UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Technology

Junyi Gu

Towards Faster Masking of Dynamic Objects for Visual Simultaneous Localization and Mapping

Master's Thesis (30 ECTS)
Robotics and Computer Engineering

Supervisor: Amnir Hadachi, PhD
Co-Supervisor: Artjom Lind, Msc

Tartu 2020

Towards Faster Masking of Dynamic Objects for Visual Simultaneous Localization and Mapping

Abstract:
Visual SLAM is a technology able to build a map of an unknown environment and perform localization simultaneously in the partially built map. It uses only visual inputs to perform location and mapping, meaning that the only sensor required is a camera. Visual SLAM is also one of the most challenging type among all SLAM systems, in the case of location information calculation from the images.

There are researches focus on improving assumption of scene rigidity in SLAM algorithm to extend the applicability of Visual SLAM in real-world environments. For example, DynaSLAM [2] provided the capabilities to detect dynamic objects in the images. It relied on a Convolution Neural Network (CNN) to mask the dynamic objects from images and this usually means a heavy computation work.

In this thesis, we proposed a new solution for decreasing the time complexity of performing masking of Dynamic object during visual SLAM process based on ORB-SLAM2 [19] and YOLOv3 network [25]. The outline masking of dynamic objects relied on stereo matching step. We embedded both image processing and ORB-SLAM2 into ROS [30] system, offered an user-friendly interface to handle the input and output for system which will be convenient for other developers or researchers to continue or use our work .

Keywords:
YOLOv3, Stereo Matching, ORB-SLAM2, DynaSLAM, ROS

CERCS: P170 - Computer science, numerical analysis, systems, control

Dünaamiliste objektide kiirema maskeerimise katse visuaalses samaaegse lokaliseerimise ja kaardistamise meetodis

Lühikokkuvõte:

Visual SLAM on üks version SLAM süsteemidest, mis kasutab ainult visuaalset sisendit selleks, et sooritada lokaliseerimise ja kaardistamise operatsioone. Visual SLAMi populaarsus on tõusuteel tänu odavatele ja lihtsasti hooldatavatele sensoritele, mida kasutatakse sisendite kogumiseks. Visual SLAM on samal ajal ka üks keerukamaid SLAM süsteeme, kuna lokaliseerimise operatsioonideks vajalikud arvutused tuleb teha piltide põhjal.

Hektel on käimas mitmeid teadustöid, mille fookuseks on parandada eelduseid piltide jäikused kohta SLAM algoritmides. Need peaks võimaldama laiendada Visual SLAMi kasutatavust päris maailmas. Näiteks DynaSLAM pakub võimekust avastada dünaamilisi objekte piltides, mis on võetud SLAM kaameratega. DynaSLAM tugineb CNNil, et maskeerida liikuvad objektid piltidel. Säärane tegevus on aga väga arvutuste rohke.

Käesolevas töös pakutakse välja uus lahendus dünaamiliste objektide kiiremaks tuvastamiseks, kasutades YOLOv3 ja ORB-SLAM2 võrgustikke. Lisaks sellele proovitakse kasutada stereo sobitamist, et leida ja maskeerida objektide kontuure. Sarnaselt DynaSLAMile kasutab käesolevas töös arendatud süsteem ORB-SLAM2. Arendatud süsteem sai loodud ROSi keskkonnas, mistõttu on olemas kasutajasõbralik kasutajaliides, mis haldab väljatöödatud süsteemi sisendeid ja väljundeid. ROS keskkonda sai implementeeritud pilditöötlus ja ORB-SLAM2. Kasutajasõbralikkus saab olema mugav arendajatele järgmiste uurimistööde läbiviimisel.

Võtmesõnad:

YOLOv3, Stereo vastavus, ORB-SLAM2, DynaSLAM, ROS

CERCS: P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Acknowledgement

First and foremost, I would like to thank my supervisors Dr. Amnir Hadachi and Artjom Lind for providing direction to the correct path and their continuous support.

In addition, I would like to thank Intelligent Transportation System Lab providing an opportunity to conduct my thesis work within a research project in collaboration with industrial partner Milrem Robotics which helped me a lot in improving my technical skills.

At the last, I also would like to thank University of Tartu and Institute of Technology offering scholarship during my study life in Estonia.

Contents

List of Figures

List of Tables

1 Introduction

Simultaneous Localization and Mapping (SLAM) has been considering as one of the most important parts in robotic applications. This is a developed topic which has long research history. Various SLAM systems based on different sensors extended their application scenarios significantly.

In general, we can classify SLAM as their input sensors, Visual SLAM is a branch of it which based on cameras. Nowadays, the SLAM technologies relied on Light Detection and Ranging (LiDAR) sensors turn to a hot topic because it gave much more information than other sensors per time step. However, the cost of the LIDAR sensors limits the reach of its usage and popularization. In some specific situations, Visual SLAM systems have practical advantages with respect to the degree of size and power consumption compared with LiDAR based SLAM systems, this is one of the reasons people still interested and trying to improve Visual SLAM systems.

Computer Vision has undergone a considerable development during the last few years. There are a lot of ways to extract information from the images, neural networks are more popular to process the images at present. This has attracted the attention from research communities and high-technological companies. They proposed many valuable ideas which can improve the performance of Visual SLAM. DynaSLAM is a typical example among them, it implemented Convolution Neural Network (CNN) filtering dynamic objects out of frames to reduce the localization drifting and realize the long-term application of the maps created by SLAM algorithms. Correspondingly, the problem of the Neural Networks based algorithms is obvious, which is the processing speed. DynaSLAM used Mask R-CNN [11] to detect and process the images in the first step and He et al. [11] reported Mask R-CNN took 195ms to process one image on an Nvidia Tesla M40 GPU. The efficiency and hardware cost of it limit the practical applicability and development of this new Visual SLAM technology in many related cases, such as the Unmanned Ground Vehicles and civil used automatic drive which based on intelligent autonomous systems. If the frame rate of the processing system is too much lower than the SLAM cameras, there will be a delay of localization and mapping in populated real-world environments, the accumulated error is unacceptable in any case for precision and safety.

In this work, we proposed an efficient real-time solution to handle the dynamic objects in stereo SLAM. We focused on improving the processing speed of the objects detecting system to achieve a higher frame rate with the same hardware. Similar to DynaSLAM, we used state-of-the-art ORB-SLAM2 system, for a purpose of having accurate tracking and reusable map of the scene.

We added image processing operations to ORB-SLAM2 system as a front-end stage, excluding dynamic objects from the images to make sure ORB-SLAM2 does not extract any features from them. DynaSLAM implemented a CNN to segment *priori* dynamic objects in pixel-wise, this is a time-wise-expensive process which is the main reason

DynaSLAM running at a slow speed. Instead of it, our proposal in this stage is the YOLOv3 system, which was first introduced in 2015 by Redmon *et al.* [25], described an object detector capable of real-time object detection, obtaining 45 FPS on a GPU.

YOLOv3 system detects objects based on the pre-trained models, what system returns are the bounding boxes in images contain the objects. These bounding boxes represented by the pixel-coordinate of left-top corner point and dimension information (width and height of the boxes). For the big objects in images, corresponding boxes usually include too much unrelated background information (see Figure 1), which we want to keep for ORB-SLAM2 to do tracking and mapping. This is not a problem for DynaSLAM because Mask R-CNN is able to mask out the outline of objects directly. We introduced a solution based on stereo disparity estimation for the YOLOv3 system to detect object outline, correspondingly, this solution requires stereo image streams as input. On the other hand, the stereo disparity map also helps for locating the same objects between the stereo images. We only have to process either left or right image once by YOLOv3 network, then we can calculate the position of this object out in another image refer to corresponding disparity value, which means no need to process the image in YOLOv3 network again.

Figure 1. Detection result of YOLOv3 network

Both ORB-SLAM2 and YOLOv3 are individual systems, we organized them together under the ROS system to handle the input and output of data in an efficient way. We provided a complete structure that loads the input image topics to YOLOv3 network, after having dynamic objects removed from the images, passing the processed image topics to ORB-SLAM2 to do tracking and mapping operations, the output will be the Odometry topic in ROS message format. The entire system will be friendly to users and easy for modifications and maintenance.

We demonstrated our proposal on an Nvidia RTX2070 Super graphic card, with the supports of the OpenCV(4.2.0) and Nvidia's cuDNN libraries, we made YOLOv3 network running on graphics processing unit (GPU), the rest of the operations including the ORB-SLAM2 tracking and mapping algorithms still running on an AMD Ryzen9

3900x CPU.

The rest of the thesis is structured as follows: section 2 discusses the background and related work, section 3 describes the methods and structures of the proposed system in details, section 4 gives the analysis and experimental results, and section 5 presents the conclusions and lines of future works.

2 Background and Related Work

Both DynaSLAM and our system are front-end process of input images, the tracking and mapping operations relied on ORB-SLAM2 system. This section will first have a general review of ORB-SLAM2 system, then the DynaSLAM and main algorithm it used to detect the *priori* dynamic objects. There is also an introduction of the methods we used to realize faster objects detection.

2.1 ORB-SLAM2

ORB-SLAM2 is a complete Visual SLAM system which has loop closing, map reusing and relocalization capabilities. It supports monocular, stereo, and RGB-D cameras, compatible with variety of environments, such as closed indoors sequences and self-driving cars in a city. Refer to the KITTI Vision Benchmark Suite [8], it is one of the best open-source Visual SLAM algorithms with 1.15% translational error and 0.0027 deg/m rotational error, the runtime is 0.06s on standard CPUs which means it is able to work in real time.

Although monocular camera is the cheapest and smallest sensor setup in the field of Visual SLAM, there are problems of depth observation with only one camera. In addition, many monocular SLAM algorithms not good at handling scale drift and has poor performance in some specific situation, for example, pure rotation around itself. Stereo and RGB-D camera can solve all these problems, many Visual SLAM algorithms focused on these two kinds of camera for the purpose of a more reliable solution. ORB-SLAM2 combined close and far stereo points and monocular observation, developers claimed their stereo results are more accurate than many state-of-the-art direct stereo methods. The comparison results in KITTI benchmark also labeled ORB-SALM2 system as the stereo setting.

