

TRƯỜNG ĐẠI HỌC BÁCH KHOA TP. HỒ CHÍ MINH
KHOA ĐIỆN – ĐIỆN TỬ
BỘ MÔN ĐIỀU KHIỂN TỰ ĐỘNG

PHAN MINH TRÍ

LUẬN VĂN TỐT NGHIỆP
THIẾT LẬP XE TỰ HÀNH TRONG NHÀ
DÙNG STEREO CAMERA

KỸ SƯ NGÀNH KỸ THUẬT ĐIỀU KHIỂN & TỰ ĐỘNG HÓA

TP. HỒ CHÍ MINH, 2018

TRƯỜNG ĐẠI HỌC BÁCH KHOA TP. HỒ CHÍ MINH
KHOA ĐIỆN – ĐIỆN TỬ
BỘ MÔN ĐIỀU KHIỂN TỰ ĐỘNG

PHAN MINH TRÍ – 1414226

LUẬN VĂN TỐT NGHIỆP
THIẾT LẬP XE TỰ HÀNH TRONG NHÀ
DÙNG STEREO CAMERA
(BUILDING AN AUTOMATIC VEHICLE INDOOR
BASED ON STEREO CAMERA)

KỸ SƯ NGÀNH KỸ THUẬT ĐIỀU KHIỂN & TỰ ĐỘNG HÓA

GIẢNG VIÊN HƯỚNG DẪN
TS. NGUYỄN VĨNH HẢO

TP. HỒ CHÍ MINH, 2018

TRƯỜNG ĐẠI HỌC BÁCH KHOA TP. HỒ CHÍ
MINH

KHOA ĐIỆN – ĐIỆN TỬ

BỘ MÔN: ĐIỀU KHIỂN TỰ ĐỘNG

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM

Độc lập - Tự do - Hạnh phúc

TP. HCM, ngày.....tháng.....năm.....

**NHẬN XÉT LUẬN VĂN TỐT NGHIỆP
CỦA CÁN BỘ HƯỚNG DẪN**

Tên luận văn:

THIẾT LẬP XE TỰ HÀNH TRONG NHÀ DÙNG STEREO CAMERA

Nhóm Sinh viên thực hiện:

PHAN MINH TRÍ

Cán bộ hướng dẫn:

1414226 TS. NGUYỄN VĨNH HẢO

Đánh giá Luận văn

1. Về cuốn báo cáo:

Số trang	_____	Số chương	_____
Số bảng số liệu	_____	Số hình vẽ	_____
Số tài liệu tham khảo	_____	Sản phẩm	_____

Một số nhận xét về hình thức cuốn báo cáo:

2. Về nội dung luận văn:

3. Về tính ứng dụng:

4. Về thái độ làm việc của sinh viên:

Đánh giá chung: Luận văn đạt/không đạt yêu cầu của một luận văn tốt nghiệp kỹ sư, xếp loại
Giỏi/ Khá/ Trung bình

Điểm từng sinh viên:

Phan Minh Trí:...../10

Cán bộ hướng dẫn

(Ký tên và ghi rõ họ tên)

TRƯỜNG ĐẠI HỌC BÁCH KHOA TP. HỒ CHÍ
MINH

KHOA ĐIỆN – ĐIỆN TỬ

BỘ MÔN: ĐIỀU KHIỂN TỰ ĐỘNG

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM

Độc lập - Tự do - Hạnh phúc

TP. HCM, ngày....tháng.....năm.....

**NHẬN XÉT LUẬN VĂN TỐT NGHIỆP
CỦA CÁN BỘ PHẢN BIỆN**

Tên luận văn:

THIẾT LẬP XE TỰ HÀNH TRONG NHÀ DÙNG STEREO CAMERA

Nhóm Sinh viên thực hiện:

PHAN MINH TRÍ

Cán bộ phản biện:

1414226

Đánh giá Luận văn

1. Về cuốn báo cáo:

Số trang	_____	Số chương	_____
Số bảng số liệu	_____	Số hình vẽ	_____
Số tài liệu tham khảo	_____	Sản phẩm	_____

Một số nhận xét về hình thức cuốn báo cáo:

2. Về nội dung luận văn:

3. Về tính ứng dụng:

4. Về thái độ làm việc của sinh viên:

Đánh giá chung: Luận văn đạt/không đạt yêu cầu của một luận văn tốt nghiệp kỹ sư, xếp loại
Giỏi/ Khá/ Trung bình

Điểm từng sinh viên:

Phan Minh Trí:...../10

Người nhận xét

(Ký tên và ghi rõ họ tên)

TRƯỜNG ĐẠI HỌC BÁCH KHOA TP. HỒ CHÍ
MINH

KHOA ĐIỆN – ĐIỆN TỬ
BỘ MÔN: ĐIỀU KHIỂN TỰ ĐỘNG

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập - Tự do - Hạnh phúc

TP. HCM, ngày....tháng.....năm.....

ĐỀ CƯƠNG CHI TIẾT

**TÊN LUẬN VĂN: THIẾT LẬP XE TỰ HÀNH TRONG NHÀ DÙNG STEREO
CAMERA**

Cán bộ hướng dẫn: TS. NGUYỄN VĨNH HẢO

Thời gian thực hiện: Từ ngày.....đến ngày.....

Sinh viên thực hiện:

Phan Minh Trí - 1414226

Nội dung đề tài:

- Nghiên cứu hệ điều hành dành cho robot (ROS – Robot Operating System).
- Nghiên cứu thuật toán điều hướng cho xe tự hành dựa trên bản đồ dựng trước.
- Thiết kế mô hình xe tự hành.
- Ứng dụng máy tính nhúng và stereo camera để xây dựng nền tảng thực hiện đề tài.
- Xây dựng bản đồ và điều hướng trên bản đồ đã dựng dựa vào mô hình đã xây dựng.

Kế hoạch thực hiện:

Phan Minh Trí	Lên mục tiêu và nhiệm vụ luận văn	22 tháng 1
Phan Minh Trí	Nghiên cứu hệ điều hành dành cho robot (ROS – Robot Operating System)	05 tháng 3
Phan Minh Trí	Nghiên cứu cách xây dựng bản đồ (mapping)	12 tháng 3

Phan Minh Trí	Nghiên cứu cách điều hướng cho xe (navigation)	19 tháng 3
Phan Minh Trí	Mô phỏng	26 tháng 3
Phan Minh Trí	Thiết lập phần cứng cho xe	02 tháng 4
Phan Minh Trí	Xây dựng bản đồ môi trường trong nhà	23 tháng 4
Phan Minh Trí	Điều hướng xe dựa trên bản đồ đã dựng	07 tháng 5
Phan Minh Trí	Tích hợp hệ thống	21 tháng 5
Phan Minh Trí	Kiểm tra và sửa chữa	28 tháng 5
Phan Minh Trí	Viết báo cáo	11 tháng 6
Phan Minh Trí	Xây dựng slide PowerPoint	18 tháng 6
Xác nhận của Cán bộ hướng dẫn (Ký tên và ghi rõ họ tên)		TP. HCM, ngày....thángnăm..... Sinh viên (Ký tên và ghi rõ họ tên)

DANH SÁCH HỘI ĐỒNG BẢO VỆ LUẬN VĂN

Hội đồng chấm luận văn tốt nghiệp, thành lập theo Quyết định số
ngày của Hiệu trưởng Trường Đại học Bách khoa TP.HCM.

1. – Chủ tịch.
2. – Thư ký.
3. – Ủy viên.
4. – Ủy viên.
5. – Ủy viên.

LỜI CẢM ƠN

Lời đầu tiên xin gửi lời cảm ơn chân thành đến thầy TS. NGUYỄN VĨNH HẢO đã tận tình chỉ dẫn, giúp đỡ và định hướng trong suốt thời gian làm đồ án và luận văn, tạo điều kiện thuận lợi để hoàn thành đề tài luận văn này.

Bên cạnh đó xin cảm ơn quý thầy cô trong bộ môn Điều khiển và Tự động hóa đã trang bị cho em những kiến thức nền tảng, bổ ích trong khoảng thời gian học tập tại trường.

Tôi xin gửi lời biết ơn sâu sắc đến cha mẹ, anh chị trong gia đình luôn ủng hộ, sát cánh và là nguồn động lực vô cùng to lớn để tôi có thể hoàn thành tốt khoảng thời gian học tập và nghiên cứu tại trường.

Cuối cùng, xin gửi lời cảm ơn đến bạn Lê Thành Long (DD16KSVT) và câu lạc bộ Pay It Forward đã có những sự hỗ trợ nhiệt tình trong quá trình thực nghiệm đề tài để đề tài luận văn có thể hoàn thành tốt.

Tp. Hồ Chí Minh, ngày 11 tháng 06 năm 2018

Phan Minh Trí

MỤC LỤC

TÓM TẮT LUẬN VĂN	1
Chương 1. TỔNG QUAN VỀ ĐỀ TÀI	2
1.1. Định hướng cho đề tài	2
1.2. Mục tiêu của đề tài.....	3
1.3. Đối tượng và phạm vi nghiên cứu	3
1.4. Các bước thực hiện đề tài	3
Chương 2. TỔNG QUAN VỀ ROBOT OPERATING SYSTEM VÀ CƠ SỞ LÝ THUYẾT	5
2.1. Giới thiệu về Robot Operating System.....	5
2.1.1. Tổng quan về hệ điều hành dành cho robot – ROS.....	5
2.1.2. Cấu trúc ROS.....	7
2.1.2.1. ROS Filesystem Level	8
2.1.2.2. ROS Computation Graph Level	10
2.1.2.3. ROS Community Level	12
2.1.3. 2D Navigation stack	13
2.1.3.1. Tổng quan về Navigation stack	15
2.1.3.2. Costmap 2D	15
2.1.3.3. Cách cấu hình costmap	18
2.1.3.4. Cấu hình base_local_planner.....	20
2.2. Nguyên lý tính khoảng cách vật thể từ stereo camera.....	21
2.3. Thuật toán Dijkstra cho global planner	24
2.4. Thuật toán Dynamic Window Approach để tránh vật cản cho local planner .	
.....	28

2.4.1.	Không gian tìm kiếm.....	28
2.4.2.	Tối ưu	29
Chương 3. THIẾT LẬP XE TỰ HÀNH VÀ LẬP TRÌNH TÍCH HỢP HỆ THỐNG		32
3.1.	Thiết lập xe tự hành	32
3.1.1.	Board điều khiển	32
3.1.2.	Camera.....	35
3.1.3.	Mobile base	38
3.1.3.1.	Thiết kế phần cứng	38
3.1.3.2.	Thiết kế chương trình nhúng	43
3.1.3.3.	Thiết kế cơ khí cho xe.....	46
3.2.	Lập trình tích hợp hệ thống	48
3.2.1.	Thiết lập serial node giao tiếp với Mobile base	48
3.2.2.	Xây dựng bản đồ (mapping).....	50
3.2.2.1.	Thu thập dữ liệu từ hai camera và chuyển dữ liệu về ảnh độ sâu..	51
3.2.2.2.	Chuyển ảnh độ sâu về kiểu dữ Laser scan.....	52
3.2.2.3.	Ước lượng tọa độ của robot từ laser scan:	53
3.2.2.4.	Thiết lập joytick để điều khiển Mobile base di chuyển.....	55
3.2.2.5.	Sử dụng package gmapping để dựng bản đồ	56
3.2.2.6.	Lưu bản đồ	57
3.2.2.7.	Tổng hợp các node trong dựng bản đồ	58
3.2.3.	Điều hướng trên bản đồ đã dựng (navigation)	61
3.2.3.1.	Các bước chuẩn bị dữ liệu cho navigation	61

3.2.3.2. Ước lượng vị trí của robot dùng Adaptive Monte Carlo Localization (AMCL).....	61
3.2.3.3. Thiết lập các thông số cấu hình để cung cấp cho move_base hoạt động	62
3.2.3.4. Thiết lập move_base để điều hướng cho xe	64
3.2.3.5. Tổng hợp lại các node được dùng trong điều hướng	66
Chương 4. KẾT QUẢ THỰC NGHIỆM.....	68
4.1. Lấy các dữ liệu cần thiết từ camera trên hệ điều hành ROS	68
4.2. Chuyển ảnh độ sâu thành laser scan:	69
4.3. Quá trình mapping và kết quả bản đồ.....	70
4.4. Quá trình điều hướng cho robot.....	74
Chương 5. KẾT LUẬN, ĐÁNH GIÁ, HƯỚNG PHÁT TRIỂN.....	79
5.1. Đánh giá.....	79
5.2. Kết luận.....	79
5.3. Hướng phát triển	80
TÀI LIỆU THAM KHẢO.....	81

DANH MỤC HÌNH VẼ

Hình 2.1 Một số robot được chế tạo trên nền tảng ROS.....	5
Hình 2.2 Cộng đồng phát triển ROS trên thế giới [1].....	6
Hình 2.3 Một số phiên bản ROS [2].....	7
Hình 2.4 Cấu trúc package trong ROS.....	8
Hình 2.5 Cấu trúc Stack trong ROS.....	9
Hình 2.6 Mô hình Server – Client trong ROS.....	9
Hình 2.7 Quan hệ giữa các node trong ROS.....	12
Hình 2.8 Một số phiên bản của ROS ở từng giai đoạn khác nhau [2].....	13
Hình 2.9 Mô hình Navigation stack trong ROS [3]	15
Hình 2.10 Bản đồ chi phí (<i>costmap</i>) trong ROS [4]	16
Hình 2.11 Giá trị chi phí ảnh hưởng đến hoạt động của robot [4].....	17
Hình 2.12 Nguyên lý hoạt động stereo camera từ nguồn sáng và một camera [5]	21
Hình 2.13 Nguyên lý hoạt động stereo camera [5]	22
Hình 2.14 Cấu trúc stereo camera [6].....	23
Hình 2.15 Sơ đồ nút và trọng số trên mỗi cạnh trong đồ thị [7].....	24
Hình 2.16 Hoạch định đường đi dùng thuật toán Dijkstra [9].....	27
Hình 2.17 Vận tốc cho phép V_a trong DWA [10].....	29
Hình 2.18 Vận tốc trong cửa sổ động V_d trong DWA [10]	29
Hình 2.19 Heading của robot trong DWA [10]	30
Hình 3.1 Máy tính Jetson TX2	32
Hình 3.2 Board phát triển được tích hợp máy tính nhúng Jetson TX2	33
Hình 3.3 Một số cảm biến laser find-ranger trên thị trường.....	35
Hình 3.4 Một số camera có tích hợp cảm biến đo khoảng cách dùng hồng ngoại	36
Hình 3.5 Stereo camera.....	36
Hình 3.6 Sơ đồ khối phần cứng.....	38

Hình 3.7 Mạch nguồn	38
Hình 3.8 Board Discovery STM32F4	39
Hình 3.9 Mạch đệm tín hiệu	39
Hình 3.10 Mạch nắn xung encoder	40
Hình 3.11 Module giao tiếp không dây CC1101	40
Hình 3.12 Hình ảnh mạch điện lớp trên	41
Hình 3.13 Hình ảnh mạch điện lớp dưới	41
Hình 3.14 Mạch sau khi hàn lắp linh kiện	42
Hình 3.15 Mạch sau khi được tích hợp lên khung xe	42
Hình 3.16 Giải thuật điều khiển PID	45
Hình 3.17 Chương trình điều khiển Mobile base	45
Hình 3.18 Mặt trước của khung xe thiết kế	46
Hình 3.19 Mặt sau của khung xe thiết kế	47
Hình 3.20 Mobile base khi hoàn thành thiết kế	48
Hình 3.21 Module joytick	55
Hình 3.22 Sơ đồ thiết lập joytick điều khiển Mobile base	56
Hình 3.23 Hình ảnh bản đồ sau khi dựng	57
Hình 3.24 Sơ đồ các node trong mapping	59
Hình 3.25 Cấu trúc frame trong mapping	60
Hình 3.26 Các node trong move_base	66
Hình 3.27 Sơ đồ frame trong move_base	67
Hình 4.1 Hình ảnh từ camera trái	68
Hình 4.2 Hình ảnh từ camera phải	68
Hình 4.3 Hình ảnh độ sâu tính toán được	69
Hình 4.4 Ảnh độ sâu thu được từ stereo camera	69
Hình 4.5 So sánh độ chính xác của laser scan so với point cloud	70
Hình 4.6 Kết quả laser scan thu được	70
Hình 4.7 Không gian thực tế để dựng bản đồ	71
Hình 4.8 Hình thành bản đồ khi di chuyển Mobile base	72

Hình 4.9 Lưu bản đồ đã dựng.....	72
Hình 4.10 Kết quả bản đồ sau khi dựng.....	73
Hình 4.11 Bản đồ sau khi loại bỏ nhiễu	73
Hình 4.12 Bản đồ trong môi trường có diện tích lớn hơn	74
Hình 4.13 Hoạch định đường đi cho robot.....	74
Hình 4.14 Robot đi theo đường đi đã hoạch định	75
Hình 4.15 Khi robot đến đích	75
Hình 4.16 Môi trường có vật cản	76
Hình 4.17 Robot phát hiện vật cản và cập nhật vào bản đồ.....	76
Hình 4.18 Hoạch lại đường đi cho phù hợp với vật cản vừa mới nhận diện được	77
Hình 4.19 Robot đến đích theo đường đi đã hoạch định khi có vật cản.....	77
Hình 4.20 Robot di chuyển theo đường đi đã hoạch định.....	78
Hình 4.21 Khi robot đến đích đã được thiết lập trước đó	78
Hình 4.22 Terminal thông báo là robot đã đến đích	78

DANH MỤC BẢNG

Bảng 2.1 File costmap_common_params.yaml	18
Bảng 2.2 File global_costmap_params.yaml	19
Bảng 2.3 File local_costmap_params.yaml	20
Bảng 2.4 File base_local_planner_params.yaml.....	21
Bảng 2.5 Thuật toán Dijkstra	26
Bảng 3.1 Thông số kỹ thuật board Jetson TX2.....	34
Bảng 3.2 Thông số kỹ thuật của ZED camera	37
Bảng 3.3 Khởi tạo serial node	49
Bảng 3.4 Hàm callback của serial node	50
Bảng 3.5 File launch cho serial node	50
Bảng 3.6 Các biến cần cấu hình cho zed_ros_wrapper.....	52
Bảng 3.7 Cấu hình các thông số cần thiết cho stereo camera	52
Bảng 3.8 Các thông số cần cấu hình cho depthimage_to_laserscan.....	53
Bảng 3.9 Các thông số cần cấu hình cho laser_scan_matcher.....	55
Bảng 3.10 Các thông số cần cấu hình cho gmapping	56
Bảng 3.11 Thông số bản đồ đã dựng.....	58
Bảng 3.12 Các thông số cấu hình cho amcl	61
Bảng 3.13 File costmap_common_params.yaml	62
Bảng 3.14 File global_common_params.yaml	63
Bảng 3.15 File local_common_params.yaml.....	63
Bảng 3.16 File dwa_local_planner_params.yaml cho mô hình thực tế.....	64
Bảng 3.17 Các thông số cấu hình cho move_base	65

DANH MỤC TỪ VIẾT TẮT

ROS	Robot Operating System	Hệ điều hành dành cho robot
-----	-------------------------------	-----------------------------

TÓM TẮT LUẬN VĂN

Đề tài được xây dựng trên nền tảng hệ điều hành dành cho robot (ROS – Robot Operating System), đây là một hệ điều hành mã nguồn mở được tích hợp rất nhiều công cụ, thư viện lớn của đông đảo các nhà khoa học, các công ty, phòng thí nghiệm trên khắp thế giới. ROS đang dần trở thành một nền tảng phổ biến và đang được sử dụng rộng rãi để nghiên cứu, chế tạo, phát triển robot.

Xe tự hành trong nhà hiện đang bắt đầu có nhiều ứng dụng trong công nghiệp trên các lĩnh vực khác nhau. Đây cũng là một hướng đi mang tính trọng tâm trên nền tảng ROS. Đây là quy trình thực hiện để hoàn thành mục tiêu của đề tài:

- Tìm hiểu hệ điều hành ROS.
- Nghiên cứu và mô phỏng quá trình xây dựng bản đồ và điều hướng trên bản đồ có sẵn trên Gazabo kết hợp với ROS.
- Thiết lập phần cứng di động (Mobile base) cho hệ thống dùng vi điều khiển STM32F4.
- Thiết lập hệ điều hành Ubuntu, ROS trên máy tính nhúng Nvidia Jetson TX2.
- Đọc cảm biến ZED camera (stereo camera) và giao tiếp với phần cứng di động.
- Tích hợp hệ thống để xây dựng bản đồ và điều hướng trên bản đồ đã dựng dựa trên hệ thống đã xây dựng.
- Hiệu chỉnh các thông số của gói ứng dụng đã thiết lập trong môi trường ROS.

