiation on sensors, to how this information is made into computational data using the Operating System, Python, C++ and ROS. Then, understanding how to manipulate the data to be able to see the information that is captured(Image, PointCloud) but also the one that is deduced from this information(estimation of Path).

The technical part aimed to create an node that contains frame rate control. Implementing the node was successful. The implementation of an edge detection node made the student experiment and learn about image processing pipelines, filters and computer vision in general.

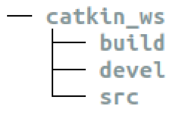
In conclusion, the project was successful since a lot of scientific and technical knowledge was acquired in the domain of computer vision, robot operating systems and image processing.

References

[1] P. K. Allen, *Robotic Object Recognition Using Vision and Touch*, ser. The Kluwer International Series in Engineering and Computer Science 34. Springer US, 1987.
[2] C. Tan, J. Z. Leibo, T. Poggio, R. Cipolla, S. Battiato, and G. M. Farinella, *Machine learning for computer vision*, ser. Studies in Computational Intelligence 411. Springer-Verlag Berlin Heidelberg, 2013.
[3] J. E. Solem, *Programming Computer Vision with Python*. O'Reilly Media, june 2012.
[4] K. Kar, *Mastering Computer Vision with TensorFlow 2.x: Build advanced computer vision applications using machine learning and deep learning techniques*. Packt Publishing, 2020.
[5] J. Howse and J. Minichino, *Learning OpenCV 4 Computer Vision with Python 3: Get to grips with tools, techniques, and algorithms for computer vision and machine learning*. Packt Publishing, 2020.
[6] A. Distante and C. Distante, *Handbook of Image Processing and Computer Vision*. Springer, 2020, vol. Volume 2: From Image to Pattern.
[7] S. Liang and J. Wang, *Advanced Remote Sensing: Terrestrial Information Extraction and Applications*. Academic Press, 2019.
[8] A. Mahtani, L. Sanchez, E. Fernandez, A. Martinez, and L. Joseph, *ROS Programming: Building Powerful Robots*. Packt, March 2018.
[9] A. Distante and C. Distante, *Handbook of Image Processing and Computer Vision*. Springer, 2020, vol. Volume 1: From Energy to Image.
[10] M. Note, *Managing Image Collections. A Practical Guide*, ser. Chandos Information Professional Series. Chandos Publishing, 2011.
[11] T. Vipin, *Understanding digital image processing*. Taylor and Francis Group, 2018.
[12] *ROS Tutorials*. [Online]. Available: http://wiki.ros.org/ROS/Tutorials
[13] C. R. Inc, "Husky unmanned ground vehicle," 2020. [Online]. Available: https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/
[14] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "The kitti vision benchmark suite," 2020. [Online]. Available: http://www.cvlibs.net/datasets/kitti/index.php
[15] R. F. Schmidt, *Software Engineering: Architecture-driven Software Development*. Morgan Kaufmann, 2013.
[16] B. Pitzer, "usb_cam," 2019. [Online]. Available: http://wiki.ros.org/usb_cam
[17] P. Mihelich and J. Bowman, "vision_opencv," 2020. [Online]. Available: http://wiki.ros.org/vision_opencv
[18] OpenCV, "Open source computer vision library," 2020. [Online]. Available: https://opencv.org/about/
[19] ——, "Opencv modules documentation," 2020. [Online]. Available: https://docs.opencv.org/3.4/modules.html
[20] A. Rosebrock, "Imutils," 2015. [Online]. Available: https://github.com/jrosebr1/imutils/
[21] N. Guelfi, "Bics semester projects reference document," University of Luxembourg, Tech. Rep., 2018.
[22] ——, "Bics bachelor semester project report template," University of Luxembourg, Tech. Rep., 2017. [Online]. Available: https://github.com/nicolasguelfi/lu.uni.course.bics.global

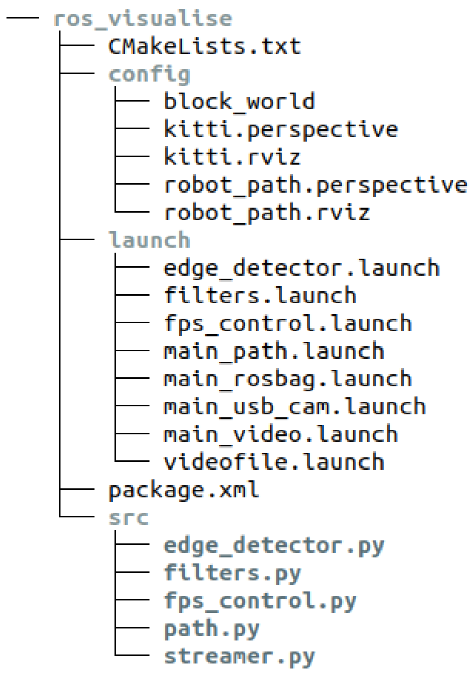
6. Appendix

Fig. 1. ROS package file system



```
─ catkin_ws
  ├── build
  ├── devel
  └── src
```