ORB-SLAM2 is built on their monocular featured-based ORB-SLAM [18], therefore, it also works with a monocular input. The main improvement of ORB-SLAM2 is new functions for stereo and RGB-D inputs. In general, the constitution of ORB-SLAM2 can be summarized as three main parallel threads, which are Tracking, Local Mapping and Loop Closing, respectively (see Figure 2). Like other modern stereo SLAM systems [31] [17] [21], ORB-SLAM2 also used Bundle Adjustment (BA) optimization in local keyframes sets to achieve more accuracy and run system in large environments. The Tracking thread applying motion-only BA to minimizing reprojection error when trying to localize the camera with every frame by matching the frame features to the local map. Tracking thread pre-processes the stereo and RGB-D input so that the rest parts of the system work independently of the input sensors. Local BA also performed in thread Local Mapping to manage and optimize the local map. The last thread Loop Closing detects the large loops in environment, it can correct the accumulated drift by applying a pose-graph optimization. This function helps to improve accuracy a lot in some specific

situations, for example, KITTI seq 00. This thread initiates another thread performing full BA after the pose-graph optimization, to optimize the structure and compute motion solution.

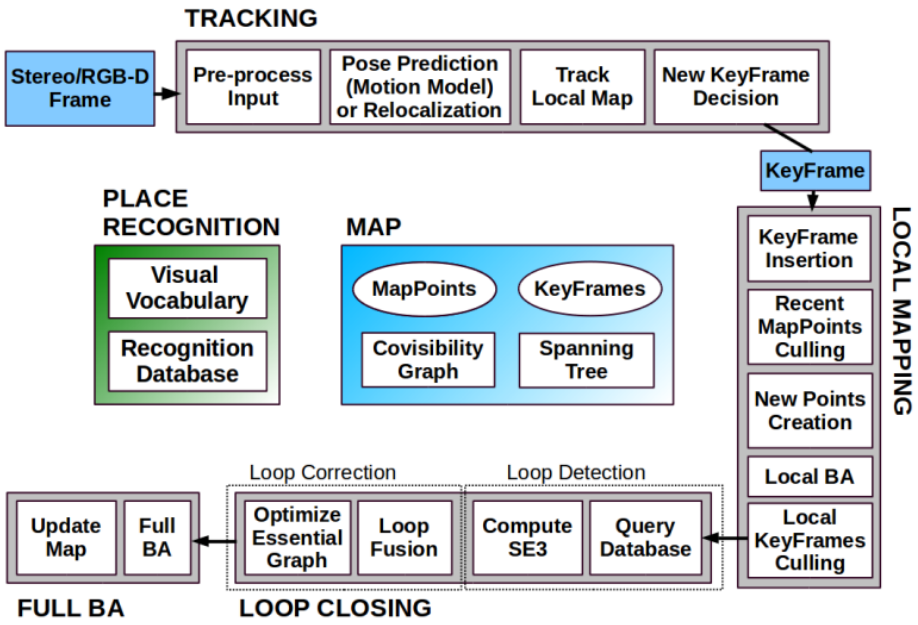


Figure 2. System Threads and Modules of ORB-SLAM2 [19]

ORB-SLAM2 system has a covisibility graph [31] allows Tracking and Local Mapping operations done locally by retrieving local windows of keyframes, therefore ORB-SLAM2 system can work in large scale environment. Pose-graph optimization in the last thread Loop Closing also relied on these graph structures. For the cases like tracking failure or reinitialization in the mapped scene, ORB-SLAM2 system has another relocalization module based on Bags-of-Words model version 2 (DBoW2) [7].

Both ORB-SLAM2 and ORB-SLAM using ORB feature [28] for all tasks (tracking, mapping, relocalization and loop closing). This method provided a good invariance viewpoint (auto-gain, auto-exposure) and illumination changes of the camera, it also fast in features extraction and matching operation which allows real-time performance without any GPUs. These two SLAM systems took account all features from sensors, which has limits in some highly dynamic scenarios, for example, in city urban areas, moving objects such as cars, bikes, walking people will complicate the tracking and relocalization tasks if they appear in the map of SLAM systems. Detecting and occluding dynamic objects is requisite of stable and reusable maps, helpful in long-term applications. Many solutions were proposed for this perspective, DynaSLAM is one of the best among

them. However, DynaSLAM is relatively heavy in computation, limits the use of it in populated real-world environments. We need objects detection is fast enough to match the frame rate of the cameras, which is the issue addressed by our work.

2.2 DynaSLAM

Visual SLAM algorithms highly based on the assumption that the scene is always rigid enough, but in some specific scenarios, this limits the use of the most Visual SLAM systems. Highly dynamic environment is one of the typical situations which is challenging to most of standard Visual SLAM algorithms.

The difficulties for Visual SLAM to detect and handle dynamic objects concentrate on tracking and mapping operations, for example, prevent tracking algorithms using the features or matches from the dynamic objects and exclude the moving objects from the 3D map in mapping algorithms. In some specific environment which has a lot of dynamic objects, the normal 3D maps created by Visual SLAM has no use for any long-term applications because these kinds of maps are not reusable at all. DynaSLAM is a solution designed to solve this problem, which outperforms the accuracy of standard Visual SLAM baselines in highly dynamic scenarios [2]. DynaSLAM is an online algorithm to handle dynamic objects in RGB-D, stereo, monocular scenarios. It relies on ORB-SLAM2 for tracking and mapping operations, realizes the processing of the dynamic objects by adding a front-end stage to ORB-SLAM2 system.

DynaSLAM is not the first SLAM system involved dynamic object detection. There are some other ideas proposed before it. Among feature-based SLAM methods, Tan *et al.* [35] projected features back to the last frame to valid the appearance and structure to detect the changes happened for objects in image. Wangsiripitak and Murray [34] implemented a 3D object tracker to Monocular SLAM to occlude the moving objects. Riazuelo *et al.* [27] developed a system has a reliable performance in human detection and tracking, used RGB-D camera as the only input to the system, computed the map with an unpopulated geometrical layer and semantic human activity layer. For direct methods, many 3-D SLAM algorithms proposed to use RGB-D cameras, Wang and Huang [33] used RGB optical flow to segment the dynamic objects in the image. Some other systems relied on depth images to segment out the dynamic objects [14], [32]. A proposal for the stereo camera is from Alcantarilla *et al.* [1], detected moving objects by means of a dense scene flow representation.

The difference of DynaSLAM to these algorithms is the combination of multi-view geometry and deep learning for all monocular, RGB-D, and stereo cameras. DynaSLAM is able to detect all kinds of dynamic objects:

- *priori* dynamic object is moving, *e.g.*, cars moving on the street.
- *priori* dynamic object stays static, *e.g.*, cars parked by the road.

- Static objects moved by force, *e.g.*, a ball threw by someone.

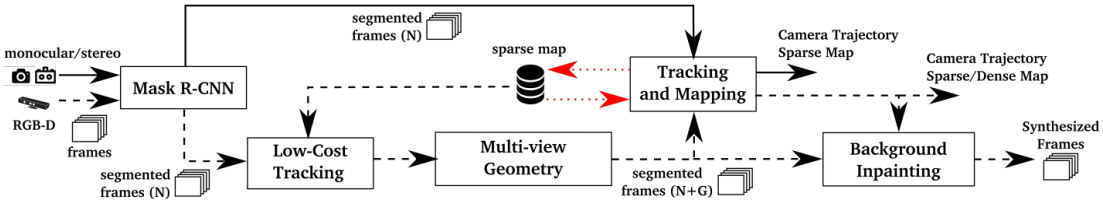


Figure 3. Block Diagram of DynaSLAM [2]

The overview of DynaSLAM system showed in Figure 3. At the very first, all kinds of image channels pass through a Mask R-CNN network to segment out objects labeled as classes within the network in pixel-wise. In RGB-D case, an additional multi-view geometry approach was implemented before tracking and mapping for more accurate motion segmentation. At the last, inpainting the background occludes the dynamic objects for a reusable map. After knowing the position of dynamic objects, comparing the current frame with a set of previous keyframes can help to occlude the dynamic objects and fill the gaps with the static structure in the same position. They left blank for the areas have no correspondences (see Figure 4).

(a) Original RGB image (b) Impainted RGB image

Figure 4. DynaSLAM RGB Images Impainting [2]

The main problem of DynaSLAM is computational time. The developers of DynaSLAM claimed that it is not optimized for real-time operation and works better on offline mode. The most computation-wise-expensive process is Mask R-CNN and multi-view geometry is another slowdown to entire system. This limits the usage of DynaSLAM in some real-world populated situations such as automatic driving vehicles

which require tracking and mapping result feedback in real time. Our work in this thesis is substituting the heavy computation Mask R-CNN with other light-weight networks to have a faster working speed of the whole pipeline. The neural network introduced in our work is YOLOv3, an extremely fast approach to object detection.

2.3 Mask-RCNN

Mask R-CNN is an effective and general framework for object instance segmentation. The method is an extension of Faster R-CNN [26] which able to segment a mask of each instance from objects in parallel with bounding box recognition.

The essence of the Mask R-CNN is the instance segmentation part. This part can be divided into two sub-tasks: 1) Detect and locate the objects. 2) Segment the detected objects pixel-wise. Instance segmentation is a challenging task because it requires accurate detection and precise segmentation happened at the same time. Mask R-CNN is a simple and flexible system achieved a good result in this perspective.

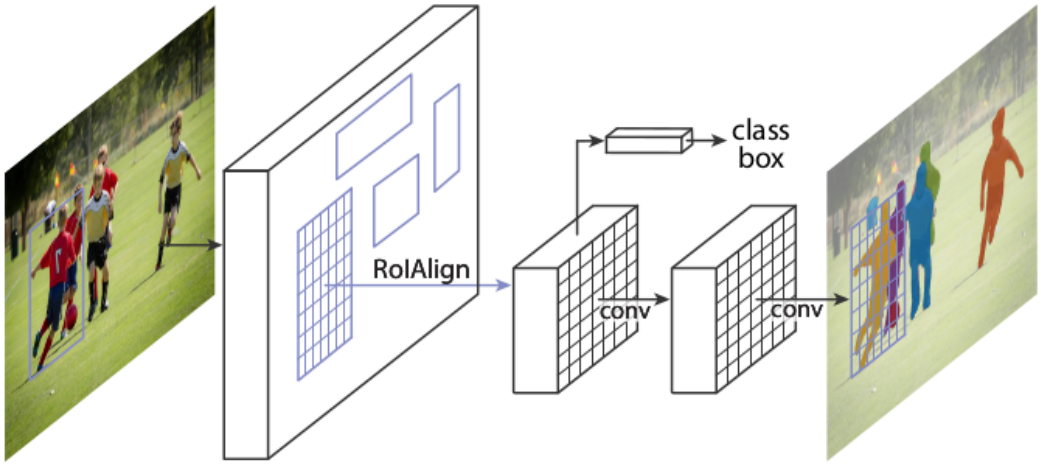


Figure 5. The Mask R-CNN framework for instance segmentation [11]

The general architecture of Mask R-CNN illustrated In Figure 5. Mask R-CNN shared the same algorithm with Faster R-CNN to extract the bounding boxes of objects, which also referred to Region of Interest (ROI). The fact of the Faster R-CNN is it was not constructed for pixel-to-pixel alignment between inputs and outputs, which will result coarse spatial quantization for feature extraction. Mask R-CNN solved this problem by implementing an algorithm ROIAlign [11], a quantization-free layer on top of Faster R-CNN. The subsequent operation after getting precise spatial locations of objects is semantic segmentation, applying a small Fully Convolutional Network (FCN) [16] to

each ROI to classify every pixel into a specific set of classes to get the segmentation mask. Unlike the usual FCNs perform per-pixel multi-class categorization, Mask R-CNN separated the class and mask prediction independent to each other to have a better performance for instance segmentation.