Chương 1. TỔNG QUAN VỀ ĐỀ TÀI

1.1. Định hướng cho đề tài

Robot tự hành là robot có khả năng tự di chuyển, tự vận động theo một quỹ đạo được xác định trước hoặc chưa biết trước và được thực hiện một công việc được giao. Robot có thể hoạt động ở nhiều môi trường khác nhau như trên không, trên mặt đất, dưới nước hay thậm chí là ngoài vũ trụ. Đây là một hướng đi có tính ứng dụng rất lớn trên nhiều lĩnh vực từ dân sự, quân sự, vũ trụ và đang được nghiên cứu phát triển rộng rãi từ các công ty lớn, những phòng nghiên cứu ở khắp nơi thế giới. Một trong những yếu tố quan trọng của robot phải đảm bảo về mặt năng lượng cho nó hoạt động đủ lâu trong thời gian thực hiện nhiệm vụ và có thể tự giải quyết các vấn đề nằm ngoài dự tính như khả năng nhận diện, phát hiện và tránh vật cản hoặc đi vào môi trường chưa biết.

Robot tự hành được thiết kế dựa trên hệ điều hành dành cho robot (Robot Operating System, là nền tảng cho việc thiết kế phần mềm dành cho robot). Bằng việc ứng dụng ROS cho robot, ta có thể điều khiển robot một cách dễ dàng hơn so với việc chúng ta tự xây dựng thuật toán điều khiển cho robot, giúp cho robot có thể tự định vị và xử lý được các tình huống trong môi trường thực tế.

Để robot có thể tự định hướng và xác định vị trí hiện tại, ta cần phải mô phỏng lại bản đồ khoảng không gian di chuyển của robot – việc này ta gọi là mapping. Từ việc có được bản đồ của khoảng không gian di chuyển, ta cần phải xác định vị trí của nó trong khoảng không gian ấy – việc này gọi là localization. Sau những công việc trên, ta sẽ xác định đường đi cho robot và điều khiển để robot đi theo đúng quỹ đạo của đường đi mà ta sẽ vẽ - việc này gọi là path-planning. Bài toán cho vấn đề trên thực hiện dựa trên bài toán **Simultaneous Localization and Mapping** – SLAM đang được nghiên cứu rộng rãi và tạo ra một cộng đồng phát triển rất mạnh.

Để xây dựng một hệ thống hoàn thiện thì đây là một đề tài khá rộng và cần có những chuyên gia có kiến thức, kinh nghiệm để có thể phát triển một hệ thống ổn định, hoạt động trôi chảy trên nhiều điều kiện môi trường khác nhau. Vì tính ứng

dụng rộng lớn và tính phức tạp của vấn đề, luận văn chỉ ở mức tìm hiểu thuật toán và ứng dụng các thuật toán cơ bản để có thể xây dựng được mô hình hoạt động hoàn thiện.

1.2. Mục tiêu của đề tài

- Hiểu được cơ bản các thuật toán trong điều hướng cho xe tự hành dựa trên bản đồ đã dựng từ môi trường thực tế.
- Thiết lập được một robot di động để nhận tín hiệu điều khiển và thực thi đúng với yêu cầu do phần mềm điều hướng điều khiển.
- Ứng dụng được chương trình phần mềm trên máy tính nhúng và dùng stereo camera để xây dựng bản đồ và phát hiện vật cản.
- Hiểu và thực hiện các quá trình để xây dựng một xe tự hành hoàn chỉnh từ công việc dựng bản đồ, hoạch định đường đi ngắn nhất, đi theo đường đi đã hoạch định đồng thời phát hiện và tránh các vật cản.

1.3. Đối tượng và phạm vi nghiên cứu

- Thiết xe tự hành có thể mang được máy tính nhúng, stereo camera và hoạt động trong môi trường trong nhà.
- Nghiên cứu thuật toán hoạch định đường đi ngắn nhất và di chuyển theo đường đi đã hoạch định.
- Thiết lập và xây dựng ứng dụng dựa trên các gói phần mềm có sẵn trên cộng đồng ROS. Bên cạnh đó, tự xây dựng các gói phần mềm khác để phù hợp với yêu cầu bài toán của luận văn đưa ra.

1.4. Các bước thực hiện đề tài

- Tìm hiểu hệ điều hành ROS.
- Nghiên cứu và mô phỏng quá trình xây dựng bản đồ và điều hướng trên bản đồ có sẵn thông qua Gazabo kết hợp với ROS.

- Thiết lập phần cứng di động cho hệ thống dùng vi điều khiển STM32F4.
- Thiết lập hệ điều hành Ubuntu, ROS trên máy tính nhúng Nvidia Jetson TX2.
- Đọc cảm biến ZED camera (stereo camera) và giao tiếp với phần cứng di động.
- Tích hợp hệ thống để xây dựng bản đồ và điều hướng trên bản đồ đã dựng dựa trên hệ thống đã xây dựng.
- Hiệu chỉnh các thông số của gói ứng dụng đã thiết lập trong môi trường ROS.

Chương 2. TỔNG QUAN VỀ ROBOT OPERATING SYSTEM VÀ CƠ SỞ LÝ THUYẾT

2.1. Giới thiệu về Robot Operating System

2.1.1. Tổng quan về hệ điều hành dành cho robot – ROS

Robot Operating System – ROS là hệ điều hành mã nguồn mở dành cho robot, là một framework được dùng rất rộng rãi trong lĩnh vực robotic với nhiều ưu điểm. Nó tạo ra một nền tảng phần mềm có thể hoạt động trên rất nhiều robot khác nhau mà không cần sự thay đổi quá nhiều trong chương trình phần mềm.

ROS được bắt đầu với ý tưởng tạo ra sự thuận tiện là có thể chia sẻ dễ dàng và có thể được sử dụng lại trên những phần cứng robot khác nhau mà không cần phải xây dựng lại từ đầu. Việc xây dựng lại từ đầu cho một nền tảng robot riêng biệt sẽ tốn rất nhiều thời gian và công sức; bên cạnh đó, việc ứng dụng lại những thành quả của việc nghiên cứu trước đó để xây dựng những thuật toán cao hơn cũng gặp rất nhiều khó khăn.

Từ những lợi ích của ROS mang lại, những tổ chức nghiên cứu phát triển trên thế giới về ROS góp phần mang lại nguồn thông tin dồi dào về cả phần cứng lẫn phần mềm. Trên thế giới, cũng có rất nhiều công ty và tổ chức đã bắt đầu ứng dụng nền tảng này vào ứng dụng của họ để tạo ra những sản phẩm có chất lượng tốt hơn. Nhờ vào đó, hiện nay trên thế giới có rất nhiều thiết bị được hỗ trợ framework này.



Hình 2.1 Một số robot được chế tạo trên nền tảng ROS

ROS cung cấp, hỗ trợ các dịch vụ (services) như một hệ điều hành như phần cứng trừu tượng (hardware abstraction), kiểm soát các thiết bị cấp thấp (low-level

device control), thực thi (implementation) các chức năng phổ biến, tin nhắn (message) qua lại giữa các quá trình, và quản lý gói. Ngoài ra, nó cũng cung cấp nhiều công cụ và thư viện cho việc tham khảo, biên dịch, viết và chạy chương trình trên nhiều máy khác nhau.

Việc chạy các chuỗi quy trình (processes) dựa trên ROS được thể hiện dưới kiến trúc graph, biểu diễn mối quan hệ của các thành phần trong hệ điều hành.

ROS thực thi một số loại giao tiếp khác nhau như giao tiếp kiểu RPC dạng đồng bộ thông qua services, truyền dữ liệu bất đồng bộ thông qua topics và lưu trữ dữ liệu trên Parameter Server.

ROS không phải là một framework thời gian thực, nhưng thông qua nó ta có thể viết chương trình thời gian thực.

ROS là một hệ điều hành mã nguồn mở nên thu hút sự quan tâm của cộng đồng, đồng nghĩa với các công cụ, thư viện sẽ được xây dựng và phát triển phong phú. Hiện nay, mô hình robot đã và đang được xây dựng trên hệ điều hành này với nhiều ứng dụng có tính thực tiễn cao.



Hình 2.2 Cộng đồng phát triển ROS trên thế giới [1]

Hiện nay, ROS chỉ chạy trên nền tảng Unix. Các phần mềm được chạy trên ROS chủ yếu được thử nghiệm trên hệ điều hành Ubuntu và Mac OS X. Bên cạnh đó,

thông qua cộng đồng này, nó còn đang được xây dựng để hỗ trợ cho các nền tảng khác như Fedora, Gentoo, Arch Linux và các nền tảng Linux khác.

Chương trình cốt lõi của ROS, các công cụ tiện ích và các thư viện cũng được phát hành các phiên bản mới được gọi là ROS Distribution. Những Distribution này giống như Linux Distribution và đồng thời cung cấp chuỗi các phần mềm tương thích.



Hình 2.3 Một số phiên bản ROS [2]

Khi ROS được ứng dụng cho robotic thì khối lượng công việc kỹ thuật cơ bản sẽ giảm, bên cạnh đó khối lượng công việc dành cho xây dựng hệ thống tăng một cách đáng kể. Do đó, chúng ta có thể dành thời gian cho việc nghiên cứu các ứng dụng chuyên sâu, đạt hàm lượng khoa học cao hơn trong dự án.

2.1.2. Cấu trúc ROS

Kiến trúc ROS có ba cấp khái niệm: Filesystem, Computation Graph và Community.

Cấp thứ nhất – Filesystem: giải thích về các dạng hình thức bên trong, cấu trúc thư mục và các tập tin tối thiểu để ROS hoạt động. Nó chủ yếu là các tài nguyên của ROS và được thực hiện trên đĩa cứng.

Cấp thứ hai – Computation Graph: nơi giao tiếp giữa các quá trình và hệ thống. Trong cấp khái niệm này, ta sẽ phải thiết lập hệ thống, quản lý các quá trình, giao tiếp giữa nhiều máy tính với nhau,...

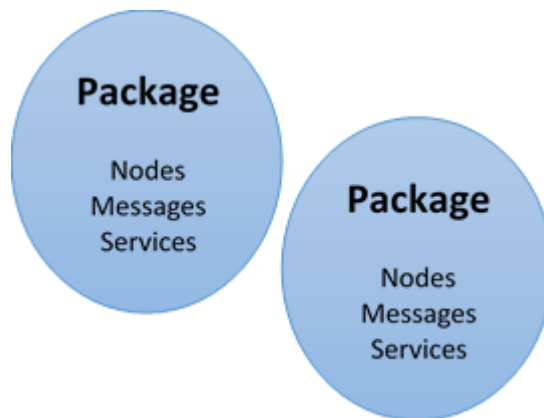
Cấp thứ ba – Community: giải thích/ hướng dẫn các công cụ và các khái niệm để chia sẻ kiến thức, thuật toán chương trình từ bất kỳ nhà phát triển nào. Đây là cấp độ quan trọng vì nó ảnh hưởng đến sự phát triển lớn mạnh của cộng đồng ROS.

2.1.2.1. ROS Filesystem Level

Filesystem chủ yếu là các nguồn tài nguyên ROS được thực thi trên bộ nhớ lưu trữ hệ thống, bao gồm:

Packages: là đơn vị chính để tổ chức phần mềm trên ROS. Một package có thể chứa các nodes (ROS runtime processes), các thư viện đặc thù của ROS, các file cài đặt hoặc bất cứ file nào cho việc tổ chức.

Mục đích của package là để tạo ra tập hợp chương trình có kích thước nhỏ nhất để có thể dễ dàng sử dụng lại.

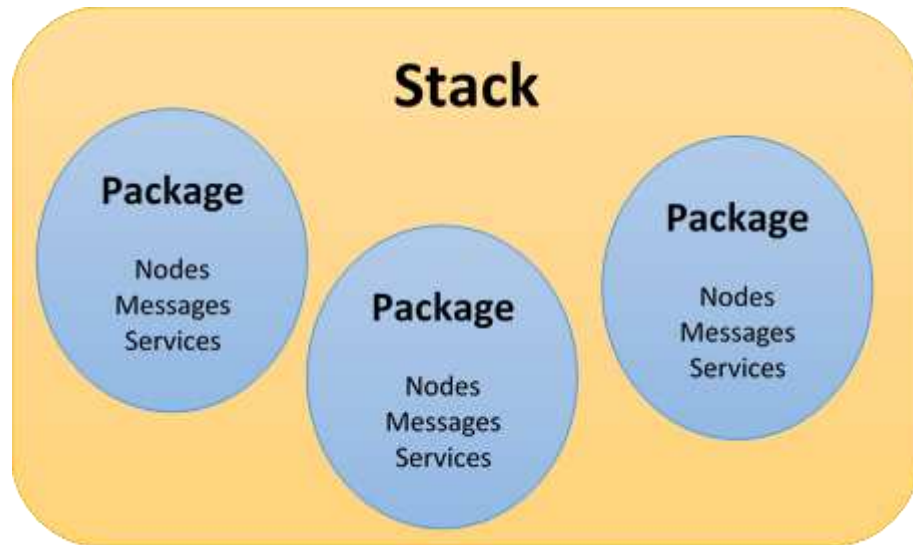


Hình 2.4 Cấu trúc package trong ROS

Metapackages: là một package chuyên biệt, dùng để phục vụ cho việc thể hiện mối quan hệ một nhóm các package khác với nhau. Metapackage thường được dùng như nơi để giữ các tương thích ngược cho việc chuyển đổi sang rosbuilt Stacks.

Package Manifests: là một bảng mô tả về một package như tên, version, mô tả, thông tin license,... Manifests được quản lý bởi một file tên là manifests.xml.

Stacks: khi chúng ta kết hợp các package với nhau với một vài chức năng thì được gọi là Stack. Trong ROS, có rất nhiều stack với công dụng khác nhau. Tương tự Package, nơi chứa thông tin về stack gọi là Stack Manifests.

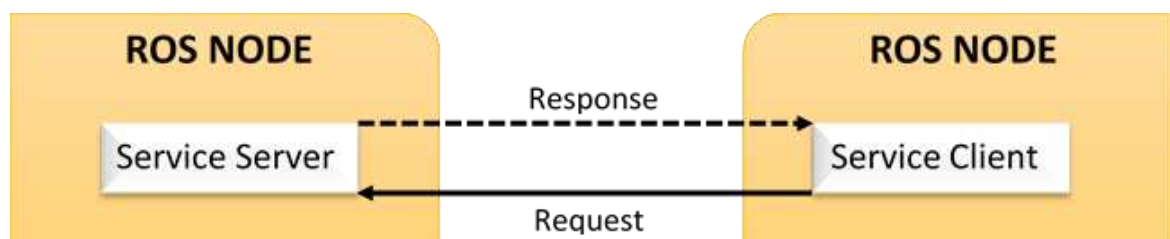


Hình 2.5 Cấu trúc Stack trong ROS

Mục tiêu của việc dùng Stack là để đơn giản hóa cho quá trình chia sẻ chương trình ứng dụng.

Message types: là mô tả của một thông điệp được gửi qua lại giữa các quá trình, được lưu trữ dưới dạng `my_package/msg/MyMessageType.msg`. Message định nghĩa cấu trúc dữ liệu cho các thông điệp được gửi đi. Trong ROS, có rất nhiều loại message tiêu chuẩn phục vụ cho quá trình giao tiếp giữa các node với nhau. Ngoài ra, ta cũng có thể tự định nghĩa lại một kiểu message theo nhu cầu sử dụng của chúng ta.

Service types: là mô tả một service, được lưu trữ dưới dạng `my_package/-msg/MyServiceType.srv`. Service định nghĩa cấu trúc dữ liệu request và response giữa các node trong ROS. Để gọi một service, ta cần phải sử dụng tên của service cùng với tên của package chứa service đó.



Hình 2.6 Mô hình Server – Client trong ROS

2.1.2.2. ROS Computation Graph Level

Computation Graph là một mạng nơi các quy trình trong ROS được kết nối với nhau. Bất kỳ một node nào trong hệ thống cũng có thể truy cập vào mạng này, tương tác với các node khác, trao đổi các dữ liệu nằm trong mạng. Các khái niệm cơ bản của Computation Graph là nodes, Master, Parameter Server, messages, services, topics và bags.

Nodes: là một quy trình dùng để tính toán, điều khiển. Một node có thể được tạo ra khi biên dịch một package thành công và trong cùng một package có thể tạo nhiều node. Khi một node muốn giao tiếp, tương tác các node khác thì bản thân node đó phải được kết nối với mạng ROS. Trong một hệ thống, mỗi node sẽ có một chức năng khác nhau.

Để tạo ra sự dễ dàng trong quản lý và chuyên biệt hóa trong một chương trình, chúng ta nên tạo ra nhiều node với mỗi node có một chức năng riêng biệt; không nên để một node quản lý quá nhiều tác vụ sẽ gây khó khăn trong lập trình và bảo trì.

Master: ROS Master cung cấp một tên đăng ký và tra cứu phần còn lại của Computation Graph. Nếu không có ROS Master thì các node không thể tìm thấy nhau, trao đổi các message hay gọi các service.

Parameter Server: Parameter Server cho phép dữ liệu được lưu trữ bởi các từ khóa trong một vị trí trung tâm. Nó là một phần của Master.

Với các biến này, nó có thể được cấu hình các node trong khi nó đang hoạt động hoặc để thay đổi hoạt động của node.

Messages: các node giao tiếp với nhau thông qua messages. Một message đơn giản là một cấu trúc dữ liệu, bao gồm các trường được định nghĩa như integer, floating point, boolean,... Messages có thể bao gồm các kiểu cấu trúc và mảng lồng nhau (giống kiểu struct trong C). Chúng ta cũng có thể tự phát triển kiểu message dựa trên các message chuẩn.

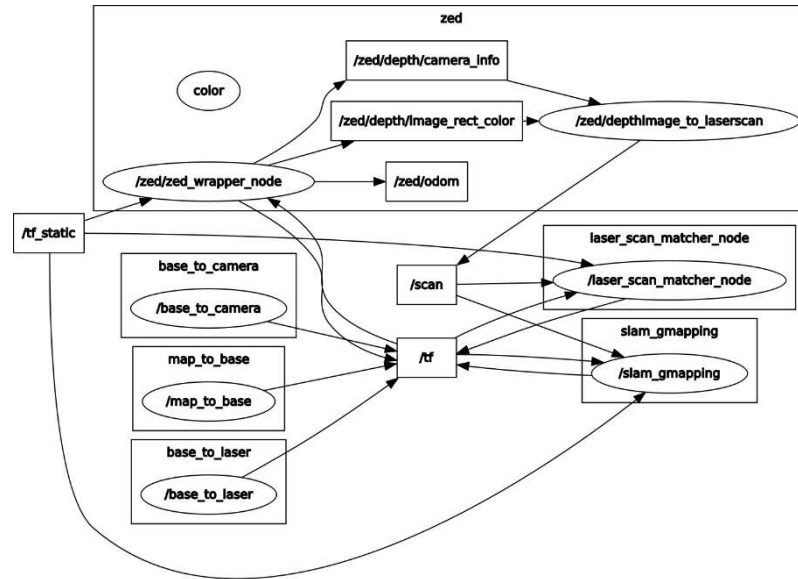
Topics: messages được định tuyến thông qua hệ thống vận chuyển, trong đó được phân loại thành publish và subscribe. Một node sẽ gửi một messages bằng việc publishing message đó lên một topic định trước. Topic chỉ là một cái tên để nhận dạng nội dung của message. Một node chỉ có thể subscribe một topic có tên và kiểu dữ liệu được khai báo. Trong cùng một thời điểm, có thể nhiều publishers và subscribers cùng truy cập vào cùng một topic, và một node có thể publish và subscribe nhiều topic. Nhìn chung, publishers và subscriber sẽ không thể nhận thức được sự tồn tại của nhau. Ý tưởng xây dựng trao đổi thông tin ở ROS nhằm tách rời giữa nguồn tạo thông tin và nơi sử dụng thông tin.

Tên của topic phải là duy nhất để tránh các vấn đề và xung đột giữa các topic có cùng tên với nhau.

Services: mô hình publish/subscribe thì rất linh hoạt trong việc giao tiếp nhưng đặc điểm là truyền được đa đối tượng và một chiều. Nhưng đôi khi lại không thích hợp cho việc truyền theo dạng request/reply, thường được dùng trong kiểu hệ thống phân bổ. Do đó, việc truyền nhận theo dạng request/reply được dùng thông qua services. Service được định nghĩa một cặp cấu trúc dữ liệu: một cho request và một cho reply. Một node cung cấp một service thông qua một thuộc tính name, và một client sử dụng service bằng việc gửi một request message và đợi phản hồi.

Tương tự như message, service phải cần có một tên duy nhất trong mạng ROS để tránh những lỗi không mong muốn xảy ra.

Bags: là một định dạng để lưu trữ và phát lại dữ liệu message ROS. Bags là một cơ chế quan trọng cho việc lưu trữ dữ liệu; ví dụ như giá trị của các cảm biến rất khó để thu thập cho sự nghiên cứu, phát triển và kiểm tra thuật toán. Vì thế việc dùng bags là rất quan trọng trong việc phát triển robot, đặc biệt là những robot mang tính phức tạp cao.