(a) catkin workspace



```
─ ros_visualise
  ├── CMakeLists.txt
  ├── config
  │   ├── block_world
  │   ├── kitti.perspective
  │   ├── kitti.rviz
  │   ├── robot_path.perspective
  │   └── robot_path.rviz
  ├── launch
  │   ├── edge_detector.launch
  │   ├── filters.launch
  │   ├── fps_control.launch
  │   ├── main_path.launch
  │   ├── main_rosbag.launch
  │   ├── main_usb_cam.launch
  │   ├── main_video.launch
  │   └── videofile.launch
  ├── package.xml
  └── src
      ├── edge_detector.py
      ├── filters.py
      ├── fps_control.py
      ├── path.py
      └── streamer.py
```

(b) ros visualize

Fig. 2. install package

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ git clone https://github.com/BarbaraMMCS/UAV_navigation_ROS.git
$ cd ~/catkin_ws/
$ catkin_make
$ source devel/setup.bash
```

Fig. 3. package.xml

```
cmake_minimum_required(VERSION 2.8.3)
project(ros_visualise)

find_package(catkin REQUIRED COMPONENTS
  cv_bridge
  rospy
  sensor_msgs
  std_msgs)

find_package(OpenCV REQUIRED)

include_directories(${catkin_INCLUDE_DIRS})
```

Fig. 4. CMakeLists.txt

```
<?xml version="1.0"?>

<package format="2">

  <name>ros_visualise</name>
  <version>0.0.1</version>
  <description>Visualise image and video with ros</description>
  <maintainer email="barbara.symeon.001@uni.lu">Barbara Symeon</maintainer>
  <license>...</license>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>cv_bridge</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>sensor_msgs</build_depend>
  <build_depend>std_msgs</build_depend>

  <build_export_depend>cv_bridge</build_export_depend>
  <build_export_depend>rospy</build_export_depend>
  <build_export_depend>sensor_msgs</build_export_depend>
  <build_export_depend>std_msgs</build_export_depend>
  <build_export_depend>message_runtime</build_export_depend>

  <exec_depend>cv_bridge</exec_depend>
  <exec_depend>rospy</exec_depend>
  <exec_depend>sensor_msgs</exec_depend>
  <exec_depend>std_msgs</exec_depend>
  <exec_depend>message_runtime</exec_depend>

</package>
```

Fig. 5. Simple node

```
#!/usr/bin/env python
```

```
import rospy
from std_msgs.msg import String
```

```
pub = rospy.Publisher('publisher', String, queue_size=10)
rospy.init_node('publisher', anonymous=True)
```

```
rate = rospy.Rate(10)
```

```
while not rospy.is_shutdown():
      hello_str = "Hello world!"
      pub.publish(hello_str)
      rate.sleep()
```

```
rospy.init_node('subscriber', anonymous=True)
rospy.Subscriber("subscriber", String, callback)
rospy.spin()
```

Fig. 6. filters.py trackbar and threshold

```
import rospy
import cv2
import imutils
from cv_bridge import CvBridge, CvBridgeError
from sensor_msgs.msg import Image
```

```
class Filter:
```

```
   def __init__(self):
      self.bridge = CvBridge()
```

```
      # msgs conversion
   def to_cv2(self, data, encoding="passthrough"):
      try:
         data = self.bridge.imgmsg_to_cv2(data, desired_encoding=encoding)
      except CvBridgeError as e:
         print(e)
      return data

   def to_imgmsg(self, data, encoding="passthrough"):
      try:
         data = self.bridge.cv2_to_imgmsg(data, encoding=encoding)
      except CvBridgeError as e:
         print(e)
      return data
```

```
      # filters
   def filters(self, data):
      value_th = cv2.getTrackbarPos("Threshold", self.window)
      _, data = cv2.threshold(data, value_th, 255, cv2.THRESH_TRUNC)
      switch = cv2.getTrackbarPos("Gray", self.window)
      if switch == 0:
         pass
      else:
         data = cv2.cvtColor(data, cv2.COLOR_BGR2GRAY)
      return data
```

Fig. 7. streamer.py for FPS algorithm using an existing library

```
from imutils.video import WebcamVideoStream
from imutils.video import FPS
```