Broadly speaking, Mask R-CNN is a simple and general system surpasses a lot of state-of-art models for object instance segmentation. Its open-source characteristic facilitates implementation and embedding across various platforms such as OpenCV, CUDA, etc. This helps developers easy to test it in any situation and populate it for future research in other fields, like DynaSLAM. However, the relatively slow running speed limit the usage of this powerful method into any populated real-world environments require real-time processing.

2.4 YOLO

YOLO [24] is a significant work in the field of deep learning-based object detection. Compared with other algorithms such as DPM [6] and R-CNN, it runs fast with a tiny trade-off in performance. Instead of two detection stages which first generate bounding boxes in the image then run the classifier on boxes, YOLO organized the object detection into a single regression stage, extract the bounding boxes and information of classes from the pixels straightaway. There are no post-processes after classification in YOLO like duplicated object elimination, bounding boxes refining, etc. Single-stage detection and fewer complex processing make YOLO model processes images in real-time at 45 FPS, another light version Fast YOLO, able to process at 155 FPS [24].

There are three processes for images to detect the objects in YOLO. First, resize the image to 448x448. Then run a single convolutional network simultaneously locates bounding boxes, predicts their corresponding labels and probabilities. At the last, apply non-max suppression filtering detection results by predefined threshold and confidence. The advantages of this simple pipeline are:

- Extreme fast running speed, able to process real-time streams in most of the situations.
- Taking the whole image into account when predicting boundaries, make fewer background errors compared with ROI populated [11], [16] and sliding window [6] methods.
- More generalized when applying natural images to other domains like artwork.

2.4.1 YOLOv2

YOLO was first introduced in 2015 by Redmon *et al.* [24]. In 2017, they published the second version, called YOLOv2 [23] which aimed to improve the accuracy while making

the detection faster. Next, I will list the improved properties of YOLOv2 in the way the author stated in their paper:

- **Batch Normalization** [12]. With the help of BA, YOLOv2 removed the dropout from the model and pushed 2% up of the mAP.
- **High Resolution Classifier**. Original YOLO network trained classifier with 224x224 pictures, then increase the resolution to 448x448 for detection. YOLOv2 retuned classifier by resolution 448x448 with fewer epochs after trained it with 224x224 images. Higher resolution classification network increase mAP for 4%.
- **Convolutional with Anchor Boxes**. YOLOv2 removed all fully connected layers and use anchor boxes to predict bounding boxes. This increase recall from 81% to 88%, with a tiny drop in mAP from 69.5 to 69.2, respectively.
- **Dimension Clusters**. Using k-means clustering instead of choosing prior by hand, got a similar Intersection over Union (IoU) of 9 anchor boxes model (Fast R-CNN used) with only 5 anchor boxes.
- **Direct Location Prediction**. Stabilize the bounding boxes close to the original grid location in early iterations by adding a logistic activation.
- **Fine-Grained Features**. As the resolution decreasing gradually in convolution layers, it is getting harder to detect small objects. YOLOv2 mapped 26x26x512 feature maps from the earlier layer into 13x13x2048, then concatenated with the original 13x13x1024 featured maps for small objects detection. mAP was increased by 1% in this way.
- **Multi-Scale Training**. Instead of fixed resolution 448x448 image resizing as YOLO, YOLOv2 adjusted the input image size on the fly. New image dimensions are randomly choosing after every 10 batches. This makes YOLOv2 work more robust on images of varied sizes.

2.4.2 YOLOv3

YOLOv3 was introduced in 2018 as a tech report, which is a rare format for academic papers. Compared with the last version, YOLOv3 traded off the speed to boost the accuracy, especially the detection for small objects.

YOLOv3 used logistic regression to predicts an objectness score for each bounding box, which represents the probability of this area refer to the ground truth objects. This operation happened before the prediction, helps to ignore the anchor boxes (named as *prior* in the original paper). YOLOv3 also used k-means clustering to generate 9 anchor boxes in total, but different from Fast R-CNN [9], it only processes one anchor box

which has the highest score. This is one of the reasons that YOLOv3 has a faster running speed.

YOLOv2 used a customized architecture Darknet-19, a network has 19 original layers and additional 11 layers for objection detection. This is a relatively fast network but missing some important elements like residual blocks, shortcut connections, and up-sampling, it also struggles in small object detection. YOLOv3 introduced a new hybrid network called Darknet-53 to solve these problems. As its name, the new network has 53 layers trained on ImageNet and another 53 layers for object detection, 106 convolutional layers in total. This improved the capability of small object detection but will slow down the processing speed.

The most obvious character for YOLOv3 is predicting boxes in three different scales features maps, the detection was done by applying 1x1 convolutional kernels in these three feature maps. In this perspective, it is different from the YOLOv2 to use passthrough structure.

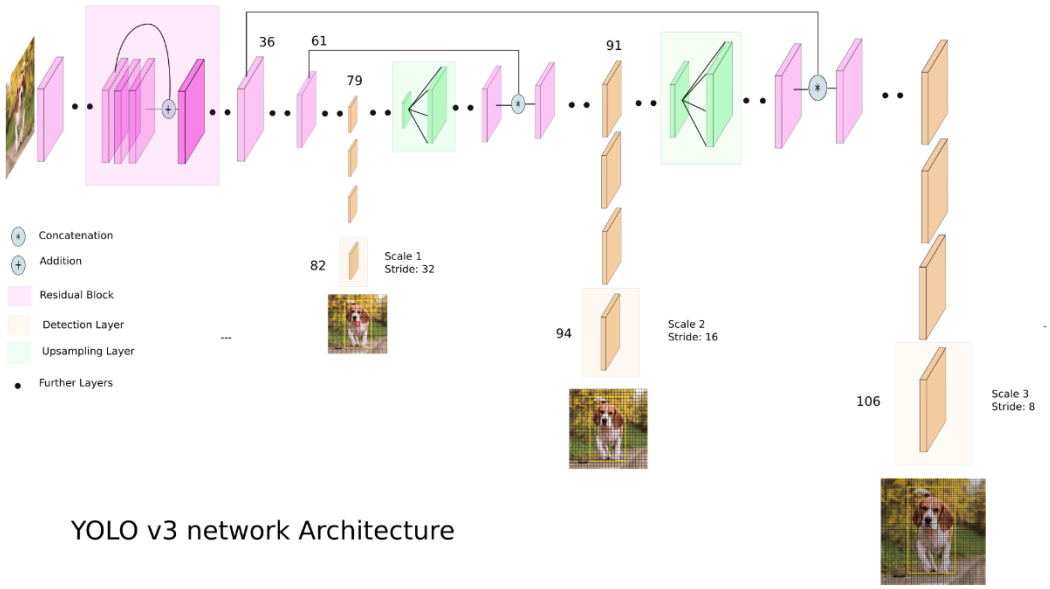


Figure 6. YOLOv3 Network Architecture [13]

Refer to Figure 6 [13]. We can see YOLOv3 processed several more convolutional layers after layer 79 to get the first detection. Compared with the input image, the stride of the feature map here is 32. More specifically, if the dimension of the input image is 416x416(YOLOv3 usually normalizes the image at the first), the feature map turns to 13x13 here. Because of the high stride value, the feature map in this stage has a relatively big receptive field, good at big object detection.

To be able to detect smaller objects, YOLOv3 up-sampled the feature maps by 2x

since layer 79, then concatenate it with the layer 61 to get layer 91 with more finer-grained information. The stride of the feature map in layer 94 turns to 16 refer to the input image.

The last detection repeats the up-sampling process like the last stage but concatenates with layer 36. The stride here is 8 which is sensitive enough to detect even smaller objects.

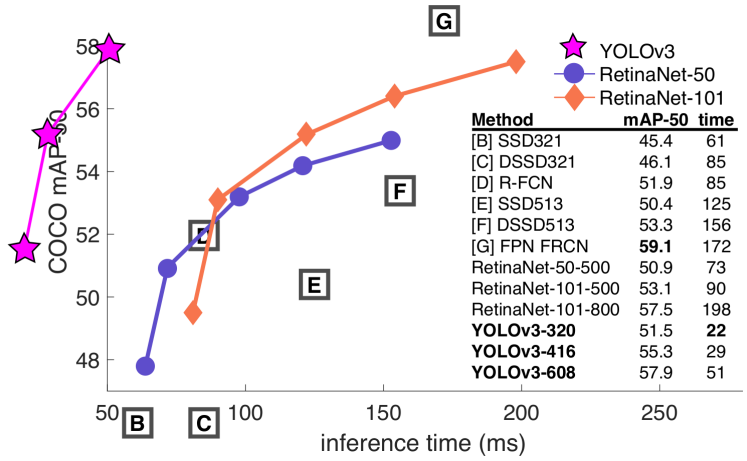


Figure 7. COCO mAP-50 benchmark [25]

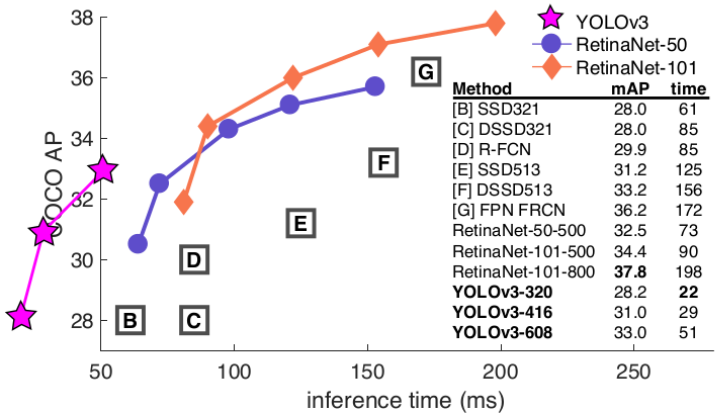


Figure 8. COCO AP benchmark [25]

In summary, YOLOv3 has a deeper network with the help of residual structure, in addition to multi-scales across prediction, it improved the capability of small object

detection and increased mAP significantly. According to the statement from authors, if using COCO [15] mAP50 as the benchmark, the speed of the YOLOv3 is almost 4 times faster than other models under the same mAP, see Figure 7. However, if using a more precise benchmark COCO AP, YOLOv3 has a small drop of mAP compared with other models, see Figure 8.