Hình 2.7 Quan hệ giữa các node trong ROS

Hình 2.7 trên cho ta thấy mối quan hệ giữa các node bên trong ROS, node nào publish dữ liệu cho topic nào và node nào subscribe topic nào.

2.1.2.3. ROS Community Level

Các khái niệm về ROS Community Level là các nguồn tài nguyên của ROS được cộng đồng người dùng trao đổi với nhau về phần mềm và kiến thức. Các nguồn tài nguyên đó là:

Distributions: là tổng hợp các phiên bản của stack mà chúng ta có thể cài đặt. ROS Distributions có vai trò tương tự như Linux Distributions.

Distro	Release date	Poster	Tuturtle, turtle in tutorial	EOL date
ROS Melodic Morenia	May, 2018	TBD	TBD	May, 2023
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame (Recommended)	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017
ROS Indigo Igloo	July 22nd, 2014			April, 2019 (Trusty EOL)
ROS Hydro Medusa	September 4th, 2013			May, 2015

Hình 2.8 Một số phiên bản của ROS ở từng giai đoạn khác nhau [2]

Respositories: là nguồn tài nguyên dựa trên cộng đồng mạng lưới các tổ chức khác nhau phát triển và phát hành những mô hình riêng của họ.

The ROS Wiki: gồm nhiều tài liệu về ROS. Bất cứ ai cũng có thể chia sẻ tài liệu, cung cấp các bản cập nhật, viết các bài hướng dẫn,... bằng tài khoản mà họ đã đăng ký.

2.1.3. 2D Navigation stack

2D Navigation stack dùng để lấy các thông tin, dữ liệu từ odometry, cảm biến (sensor), điểm mục tiêu (goal pose) và xuất ra tín hiệu vận tốc gửi xuống robot.

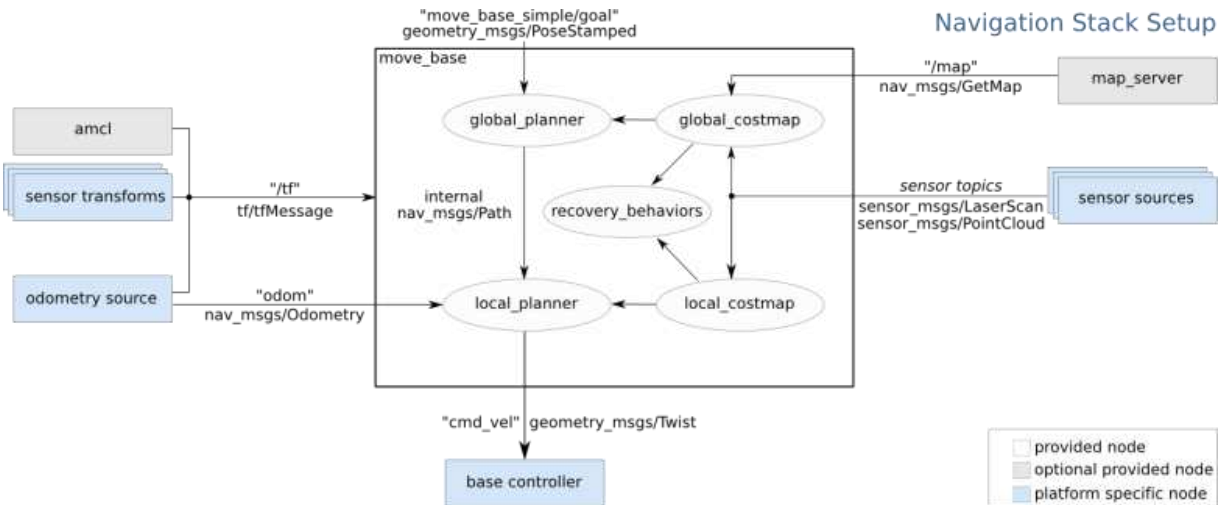
Việc sử dụng Navigation stack trên một robot bất kỳ thì khá là phức tạp. Điều kiện tiên quyết để sử dụng Navigation stack là robot phải được chạy trên ROS, phải

có một tf transform tree, và publish dữ liệu của cảm biến theo đúng kiểu Message trong ROS. Thứ hai, Navigation stack cần được cấu hình về hình dạng và đặc tính của robot để có thể hoạt động ở một cấp độ cao hơn.

Mặc dù Navigation stack được thiết kế sao cho dùng trong mục đích chung nhất có thể, nhưng vẫn có một số yêu cầu bắt buộc về phần cứng như sau:

- Nó được thiết kế dành cho cả mô hình robot lái bằng hiệu tốc độ (differential drive) và mô hình robot có bánh xe đa hướng (holonomic wheeled robots). Nó chỉ việc gửi trực tiếp giá trị vận tốc xuống cho bộ phận điều khiển di chuyển của robot để đạt được các giá trị của robot mong muốn như vận tốc theo trục x, vận tốc theo trục y và vận tốc xoay.
- Navigation stack cần phải được lắp một cảm biến laser scan được gắn lên robot. Laser scan này dùng để xây dựng bản đồ (mapping), định vị (localization) và phát hiện vật cản.
- Nó được phát triển trên những robot có hình vuông vì thế để được hiệu năng tốt nhất thì hình dạng robot nên là hình vuông hoặc hình tròn. Navigation stack cũng làm việc được trên robot có hình dạng và kích thước bất kỳ nhưng sẽ khó khăn với những robot có kích thước lớn khi làm việc trong những không gian hẹp.

2.1.3.1. Tổng quan về Navigation stack

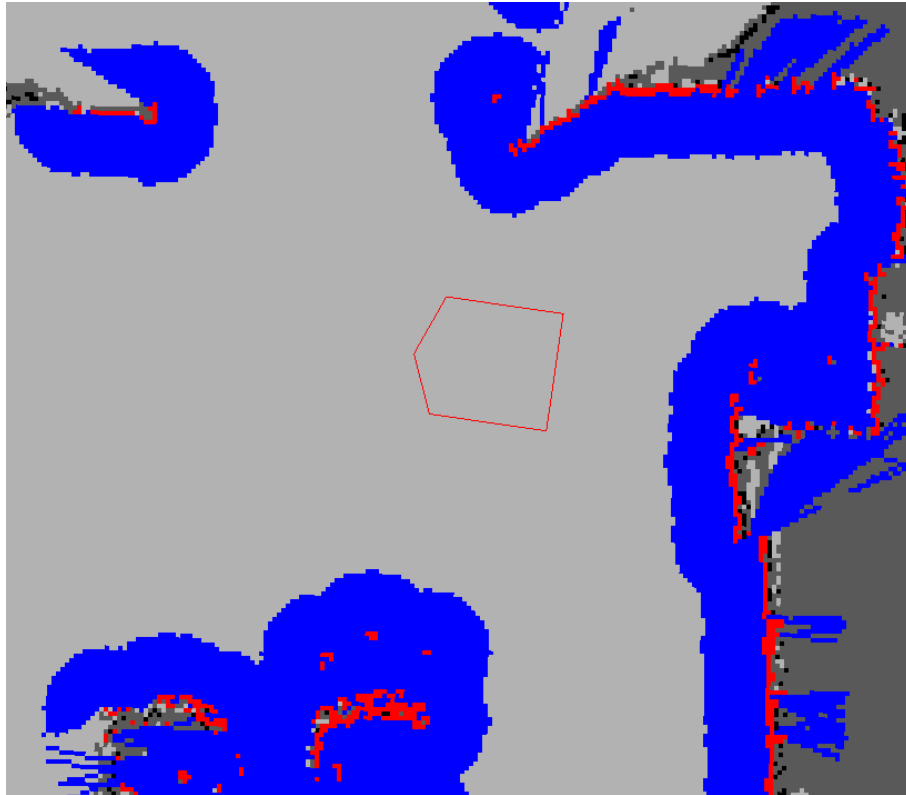


Hình 2.9 Mô hình Navigation stack trong ROS [3]

Hình 2.9 trên cho ta thấy được cái nhìn tổng quan về cách cài đặt cũng như những thành phần có trong Navigation stack. Những thành phần bên trong **move_base** (màu trắng) là những thành phần bắt buộc phải có. Những thành phần như **amcl**, **map_server** (màu xám) là những thành phần tùy chọn. Và những thành phần còn lại (màu xanh) là những thành phần phải có tùy thuộc vào robot mà ta thiết kế.

2.1.3.2. Costmap 2D

Costmap 2D là một package để triển khai bản đồ chi phí (**costmap**) từ việc lấy dữ liệu của cảm biến từ môi trường, xây dựng một lưới các nơi bị chiếm chỗ và tăng vùng chi phí trong bản đồ chi phí 2D dựa trên lưới các nơi bị chiếm chỗ và bán kính tăng chi phí do người dùng định nghĩa.

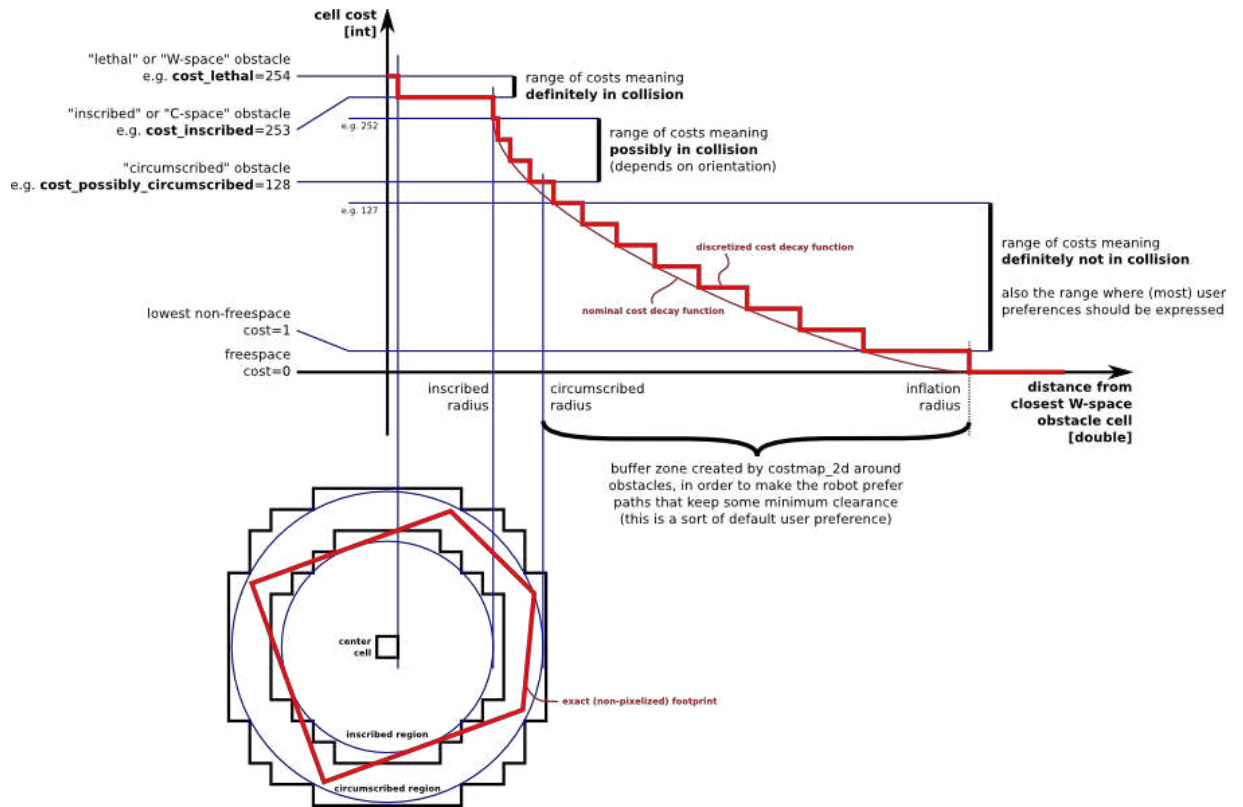


Hình 2.10 Bản đồ chi phí (*costmap*) trong ROS [4]

Costmap 2D dùng để chứa thông tin các vật cản cũng như những nơi mà robot không thể tới được. Costmap sử dụng dữ liệu của các cảm biến từ bản đồ đã được xây dựng trước đó và cập nhật thông tin của vật cản vào *costmap*.

Các phương pháp cơ bản sử dụng để ghi dữ liệu lên *costmap* được cấu hình đầy đủ, mỗi bit chức năng nằm trên một lớp khác nhau. Ví dụ, bản đồ tĩnh (bản đồ đã được dựng trước đó) được nằm trên một lớp, và các vật cản được nằm trên một lớp khác. Lớp chứa vật cản lưu trữ thông tin các vật thể theo ba hướng nhằm cho vấn đề xử lý các vật cản trong bản đồ thông minh hơn.

Costmap tự động đăng ký nhận dữ liệu từ các topic của cảm biến thông qua ROS và cập nhật các thông tin cho chính *costmap* này một cách thích hợp. Mỗi cảm biến được sử dụng để đánh dấu các vật cản hoặc loại bỏ các vật cản trong costmap. Nếu các vật thể được lưu trữ thông tin ở dạng ba chiều thì được chuyển thành hai chiều khi đặt vào *costmap*.



Hình 2.11 Giá trị chi phí ảnh hưởng đến hoạt động của robot [4]

Hình 2.11 trên chỉ cho chúng ta thấy được giá trị chi phí ảnh hưởng đến hoạt động của robot như thế nào.

- **Cost_lethal** là giá trị của một cell chứa vật cản. Nếu trọng tâm của robot đi vào vùng này thì chắc chắn robot sẽ bị va chạm.
- **Cost_inscribed** là giá trị của một cell từ vật cản đến bán kính nội tiếp của robot. Nếu tâm của robot nằm trong vùng này thì robot chắc chắn sẽ bị va chạm vào vật cản.
- **Cost_possibly_circumscribed** là giá trị của chi phí tương tự như **cost_inscribed** nhưng đó là giá trị từ vật cản đến bán kính ngoại tiếp của robot. Nếu tâm của robot đi vào vùng này thì khả năng robot va chạm vật cản phụ thuộc vào hướng đi của robot.
- **Cost_freespace** có giá trị bằng 0, thể hiện rằng không có vật cản nơi này; robot có thể di chuyển tự do trong vùng này.

- **Cost_unknown** là những cell chưa có thông tin về giá trị của chi phí.

2.1.3.3. Cách cấu hình costmap

Để bắt đầu cấu hình, chúng ta cần biết mỗi costmap được dùng để làm gì. Robot sẽ di chuyển trong map với hai định hướng đường đi là toàn cục (global) và cục bộ (local): định hướng đường đi toàn cục (global navigation) được dùng để tạo đường đi cho mục tiêu trong bản đồ hoặc đường đi có khoảng cách xa; định hướng đường đi cục bộ (local navigation) được dùng để tạo đường đi trong khoảng cách gần và để tránh vật cản khi robot di chuyển.

Các module này sử dụng costmap để giữ thông tin về bản đồ. Định hướng đường đi toàn cục thì dùng costmap toàn cục (global costmap) và costmap cục bộ (local costmap) dùng cho định hướng cục bộ. Mỗi **costmap** sẽ có những thông số riêng của nó và cũng có những thông số dùng chung. Chúng được cấu hình dựa vào các file: `costmap_common_params.yaml`, `global_costmap_params-.yaml`, `local_costmap_params.yaml`.

<pre>obstacle_range: 2.5 raytrace_range: 3.0 footprint: [[-0.2, -0.2],[-0.2, 0.2],[0.2, 0.2],[0.2, -0.2]] inflation_radius: 0.3 observation_sources: laser_scan_sensor laser_scan_sensor: {sensor_frame: laser_base_link, data_type: LaserScan, topic: /base_scan/scan, marking: true, clearing: true}</pre>
--

Bảng 2.1 File `costmap_common_params.yaml`

Đoạn chương trình trên nằm trong file `costmap_common_params.yaml` dùng để cấu hình các biến dùng chung cho cả `local_costmap` và `global_costmap`.

Thuộc tính `obstacle_range` và được dùng để chỉ khoảng cách tối đa mà cảm biến có thể đọc và sinh ra thông tin mới về môi trường trong costmap. Nếu robot phát hiện ra vật cản có khoảng cách nhỏ hơn `obstacle_range` thì nó sẽ đặt một vật cản

vào costmap. Nếu robot di chuyển, nó có thể xóa *costmap* và cập nhật freespace. Lưu ý rằng chúng ta chỉ có thể cập nhật các giá trị được trả về từ thông tin laser trên bề mặt vật cản mà không thể nhận biết được toàn bộ vật cản. Nhưng với những thông tin này, ta đã có thể xây dựng được bản đồ.

Thuộc tính `footprint` dùng để chỉ cho Navigation stack nhận biết được hình thái của robot. Các thông số này dùng để giữ khoảng cách giữa vật cản và robot để không xảy ra sự va chạm hoặc robot có thể di chuyển qua những nơi có khoảng cách hẹp.

Thuộc tính `inflation_radius` là thông số để giữ khoảng cách nhỏ nhất giữa tâm của robot và vật cản.

Thuộc tính `laser_scan_sensor` dùng để khai báo các thông tin về loại cảm biến được dùng (ở đây là dùng laser scan) và được dùng để thêm/xóa vật cản trong costmap.

```
global_costmap:
  global_frame: /map
  robot_base_frame: /base_footprint
  update_frequency: 1.0
  static_map: true
```

Bảng 2.2 File `global_costmap_params.yaml`

Các thông tin trên là dùng để khai báo các biến cho file `global_costmap_params.yaml`.

Thuộc tính `global_frame` và `robot_base_frame` định nghĩa các frame toàn cục và frame gắn với Mobile base.

Thuộc tính `update_frequency` dùng để khai báo tần số cập nhật cho costmap, ở đây giá trị là 1Hz.

Thuộc tính `static_map` để dùng khai báo có hay không dùng bản đồ đã dựng trước đó.

```

local_costmap:

  global_frame: /map

  robot_base_frame: /base_footprint

  update_frequency: 1.0

  publish_frequency: 2.0

  static_map: true

  rolling_window: false

  width: 10.0

  height: 10.0

  resolution: 0.1

```

Bảng 2.3 File local_costmap_params.yaml

Các thuộc tính `global_frame`, `robot_base_frame`, `update_frequency` và `static_map` giống với file `global_costmap_params.yaml`.

Thuộc tính `publish_frequency` để xác định tần số cập nhật thông tin.

Thuộc tính `rolling_window` dùng để giữ costmap nằm ở tâm của robot khi di chuyển.

Các thuộc tính `width`, `height` và `resolution` để cấu hình khoảng cách và độ phân giải của costmap. Các thông số này có đơn vị là mét.

2.1.3.4. Cấu hình base_local_planner

Cấu hình này dùng để khai báo các thông số khi tạo ra các giá trị của vận tốc gửi xuống cho robot di chuyển.

```

TrajectoryPlannerROS:

  max_vel_x: 1

  min_vel_x: 0.5

  max_rotational_vel: 1.0

  min_in_place_rotational_vel: 0.4

  acc_lim_th: 3.2

```

```

acc_lim_x: 2.5

acc_lim_y: 2.5

holonomic_robot:false

```

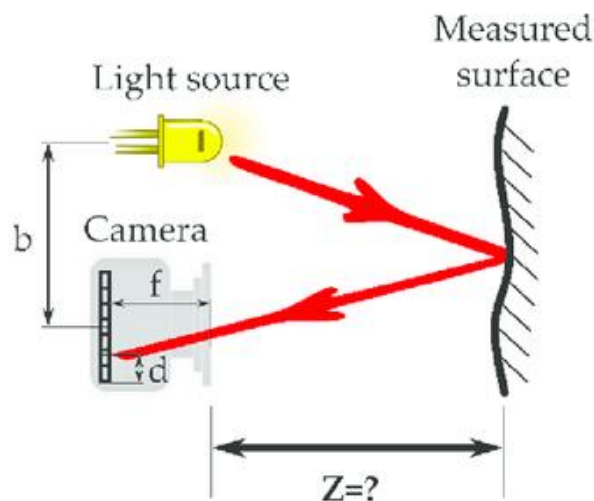
Bảng 2.4 File `base_local_planner_params.yaml`

Cấu hình trên được lưu lại trong file `base_local_planner_params.yaml` dùng để thiết lập vận tốc tối đa/tối thiểu và gia tốc của robot mà chúng ta cần thiết lập để phù hợp với phần cứng thiết kế.

Thuộc tính `holonomic_robot` là biến khai báo xem chúng ta có xây dựng robot có cấu trúc di chuyển dựa trên phần cứng có mô hình holonomic hay không. Nếu chúng ta xây dựng robot không dựa trên nền tảng này thì khai báo là `false`.