```
class Video:

   def __init__(self):
      self.num_frames = rospy.get_param('~num_frames')
      self.display = rospy.get_param('~display')
      self.video_path = rospy.get_param('~video_path')
      self.bridge = CvBridge()
      self.vs = WebcamVideoStream(src=self.video_path).start()
      self.fps = FPS().start()

   def clean_shutdown(self):
      cv2.destroyAllWindows()
      self.vs.stop()
```

```
   def stream(self):
      while self.fps._numFrames < self.num_frames:
         frame = self.vs.read()
         if self.display > 0:
            self.show(frame)
         self.publisher(frame)
      self.fps.update()
      self.fps.stop()
      cv2.destroyAllWindows()
      self.vs.stop()
```

Fig. 8. fps-control.py for FPS algorithm and callback function

```
class Fps:

   def __init__(self):
      self.bridge = CvBridge()
      self.window = "fps_control"
      self.fps_param = rospy.get_param('~fps')
      cv2.namedWindow(self.window)
      cv2.createTrackbar("FPS", self.window, self.fps_param, 30, self.nothing)
```

```
   def update(self):
      fps = float(cv2.getTrackbarPos("FPS", self.window))
      if fps == 0:
         fps = 1
      else:
         pass
      return fps
```

```
   def callback(self, data):
      topic_pub = rospy.get_param('~topic_pub')
      pub = rospy.Publisher(topic_pub, Image, queue_size=1)

      fps = self.update()
      frame_time = rospy.Duration(1/fps)
      begin_time = rospy.Time.now()
      end_time = frame_time + begin_time
      while rospy.Time.now() < end_time:
         pub.publish(data)
         self.show(data)
         rospy.sleep(frame_time)
```

```
   def subscriber(self):
      topic = rospy.get_param('~topic_sub')
      rospy.Subscriber(topic, Image, self.callback)
```

Fig. 9. edge-detector.py algorithm and trackbars

```
class Edge:

   def __init__(self):
      self.bridge = CvBridge()
      self.min_val = 200
      self.max_val = 300
      self.aperture_size = 3
      self.gray = None
      self.window = "edge_detector"
      cv2.namedWindow(self.window)
      cv2.createTrackbar('Min', self.window, 200, 800, self.min_change)
      cv2.createTrackbar('Max', self.window, 300, 800, self.max_change)

   def change_params(self, name, value):
      edge_params = value
      print(self.edge_params)
      self.redraw_edges()

   def redraw_edges(self):
      edges = cv2.Canny(self.gray, self.min_val, self.max_val, self.aperture_size)
      self.show(edges)

   def min_change(self, new_val):
      self.min_val = new_val

   def max_change(self, new_val):
      self.max_val = new_val

   def edge(self, data):
      self.gray = data
      self.redraw_edges()
      return data
```

Fig. 10. Draw path from Odometry message

```python
#!/usr/bin/env python
import rospy

from nav_msgs.msg import Path
from nav_msgs.msg import Odometry
from geometry_msgs.msg import PoseStamped

path = Path()

def odom_cb(data):
   global path
   path.header = data.header
   pose = PoseStamped()
   pose.header = data.header
   pose.pose = data.pose.pose
   path.poses.append(pose)
   path_pub.publish(path)

rospy.init_node('path')

topic_sub = rospy.get_param('~topic_sub')
topic_pub = rospy.get_param('~topic_pub')
odom_sub = rospy.Subscriber(topic_sub, Odometry, odom_cb)
path_pub = rospy.Publisher(topic_pub, Path, queue_size=1)

if __name__ == '__main__':
   rospy.spin()
```

Fig. 11. launch files

```xml
<launch>

<!-- Navigation launch

  <include file="$(find husky_navigation)/launch/amcl_demo.launch"/>
  <include file="$(find husky_navigation)/launch/exploration_demo.launch"/>

  <include file="$(find husky_gazebo)/launch/husky_playpen.launch"/>

 -->

  <include file="$(find husky_navigation)/launch/gmapping_demo.launch"/>
  <include file="$(find husky_gazebo)/launch/husky_empty_world.launch"/>

  <node name="path" pkg="ros_visualise" type="path.py" output="screen" >
    <param name="topic_sub" value="odometry/filtered" />
    <param name="topic_pub" value="/path" />
  </node>

  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
      ros_visualise)/config/robot_path.rviz" output="screen"/>

  <node name = "rqt_gui" pkg = "rqt_gui" type = "rqt_gui" respawn = "false" output = "screen"
      args = "--perspective-file $(find ros_visualise)/config/robot_path.perspective"/>