2.5 Stereo Matching

Figuring out the distance to the objects is easy to some reflectivity-based sensors like LIDAR and sonar, but a very challenging task to the cameras. Human beings are relatively perceptible to the depth with our eyes, which can be regarded as an elegant binocular vision system. Following this principle, stereo matching, also called disparity mapping is an important sub-class of a wide range of algorithms in the field of computer vision to find the depth. Compared with other methods, stereo matching has undeniable advantages in simplicity and robustness because it only needs two rectified images.

The stereo matching is aimed to extract the depth information from the stereo camera, namely, horizontal placed two cameras. Depth information usually stored and visualized by disparity map, represent corresponding points that are shifted between the left image and right image horizontally. There are two main practical meanings out of disparity value, first is computing the point position in any of stereo rectified image by giving its position in another image. This is the usage of disparity computation in our work, we only need to do objects detection and mask one time and shift the mask to another image by corresponding disparity value, which reduces duplicated heavy calculation and make the pipeline works more efficient in real-time situations. Another usage of disparity value is calculating the exact distance between objects and camera. Figure 9 is an illustration which explains this intuition. X is a 3D scene point, O and O' are the center of stereo cameras.

The equivalent equation of diagram in Figure 9 can be written as:

$$disparity = x - x' = B * f/Z \tag{1}$$

x and x' are the distance between 3D scene projected points in image plane corresponding to their camera center. B is the baseline (horizontal distance) between two cameras and f is the focal length. From the equation, we can see the depth of a point in 3D scene is inversely proportional to the distance between their corresponding projected points in the image plane. Since B and f are always known, we can derive the depth of all pixels in an image if have their corresponding disparity value.

From a mathematical perspective, there are two main classes of stereo matching algorithms: local methods and global methods. Local algorithms usually are correlation-based methods that use only a small number of pixels surrounding the interested pixel. Global algorithms are much more complicated, usually implement constraints into a

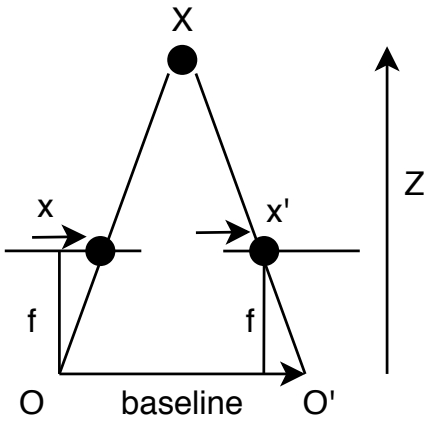


Figure 9. Relationship between Depth and Disparity

bigger area or the whole image. Compared with the local algorithms, global algorithms are more complex and expensive in computation, correspondingly, perform better to reduce sensitivity to regions that fail to match. Since our work is a real-time application, we cannot afford the heavy calculation of the global algorithms. Hence, we used local methods Stereo BM which built in the OpenCV library, it is an intensity-based local algorithm rely on Block Matching algorithm. There is also another Stereo SGBM(semi-global block-matching) algorithm available in OpenCV, but it fails to compete with Stereo BM in term of speed, which conflicts with our original intention toward this work, but we still have a result comparison between them in Chapter 4.

2.6 Robot Operating System (ROS)

Robot Operating System (ROS) is a flexible framework for developers writing robot software across a variety of robotics platforms, sensors, and systems. It is not an actual operating system but a meta-operating system provides a communication layer running on top of operating systems of host computers [22]. It also provides packages management, messages passing, and other services across multiple computers. Because of open-source characteristic, ROS has a tremendous of libraries and tools covered almost all aspects in the field of robotics.

SLAM algorithms solved tracking and localization issues for the robot, but in general perspective, there is need a tool to connect the hardware and software. For Visual SLAM algorithms, data transferring between the cameras and algorithms will decide the efficiency of the entire system. There are also post-processes for the location and map information computed out by SLAM algorithms. ROS is a powerful tool organizing all these modules together and transferring data between them fast and efficient.

One of the original intentions of our work is building a user-friendly general Visual SLAM system. We organized all our modules under the ROS structure, which the authors of ORB-SLAM2 and DynaSLAM do not provide. In our pipeline, input images and output odometry described as ROS topics, essential processes managed as ROS nodes, all under clear architecture, and have unified format. We hope this can offer some simplicity and convenience to other developers who interested in our work.

3 Methodology

In this section, we will present design and architecture of the entire system. At first, we will summarize the general structure of the pipeline, give an overview of how our system works, what input it takes and what output it produces. The second part is a detailed description of processes to input stereo images, which is the core of our work. The last part is about the configuration to ORB-SLAM2, which we used for tracking and mapping operations.

3.1 General Structure

The aim of our work is to build a real-time Visual SLAM system, similar to DynaSLAM which able to detect and occlude moving objects out of images but running faster and more accurate. Figure 10 shows an overview of our system.

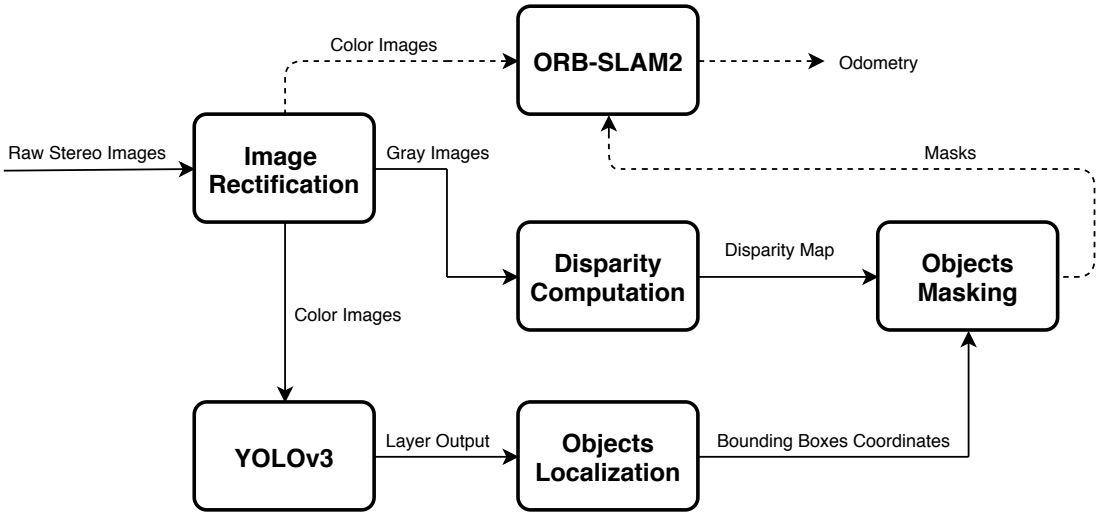


Figure 10. System Block Diagram

In our work, we focused on the stereo case. First of all, the stereo channels pass through a node rectifying the images. This is a necessary and important process, because both *ORB-SLAM2* and *Disparity Computation* require well-rectified images as input, especially the disparity estimation, the rectified images are essential to it to calculate the correct disparity values which we used to transfer the objects masks between the left and right images. The input of the *Disparity Computation* must be single-channel-grayscale images, for *ORB-SLAM2* and *YOLOv3* network, they are able to process either grayscale or colorful images. We assumed the raw stereo stream is colorful images, therefore

there is need to convert the colorful images to single-channel-grayscale images after rectification process, then send them to *Disparity Computation* node.

YOLOv3 network actually predicts 10647 bounding boxes regards to the predefined classes in total, where is the *Layer Output* in the diagram. There is a more detailed explanation of this part in Chapter 3.2.2. To process these amounts of bounding boxes in an efficient and fast way, we implemented a simple node handle them in array-wise, extract the coordinates and dimensions of effective bounding boxes among all of them. The information of effective bounding boxes will be passed to node *Objects Masking*, together with *Disparity Map*, we are able to mask the outline of objects. At the last, we overlaid the masks with the original color images and past them to *ORB-SLAM2* as input to do tracking and localization jobs.

The details of different stages are described in the next few subsections.

3.2 YOLOv3 Implementation

YOLOv3 is the neural network we used to detect the objects in images. Compared with other networks such as Fast R-CNN and Mask R-CNN, it has a much higher frame rate which able to implement to real-time applications when running on GPU. This section will describe our configuration of the YOLOv3 network on GPU and post-processing to the network result.

3.2.1 CUDA backend of YOLOv3 in OpenCV

There is a standalone network of YOLOv3 called Darknet, which is a high-level structure able to do detection using pre-trained model or train new models with different datasets. Darknet is a well-constructed network originated from the authors, covered most of the functions and specialties of the YOLOv3 system. Except for detecting single or multiple images by pre-trained model statically, it is even possible to implement Darknet into real-time webcam with some auxiliary dependencies. Darknet also integrated the function to train customized models, it provided configuration and setup files for some popular open datasets such as VOC [5], COCO, ILSVRC [29], etc. Users can easily customize desired labels for detection and modify parameters for training to control the time usage.

Darknet is compatible with several optional dependencies like CUDA, cuDNN, OpenCV, and OpenMP. A variety of dependencies supports extend the application of the YOLOv3 system in populated environment. CUDA and cuDNN backend significantly boost the running speed to make it applicable to the real-time situation. OpenCV helps to handle more image formats and load the data faster, it is a bridge connect the YOLOv3 system to other OpenCV applications.

In our work, we compiled the YOLOv3 system with both CUDA and OpenCV to match the frame rate of cameras and simplify other images processing to mask the

objects out. Even though Darknet originated support the compiling with OpenCV, it is not convenient to handle the input images and process the detection output in other operations. Instead of Darknet, we implemented YOLOv3 system by OpenCV 'deep neural network'(dnn) module. OpenCV dnn module first released in version 3.3, supports a number of deep learning frameworks, including Caffe, TensorFlow, and PyTorch, etc. Starting with OpenCV 3.4.2, dnn module is capable to detect objects in images with pre-trained YOLOv3 model. Furthermore, we can post-process detection output with the disparity map to get the masks of objects. In addition, the dnn module is relatively easy and fast to implement since it bounds with Python.