2.2. Nguyên lý tính khoảng cách vật thể từ stereo camera

Stereo camera là một cặp gồm hai camera giống nhau về các thông số vật lý cấu hình nên camera. Tương tự như với mắt người, stereo camera có thể cho được ảnh với không gian ba chiều từ môi trường xung quanh.

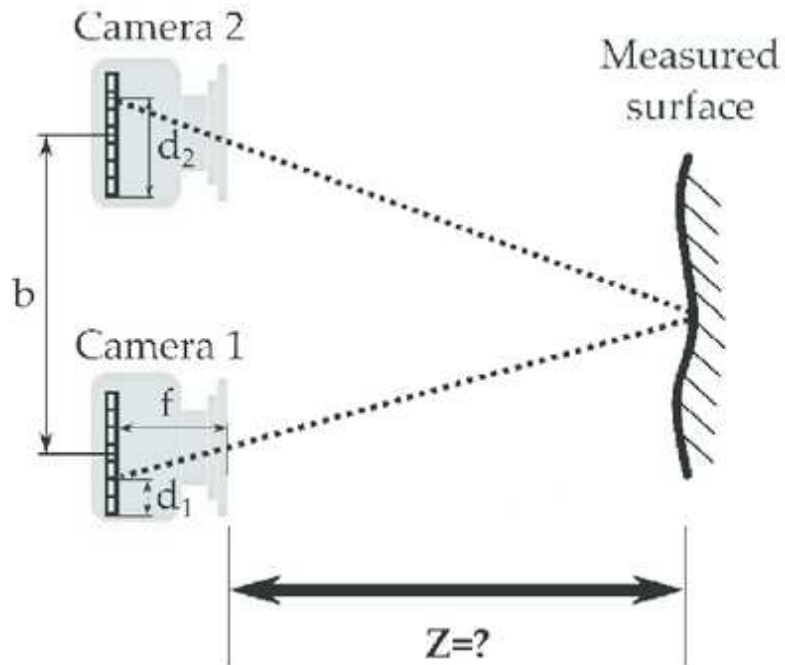


Hình 2.12 Nguyên lý hoạt động stereo camera từ nguồn sáng và một camera [5]

Giả sử, ta có một nguồn sáng được đặt ngang với camera cách một đoạn là b , thấu kính camera có tiêu cự là f và nguồn sáng nằm trên ảnh có khoảng cách là d . Từ đó, ta có thể dễ dàng tính được khoảng cách từ camera đến mặt phẳng cần đo là

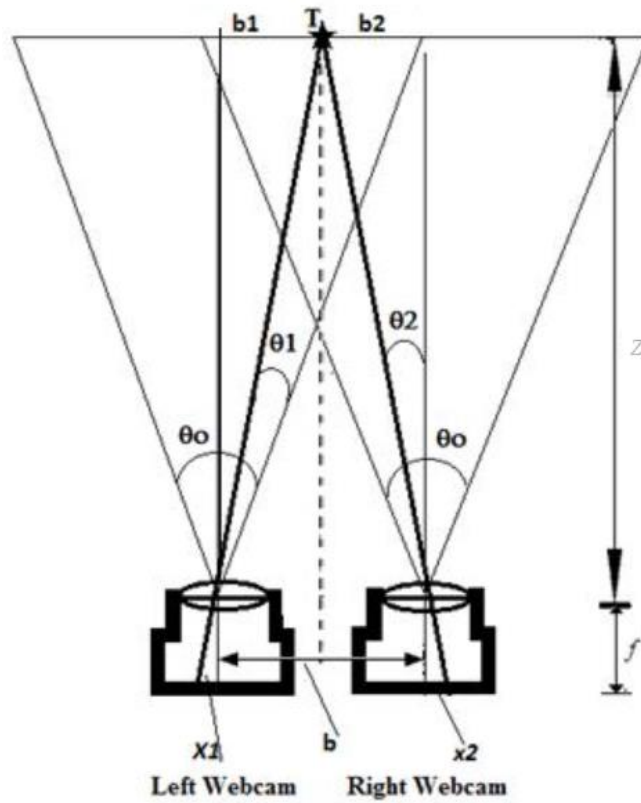
$$Z = \frac{bf}{d}$$

Nếu ta thay nguồn sáng đó thành một camera khác thì ta có sơ đồ như sau



Hình 2.13 Nguyên lý hoạt động stereo camera [5]

Như giả sử ban đầu, ta có hai camera có cùng các thông số vật lý như tiêu cự, góc nhìn,... Để dễ dàng tính toán, ta xét ảnh sau:



Hình 2.14 Cấu trúc stereo camera [6]

Dựa vào tam giác đồng dạng, ta có:

$$\frac{b_1}{Z} = \frac{-x_1}{f}, \frac{b_2}{Z} = \frac{x_2}{f}$$

Do $b = b_1 + b_2$ nên

$$b = \frac{Z}{f}(x_2 - x_1) \Rightarrow Z = \frac{bf}{x_2 - x_1} = \frac{bf}{d_2 - d_1} \quad (2.1)$$

Xét một trong hai máy ảnh trên, ta có góc nhìn của ảnh là θ_0 và X là bề rộng của ảnh nên suy ra được:

$$f = \frac{X}{2 \tan\left(\frac{\theta_0}{2}\right)} \quad (2.2)$$

Thay biểu thức (2.2) vào (2.1), ta được khoảng cách từ camera đến vật thể theo công thức:

$$Z = \frac{bX}{2(x_2 - x_1)\tan\left(\frac{\theta_0}{2}\right)}$$

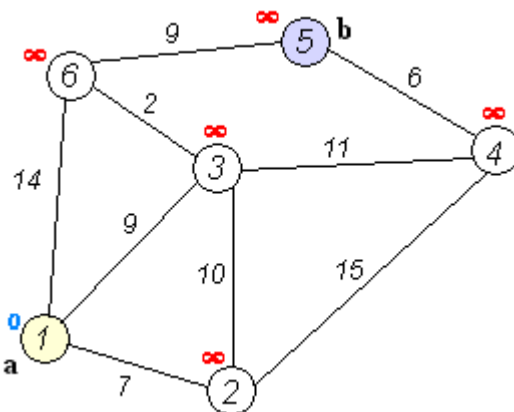
Trong đó:

- b là khoảng cách giữa hai tâm của hai camera (đơn vị là mét).
- X là chiều rộng của ảnh (đơn vị pixel).
- x_1, x_2 lần lượt là tọa độ của vật thể trên ảnh của camera trái và camera phải (đơn vị pixel).

2.3. Thuật toán Dijkstra cho global planner

Trong điều hướng cho robot để robot đến được vị trí mong muốn, ta phải đưa ra một hoạch định đường đi từ điểm bắt đầu (vị trí hiện tại của robot) đến điểm kết thúc (vị trí đích mong muốn). Nhằm tối ưu hóa cho đường đi, ta cần phải lựa chọn đường đi ngắn nhất giữa hai điểm ấy. Với yêu cầu này, thuật toán Dijkstra là một trong những thuật toán đơn giản để tìm được đường đi tối ưu nhất.

Bài toán đặt ra cho thuật toán Dijkstra là có một đồ thị $G = (V, E)$ (với V là tập hợp các đỉnh và E là các cạnh vô hướng hoặc có hướng), mỗi cạnh sẽ có trọng số luôn dương và một đỉnh nguồn s . Chúng ta cần tính toán được đường đi ngắn nhất từ đỉnh nguồn s đến mỗi đỉnh trên đồ thị. Để hiểu rõ hơn, ta hãy xem hình sau:



Hình 2.15 Sơ đồ nút và trọng số trên mỗi cạnh trong đồ thị [7]

Ta có một đồ thị gồm 6 nút, 9 cạnh, các trọng số trên mỗi cạnh và có điểm nguồn a, điểm đích b. Vậy chúng ta cần tìm đường đi ngắn nhất từ a đến b. Để thực hiện được thuật toán này, ta sẽ có phương pháp giải như sau:

1. Trước khi bắt đầu, ta cần gán mỗi nút một giá trị khởi tạo: bằng 0 đối với nút bắt đầu và bằng vô cùng cho tất cả các nút còn lại.
2. Đưa tất cả các nút vào trong một tập dữ liệu Q để ta tính khoảng cách từ điểm khởi tạo đến điểm đó.
3. Đối với điểm nút hiện tại, lần lượt xét các điểm nút xung quanh chưa từng xét trước đó và tính khoảng cách dự kiến của nó từ trọng số của nút hiện tại. Nếu giá trị dự kiến nhỏ hơn giá trị hiện tại của nút đó thì gán lại giá trị của nút đó bằng giá trị dự kiến.
4. Đối với mỗi nút đã xét, ta loại bỏ nút này khỏi tập Q để không cần phải xét lại.
5. Chúng ta lần lượt kiểm tra các nút trong tập Q. Nếu nút đích đã được kiểm tra thì dừng lại và kết thúc thuật toán.

Từ phương pháp giải trên, chúng ta sẽ thành lập đoạn mã giả (pseudo code) sau để có thể tìm được đường đi ngắn nhất.

1	Function Dijkstra (<i>Graph</i> , <i>source</i>):	
2	for each vertex <i>v</i> in <i>Graph</i> :	<i>// Initializations</i>
3	dist[<i>v</i>]:= infinity;	<i>// Unknown distance function from</i>
4		<i>// source to v</i>
5	previous[<i>v</i>]:= undefined;	<i>// Previous node in optimal path</i>
6	end for	<i>// from source</i>
7		
8	dist [<i>source</i>]:= 0;	<i>// Distance from source to source</i>
9	<i>Q</i> : = the set of all nodes in <i>Graph</i> ;	<i>// All nodes in the graph are</i>
10		<i>// unoptimized – thus are in Q</i>

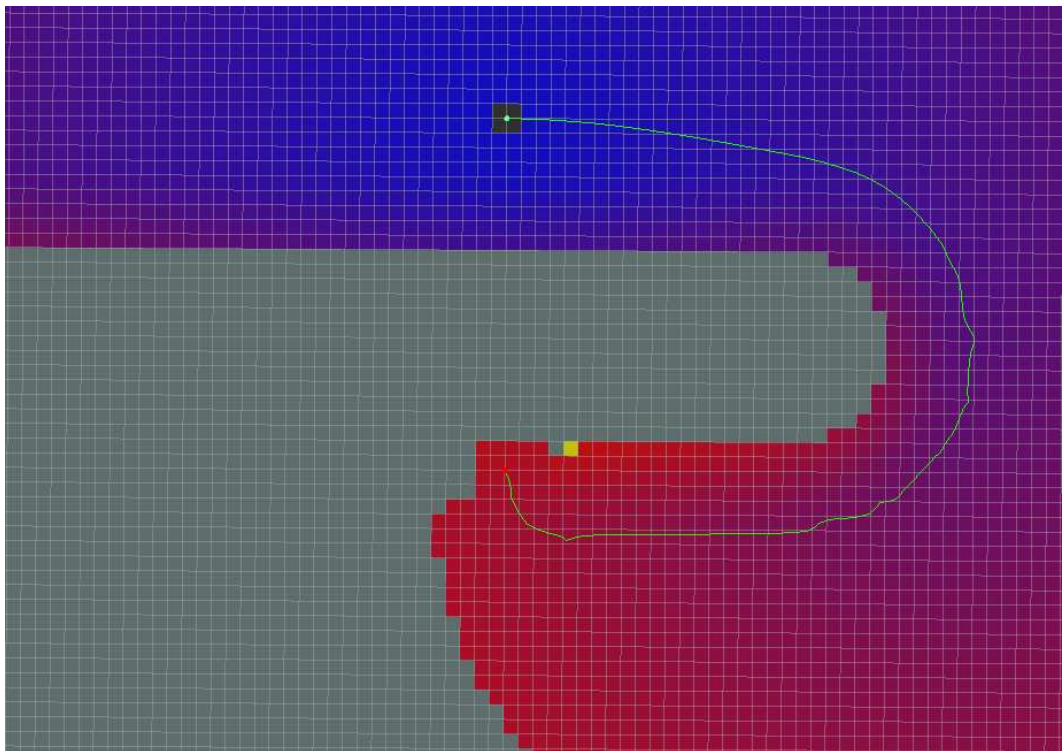
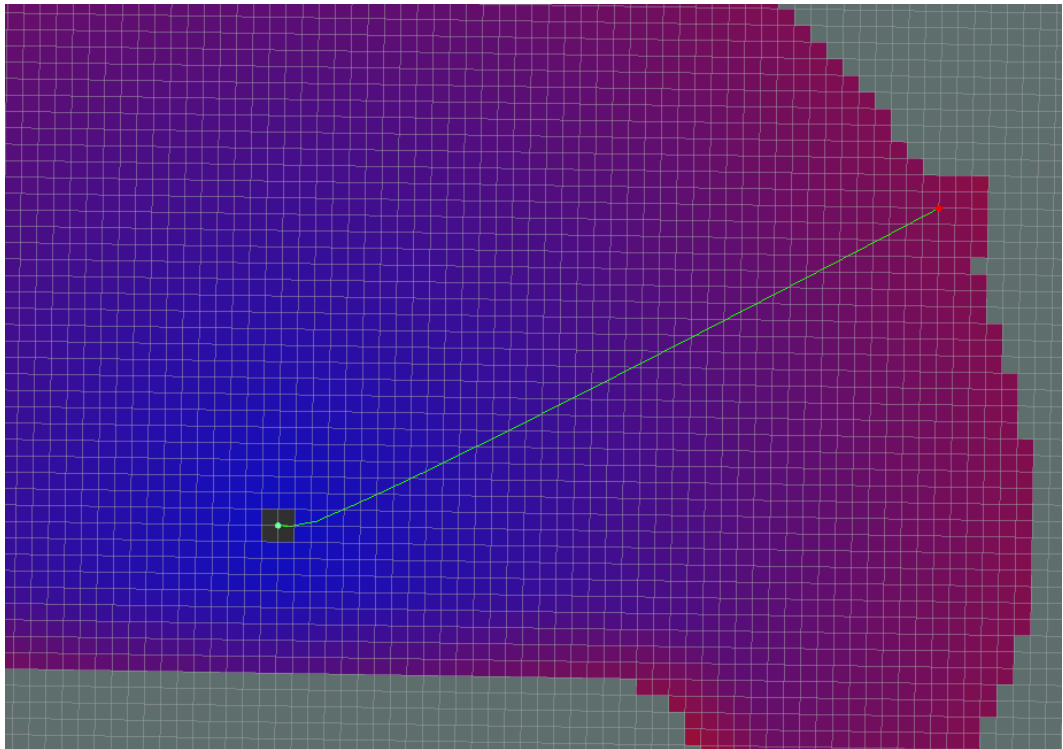
```

11  while  $Q$  is not empty:           // the main loop
12       $u$  = vertex in  $Q$  with smallest distance in  $\text{dist}[]$ ; // Source node in
13                                          // first case
14      remove  $u$  from  $Q$  ;
15      if  $\text{dist}[u] = \text{infinity}$ :
16          break;                     // all remaining vertices are
17      end if                           // inaccessible from source
18
19      for each neighbor  $v$  of  $u$ :       // where  $v$  has not yet been
20                                          // removed from  $Q$ .
21           $\text{alt} := \text{dist}[u] + \text{dist\_between}(u, v)$ ;
22          if  $\text{alt} < \text{dist}[v]$ : // Relax ( $u, v, a$ )
23               $\text{dist}[v] := \text{alt}$ ;
24               $\text{previous}[v] := u$ ;
25              decrease-key  $v$  in  $Q$ ; // Reorder  $v$  in the Queue
26          end if
27      end for
28  end while
29  return  $\text{dist}[], \text{previous}[]$ ;
30 end function

```

Bảng 2.5 Thuật toán Dijkstra [8]

Qua thuật toán trên, ta có thể dễ dàng tìm được đường đi ngắn nhất từ điểm bắt đầu đến điểm đích.



Hình 2.16 Hoạch định đường đi dùng thuật toán Dijkstra [9]

2.4. Thuật toán Dynamic Window Approach để tránh vật cản cho local planner

Thuật toán Dynamic Window Approach (DWA) là dùng để tìm ra một tín hiệu điều khiển hợp lý gửi xuống robot nhằm mục đích điều khiển nó đến đích an toàn, nhanh chóng dựa trên global planner đã hoạch định từ trước. Thuật toán này gồm hai bước chính là cắt giảm không gian tìm kiếm (search space) của vận tốc và tìm được vận tốc tối ưu trong không gian tìm kiếm đó.

2.4.1. Không gian tìm kiếm

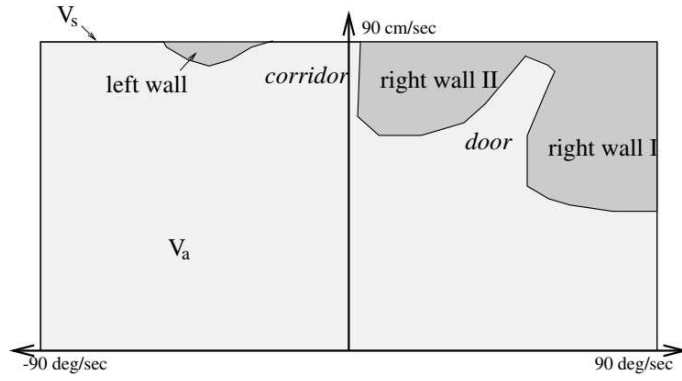
Các vận tốc có thể điều khiển được trong không gian tìm kiếm được cắt giảm theo ba bước sau:

- Quỹ đạo tròn: thuật toán DWA chỉ xét đến quỹ đạo là hình tròn (đường cong) được xác định duy nhất bởi một cặp vận tốc thẳng và vận tốc xoay (v, ω) .
- Vận tốc cho phép: nhằm tạo ra một quỹ đạo an toàn cho robot để tránh vật cản. Một cặp vận tốc (v, ω) được cho phép là khi robot có thể dừng trước vật cản gần nhất mà không có sự va chạm trên đường cong tương ứng với vận tốc đó. Vận tốc cho phép được định nghĩa như sau

$$V_a = \{(v, \omega) | v \leq \sqrt{2 \cdot \text{dist}(v, \omega) \cdot v_b} \wedge \omega \leq \sqrt{2 \cdot \text{dist}(v, \omega) \cdot \omega_b}\}$$

Trong đó:

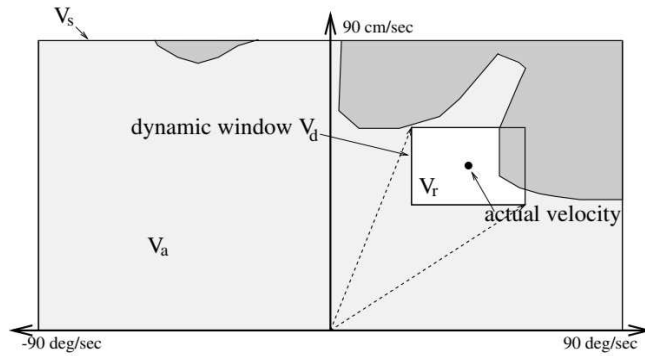
- + V_a là chuỗi các giá trị vận tốc (v, ω) cho phép robot dừng trước vật cản mà không có sự va chạm.
- + $\text{dist}(v, \omega)$ là khoảng cách nhỏ nhất mà robot dừng trước vật cản để không có sự va chạm.
- + v_b, ω_b là gia tốc của vận tốc thẳng và vận tốc xoay tối đa nếu robot di chuyển sẽ gây va chạm với vật cản.