</launch>
```

Fig. 12. Digital image with Python, Numpy and OpenCV

```
import numpy as np
import cv2

# Numpy array with one channel
img = np.zeros((3, 3), dtype=np.uint8)
```

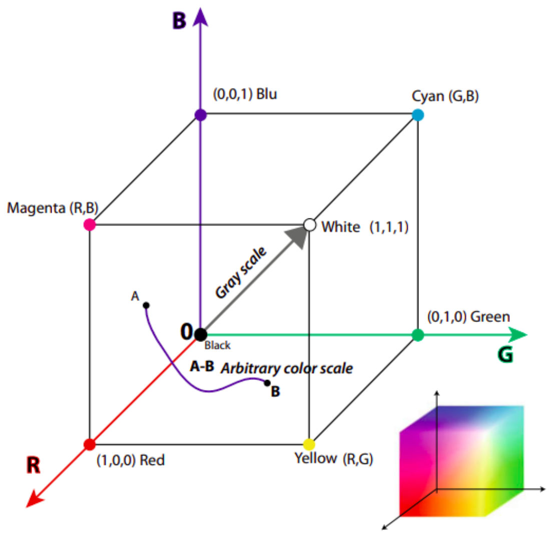
```
# Black square 3x3
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]], dtype=uint8)
```

```
# Conversion one channel to three channels
img = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
```

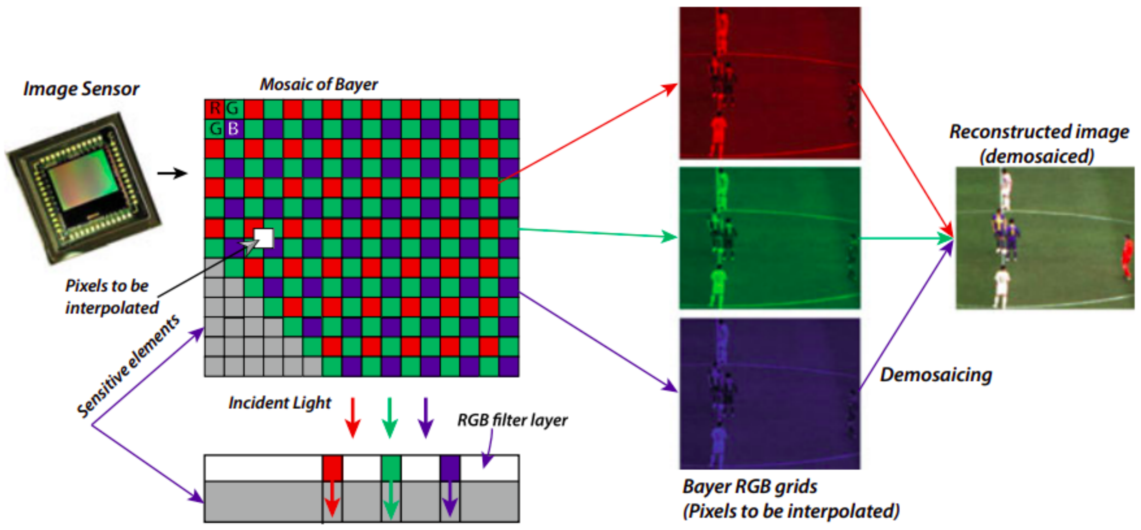
```
# Multidimensional image for B, G and R channels
array([[[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]],

        [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]],

        [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]]], dtype=uint8)
```

Fig. 13. digital image conversion encoding with CvBridge