In the beginning, OpenCV dnn module only able to run YOLOv3 network in CPU, the FPS of detection is hard to break 10 even with the OpenMP optimization. Fortunately, CUDA backend is available with the release of version 4.2.0. To run the YOLOv3 system in OpenCV with CUDA backend, we compiled the OpenCV 4.2.0 with the Nvidia cuDNN dependency, related libraries are available in Nvidia developer website [20], the version we used is 'libcudnn7.6.5.32-1+cuda 10.0'. At the last, we managed to run YOLOv3 system over 40 FPS in an Nvidia RTX 2070 Super graphic card.

We made both Darknet and OpenCV dnn module work in the same computer. In Chapter 4.1, there is a comparison of the time usage for them to process single image and video under different frameworks.

3.2.2 Processing of YOLOv3 results in array wise

YOLOv3 actually provided three outputs, as shown in Figure 11.

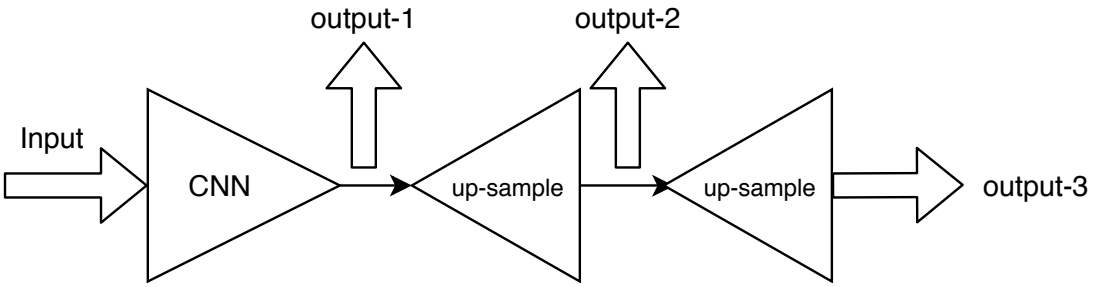


Figure 11. YOLOv3 Network Output

As mentioned in Chapter 2.4.2, the size of input images to YOLOv3 network usually is 416x416. The network down-samples it by scale 32 to a feature map of size $13 * 13$ at the first, where *output-1* is provided. To have a better performance of detection, YOLOv3 up-samples the 13x13 feature map to the size $26 * 26$ and $53 * 53$, where are the result of *output-2* and *output-3*, respectively. Every cell in feature map for different stages predicts three bounding boxes which correspond to three predefined anchors. In total,

the network will predict $(13 * 13 + 26 * 26 + 52 * 52) * 3 = 10647$ bounding boxes, alongside with their corresponding label and label scores.

Bounding boxes in YOLOv3 system represented as its left-top point coordinate, width, and height, four elements in total. YOLOv3 network assigns labels scores for each bounding box, where stored after the coordinate and dimension. The amount of labels scores for each bounding box decided by the pre-trained model. For example, the template model provided by authors of YOLOv3 system, was trained by COCO dataset which able to detect 80 classes objects, this means there will be another 80 elements for each bounding box represent the probability of each class. Searching the index of biggest scores back to the label list of the model, we can know what object this bounding box is most probably for.

In summary, for each image, if we use the provided pre-trained model and stack all results together as an array, the dimension of it will be 10647 by 85. In our system, we do not need to know the exact label name for each bounding box. Therefore, we do not have to iterate all bounding boxes to figure out index value, we only need filtering all bounding boxes which have labels scores higher than threshold and passing their coordinate and dimension to object masking operation.

To increase the speed of the entire system, we implemented a fast method to handle the detection results. Since our system is in Python, we programmed our method through Numpy, which is reliable and efficient to process the data in array wise. As the fact listed in the last paragraph, we do not have to know the exact names of labels. Therefore we stack labels for all 10647 bounding boxes as an array $(10647 * 80)$, then select out elements which scores higher than predefined threshold and their position as a mask. We also stack the coordinate and dimension of bounding boxes to another array $(10647 * 4)$. Applying the mask to this array can simplify the original redundant results to a few effective bounding boxes only.

There is an efficiency evaluation to this algorithm in Chapter 4.2, compared against the method iterate all bounding boxes one by one.

3.3 Image Processing

We relied on YOLOv3 network to detect objects from images. Compared with Mask R-CNN which DynaSLAM used for detection, YOLOv3 has incomparable speed which can be implemented into any real-time applications, but instead of masking out the outline of objects, YOLOv3 produces the bounding boxes of them. If the objects are too close to the camera, bounding boxes usually contain too much information of background, which is what we want to keep for localization and tracking operations in SLAM algorithm.

Briefly speaking, we want to mask out objects outlines after getting their corresponding bounding box from YOLOv3 system. To increase the speed of the whole system, we detected and masked the outlines of objects by pixel-wise image processing but the neural network approaches. The input data of our system is stereo images, which is widely

used to generate depth image. According to the statements discussed in Chapter 2.5, depth information usually stored and visualized as the disparity map. We do not need the exact depth information for each pixel, but disparity maps can help us classify the objects from the background then to get the outlines. There is another use of disparity map in our system which is helping to transfer the masks between stereo images. We only have to pass one image through YOLOv3 network and image processing operations, either left image or right image. We can apply the same mask to another image with their corresponding disparity values.

In the next sub-sections, we will describe how we generate the disparity maps and compute objects masks based on it.

3.3.1 Disparity Estimation

The algorithm we used to generate disparity map is Block Matching, an intensity-based local method estimates disparity at a point by comparing a small region around it with congruent regions extracted from the other image. Same as our CUDA backend YOLOv3 network, we generated disparity map by OpenCV functions Stereo BM and Stereo SGBM. They all based on Block Matching algorithm but Stereo SGBM is a semi-global method, produce more accurate result than local method Stereo BM, correspondingly, more expensive in computation.

Both Stereo SGBM and Stereo BM functions in OpenCV take single-channel rectified grayscale stereo images as input, the performance of disparity estimation highly depends on predefined parameters. Here is the list and a brief explanation of these parameters:

- **minDisparity**. The minimum disparity value, zero by default. When it is bigger than 0, it is easier to find the objects closer to the camera and objects at a larger distance will be ignored. It can be set to negative if the stereo cameras are 'verged' (inclined toward each other).
- **numDisparities**. The size of the disparity search window in pixels. It is the subtraction value of the maximum and minimum disparity.
- **blockSize**. The size of the correlation window for matching, must be an odd number in the range 0 to 255. Larger values have smoother disparity results but smear out small features and depth discontinuities.
- **P1, P2**. Controlling smoothness of disparity. P2 must bigger than P1, the larger the values are, the smoother the disparity is.
- **disp12MaxDiff**. Maximum allowed difference in pixels for left and right disparity check.
- **preFilterCap**. Truncation value for the prefiltered image pixels.

- **uniquenessRatio**. Margin in percentage filtering disparity readings based on a comparison to the second-best correlation value along the epipolar line.
- **speckleWindowSize**. Size of the disparity regions to be ignored. For example, a value of 100 means that all regions with fewer than 100 pixels will be rejected in the final disparity map.
- **speckleRange**. Maximum disparity variations based on their connected component. Disparities are grouped together in the same region if they are within distance set in this parameter.

The tuning of these parameters will directly affect the performance of disparity estimation. We used ROS packages *stereo_image_proc* and *stereo_view* to tune these parameters to improve the efficiency and have a straightaway inspection. *stereo_image_proc* provides a graphic interface where all parameters controlled by track bars, see Figure 12. The original images and disparity map correspond to parameters set in the graphic interface can be visualized by *stereo_view* package.

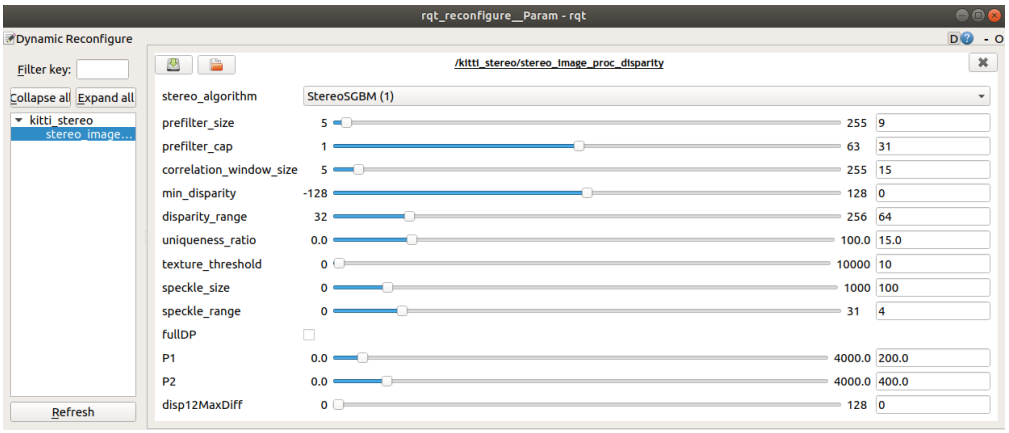


Figure 12. ROS graphic interface tuning Block Matching parameters

stereo_image_proc subscribes the topics of stereo images and their camera information, publishes disparity map in ROS message format(*stereo_msgs/DisparityImage*). There are rectification and grayscale conversion functions built in it and the results published as ROS topics.

3.3.2 Disparity Map Filtering

There are two uses of disparity map in our system, first is outlining the objects detected by YOLOv3 network, second is transferring the objects masks between the stereo images.

Both these two tasks require accurate and smooth disparity map, but most stereo matching algorithms, especially highly-optimized ones that intended for real-time processing, tend to make quite a few errors on some specific sequences. Stereo BM and Stereo SGBM implement some techniques to reduce and invalidate unreliable disparity values, but it always means more computations and information loss in other perspectives. We have to make the best trade-off between the performance and speed. As a result, we introduced a post-filter to align the disparity map edges with their source images to propagate the disparity values from high confidence regions to low confidence regions. The OpenCV-in-built filter we implemented called WLS (Weighted Least Squares) filter, a well-known edge-preserving smoothing technique highly depends on the gradients of images. It helps to eliminate the disparity errors and refine the results in uniform texture-less areas, half-occlusions, and regions near depth discontinuities. Our system lost some speed after implementing this filter into disparity map but gain a better performance in objects outlines masking. Figure 13 and Figure 14 are the comparisons before and after WLS filtering for Stereo SGBM algorithm result with the same parameter configuration.