Hình 2.17 Vận tốc cho phép V_a trong DWA [10]

- Dynamic window: nhằm hạn chế vận tốc cho phép đối với những vận tốc có thể đạt được trong khoảng chu kỳ cho trước với gia tốc tối đa của robot. Để Δt là khoảng thời gian mà trong đó gia tốc $\dot{v}, \dot{\omega}$ sẽ được thực thi để (v_a, ω_a) là vận tốc thực được gửi xuống robot. Từ đó, vận tốc V_d sẽ được định nghĩa như sau

$$V_d = \{(v, \omega) | v \in [v_a - \dot{v} \cdot \Delta t, v_a + \dot{v} \cdot \Delta t] \wedge \omega \in [\omega_a - \dot{\omega} \cdot \Delta t, \omega_a + \dot{\omega} \cdot \Delta t]\}$$



Hình 2.18 Vận tốc trong cửa sổ động V_d trong DWA [10]

- Kết thúc ba bước trên thì ta tìm được không gian tìm kiếm

$$V_r = V_s \cap V_a \cap V_d$$

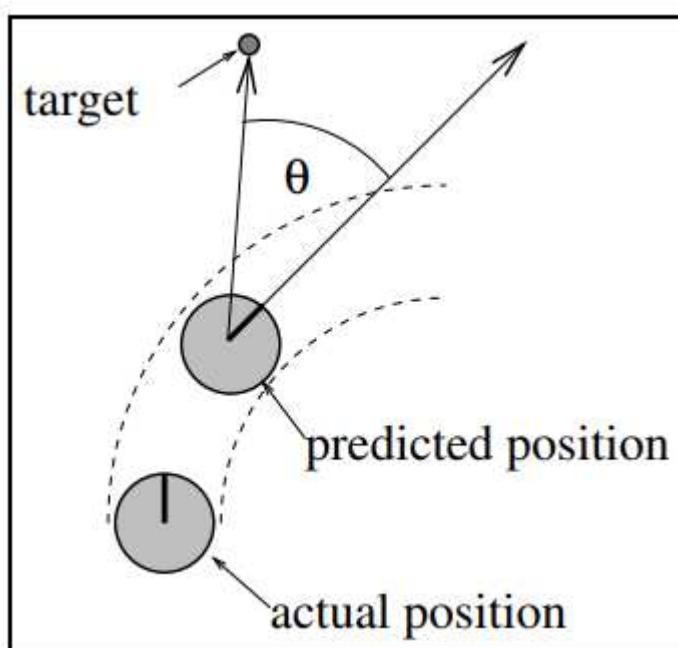
2.4.2. Tối ưu

Ta có hàm mục tiêu được định nghĩa như sau

$$G(v, \omega) = \alpha \cdot heading(v, \omega) + \beta \cdot dist(v, \omega) + \gamma \cdot vel(v, \omega)$$

Để có thể tối ưu hóa vận tốc ngõ ra thì hàm mục tiêu phải có giá trị tối đa. Để thực hiện được điều này, ta thực hiện các bước sau:

- Target heading: heading là giá trị đo tiến độ hướng đến đích của robot. Giá trị sẽ mang giá trị tối đa khi robot di chuyển trực tiếp về phía đích. Giá trị của $heading(v, \omega)$ được tính bởi công thức $180 - \theta$, với θ là góc giữa hướng của robot và điểm đích.



Hình 2.19 Heading của robot trong DWA [10]

- Không gian trống (clearance): hàm $dist(v, \omega)$ thể hiện khoảng cách tính từ robot đến vật cản gần nhất nằm trên quỹ đạo cong của nó. Giá trị này sẽ rất lớn nếu không có vật cản nằm trên quỹ đạo cong di chuyển của nó. Giá trị này càng nhỏ thì việc nó đối mặt với vật cản càng cao, khi đó nó sẽ di chuyển xung quanh vật cản ấy.
- Vận tốc: hàm $vel(v, \omega)$ là vận tốc di chuyển thẳng của robot và hỗ trợ di chuyển nhanh hơn.
- Các hệ số α, β, γ được chọn sao cho phù hợp với đặc tính của robot và môi trường hoạt động. Khi hàm mục tiêu có giá trị lớn nhất thì quỹ đạo

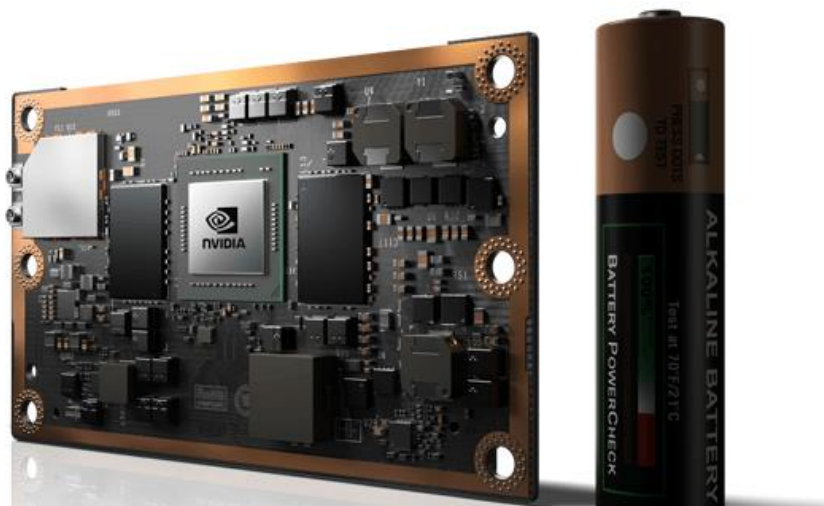
tối ưu sẽ được chọn với vận tốc (v, ω) tốt nhất và là kết quả của thuật toán.

Chương 3. THIẾT LẬP XE TỰ HÀNH VÀ LẬP TRÌNH TÍCH HỢP HỆ THỐNG

3.1. Thiết lập xe tự hành

3.1.1. Board điều khiển

Để có thể đáp ứng được khả năng cũng như tốc độ xử lý, chúng ta cần một board có khả năng xử lý đủ mạnh để chạy các thuật toán điều khiển có độ phức tạp cao. Bên cạnh đó, nhằm giảm kích thước và tăng tính linh động cho sản phẩm thì việc lựa chọn một bộ vi xử lý tích hợp có kích thước nhỏ là một sự lựa chọn hợp lý. Trên thị trường hiện nay, board nhúng có rất nhiều giá thành khác nhau dẫn đến khả năng xử lý khác nhau.



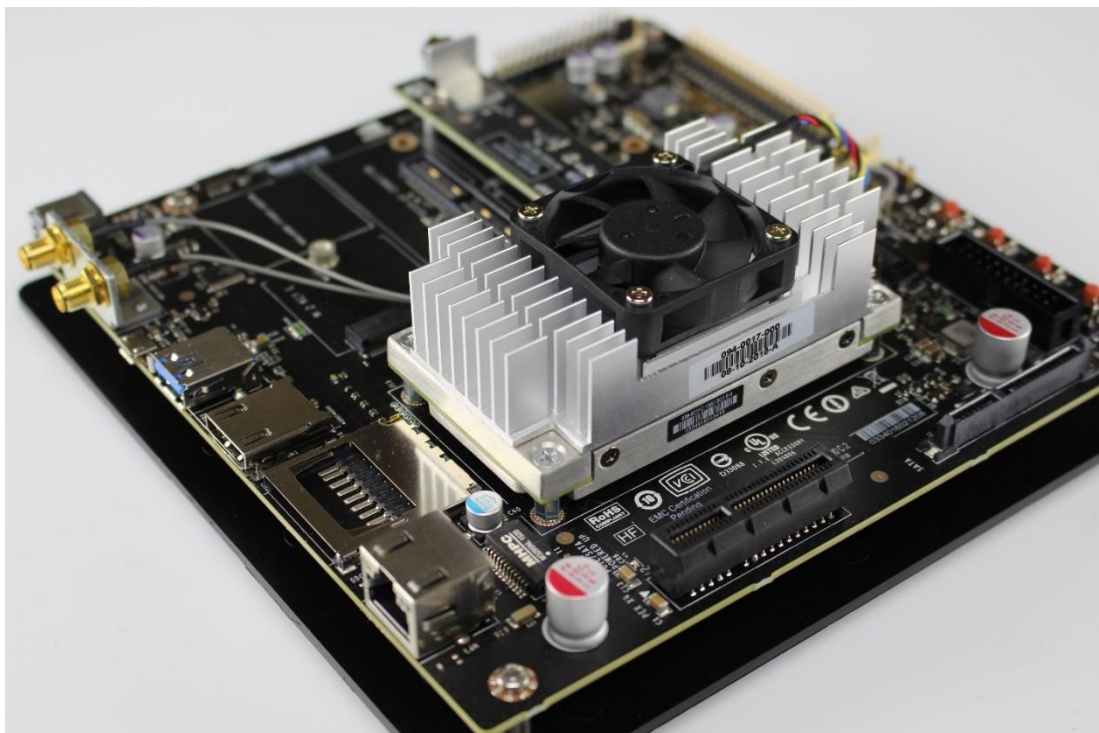
Hình 3.1 Máy tính Jetson TX2

Từ đó, việc phát triển ứng dụng cho robot tự hành trong nhà dựa trên board NVIDIA Jetson là một sự lựa chọn tối ưu. Trước sự thành công của Jetson TX1, thì Jetson TX2 là một phiên bản được nâng cấp về phần cứng cho nên đây là một lợi thế tốt hơn.

Jetson TX2 có GPU được xây dựng trên kiến trúc NVIDIA Pascal GPU với lõi chứa 256 CUDA (CUDA là nền tảng cho việc tính toán song song và cũng là một kiểu ngôn ngữ lập trình). CPU của Jetson TX2 có chứa 2 nhóm CPU ARM v8 64 bits được kết nối với nhau nhằm tăng hiệu năng xử lý của board. Một nhóm CPU Denver

2 (Dual-Core) được tối ưu cho việc tính toán luồng đơn, và một nhóm Cortex-A57 QuadCore được dùng cho các ứng dụng đa luồng. Bộ nhớ chính của board là 8GB 128 bits LPDDR4 với tốc độ trao đổi dữ liệu lên đến 58.4 GB/s, và một bộ nhớ eMMC có dung lượng 32GB được tích hợp sẵn trong board nhằm để lưu hệ điều hành cùng các dữ liệu mà chúng ta xử lý. Bên cạnh đó, module còn hỗ trợ phần cứng cho việc mã hóa và giải mã hình ảnh/video có chất lượng lên đến 4K với tốc độ 60 fps. Ngoài ra, Jetson TX2 còn hỗ trợ các kết nối không dây như Wifi và Bluetooth. Jetson TX2 chạy trên hệ điều hành Ubuntu phiên bản 16.04 với kernel là 4.4.

Để thuận lợi cho việc nghiên cứu và phát triển, Jetson TX2 Module được thiết kế bởi NVIDIA là sự lựa chọn tốt hơn bởi chúng ta sẽ không cần phải thiết kế các module kết nối với các thiết bị khác như HDMI, USB,...



Hình 3.2 Board phát triển được tích hợp máy tính nhúng Jetson TX2

Một số thông số kỹ thuật của board Jetson TX2:

GPU	NVIDIA Pascal™, 256 CUDA cores
CPU	HMP Dual Denver 2/2 MB L2 + Quad ARM® A57/2 MB L2
Video	4K x 2K 60 Hz Encode (HEVC) 4K x 2K 60 Hz Decode (12-Bit Support)
Memory	8 GB 128 bit LPDDR4 59.7 GB/s
Display	2x DSI, 2x DP 1.2 / HDMI 2.0 / eDP 1.4
CSI	Up to 6 Cameras (2 Lane) CSI2 D-PHY 1.2 (2.5 Gbps/Lane)
PCIE	Gen 2 1x4 + 1x1 OR 2x1 + 1x2
Data storage	32 GB eMMC, SDIO, SATA
Other	CAN, UART, SPI, I2C, I2S, GPIOs
USB	USB 3.0 + USB 2.0
Connectivity	1 Gigabit Ethernet, 802.11ac WLAN, Bluetooth
Mechanical	50 mm x 87 mm

Bảng 3.1 Thông số kỹ thuật board Jetson TX2

3.1.2. Camera

Để có thể thực hiện được yêu cầu xây dựng bản đồ và điều hướng trong bản đồ đó thì chúng ta cần một cảm biến laser scan có thể quét được không gian 2D hoặc 3D. Một số loại cảm biến này có thể kể đến như Hokuyo Scanning Laser Rangefinder, SICK laser, RPLIDAR,...



Hình 3.3 Một số cảm biến laser find-ranger trên thị trường

Tuy nhiên, giá thành của các cảm biến này rất đắt (từ vài trăm đến hàng chục nghìn USD). Do đó, chúng ta cần một loại cảm biến khác có giá thành rẻ hơn nhưng vẫn có kết quả ngõ ra tương tự như các loại cảm biến laser scan này. Mục đích của chúng ta là để robot cảm nhận được chiều sâu của môi trường để xây dựng hình dạng lại môi trường ấy, vì thế ta cần chọn loại cảm biến nào có thể trả về giá trị độ sâu của trường. Sau đó chúng ta cần chuyển độ sâu nhận được về kiểu dữ liệu như của laser scan để phù hợp với các package mà chúng ta sẽ sử dụng.

Với yêu cầu đó, ta có hai sự lựa chọn phù hợp là:

- + Một camera để thu thập dữ liệu hình ảnh và một cảm biến đo độ sâu (phương pháp đo độ sâu bằng hồng ngoại). Với sự lựa chọn này, ta có một số cảm biến như Kinect của Microsoft, Astra của Orbbec,...



Hình 3.4 Một số camera có tích hợp cảm biến đo khoảng cách dùng hồng ngoại
 + Hai camera – stereo camera để thu thập dữ liệu hình ảnh từ hai camera. Từ hai hình ảnh thu được, ta có thể suy ra được độ sâu của các đặc trưng trong môi trường. ZED camera là một thiết bị có thể đáp ứng được yêu cầu này.



Hình 3.5 Stereo camera

Qua đó, ta thấy nếu ta sử dụng stereo camera kết hợp với cảm biến đo độ sâu thì sẽ được dữ liệu tốt hơn. Trong giới hạn đề tài này về mặt kiến thức lẫn thời gian, việc tìm hiểu stereo camera sẽ có lợi thế nên trong báo cáo sẽ tập trung vào việc tìm hiểu stereo camera.

ZED camera được phát triển dưới Stereo LABS với nhiều tính năng và công cụ hỗ trợ. Khi dùng ZED, ta hoàn toàn lấy được độ sâu và theo dõi chuyển động một cách dễ dàng hơn. Camera có thể cho ta hình ảnh có độ phân giải tối đa 2K với độ nhạy sáng thấp, thích hợp cho việc sử dụng trong những môi trường khó khăn. ZED hỗ trợ tỷ lệ khung hình khá cao và có góc nhìn rộng. Về mặt đo độ sâu, thì ZED camera có thể đo chính xác với khoảng cách lên đến 20m trong điều kiện trong nhà và ngoài trời.

Về các hoạt động về mặt chuyển động, ZED cho độ chính xác khoảng cách ở 6 trục với độ chính xác vị trí là $\pm 1\text{mm}$ và độ chính xác về định hướng là 0.1° . Tần số cập nhật lên đến 100Hz.

Hiện tại, Stereo LABS cung cấp cho các nhà phát triển bộ thư viện ZED SDK nhằm thực hiện một số sample về khả năng của camera như hiển thị video trực tuyến, calibration, chuẩn đoán lỗi, phát hiện độ sâu môi trường,...

Một số thông số cấu hình của camera:

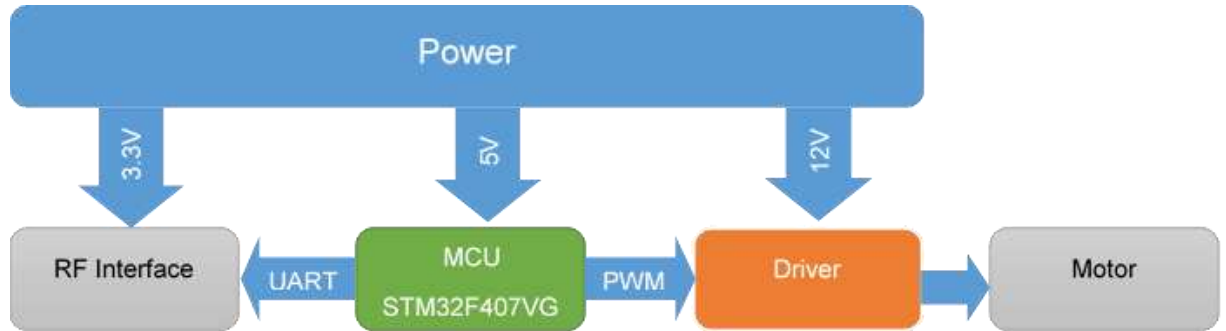
Video	Mode	FPS	Resolution
	2.2K	15	4416x1242
	1080p	30	3840x1080
	720p	60	2560x720
	WVGA	100	1344x376
Depth	Range		Format
	0.5 – 20m		32-bits
Motion	6-axis Pose Accuracy		Frequency
	Position: +/- 1mm Orentation: 0.1°		Up to 100Hz

Bảng 3.2 Thông số kỹ thuật của ZED camera

3.1.3. Mobile base

3.1.3.1. Thiết kế phần cứng

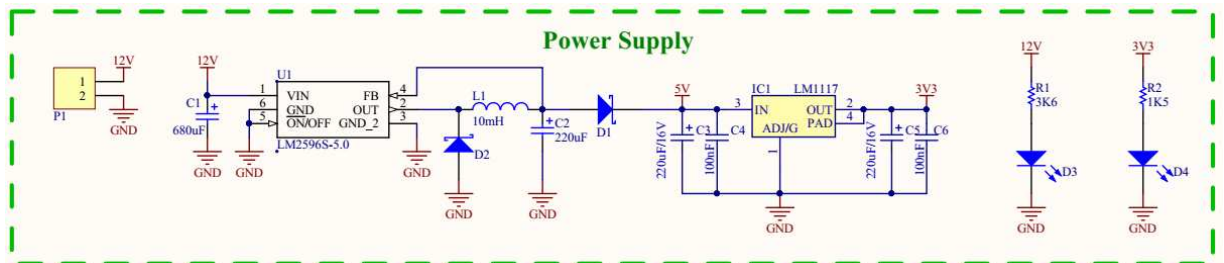
Sơ đồ khối của phần cứng:



Hình 3.6 Sơ đồ khối phần cứng

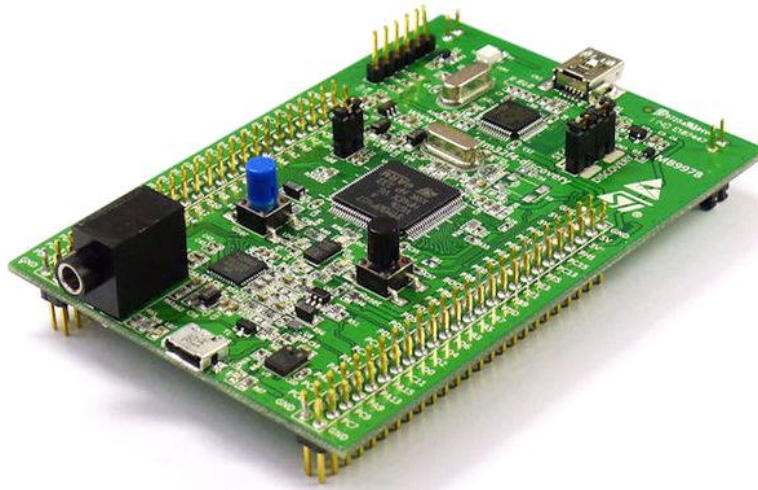
Các thành phần của board bao gồm nguồn cung cấp, vi điều khiển, RF, driver điều khiển motor và motor.

Phần nguồn được cấp từ pin Lipo (11.1V) và sau đó được hiệu chỉnh thành các điện áp thành phần nhằm cung cấp mức điện áp phù hợp cho từng module.



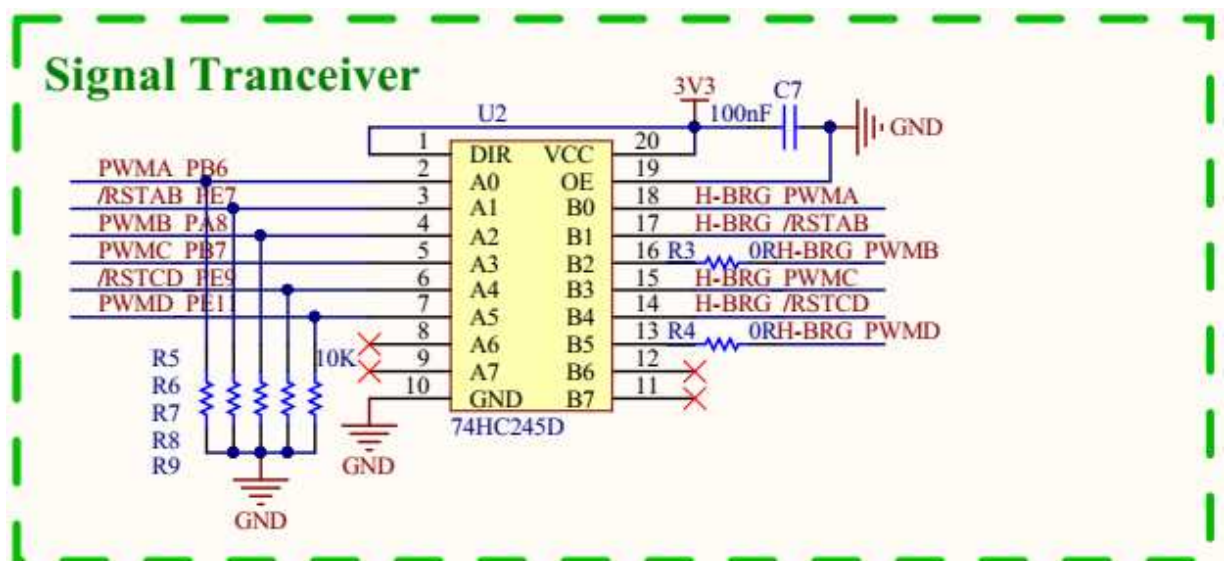
Hình 3.7 Mạch nguồn

Thành phần khá quan trọng trong mạch điều khiển đó chính là vi điều khiển. Ở đây, mạch được thiết kế phù với board Discovery STM32F4. Đây là một vi điều khiển có khả năng đáp ứng và hiệu năng hoạt động khá tốt với nhiều thành phần ngoại vi được hỗ trợ. STM32F4 được xây dựng trên nền tảng ARM Cortex-M4 32bits MCU+FPU hoạt động ở tần số 168MHz nhằm đem lại khả năng tính toán nhanh, hiệu quả.