```
8-bit grayscale image, mono8: CV_8UC1

16-bit grayscale image, mono16: CV_16UC1

8-bit color image with blue-green-red color order, bgr8: CV_8UC3

8-bit color image with red-green-blue color order, rgb8: CV_8UC3
```

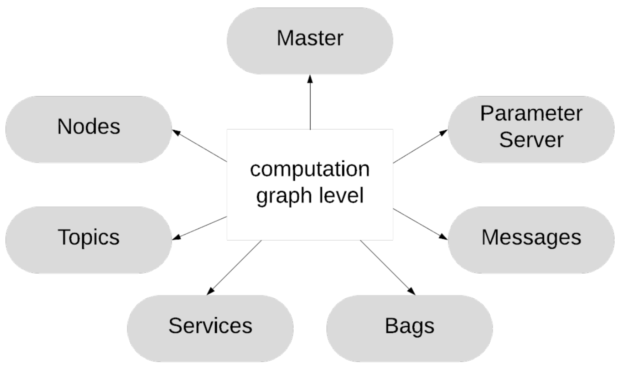


(a) RGB 3D color space. Taken from [9]

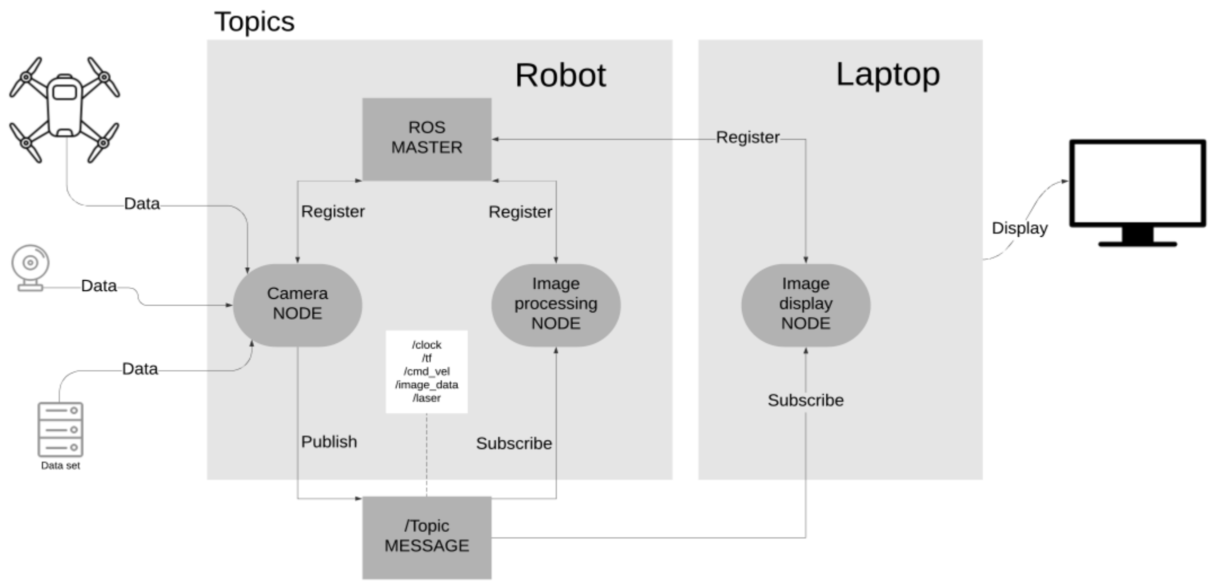


(b) Digital image processing. Taken from [9]

Fig. 14. Digital Image

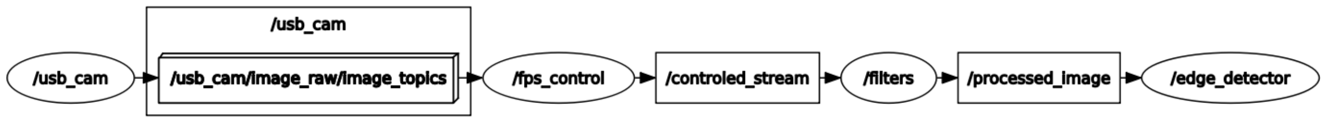


(a) ROS Computation graph

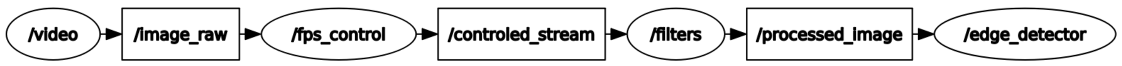


(b) Nodes and topics

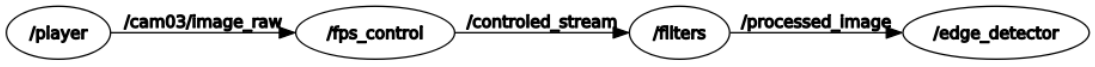
Fig. 15. Robot Operating System



(a) Graph system for main_usb_cam.launch

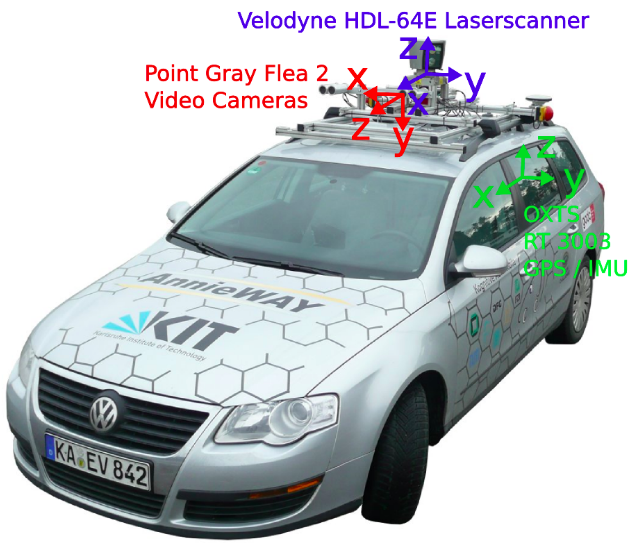


(b) Graph system for main_video.launch

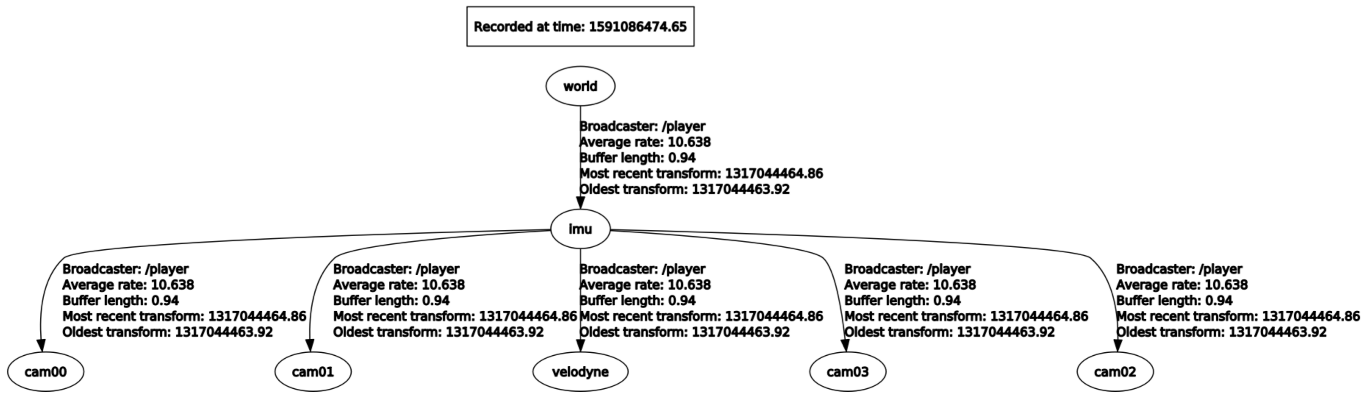


(c) Graph system for main_rosbag.launch

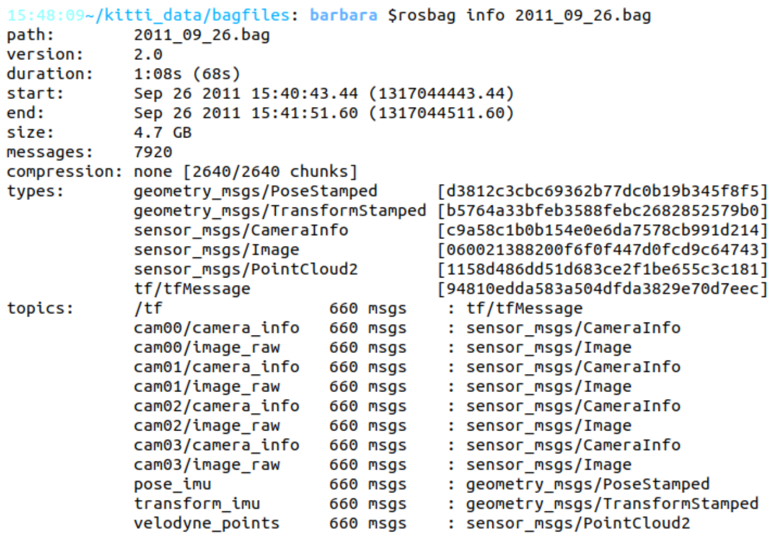
Fig. 16. Graph system of Image nodes



(a) Autonomous driving platform Annieway. Taken from [14]