Figure 13. Stereo SGBM result before WLF filtering

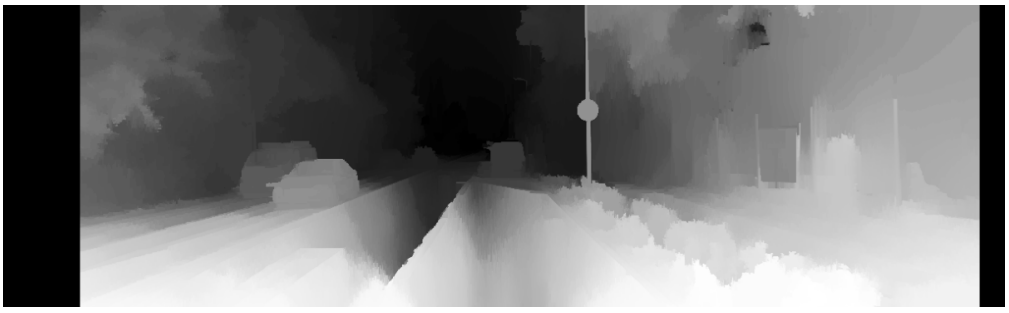


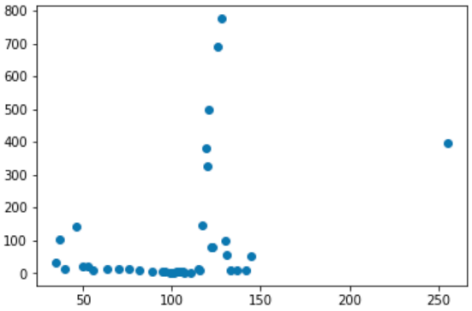
Figure 14. Stereo SGBM result after WLF filtering

3.3.3 Objects masking based on disparity map

To have a proper visualization of disparity map, we introduced the normalization process and converted the format of pixel values to unsigned integer 8, which means the disparity map can be visualized as a grayscale image with the pixels values distributed in range 0 to 255, as the Figure 14. Besides, there is another algorithm to normalize the brightness and increase the contrast of the normalized disparity map in case of scenarios with poor disparity (for example the image which has too much sky or plain ground).

We masked the outlines of objects by normalized disparity maps and bounding boxes of objects from the YOLOv3 network. At first, we calculated the value of pixels and their corresponding quantities for bounding box areas in visualized disparity map, see Figure 15.

(a) Disparity Map with Bounding Box



(b) Histogram of Bounding Box of Dsparity

(c) Masked Grayscale Image

(d) Fill-Flooded Masked color Image

Figure 15. Objects Masking Processing

The rectangular area in Figure 15a is the object bounding box in the visualized disparity map. Figure 15b is the statistical plot of the pixel values and their quantities of rectangular box area in Figure 15a. In visualized disparity map, object is distinguished from the background, which takes majority amount of pixels in rectangular box and the values to these pixels should be in a narrow range, therefore in the plot of Figure 15b, the highest histogram represents the pixels of the object in visualized disparity map, which is the exact mask of object outline. We selected the mask out and applied it back to the original image as showed in Figure 15c. At the last, we implemented a flood-fill algorithm to fill the blank gaps inside the mask, the final masking result showed in Figure 15d.

Figure 15d shows the making result for the left image, there is no need to repeat the whole procedure for right image. The shapes of objects outlines should be similar in left and right images for ordinary stereo camera systems, we can simply applying the same mask to another image if we know the corresponding offset of mask between stereo images, which is exactly the disparity value from the stereo matching algorithm. In our system, these disparity values stored in the disparity map before the visualization process. The value in raw disparity map divided by 16 is the offset in pixels unit of that point between stereo images. We can easily transfer the objects masks from left image to right image by disparity values we got from the stereo matching algorithm.

The overview structure of our object masking processing was shown in Figure 16.

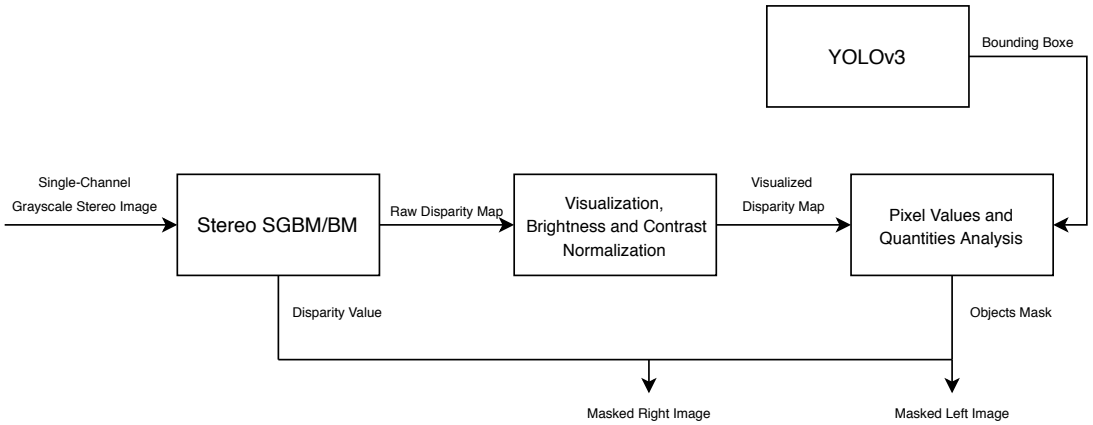


Figure 16. Masking Processing Block Diagram

The output of *Stereo SGBM/BM*, which the *Raw Disparity Map* contains the actual disparity values, converted to *Visualized Disparity Map* at the first by normalization to its pixel values, brightness, and contrast. Together with *Bounding Box* from *YOLOv3* network, we can acquire the masks of the objects for left or right images, then transfer the masks to another image by their disparity values stored in the *Raw Disparity Map*.

3.4 ORB-SLAM2 Configuration

The ROS-based-ORB-SLAM2[3] system we used subscribes the left and right images as ROS topics in the stereo case. The input images can be either colorful or grayscale, as long as predefined in the configuration file. The output is the Odometry of the camera in ROS message format(nav_msgs/Odometry). We modified the source codes to add a switch writing Odometry information as a text file.

In ORB-SLAM2 system, stereo calibration of the camera directly affects the performance for the stereo case. The related stereo calibration functions [4] are available in

OpenCV. ROS provided the calibration node [36] straightly connects to the camera, also works with the static images sequences.

In configuration file, we set the number of features extracted for every image to 4000 and levels of scale pyramid as 16. This will increase the computation work but produce more accurate results.

4 Analysis and Results

4.1 YOLOv3 speed performance between Darknet and OpenCV

There are many ways to construct a YOLOv3 network. Authors provided a network called Darknet includes the images detection, customized model training, and webcam application. According to the statements discussed in Chapter 3.2.1, Darknet is a standalone network, which is not convenient to do pre-processing to input images and post-processing to detection results. Another popular substitution of Darknet is the OpenCV dnn module, which has the API to construct the YOLOv3 network easily. The entire network wrapped inside the OpenCV environment so that it is possible to make use of abundant OpenCV libraries to process the input and output of the detection network. After version 4.2.0, YOLOv3 network inside the OpenCV dnn module can be powered by GPU, which is what implemented in our system.

Speed is another reason we considered using OpenCV dnn module. In our test, we reached approximate 50 FPS of CUDA backend YOLOv3 network in OpenCV with an Nvidia RTX 2070 Super graphic card. This speed is a little bit better than the result from cuDNN compiled Darknet network, which we ran in the same machine and operating system.

Darknet can runes in CPU standalone or optimized by OpenMP, and in GPU with cuDNN backend. We test all these configurations in a machine with 24 cores AMD Ryzen9 3900x CPU and Nvidia RTX 2070 Super GPU, the operating system is Ubuntu 18.04. Table 1 shows the time for different configurations to process the same image, as Figure 1. During the test, time always fluctuated within a tiny range, we ran every configuration to process the same image for 5 times, take their average as the final time usage.

Framework	**Hardware**	**Time(ms)**
Darknet	24-cores AMD Ryzen9 3900x CPU	4516
Darknet + cuDNN	NVIDIA RTX 2070 Super GPU	25.6
Darknet + OpenMP	24-cores AMD Ryzen9 3900x CPU	1612

Table 1. Darknet speed of different frameworks

For YOLOv3 network in OpenCV dnn module, we compared the speed of network in both CPU and GPU. The hardware and operating system are same as Darknet test. The version of related software was list in Chapter 3.2.1. We processed the same image (Figure 1) for 5 times in CPU and GPU as well, results were shown in Table 2.

Refer to Table 2, compared with running standalone in CPU, the speed of single image detection only increased 9.5% in GPU for OpenCV dnn module, much smaller than the increase between Darknet standalone and cuDNN backend Darknet, 99.4%.

Framework	Hardware	Seq. Time(ms)	Avg.(ms)
OpenCV	24-cores AMD Ryzen9 3900x CPU	(1) 4228	4216
		(2) 4074	
		(3) 4424	
		(4) 4048	
		(5) 4304	
OpenCV + cuDNN	NVIDIA RTX 2070 Super GPU	(1) 3801	3816
		(2) 3798	
		(3) 3856	
		(4) 3804	
		(5) 3819	

Table 2. OpenCV single image detection speed in CPU and GPU

This is because of the way OpenCV handles the data inside the network, there are part of works still done by CPU for cuDNN backend YOLOv3 network, for example, the function *blobFromImage*, a preprocessing in OpenCV for deep learning classification to do mean subtraction and scaling for input images. This involves the transmission of data between the CPU and GPU, which is a slowdown of the speed.

However, we found there is a significant improvement in the efficiency to process videos in cuDNN backend OpenCV YOLOv3 network. Table 3 is the comparison of time usage in CPU and GPU for the same video. We extracted the speeds of 10 frames and calculated the average of them, the increase of the speed reached 88.9%.

Framework	Hardware	Time(ms)
OpenCV	24-cores AMD Ryzen9 3900x CPU	171
OpenCV + cuDNN	NVIDIA RTX 2070 Super GPU	20.2

Table 3. OpenCV video detection speed in CPU and GPU

4.2 Efficiency Evaluation of Detection Results Processing

According to the facts listed in Chapter 2.4.2 and Chapter 3.2.2. The direct detection results from YOLOv3 network are 10647 bounding boxes with probabilities of all labels predefined in model. Take the pre-trained model offered by YOLOv3 authors as an example, the size of the final result will be 10647 by 85 if stacking all predicted bounding boxes together as an array. However, there is too much redundant information in this array, we have to filter out the effective bounding boxes which truly represent the objects in the image. Since our system was designed for real-time applications, we have to process these results in a fast and efficient way.