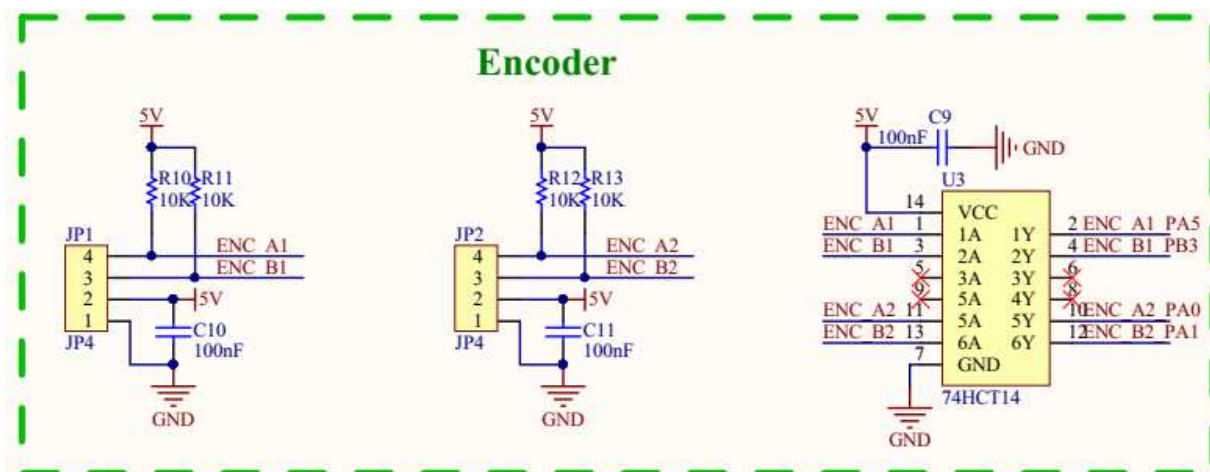
Hình 3.8 Board Discovery STM32F4

Ngoài ra nhằm đảm bảo tín hiệu ổn định và di chuyển theo một chiều, chúng ta cần thêm một IC đệm tín hiệu để tín hiệu có dòng, áp ngõ ra ổn định và đi theo một chiều.



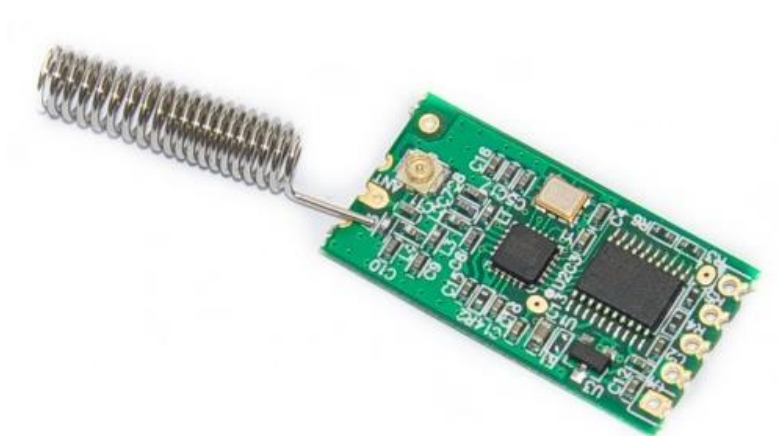
Hình 3.9 Mạch đệm tín hiệu

Bên cạnh đó, mạch phân cứng còn được thiết kế để đọc tín hiệu encoder nhằm hỗ trợ cho việc tích hợp encoder vào hệ thống để tính toán giá odom và pose cho robot. Để hạn chế nhiễu cho tín hiệu encoder khi đọc vào vi điều khiển, mạch được hỗ trợ IC đệm có hỗ trợ Schmitt trigger nhằm tạo lại xung vuông cho tín hiệu encoder.



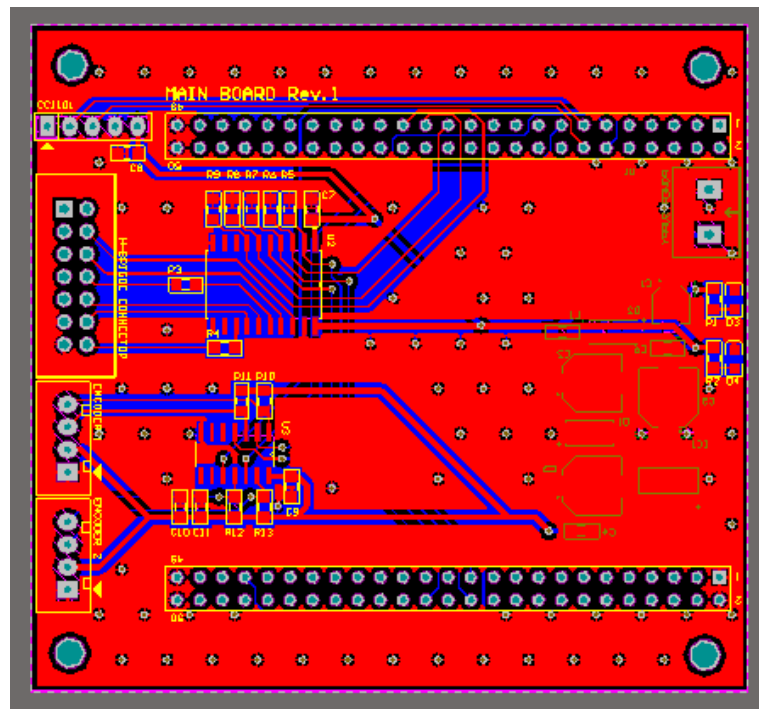
Hình 3.10 Mạch nắn xung encoder

Nhằm để trợ quá trình giao tiếp giữa hệ điều hành - mobile base (khi điều hướng xe) và joytick – mobile base (khi dựng bản đồ) thì việc truyền nhận tín hiệu điều khiển phải linh hoạt. Và việc lựa chọn truyền thông không dây là sự lựa chọn tối ưu. Phần cứng này được thiết kế cho các module hỗ trợ giao tiếp UART, trong đó module CC1101 là ví dụ điển hình cho truyền thông không dây bằng sóng RF.

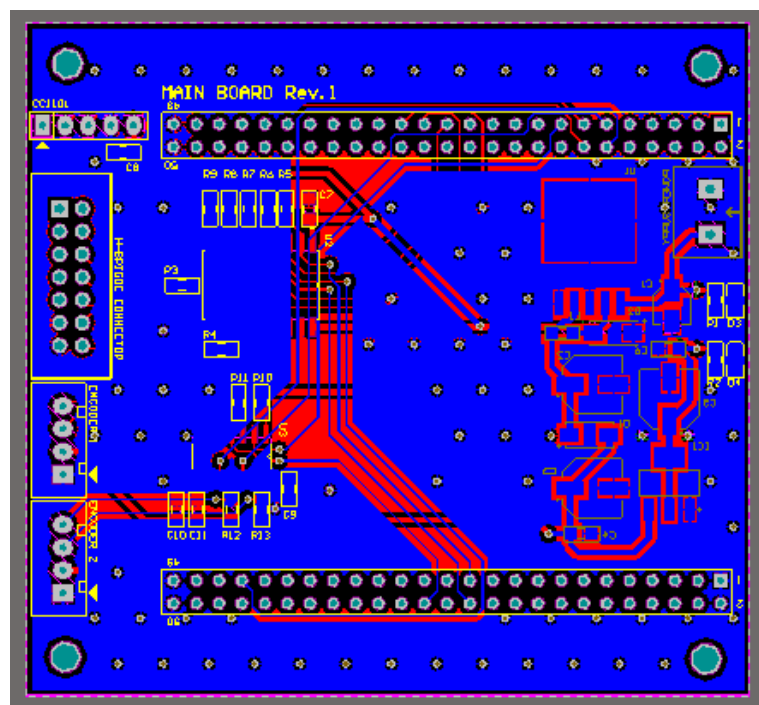


Hình 3.11 Module giao tiếp không dây CC1101

Hình ảnh sau khi thiết kế board mạch

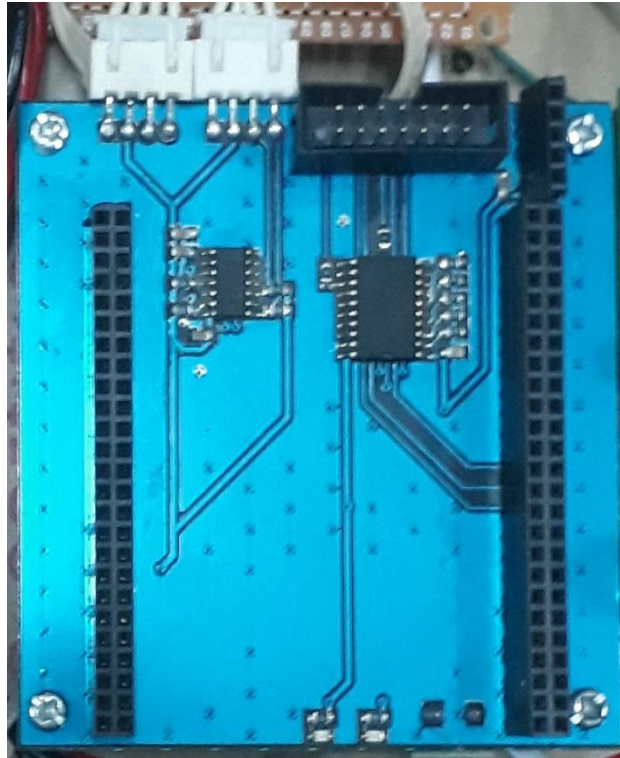


Hình 3.12 Hình ảnh mạch điện lớp trên

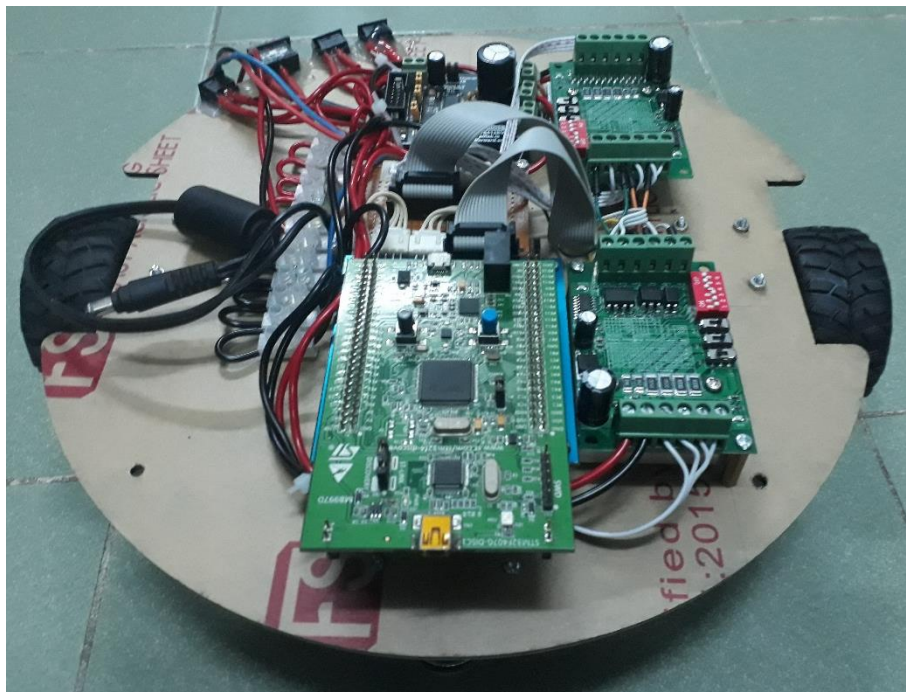


Hình 3.13 Hình ảnh mạch điện lớp dưới

Một số hình ảnh thực tế sau khi phần cứng được hoàn thành



Hình 3.14 Mạch sau khi hàn lắp linh kiện



Hình 3.15 Mạch sau khi được tích hợp lên khung xe

3.1.3.2. Thiết kế chương trình nhúng

Mobile base được thiết kế để nhận tín hiệu điều khiển từ hệ điều hành ROS gửi xuống. Tín hiệu nhận được bao gồm vận tốc thẳng v và vận tốc xoay ω . Nhiệm vụ của chương trình nhúng là có thể điều khiển mobile base thực thi đúng với các vận tốc ấy.

Từ vận tốc nhận được, ta phải chuyển về vận tốc riêng cần thiết cho bánh trái và bánh phải thực hiện đúng với lệnh mà phần mềm gửi xuống để vận tốc của tâm xe đúng với vận tốc yêu cầu. Vì xe có cấu trúc dạng tròn và hai bánh xe có tâm đối xứng nhau nên việc điều khiển xe khá dễ dàng (hình dáng của xe được trình bày rõ ràng hơn ở mục 3.1.3.3).

Vận tốc riêng phần của bánh xe trái và bánh xe phải là

$$v_l = \frac{2v - \omega L}{2}$$

$$v_r = \frac{2v + \omega L}{2}$$

Trong đó, L là khoảng cách giữa hai bánh xe (đơn vị mét).

Tùy thuộc vào từng loại động cơ mà chúng ta có cách thiết kế chương trình nhúng khác nhau. Nếu là động cơ step thì chúng ta cần cấp số xung cần thiết trong khoảng thời gian lấy mẫu cho mỗi động cơ để có thể đáp ứng được vận tốc đặt ra. Số xung cần thiết cấp cho từng động cơ là:

$$N_l = \frac{\frac{T_{sample}}{T_{pulse}}}{2 \cdot \frac{v_l \cdot T_{sample} \cdot PPR}{2\pi R}}, N_r = \frac{\frac{T_{sample}}{T_{pulse}}}{2 \cdot \frac{v_r \cdot T_{sample} \cdot PPR}{2\pi R}}$$

Trong đó:

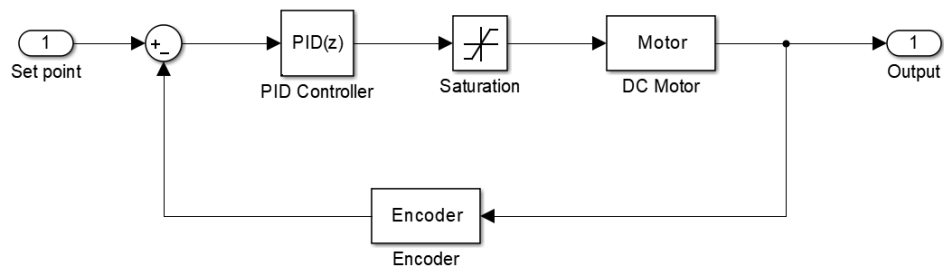
- T_{sample} là thời gian lấy mẫu của hệ thống hay là thời gian tính toán lại các giá trị vận tốc, số xung cho hệ thống (đơn vị là giây).

- T_{pulse} là khoảng thời gian nhỏ nhất giữa hai xung liên tiếp cấp cho driver điều khiển động cơ (đơn vị là giây).
- v_l, v_r lần lượt là vận tốc tịnh tiến của bánh xe trái và bánh xe phải (đơn vị là mét/giây).
- PPR (Pulse per Round) là số xung cấp cho động cơ để động cơ quay được một vòng. Thông số này tùy thuộc số bước động cơ và tỉ lệ bước trên driver.
- R là bán kính của bánh xe (đơn vị là mét).

Đối với động cơ DC thì chúng ta lại điều khiển bằng áp và driver điều khiển động cơ lại nhận tín hiệu phản xung ở mức cao (T_{on}) để ra tín hiệu áp cung cấp cho động cơ. Để có thể đáp ứng được chính xác vận tốc ngõ ra, chúng ta cần một bộ điều khiển để điều khiển cho động cơ. Với điều khiển động cơ DC thì bộ điều khiển PID là bộ điều khiển khá thông dụng cho việc điều khiển vận tốc. Việc dùng bộ điều khiển PID thì cần có một nguồn cảm biến để đo giá trị phản hồi từ động cơ về để điều chỉnh lại tín hiệu cung cấp cho động cơ. Encoder là một cảm biến thường dùng để đo vận tốc, vị trí của động cơ. Giá trị trả về của Encoder là số xung tương ứng góc quay của động cơ. Để điều khiển được vận tốc ngõ ra, chương trình được thiết kế theo điều khiển số xung Encoder trong một khoảng thời gian lấy mẫu tương ứng với vận tốc nhận được. Số xung encoder mong muốn của mỗi động cơ là:

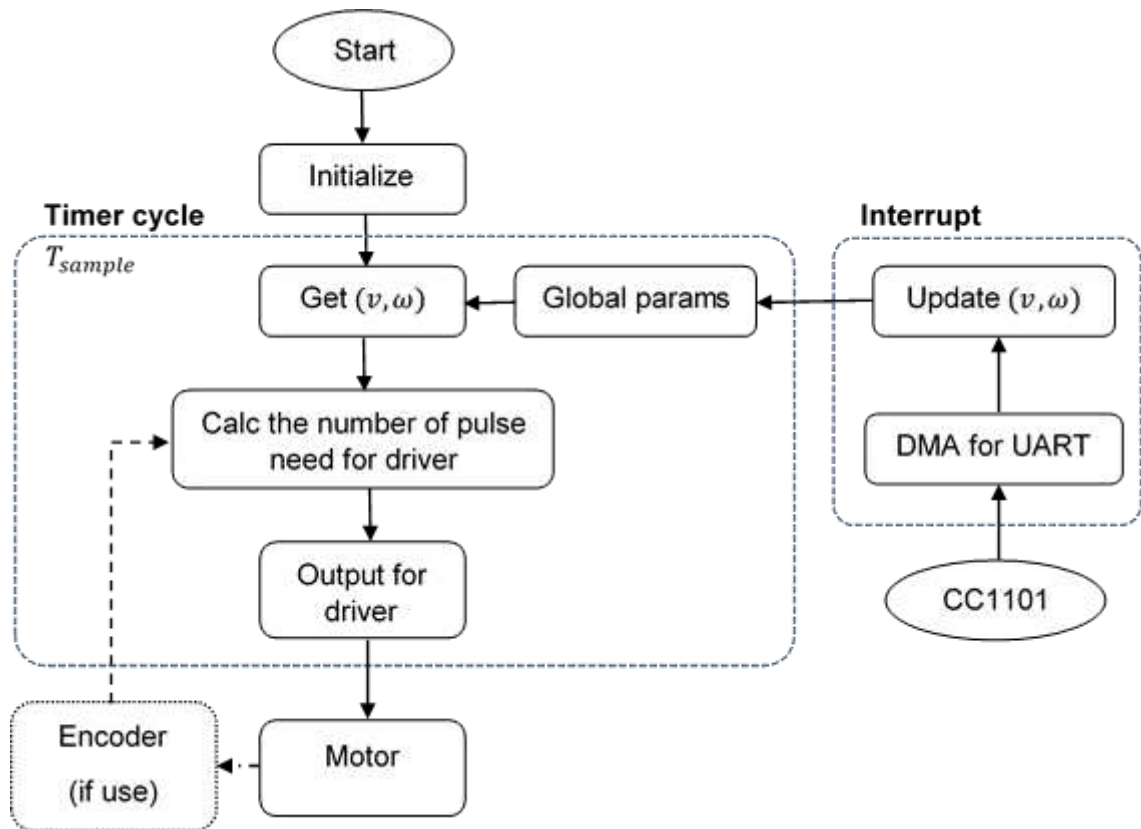
$$N_l = \frac{\frac{v_l}{R} \cdot PPR \cdot T_{sample}}{2\pi}, N_r = \frac{\frac{v_r}{R} \cdot PPR \cdot T_{sample}}{2\pi}$$

Giải thuật điều khiển động cơ dựa trên bộ điều khiển PID như sau:



Hình 3.16 Giải thuật điều khiển PID

Tổng quan chương trình điều khiển của Mobile base:



Hình 3.17 Chương trình điều khiển Mobile base

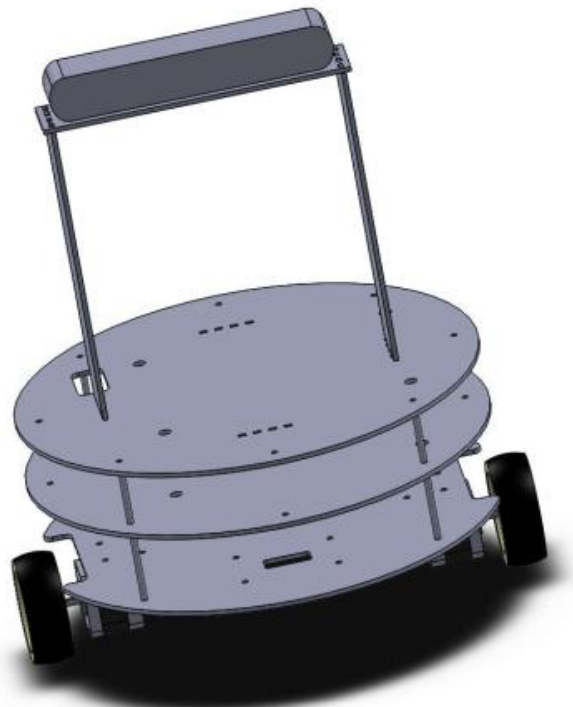
Sau khi hệ thống khởi động là quá trình khởi tạo các module ngoại vi như GPIO, Timer, UART, DMA,... Sau mỗi chu kỳ hoạt động của Timer thì vi điều khiển sẽ lấy giá trị vận tốc (v, ω) từ các biến giá trị toàn cục và tính toán số xung cần thiết cho từng động cơ trái, phải. Tiếp đó, nó sẽ xuất tín hiệu xung ra driver. Nếu hệ thống có sử dụng encoder thì tín hiệu encoder sẽ phản hồi về bộ tính toán số xung và bộ điều khiển PID sẽ lấy giá trị sai số để xuất ra số xung phù hợp cho động cơ.