Recorded at time: 1591086474.65
world
Broadcaster: /player
Average rate: 10.638
Buffer length: 0.94
Most recent transform: 1317044464.86
Oldest transform: 1317044463.92
imu
cam00
cam01
velodyne
cam03
cam02

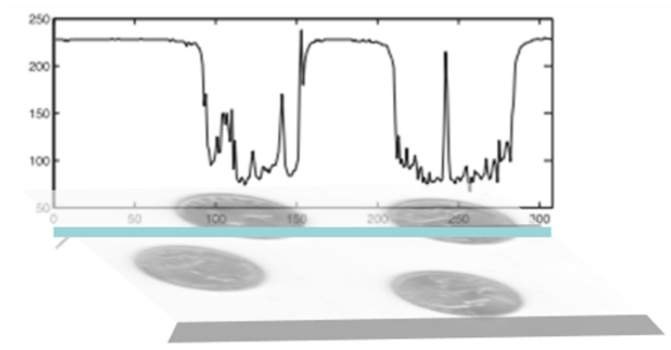
(b) KITTI rosbag transform frame tree



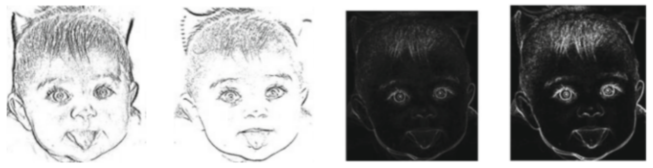
```
15:48:09~/kitti_data/bagfiles: barbara $rosbag info 2011_09_26.bag
path:        2011_09_26.bag
version:     2.0
duration:    1:08s (68s)
start:       Sep 26 2011 15:40:43.44 (1317044443.44)
end:         Sep 26 2011 15:41:51.60 (1317044511.60)
size:        4.7 GB
messages:    7920
compression: none [2640/2640 chunks]
types:       geometry_msgs/PoseStamped      [d3812c3cbc69362b77dc0b19b345f8f5]
             geometry_msgs/TransformStamped [b5764a33bfeb3588febc2682852579b0]
             sensor_msgs/CameraInfo         [c9a58c1b0b154e0e6da7578cb991d214]
             sensor_msgs/Image              [060021388200f6f0f447d0fcd9c64743]