We developed a simple algorithm to handle and filter these data in array wise with the help of Numpy library in Python. There is a detailed description of this algorithm in Chapter 3.2.2. In this chapter, we will evaluate the efficiency of this algorithm, compared to a slow method we used initially. The procedure of this slow method is shown in Algorithm 1.

Algorithm 1: Processing detection results in FOR loop

Input: Three layers outputs from YOLOv3 network
Result: Effective bounding boxes of objects

```
for layers outputs do
    for each detection in layers outputs do
        Appending effective bounding boxes information
        Appending effective labels scores and indexes
;
Applying Non-Maximum Suppression
;
for bounding boxes after suppression do
    Applying back to image select objects out
;
```

The reason this algorithm slow is because we iterated all the elements in the detection result. When the size of detection result is too big, the latency of one-by-one iteration is unacceptable for a real-time system. We test it by detecting a single image, the model is trained by COCO dataset which can detect 80 classes, the time this algorithm need is 27.9ms(average of 5 times test), as much as the time YOLOv3 network spent to detect a single image.

The general structure to processing detection results in array wise was shown in Algorithm 2. The time it spent to process the same image only 4ms, 7 times faster than previous algorithm.

Algorithm 2: Processing detection results in array wise

Input: Three layers outputs from YOLOv3 network
Result: Effective bounding boxes of objects

```
Stacking all labels scores as an array
Stacking all bounding boxes information as an array
Assembling effective scores as a mask
Applying mask to bounding boxes information array
;
Applying Non-Maximum Suppression
;
for bounding boxes after suppression do
    Applying back to image select objects out
;
```

4.3 Evaluation of Stereo SGBM and BM in speed and performance

Our system relied on disparity map to mask the objects out in stereo images. The algorithm we used to estimate the disparity is Block Matching, which is available in OpenCV library. The name of the function is Stereo BM, a local method of stereo matching algorithms which able to run fast with a trade-off in performance. There is another similar function in OpenCV called Stereo SGBM which is a semi-global method. It performs better in stereo matching but runs relatively slow. These two functions share some same parameters and we made them both available in our system. This chapter will compare these two functions in the perspectives of performance and speed.

According to the statements described in Chapter 3.3.1. The results of these two functions highly depended on the parameters, and the easiest way to tuning them is by ROS *stereo_image_proc* package. Among all parameters listed in Figure 12, the ones control post-filtering and correlation of disparity is relatively more important and affect the result directly, for example, the *speckle_size* and *speckle_range*, decide the minimum amount of pixels to be considered as a disparity blob and how close in value disparities to be regarded as the same blob. For disparity correlation parameters, *min_disparity* and *disparity_range* define the distance that stereo block matcher able to 'see', according to visualized disparity image is the 'gaps' between disparity blobs.

All parameters have to be tuned based on the scenes in input images. We used the KITTI sequences [8] in our system, Figure 17 and Figure 18 are the result of disparity estimation for Stereo SGBM and Stereo BM with the same parameter setting, respectively. The visualization of disparity map here is based on another ROS package *stereo_view*, a convenient tool can subscribe to the raw disparity map as ROS topic, then color and display them on the fly.

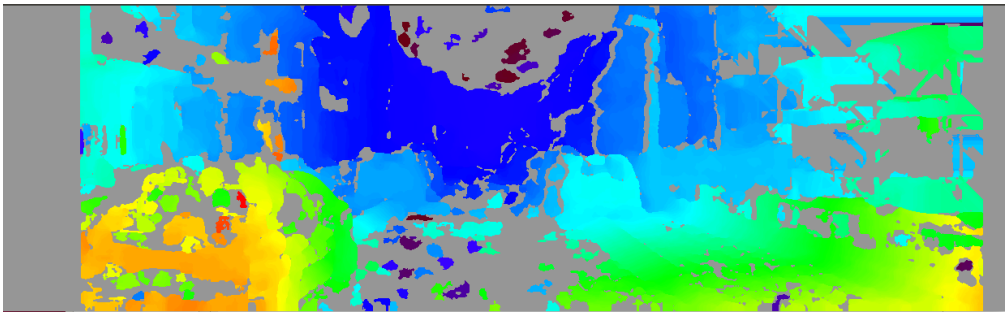


Figure 17. Stereo SGBM disparity map for KITTI sequence 00

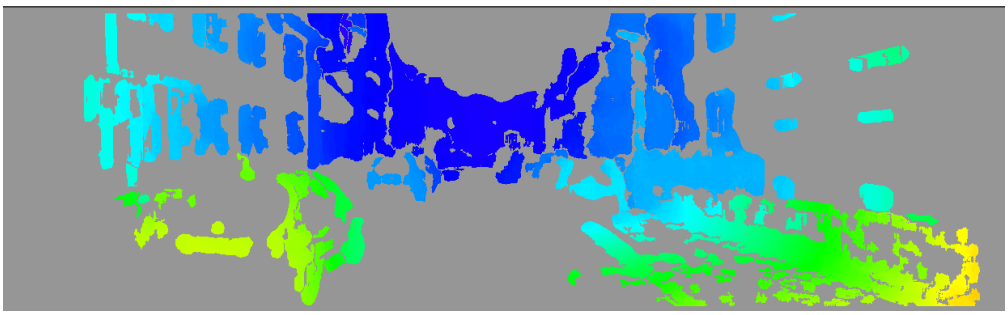


Figure 18. Stereo BM disparity map for KITTI sequence 00

Comparing with these two figures, we can see Stereo SGBM performs better in drawing the outlines of the objects, but the speed is another character we must concern. We inspected the speed of Stereo SGBM and Stereo BM in two ways, first is in ROS platform. Since *stereo_image_proc* node publishes the disparity map as a topic, we can inspect the frequency of this topic, correspondingly, it will not bigger than the frequency of input image topics. The second way we checked the speed is by computing disparity map to a single image with related OpenCV functions, we can calculate the time of the whole process needs.

For KITTI sequences 00 to 21, the images were captured at 10Hz, we prepared the input image topics at the same frequency. In our experiment, we found the Stereo BM can publish the disparity map at the same frequency. For Stereo SGBM, the frequency of the topic is around 7.46Hz, which means it cannot process KITTI dataset lively.

In OpenCV platform, we implemented Stereo SGBM and Stereo BM to process Figure 1 for five times and calculated the average of the time consuming, which are 44.1ms and 21.0ms, respectively. The Stereo BM runs around 2 times faster than Stereo SGBM in OpenCV.

4.4 Trajectory and Time Evaluation for KITTI dataset

To have a precise evaluation of the performance of our system, we used stereo images from KITTI [8], a specific dataset and benchmark widely applied in the autonomous driving field. All the sensors in KITTI dataset are calibrated and synchronized, minimizing the noises and bias of data, and providing with accurate ground truth. The sequences of KITTI are captured by driving around the mid-size city, in rural areas, and on highways, covered most of the populated environments for autonomous driving vehicles.

4.4.1 Timing Analysis

The KITTI dataset has stereo sequences for urban and motorway environments recorded by a car. We tested our system by sequence 00 to sequence 09 against DynaSLAM to evaluate the speed and the accuracy of trajectory. Table 4 shows the speed of DynaSLAM and our system based on different stereo matching algorithms. Note that the data in Table 4 are only the times for systems to mask out one grayscale image, not include the time usage for localization and tracking operations. DynaSLAM used Mask R-CNN to mask the objects from images and our system based on the YOLOv3 network to detect the bounding boxes for objects first, then mask them out by their disparity values. We also have some post-processing to optimize the object masks such as filling the blank gaps, all the time usage for these processes is included in the table.

Sequence	**DynaSLAM(ms)**	**TBD_BM(ms)**	**TBD_SGBM(ms)**
KITTI 00	169	54.6	114
KITTI 01	163	50.6	93 1
KITTI 02	162	67.4	91
KITTI 03	166	61.3	99
KITTI 04	164	53.0	110
KITTI 05	166	56.9	103
KITTI 06	166	59.0	95
KITTI 07	170	71.1	107
KITTI 08	168	52.7	101
KITTI 09	164	49.0	91

Table 4. Comparison of Speed with DynaSLAM

We selected 10 frames randomly at the beginning of the sequence and calculated the average of the time used to objects masking for different systems. In our system, the speed affected by the scenario in the image, if there are many objects in image need to be detected and masked, then it will take more time to process image because the operations like counting value and amount of object pixels in disparity map and blank gaps filling have to be repeated for each bounding boxes.

In stereo case, both left and right images have to be masked, which means the actual time DynaSLAM needs is the values in Table 4 multiply by two because the time listed in Table 4 for DynaSLAM is only for one image. In our system, we made use of disparity values to estimate the offset of objects between stereo images, therefore, we only have to pass one image, either left or right, through neural network to do detection, objects masking operation for another image is done by light-weight mathematical approach, time spent in this process included in the values of Table 4 for our system whatever Stereo BM or Stereo SGBM based.

4.4.2 Trajectory Evaluation

The trajectory evaluation of our system against DynaSLAM based on ten KITTI training sequences (00 to 09) because their ground truth is available. The metric we used to evaluate trajectories of our system and DynaSLAM against ground truth is absolute pose error, executed by a powerful Python-based public benchmarking tool, evo, proposed in [10].

In KITTI training sequences, the ground truth of poses contained in text files. Each row of the file contains the first three rows of a 4 by 4 homogeneous pose matrix flattened into one line. It is not a proper trajectory message because there are no timestamps, the timestamps for KITTI training sequences were store in another text file. Since our system built in ROS platform, the trajectory published as a ROS topic in format *nav_msgs/Odometry*. In evo, the trajectories under comparison have to be the same format, we converted the ground truth poses of KITTI training sequences to ROS topic and stored it as the static bag file by a script [10], then used *message_filters* [37] library to synchronize the timestamps of this topic same as the Odometry topics computed by our system and DynaSLAM.