Nếu trong quá trình hoạt động mà có tín hiệu mới nhận được module C1101 thì sẽ được gửi vào bộ đệm thông qua DMA. Khi bộ DMA nhận được đủ số lượng byte trong bộ đệm thì sẽ sinh ra một ngắt để vi xử lý có thể lấy khung dữ liệu trong bộ đệm để tính toán ra giá trị vận tốc cần thiết và cập nhật lại giá vận này trong biến giá trị toàn cục.

3.1.3.3. Thiết kế cơ khí cho xe

Thiết kế xe đơn giản, có cấu trúc đối xứng và dễ điều khiển là mục tiêu để thiết kế xe. Với các hình học cơ bản thì hình tròn là hình có cấu trúc đối xứng hai bên, dễ điều khiển vì tâm của xe là tâm của hình tròn.

Xe trong đề tài được thiết kế thành 3 tầng: một tầng để chứa mạch điều khiển, driver cho Mobile base; một tầng chứa máy tính nhúng; một tầng để chứa chứa khung nâng camera.



Hình 3.18 Mặt trước của khung xe thiết kế

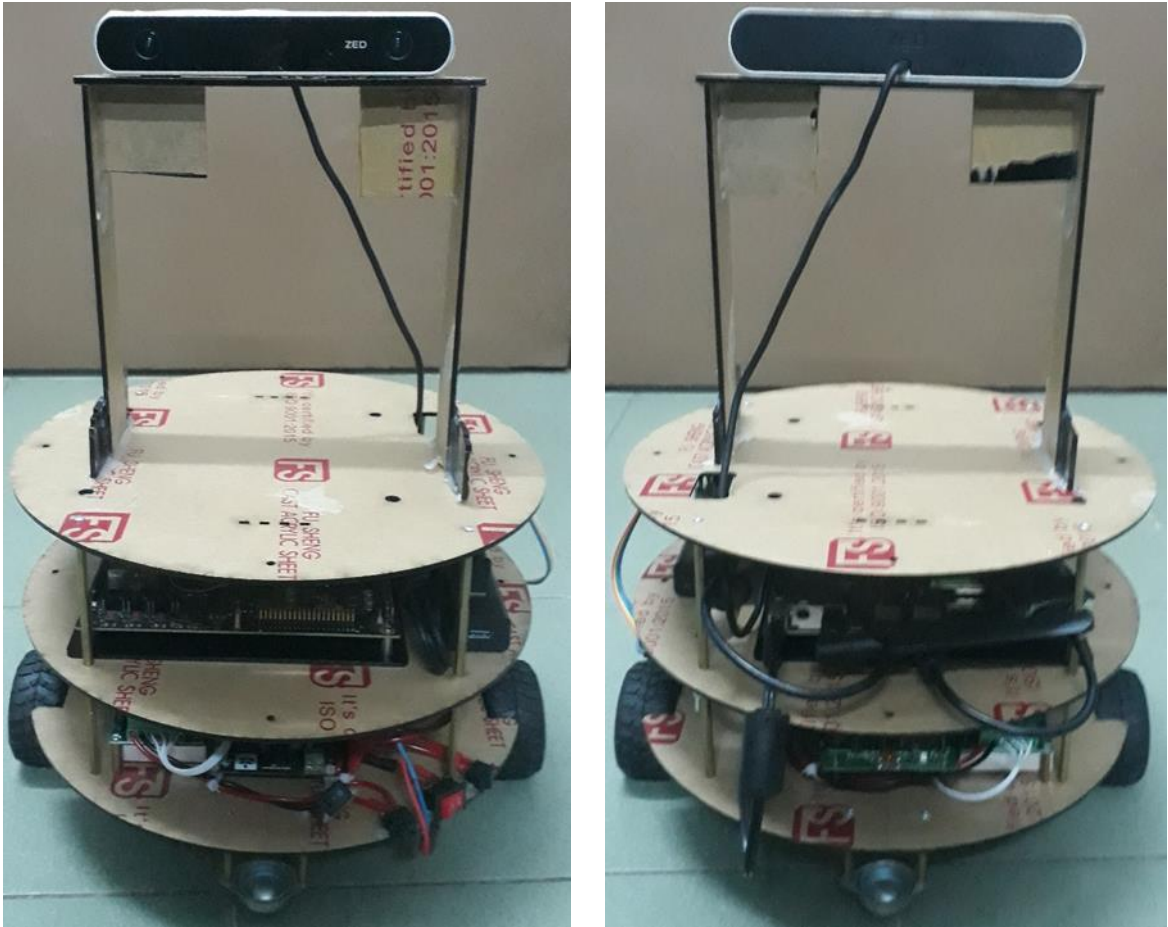


Hình 3.19 Mặt sau của khung xe thiết kế

Một số thông số kỹ thuật của khung xe:

- Đường kính bao quanh xe: 350 mm
- Chiều cao của xe: 450 mm
- Khoảng cách giữa tầng 1 và tầng 2: 50 mm
- Khoảng cách giữa tầng 2 và tầng 3: 40 mm
- Chiều cao giá đỡ của camera: 250 mm

Một số hình ảnh của xe sau khi hoàn thành:



Hình 3.20 Mobile base khi hoàn thành thiết kế

3.2. Lập trình tích hợp hệ thống

3.2.1. Thiết lập serial node giao tiếp với Mobile base

Để gửi lệnh điều khiển và nhận giá trị hồi tiếp từ *Mobile base*, ta cần một serial node để thực hiện nhiệm vụ này giao tiếp với hệ điều hành ROS. Node này cần phải subscribe một topic “cmd_vel” được publish từ *move_base*. Sau đó, cần chuyển các giá trị vận tốc nhận được gửi qua cổng giao tiếp nối tiếp. “cmd_vel” là một message kiểu `geometry_msgs/Twist` được định nghĩa như sau:

```
geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular
```

Vì **Mobile base** chỉ nhận giá trị vận tốc thẳng và vận tốc xoay (theo mô hình differential drive) nên ta cần phải thiết lập `dwa_local_planner` chỉ có vận tốc theo phương x và vận tốc xoay theo phương z (xem chi tiết ở mục 3.2.3.3). Nên khi đó, ta chỉ cần xem vận tốc linear theo phương x là vận tốc thẳng v và vận tốc angular theo phương z là vận tốc xoay ω .

Để thực thi được **serial node** này chạy trên ROS, ta có hai ngôn ngữ để lập trình là Python và C++, tùy vào lợi thế của bản thân mà chúng ta cần lựa chọn ngôn ngữ phù hợp. Đoạn chương trình sau thể hiện việc publish là subscribe các topic mà **serial node** quản lý:

```
ros::NodeHandle nh("~");

ros::Subscriber write_sub = nh.subscribe("cmd_vel", 100,
velCallback);

ros::Publisher cmd_pub =
nh.advertise<geometry_msgs::Twist>("cmd_vel_odom", 1000);
```

Bảng 3.3 Khởi tạo serial node

Trong đoạn chương trình trên, `nh` là node handle để hệ điều hành quản lý node của chúng ta. `write_sub` là một đối tượng được khai báo nhằm thông báo với hệ điều hành chúng ta cần subscribe topic có tên là “`cmd_vel`” với kích thước bộ đệm là 100. Khi có dữ liệu đến thì hàm `velCallback` sẽ được gọi. Bên cạnh đó, `cmd_pub` là đối tượng dùng để publish một message có kiểu dữ liệu `geometry_msgs::Twist` lên topic có tên là “`cmd_vel_odom`” với kích thước bộ đệm là 1000.

Như đã nói ở trên, hàm `velCallback` được gọi khi có dữ liệu nhận được từ **move_base**. Nhiệm vụ của hàm callback này là lấy giá trị vận tốc từ message nhận được và gửi qua cổng serial. Đoạn chương trình sau sẽ thể hiện được việc giao tiếp này:

```
void velCallback(const geometry_msgs::Twist::ConstPtr& vel) {
    if((vel->linear.x != vel->linear.x) || (vel->angular.z != vel-
>angular.z))
    {
```

```

        ROS_INFO("Linear: [%.3f] | Angular: [%.3f]", vel->linear.x, vel->angular.z);

        sprintf((char *)u8_bufWrite, "[%.3f,%.3f,]", vel->linear.x, vel->angular.z);

        write (mainfd, (char *) u8_bufWrite, strlen((char *) u8_bufWrite));
    }

    vel_.linear.x = vel->linear.x;

    vel_.angular.z = vel->angular.z;
}

```

Bảng 3.4 Hàm callback của serial node

Nhằm để dễ dàng cho việc khởi tạo và thiết lập môi trường cho *serial node*, ta sẽ cần phải tạo launch file nhằm để đơn giản hóa việc thực thi hơn.

```

<launch>

  <node pkg="test_serial" type="test_serial" name="test_serial"
output="screen">

    <param name="port_id" value="/dev/ttyUSB1"/>

  </node>

</launch>

```

Bảng 3.5 File launch cho serial node

Trong đó, `port_id` là tên của cổng giao tiếp nối tiếp mà hệ điều hành Ubuntu nhận dạng (để biết cổng nào đang sử dụng thì dùng lệnh `ls /dev/tty*`, các cổng thường dùng có dạng `ttyUSBx`, `ttyACMx`)

3.2.2. Xây dựng bản đồ (mapping)

Để có xây dựng được bản đồ 2D nhằm cung cấp một bản đồ tĩnh để Navigation stack hoạt động thì có một package `gmapping` được hỗ trợ trên nền tảng ROS. Đối với *gmapping* thì nó yêu cầu một dữ liệu odometry và một cảm biến laser range-finder. Với yêu cầu này từ package và như mục 3.1.2 đã đề cập, chúng ta sẽ sử dụng stereo camera để chuyển dữ liệu nhận được (depth image) về kiểu dữ liệu như laser range-finder. Sau đây là các bước để xây dựng bản đồ stereo camera:

3.2.2.1. Thu thập dữ liệu từ hai camera và chuyển dữ liệu về ảnh độ sâu

Chúng ta dùng ZED camera của Stereolabs là một lợi thế vì nó đã hỗ trợ các package cũng như các tool cần thiết cho việc sử dụng camera này. Với package này cũng cấp một gói ZED SDK để sử dụng ZED stereo Camera với hệ điều hành ROS. Nó có thể cung cấp hình ảnh từ hai camera, ảnh độ sâu, 3D point và 6-DOF tracking.

zed_ros_wrapper package cung cấp rất nhiều topic để chúng ta sử dụng nhằm tăng tính linh hoạt và tiện dụng cho sản phẩm của họ, nhưng chúng ta chỉ cần một vài topic cần thiết để sử dụng cho mục đích của chúng ta. Một số topic cần dùng được liệt kê dưới đây:

- **/zed/rgb/image_rect_color:** dùng để hiển thị hình ảnh của camera (mặc định là camera trái). Chúng ta không nên dùng topic **/zed/rgb/image_raw_color** vì hình ảnh nó chưa được cân chỉnh.
- **/zed/depth/depth_registered:** ảnh độ sâu dựa trên ảnh của camera trái. Kiểu dữ liệu của topic này có thể cài đặt với hai sự lựa chọn: 32 bits float với đơn vị là mét, hoặc 16 bits integer với đơn vị là milimét.
- **/zed/depth/camera_info:** thông tin của camera.
- **/zed/odom:** vị trí và hướng của camera.

Để sử dụng camera đúng thông số mong muốn, chúng ta phải thiết lập một số thông số cần thiết trong launch file:

```
<arg name="publish_tf"                default="false" />
<arg name="odometry_frame"            default="odom" />
<arg name="base_frame"                default="zed_center" />
<arg name="camera_frame"              default="zed_left_camera" />
<arg name="depth_frame"               default="zed_depth_camera" />
<!-- publish odometry frame -->
<param name="publish_tf"              value="$ (arg publish_tf)" />
<!-- Configuration frame camera -->
<param name="odometry_frame"          value="$ (arg odometry_frame)" />
/>
<param name="base_frame"              value="$ (arg base_frame)" />
```

<code><param name="camera_frame"</code>	<code>value="\$(arg camera_frame)" /></code>
<code><param name="depth_frame"</code>	<code>value="\$(arg depth_frame)" /></code>

Bảng 3.6 Các biến cần cấu hình cho zed_ros_wrapper

Trên đây là việc đổi các tên mặc định của package thành tên chúng ta mong muốn để có thể dễ dàng ghép nối chúng với các package nhằm tạo ra sự thông suốt trong cả hệ thống. Thẻ `<arg>` dùng để chỉ biến `name` sẽ có giá trị mặc định là `default`. Thẻ này luôn phải có một thẻ `<param>` tương ứng trong cùng một tệp để khai báo giá trị của biến `name` trong thẻ `param` có giá trị được định nghĩa trong thẻ `<arg>`.

<code><param name="resolution"</code>	<code>value="2" /></code>
<code><param name="quality"</code>	<code>value="2" /></code>
<code><param name="frame_rate"</code>	<code>value="30" /></code>
<code><param name="openni_depth_mode"</code>	<code>value="1" /></code>

Bảng 3.7 Cấu hình các thông số cần thiết cho stereo camera

Biến `resolution` dùng để chọn độ phân giải cho camera ('0': HD2K, '1': HD1080, '2': HD720, '3': VGA).

Biến `quality` để chọn chất lượng của depth map ('0': NONE, '1': PERFORMANCE, '2': MEDIUM, '3': QUALITY).

Biến `frame_rate` nhằm chọn tốc độ frame cho camera video.

Biến `openni_depth_mode` dùng để chuyển ảnh độ sâu 32bit (đơn vị là mét) sang 16bit (đơn vị là milimét).

3.2.2.2. Chuyển ảnh độ sâu về kiểu dữ Laser scan

Từ ảnh độ sâu thu được từ bước 3.2.2.1, ta cần phải chuyển sang kiểu dữ liệu mà cảm biến laser range-finder (`sensor_msgs/LaserScan.msg`). Nhằm hỗ trợ việc này, tác giả Chad Rockey đã có một package tên ***depthimage_to_laserscan*** nhằm chuyển đổi từ ảnh độ sâu sang kiểu dữ liệu `LaserScan`.

depthimage_to_laserscan package sẽ lấy ảnh độ sâu (`sensor_msgs/Image`) (hỗ trợ cả 32bit float và 16bit unsigned integer) và thông tin camera đi kèm với ảnh lấy

vào (`sensor_msgs/CameraInfo`). Sau đó, nó sẽ sinh ra một tín hiệu 2D laser scan (`sensor_msgs/LaserScan`) dựa trên các thông tin mà nó nhận được. Đây là một package chỉ subscribe `image` và `camera_info` khi có một node khác subscribe `scan` (người ta gọi đây là lazy subscribing).

Để tiện sử dụng và cấu hình, việc dùng launch file là một sự lựa chọn hợp lý nên ta cần cấu hình nó.

```
<param name="scan_height" value="50"/>
<param name="output_frame_id" value="/zed_center"/>
<param name="range_min" value="0.5"/>
<param name="range_max" value="8"/>
<remap from="image" to="/zed/depth/depth_registered"/>
<remap from="scan" to="/scan"/>
```

Bảng 3.8 Các thông số cần cấu hình cho `depthimage_to_laserscan`

`scan_height` là số thể hiện số hàng pixels được sử dụng để sinh ra tín hiệu laser scan. Đối với mỗi cột, nó sẽ trả về giá trị nhỏ nhất cho các pixel tập trung theo chiều dọc của ảnh.

`output_frame_id` là frame id của dữ liệu laser scan.

`range_min`, `range_max` là ngưỡng tối thiểu và tối đa của tín hiệu laser scan. Giá trị trả về $-Inf$ nếu nhỏ hơn `range_min` hoặc bằng $+Inf$ nếu lớn hơn giá trị `range_max`.

`image` là tên topic của ảnh độ sâu.

`scan` là tên topic của dữ liệu laser scan được publish ra.

`camera_info` sẽ tự động subscribe tới cùng một namespace với `image` nên chúng ta không cần khai báo giá trị cho nó.

3.2.2.3. Ước lượng tọa độ của robot từ laser scan:

Chúng ta có rất nhiều phương pháp để xác định vị trí của robot so với frame gốc ban đầu. Ở đây, chúng ta có thể kể đến như ước lượng tọa độ từ encoder, dùng

xử lý ảnh để xác định khoảng cách dịch chuyển, định vị vị trí dùng cảm biến đo mức quán tính (Inertial Measurement Unit - IMU), xác định tọa độ dựa vào các tín hiệu laser scan liên tiếp nhau,...

Dựa vào dữ liệu có sẵn là các message laser scan liên tiếp nhau được chuyển đổi từ camera, nên việc xác định tọa độ dựa vào các message này là việc khá thuận lợi và có thể không cần phải dùng đến cảm biến khác. Để hỗ trợ cho công việc này, chúng ta có thể dùng *laser_scan_matcher* package. Như đã nói ở trên, package có thể sử dụng mà không cần sự hỗ trợ của các cảm biến khác. Nhưng chúng ta có thể đưa thêm dữ liệu các cảm biến khác để tăng tính chính xác cho hệ thống. Ngoài việc sử dụng dữ liệu scan, nó còn có thể chấp nhận một số cảm biến khác như:

- **IMU:** cảm biến này ước tính sự thay đổi của hướng robot dưới dạng message `sensor_msgs/IMU`. Package này giả sử thành phần yaw của IMU sẽ tương ứng với hướng của robot.
- **Wheel odometry:** ước tính sự thay đổi tọa độ (x, y, θ) của robot từ các cảm biến đo lường sự thay đổi của odometry (encoder là một ví dụ điển hình).
- **Constant velocity model:** giả sử nếu robot chỉ di chuyển dựa vào ước tính của tốc độ robot, thì vận tốc ước tính này có thể lấy từ một cảm biến bên ngoài hoặc được xuất ra từ các dữ liệu laser scan tương thích nhau trong chính package của nó.
- **Zero-velocity model:** không sử dụng bất kì một dự đoán nào, ví dụ đang đứng yên một chỗ.

Để sử dụng package phù hợp với hệ thống của chúng ta, ta cần phải chỉnh lại một số thông số như sau:

```
<param name="fixed_frame" value = "odom"/>
<param name="base_frame" value="base_frame"/>
<param name="use_odom" value="false"/>
<param name="use_imu" value="false"/>
<param name="publish_pose" value="true"/>
<param name="publish_tf" value="true"/>
```


Bảng 3.9 Các thông số cần cấu hình cho laser_scan_matcher

Biến `fixed_frame` dùng để xác định đâu là frame cố định của hệ thống để package có thể tính được vị trí của base frame so với `fixed_frame`.

Biến `use_odom` để xác định xem chúng ta có cung cấp nguồn dữ liệu khác là cảm biến đo lường odometry hay không.

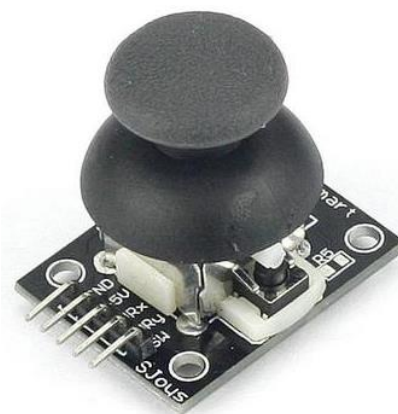
Biến `use_imu` giống như biến `use_odom` là xác định là chúng ta có đưa thêm cảm biến IMU vào hay không.

Biến `publish_pose`, `publish_tf` dùng khai báo xem package này có publish các thông số tf và pose vào môi trường ROS hay không.