             sensor_msgs/PointCloud2        [1158d486dd51d683ce2f1be655c3c181]
             tf/tfMessage                   [94810edda583a504dfda3829e70d7eec]
topics:      /tf                 660 msgs    : tf/tfMessage
             cam00/camera_info   660 msgs    : sensor_msgs/CameraInfo
             cam00/image_raw     660 msgs    : sensor_msgs/Image
             cam01/camera_info   660 msgs    : sensor_msgs/CameraInfo
             cam01/image_raw     660 msgs    : sensor_msgs/Image
             cam02/camera_info   660 msgs    : sensor_msgs/CameraInfo
             cam02/image_raw     660 msgs    : sensor_msgs/Image
             cam03/camera_info   660 msgs    : sensor_msgs/CameraInfo
             cam03/image_raw     660 msgs    : sensor_msgs/Image
             pose_imu            660 msgs    : geometry_msgs/PoseStamped
             transform_imu       660 msgs    : geometry_msgs/TransformStamped
             velodyne_points     660 msgs    : sensor_msgs/PointCloud2
```

(c) KITTI rosbag information

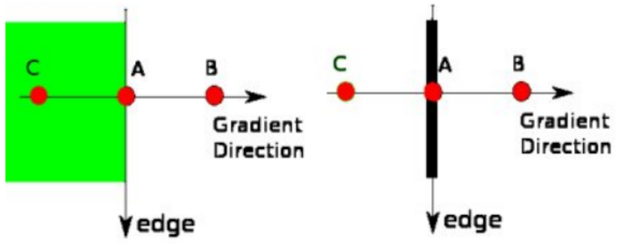
Fig. 17. Kitti dataset



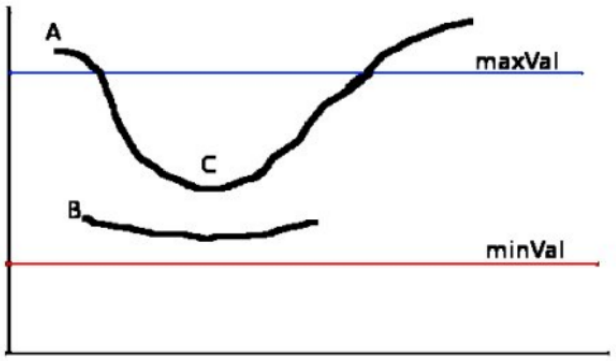
(a) Discontinuities of the edges on the image line. Taken from [6]



(b) Edge detection steps. (a) (b) and (c) are gradients, (d) is edges after adding a threshold filter. Taken from [6]



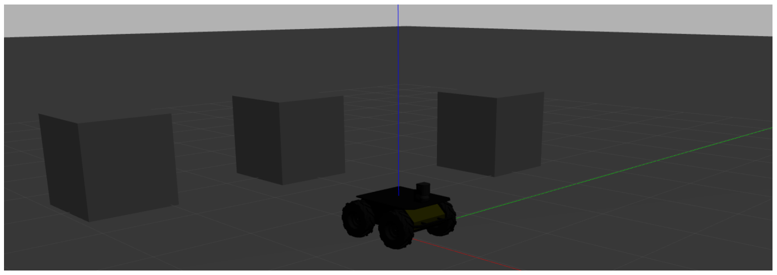
(a) Non-maximum Suppression

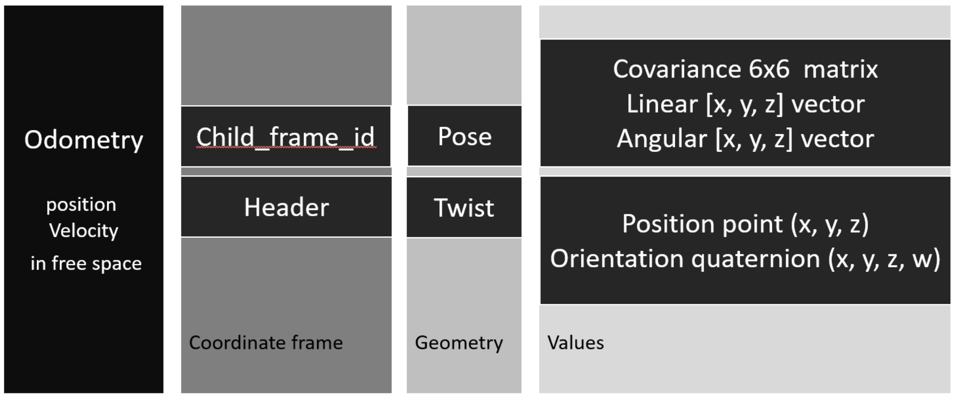


(b) Hysteresis Thresholding

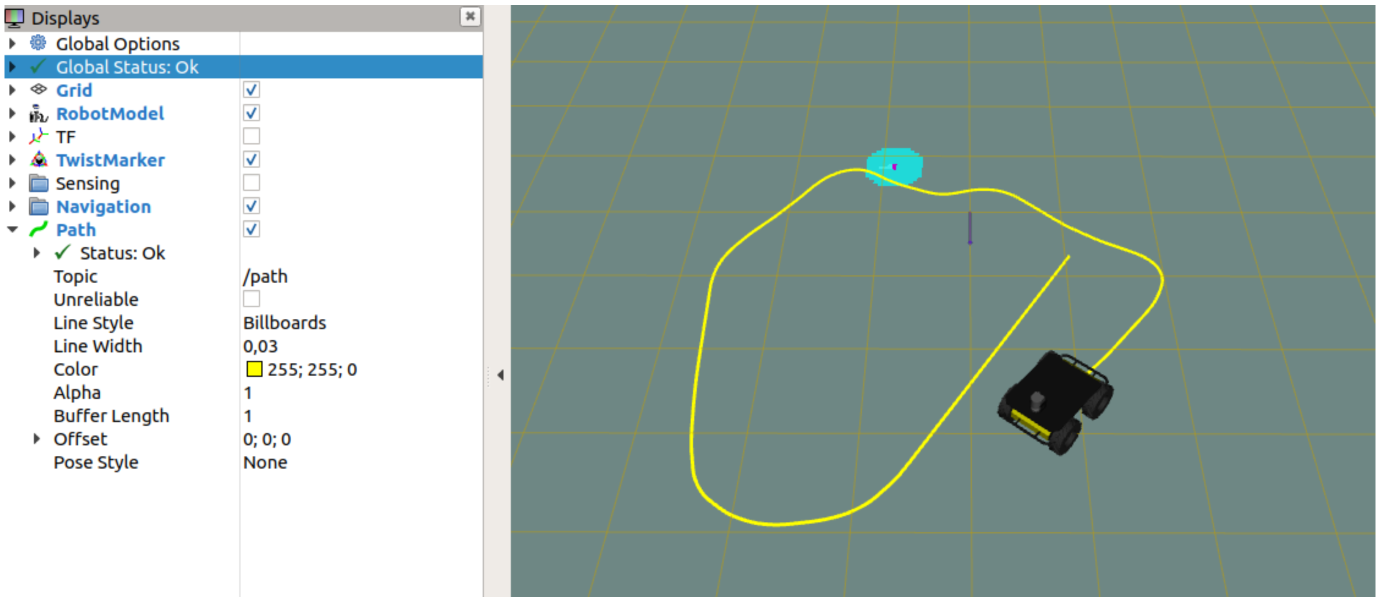
(c) Canny edge detection concepts with OpenCV. Taken from [19]

Fig. 18. Edge detection


(a) HUSKY UGV in Gazebo simulation



(b) Diagram definition of nav_msgs/Odometry Message



Topic	/path
Unreliable	☐
Line Style	Billboards
Line Width	0,03
Color	🟨 255; 255; 0
Alpha	1
Buffer Length	1
Offset	0; 0; 0
Pose Style	None

(c) HUSKY Unmanned Ground Vehicle in Rviz with path

Fig. 19. Path with Husky