Sequence	**max(m)**	**mean(m)**	**median(m)**	**min(m)**	**rmse(m)**	**sse(m)**	**std(m)**
KITTI 00	16.21	7.97	7.71	0.86	**8.71**	338225	3.50
KITTI 01	108.41	57.79	66.86	0.53	**72.85**	5668520	44.36
KITTI 02	33.59	13.99	11.85	0.76	**17.30**	1385508	10.18
KITTI 03	1.74	0.71	0.70	0.16	**0.78**	490	0.33
KITTI 04	2.29	1.64	1.62	1.27	**1.65**	732	0.23
KITTI 05	9.50	3.91	3.57	0.20	**4.65**	59442	2.52
KITTI 06	5.85	2.83	2.50	0.15	**3.12**	10687	1.33
KITTI 07	5.40	2.57	1.96	0.09	**3.01**	9956	1.57
KITTI 08	19.06	11.94	11.71	0.93	**12.35**	620541	3.14
KITTI 09	17.29	7.39	8.40	0.29	**8.58**	116853	4.37

Table 5. Absolute Pose Error of trajectories for DynaSLAM

Both DynaSLAM and our system relied on ORB-SLAM2 for localization and track-

ing. According to the statement listed in Chapter 3.4, ORB-SLAM2 setup in ROS environment and subscribes the left and right image streams as topics in stereo case. In our system, ORB-SLAM2 was integrated with the neural network and image processing, the pipeline takes raw stereo images as input and computes the Odometry of cameras as output. The frame rate of input images to ORB-SLAM2 is 10 FPS for both our system and DynaSLAM. Note that the performance of ORB-SLAM2 is affected by its running speed, in general, the slower it runs, relatively more accurate of the Odometry result.

We ran the ORB-SLAM2 standalone with the same parameters and environment to reduce the effects of the non-deterministic nature of the system. Table 5 shows the Absolute Pose Error of DyanSLAM stereo mode for KITTI sequences 00 to 09. Table 6 and Table 7 shows the results in the same sequences for our system variants of Stereo BM and Stereo SGBM algorithms.

Sequence	**max(m)**	**mean(m)**	**median(m)**	**min(m)**	**rmse(m)**	**sse(m)**	**std(m)**
KITTI 00	25.63	9.71	7.96	0.66	**11.31**	577716	5.80
KITTI 01	132.95	64.40	66.41	0.14	**83.69**	7409654	53.44
KITTI 02	30.71	14.11	10.36	0.08	**16.65**	1282034	8.85
KITTI 03	3.13	2.04	1.91	0.17	**2.10**	3518	0.52
KITTI 04	2.45	1.68	1.65	0.06	**1.72**	782	0.33
KITTI 05	10.76	4.43	3.70	0.02	**5.36**	79086	3.02
KITTI 06	5.45	3.30	3.84	0.07	**3.61**	14249	1.46
KITTI 07	9.82	3.43	3.92	0.02	**3.91**	16728	1.88
KITTI 08	18.16	11.98	12.32	1.00	**12.43**	627072	3.31
KITTI 09	11.95	7.38	6.99	0.03	**7.90**	98971	2.81

Table 6. Absolute Pose Error of trajectories for pipeline featured with Stereo BM

Sequence	**max(m)**	**mean(m)**	**median(m)**	**min(m)**	**rmse(m)**	**sse(m)**	**std(m)**
KITTI 00	27.15	9.62	8.21	0.32	**11.08**	548412	5.51
KITTI 01	139.93	81.65	92.77	0.08	**96.49**	9906023	51.42
KITTI 02	33.20	15.04	13.12	0.76	**17.79**	1466986	9.50
KITTI 03	3.23	1.91	1.73	0.07	**2.06**	3373	0.75
KITTI 04	1.30	0.60	0.49	0.07	**0.66**	118	0.28
KITTI 05	10.94	4.29	3.50	0.01	**5.41**	80491	3.31
KITTI 06	5.95	2.94	2.70	0.08	**3.25**	11569	1.37
KITTI 07	6.47	3.04	3.30	0.02	**3.69**	14854	2.09
KITTI 08	18.81	12.33	12.42	0.86	**12.70**	655859	3.08
KITTI 09	15.77	6.81	7.21	0.34	**7.74**	94838	3.68

Table 7. Absolute Pose Error of trajectories for pipeline featured with Stereo SGBM

In general, the absolute pose RMSE results are similar in all systems, even though the

objects masking is better in Mask R-CNN and Stereo SGBM cases. However, the Stereo BM based system can work in real-time applications because of the light computation.

As future work, it is interesting to test with other datasets to cover more environments and scenarios. To minimize the non-deterministic affections from the SLAM algorithm, it is wiser to run the same sequences multiple times and take the median result.

5 Conclusion

Visual SLAM keeps drawing attention from the researchers because of the simplicity and robustness of the required sensors. The advanced development of computer vision related technologies is another stimulation to Visual SLAM systems, making them applicable to more complicated real-world environments. For example, DynaSLAM used Mask R-CNN to mask the dynamic objects out of images, avoiding SLAM algorithms extract features from them, and producing re-usable maps can work in long-term applications. However, the heavy computation of Mask R-CNN limits the usage of DynaSLAM in real-time applications.

Our system focused on the perspective of running speed, proposed to use YOLOv3 system, an extreme fast neural network, substitute Mask R-CNN to detect dynamic objects in images. Since YOLOv3 network only computes out the bounding boxes, we introduced stereo matching algorithms to detect the contour of objects. It also helps to determine the offset of pixels between stereo images, therefore, there is no need to pass both images to the neural network for stereo case. In addition, we integrated all processes include objects detection neural network, image processing, and SLAM algorithms, etc. in ROS environment, to provide a user-friendly interface monitoring and controlling the entire system.

We evaluated the time and trajectory against DynaSLAM based on ten KITTI training sequences for the stereo case. The performance of objects masking operation in our system depends on the results of stereo matching, which is limited by the exact environment, for example, the poor disparity scenarios. In general, our accuracy is similar to DynaSLAM for KITTI sequences in stereo case, in the condition of that several times faster running speed, which applicable to most of real-time situations. Future extensions of this work might include the comparison with more kinds of datasets.

The part yet to be improved of this work is objects outline masking. There are reliability problems detecting outlines for each frame of disparity map because the stereo matching highly affected by the environment of images. Algorithms like multi-view geometry can be implemented to detect the displacement of the same objects in successive frames, then the best outline masks computed by the high-quality disparity map can be used to the objects in different frames. This can improve the performance of masking processing in the frames with poor-quality disparity maps. In addition, a better speed calculating approach is necessary for benchmarking. The general frame rate of the whole sequence is better than the average of a period to reflect the actual speed, since in our system, it was affected by the amount of objects in each frame.

References

[1] P. F. Alcantarilla, J. J. Yebes, J. Almazán, and L. M. Bergasa. On combining visual slam and dense scene flow to increase the robustness of localization and mapping in dynamic environments, 2012.

[2] B. Bescos, J. M. Fácil, J. Civera, and J. Neira. Dynaslam: Tracking, mapping, and inpainting in dynamic scenes, 2018.

[3] caiofis. Ros-friendly interface to orb-slam2. https://github.com/caiofis/ORB_SLAM2, Dec 2018.

[4] OpenCV Documentation. Camera calibration and 3d reconstruction. https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html.

[5] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John M. Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge, 2009.

[6] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan. Object detection with discriminatively trained part-based models, 2010.

[7] D. Galvez-López and J. D. Tardos. Bags of binary words for fast place recognition in image sequences, 2012.

[8] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite, 2012.

[9] R. Girshick. Fast r-cnn, 2015.

[10] Michael Grupp. evo: Python package for the evaluation of odometry and slam. https://github.com/MichaelGrupp/evo, 2017.

[11] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask r-cnn, 2017.

[12] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[13] Ayoosh Kathuria. What's new in yolo v3? https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b, Apr 2018.

[14] D. Kim and J. Kim. Effective background model-based rgb-d dense visual odometry in a dynamic environment, 2016.

[15] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2014.

[16] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation, 2015.

[17] Christopher Mei, Gabe Sibley, Mark Cummins, Paul Newman, and Ian Reid. Rslam: A system for large-scale mapping in constant-time using stereo, 09 2011.

[18] Raul Mur-Artal, J. M. M. Montiel, and Juan D. Tardos. Orb-slam: A versatile and accurate monocular slam system, Oct 2015.

[19] Raul Mur-Artal and Juan D. Tardos. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras, Oct 2017.

[20] NVIDIA. Nvidia developer. `https://developer.nvidia.com/`.

[21] T. Pire, T. Fischer, J. Civera, P. De Cristóforis, and J. J. Berlles. Stereo parallel tracking and mapping for robot localization, 2015.

[22] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system, 01 2009.

[23] J. Redmon and A. Farhadi. Yolo9000: Better, faster, stronger, 2017.

[24] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2015.

[25] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, 2018.

[26] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2015.

[27] L. Riazuelo, L. Montano, and J. M. M. Montiel. Semantic visual slam in populated environments, 2017.

[28] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. Orb: An efficient alternative to sift or surf, 2011.

[29] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge, 2014.

[30] Stanford Artificial Intelligence Laboratory et al. Robotic operating system.

[31] H. Strasdat, A. J. Davison, J. M. M. Montiel, and K. Konolige. Double window optimisation for constant time visual slam, 2011.

[32] Yuxiang Sun, Ming Liu, and Max Q.-H. Meng. Improving rgb-d slam in dynamic environments: A motion removal approach, 2017.

[33] Y. Wang and S. Huang. Motion segmentation based robust rgb-d slam, 2014.

[34] S. Wangsiripitak and D. W. Murray. Avoiding moving outliers in visual slam by tracking moving objects, 2009.

[35] Wei Tan, Haomin Liu, Z. Dong, G. Zhang, and H. Bao. Robust monocular slam in dynamic environments, 2013.

[36] ROS wiki. How to calibrate a stereo camera. `http://wiki.ros.org/camera_calibration/Tutorials/StereoCalibration`.

[37] ROS wiki. message_filters. `http://wiki.ros.org/message_filters`.

Appendix

I. Code

The source code and experimental data for this system is located in repository:
`https://gitlab.ds.cs.ut.ee/ds/junyigu/thesis_claude`

The access to the repository could be granted upon sending an email to:
`tfkcloud47@gmail.com`

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Junyi Gu**,
(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work from **20/05/2020** until the expiry of the term of copyright,

   Towards Faster Masking of Dynamic Objects for Visual Simultaneous Localization and Mapping,
   (title of thesis)

   supervised by Amnir Hadachi, PhD and Artjom Lind, Msc.
   (supervisor's name)

2. I am aware of the fact that the author retains the rights specified in p. 1.

3. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Junyi Gu
20/05/2020