3.2.2.4. Thiết lập joytick để điều khiển Mobile base di chuyển

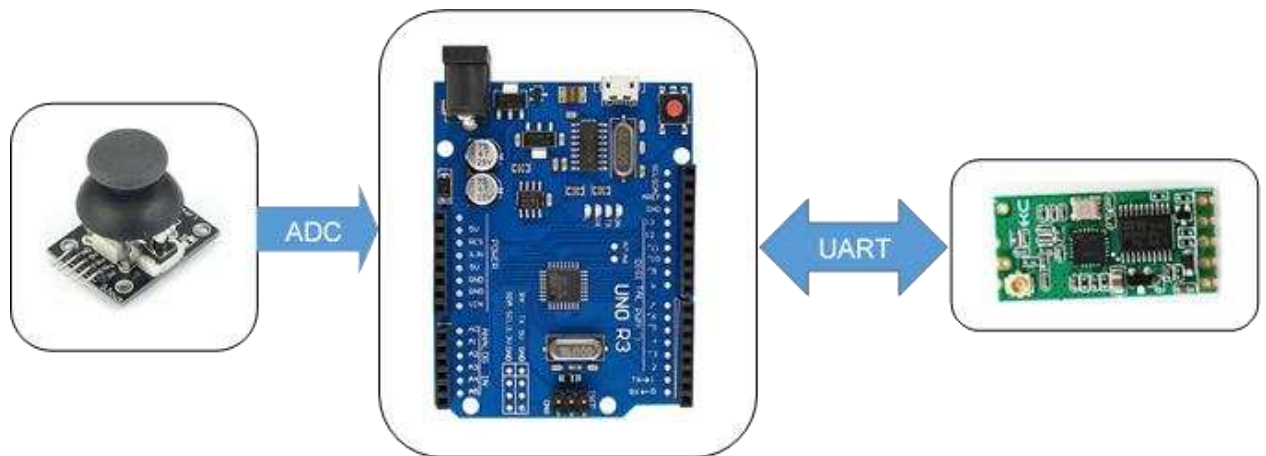
Nhằm thuận tiện cho việc điều khiển xe để nó di chuyển như mong muốn, chúng ta sẽ thiết lập một thiết bị để điều khiển xe dễ dàng hơn trong quá trình dựng bản đồ. Mục tiêu đặt ra là cần có một thiết bị để dễ dàng điều khiển xe tới/lùi, xoay trái/phải và chuyển tín hiệu từ cảm biến đó thành giá trị tương tự `cmd_vel` để **Mobile base** có thể hoạt động được.

Với mục tiêu đặt ra, joytick là một module có thể dùng điều khiển dễ dàng và phù hợp.



Hình 3.21 Module joytick

Joytick module có ngõ ra là ADC nên việc đọc và chuyển tín hiệu dễ dàng. Để tiết kiệm thời gian, dùng Arduino sẽ cho ta phương án dễ dàng và nhỏ gọn.



Hình 3.22 Sơ đồ thiết lập joytick điều khiển Mobile base

3.2.2.5. Sử dụng package gmapping để dựng bản đồ

Như đầu mục 3.2.2 đã đề cập đến các topic cần thiết để dựng bản đồ. Ngõ ra của package này là một bản đồ 2D (`nav_msgs/OccupancyGrid`), có thể nói như là mặt sàn của môi trường đang dựng.

Tương tự như các package khác, chúng ta cần thiết lập launch file cho **gmapping**. Trong đó, chúng ta có các thông số quan trọng sau:

```

<param name="map_udpate_interval" value="2.0"/>
<param name="delta" value="0.02"/>
<param name="base_frame" value="base_frame"/>
<param name="xmin" value="-20"/>
<param name="xmax" value="20"/>
<param name="ymin" value="-20"/>
<param name="ymax" value="20"/>
  
```

Bảng 3.10 Các thông số cần cấu hình cho gmapping

`map_udpate_interval` là thời gian giữa hai lần cập nhật map tính bằng giây.

`delta` là độ phân giải của map (đơn vị là mét trên một khối occupancy grid).

`base_frame` là frame được gắn với **Mobile base**.

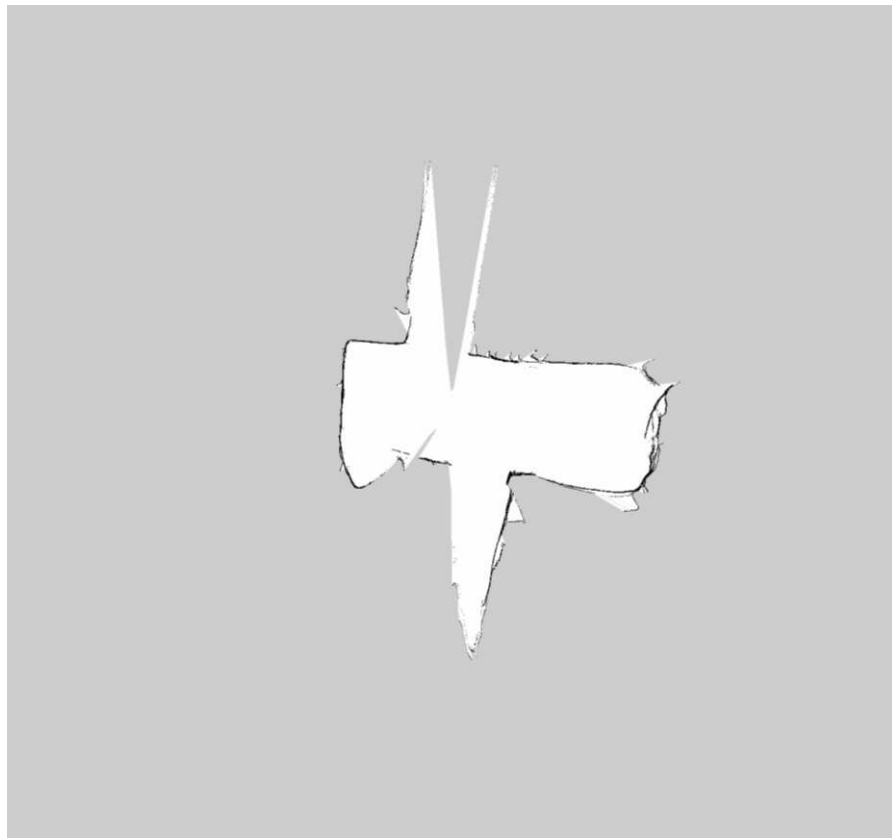
`xmin, xmax, ymin, ymax` là quy định kích thước của bản đồ (đơn vị là mét).

3.2.2.6. Lưu bản đồ

Sau khi di chuyển robot trong môi trường cần dựng và kết quả map trả về là tốt nhất thì chúng ta cần lưu lại. Việc lưu lại nên được thực hiện bằng command trên terminal. Bất cứ khi nào mà mình muốn lưu lại bản đồ đã dựng thì gõ lệnh:

```
roslaunch map_server map_saver [-f mapname]
```

Trong đó `mapname` là tên bản đồ do chúng ta đặt tên. Sau khi lưu, chúng ta sẽ nhận được hai file để mô tả cho bản đồ mà chúng ta đã lưu là file ảnh `pgm` và file mô tả ảnh `yaml`.



Hình 3.23 Hình ảnh bản đồ sau khi dựng

File ảnh sẽ mô tả các không gian bị chiếm dụng (occupancy) trong môi trường. Theo tiêu chuẩn, các pixel màu trắng sẽ là khoảng không gian trống; các pixel màu đen là nơi các pixel đã có vật thể ở đó (như tường, vật cản,...); các pixel màu xám là những vùng mà robot chưa đi qua.

File yaml sẽ có các thông tin sau:

```
image: map-a.pgm
resolution: 0.020000
origin: [-20.000000, -20.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Bảng 3.11 Thông số bản đồ đã dựng

`image` là đường dẫn đến ảnh chứa thông tin, đây có thể là đường dẫn tương đối hoặc tuyệt đối so với vị trí của file yaml.

`resolution` là độ phân giải của bản đồ (mét/pixel).

`origin` là tọa độ của pixel bên trái dưới cùng và việc xoay của bản đồ (thông số cuối thường được bỏ qua).

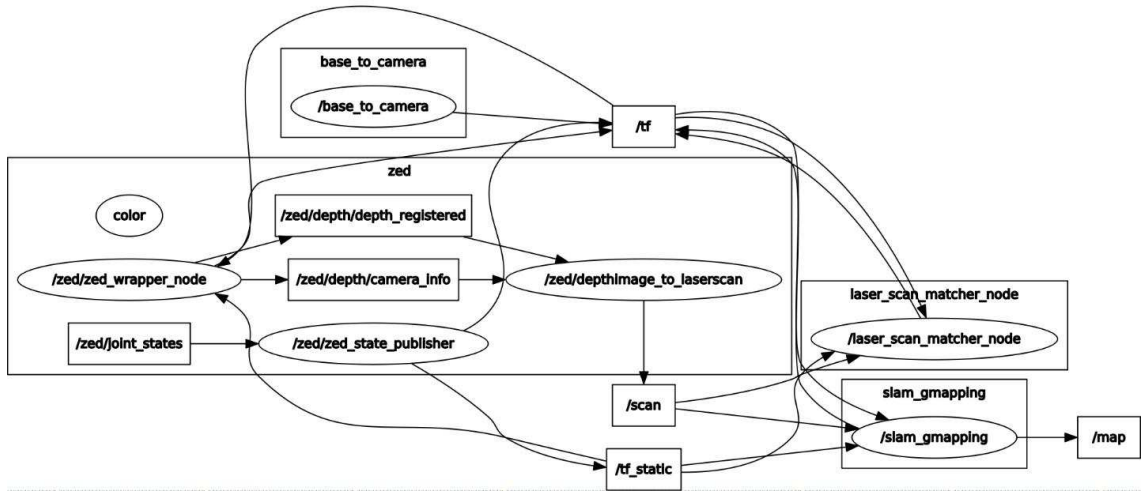
`negate` nếu bằng 0 thì màu trắng là không gian trống, màu đen là không gian bị chiếm dụng. Nếu bằng 1 thì ngược lại.

`occupied_thresh` là ngưỡng để xem xét một pixel có phải là đang bị chiếm dụng hay không.

`free_thresh` là ngưỡng để xem xét một pixel có phải đang ở trong không gian trống hay không.

3.2.2.7. Tổng hợp các node trong dựng bản đồ

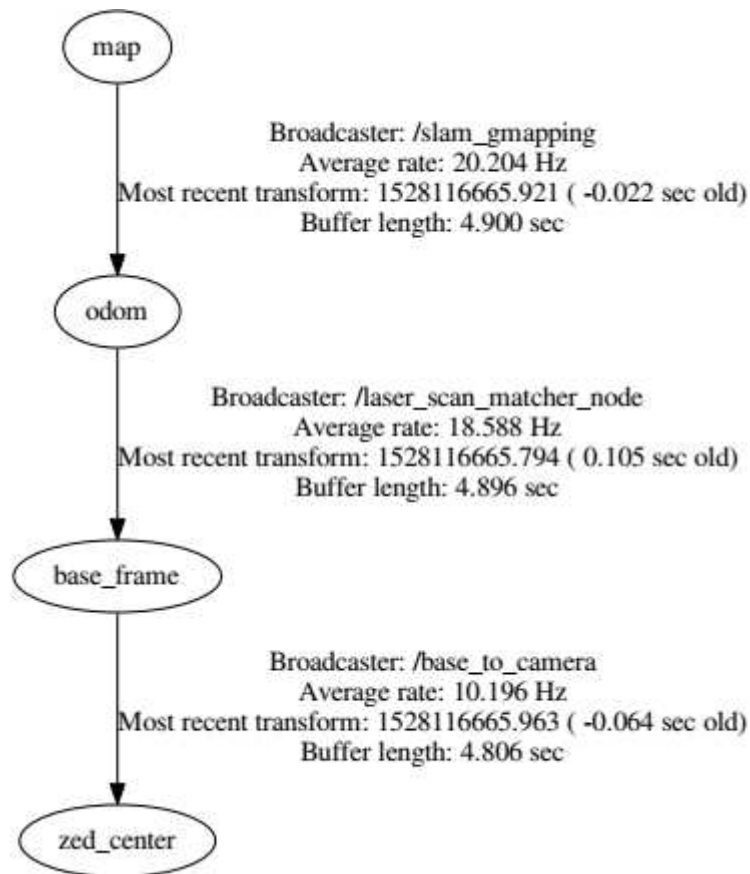
Sau khi chúng ta thiết lập hoàn tất các cấu hình cần thiết để cho hệ thống, chúng ta sẽ kiểm tra lại tính đúng đắn và luồng các dữ liệu của các node để đảm bảo chương trình của chúng ta hoạt động ổn định. Có hai đồ thị chúng ta cần phải duyệt lại là xem lại dữ liệu đi như thế nào (dùng `rqt_graph`) và sơ đồ cây của các frame (dùng `package tf` để xuất ra sơ đồ này).



Hình 3.24 Sơ đồ các node trong mapping

Qua đồ thị trên, các luồng data đã có hướng đi hợp lý, các node publish và subscribe đúng với hệ thống. Ta thấy node ***depthimage_to_laserscan*** nhận dữ liệu ảnh độ sâu (`sensor_msgs/Image`) và thông tin về cấu hình camera (`sensor_msgs/CameraInfo`) từ node ***zed_ros_wrapper***; và nó sẽ đưa ra dữ liệu scan (`sensor_msgs/LaserScan`) để các node ***laser_scan_matcher_node***, ***slam_mapping*** subscribe. Tại node ***slam_gmapping*** khi đã subscribe được các topic cần thiết thì sẽ publish bản đồ chiếm dụng thông qua topic map (`nav_msgs/OccupancyGrid`).

Bên cạnh xem các node liên kết với nhau như thế nào, chúng ta còn phải xem một việc quan trọng khác là các frame được kết nối với nhau hay chưa và những node nào sẽ publish thông số liên hệ giữa các frame.



Hình 3.25 Cấu trúc frame trong mapping

Trên đây là sơ đồ mà các frame liên kết với nhau. Frame map là frame toàn cục (world frame), đây sẽ là frame cố định mà các frame khác phải dựa trên frame này để tính ra vị trí tương đối của nó. Odom frame là frame con của map frame, việc chuyển đổi này do *slam_gmapping* node sinh ra với tần số cập nhật là 20Hz. Tương tự cho các frame còn lại, chúng ta phải xem xét những node nào đang sinh ra việc chuyển đổi từ frame này đến frame khác và tần số cập nhật giữa các lần chuyển đổi là bao nhiêu để cân chỉnh phù hợp với hệ thống. Một điều lưu ý và rất quan trọng, là tất cả các frame được tham chiếu trong hệ thống phải có liên kết với nhau và được liên kết với frame toàn cục. Mỗi một frame trong sơ đồ chỉ có thể tối đa một frame cha và có thể có nhiều frame con. Sơ đồ cây thể hiện mối liên kết giữa các frame là rất quan trọng và ảnh hưởng trực tiếp tới việc hệ thống chúng ta có hoạt động được hay không.

3.2.3. Điều hướng trên bản đồ đã dựng (navigation)

3.2.3.1. Các bước chuẩn bị dữ liệu cho navigation

Để điều hướng cho xe có thể đến được đích và an toàn (tránh vật cản) thì ta vẫn phải dùng cảm biến để xem xét môi trường. Nên các bước chuyển đổi kiểu dữ liệu từ ảnh độ sâu sang dữ liệu laser scan là giống như bước dựng bản đồ; ngoài ra, để xác định vị trí của xe trong bản đồ thì vẫn cần đến *laser_scan_matcher_node* để thực hiện nhiệm vụ này.

3.2.3.2. Ước lượng vị trí của robot dùng Adaptive Monte Carlo Localization (AMCL)

AMCL là một hệ thống probabilistic localization cho robot di chuyển trong mặt phẳng 2D dùng phương pháp adaptive Monte Carlo localization dựa trên bộ lọc hạt (particle filter) để tính ra vị trí của robot dựa trên bản đồ đã biết. Ở đây, chúng ta dùng *AMCL* package dùng để tính toán odom của hệ thống thông qua quá trình di chuyển của robot. Để package hoạt động chính xác, chúng ta cần cấu hình lại một vài thông số để phù hợp với mục tiêu của chúng ta:

```
<param name="odom_frame_id" value="odom"/>
<param name="base_frame_id" value="base_frame"/>
<param name="use_map_topic" value="true"/>
<param name="min_particles" value="20"/>
<param name="max_particles" value="400"/>
```

Bảng 3.12 Các thông số cấu hình cho amcl

`odom_frame_id` là biến để khai báo frame nào là odom frame trong hệ thống của chúng ta. Tương tự cho `base_frame_id` là dùng để xác định frame nào là `base_frame` của hệ thống.

Biến `use_map_topic` dùng để thông báo với package rằng nó sẽ phải subscribe map topic chứ không cần phải lấy dữ liệu thông qua service.

`min_particles`, `max_particles` là thông số để khai báo số hạt tối thiểu và tối đa cho việc tính toán xác suất. Nếu chúng ta đặt số này càng lớn thì kết quả có khả năng chính xác hơn nhưng bù lại phải tính toán nhiều hơn nên chúng ta cần cân nhắc hợp lý với khả năng đáp ứng của hệ thống chúng ta.

3.2.3.3. Thiết lập các thông số cấu hình để cung cấp cho `move_base` hoạt động

Thiết lập `costmap_common_params.yaml`

```
obstacle_range: 2.5
raytrace_range: 3.0
robot_radius: 0.3
inflation_radius: 0.3
observation_sources: laser_scan_sensor #point_cloud_sensor
laser_scan_sensor: {sensor_frame: zed_center, data_type: LaserScan,
topic: /scan, marking: true, clearing: true}
```

Bảng 3.13 File `costmap_common_params.yaml`

Các thông số này đã được đề cập ở mục 2.1.3.3 ở trên. Các thông số này phụ thuộc rất nhiều vào đặc tính của từng robot và môi trường mà robot hoạt động.

Thiết lập `global_common_params.yaml`

```
global_costmap:
  global_frame: map
  robot_base_frame: base_frame
  update_frequency: 1.0
  publish_frequency: 1.0
  resolution: 0.05
  static_map: true
  width: 40.0
  height: 40.0
  map_type: costmap
```


Bảng 3.14 File `global_common_params.yaml`

Ở đây, chúng ta nên khai báo kích thước của bản đồ (thông số `width`, `height`) để cho *move_base* biết được kích thước mà bản đồ của chúng ta sẽ đưa vào. Ngoài ra, ta nên cân nhắc việc lựa chọn độ phân giải đưa vào hệ thống (thông số `resolution`) vì nó sẽ ảnh hưởng đến khả năng tính toán của hệ thống để tìm ra được đường đi cho robot trong bản đồ mà chúng ta đưa vào.

Thiết lập `local_common_params.yaml`

```
local_costmap:
  global_frame: map
  robot_base_frame: base_frame
  update_frequency: 2.0
  publish_frequency: 1.0
  static_map: false
  rolling_window: true
  width: 6.0
  height: 6.0
  resolution: 0.05
```

Bảng 3.15 File `local_common_params.yaml`

Ý nghĩa các thông số tương tự như mục 2.1.3.3 ở trên.

Tiếp theo, để thiết lập cho local planner hợp lý thì ta xác định xem chúng ta dùng thuật toán nào để thông báo cho *move_base* biết được các thông số cần thiết cho thuật toán này. Vì đề tài sử dụng thuật toán DWA nên chúng ta cần cấu hình file `dwa_local_planner_params.yaml` như sau:

```
DWAPlannerROS:
  acc_lim_x: 0.1
  acc_lim_y: 0
  acc_lim_th: 0.05
  max_trans_vel: 1.0
```

```
min_trans_vel: 0.0

max_vel_x: 0.5

min_vel_x: 0.1

max_vel_y: 0

min_vel_y: 0

max_rot_vel: 0.6

min_rot_vel: 0.0

yaw_goal_tolerance: 0.7

xy_goal_tolerance: 0.2
```

Bảng 3.16 File `dwa_local_planner_params.yaml` cho mô hình thực tế

Để tương thích với `move_base`, chúng ta chỉ thiết lập vận tốc thẳng theo trục x và vận tốc xoay, còn vận tốc thẳng theo trục y thì ta cho bằng 0. Các thông số gia tốc, vận tốc tối đa, tối thiểu thì phụ thuộc vào đặc tính của robot và môi trường hoạt động. Chúng ta cần phải cân chỉnh các thông số này trong quá trình thực nghiệm để đưa ra được những thông số phù hợp nhất. Hai thông số cũng rất quan trọng là `yaw_goal_tolerance` và `xy_goal_tolerance` để xác định xem khi nào robot đã đến đích. Hai thông số này cũng có thể xác định xem sai số vị trí của robot khi nó di chuyển về đích. Ta cần phải lựa chọn thông số này hợp lý để robot có thể đến đúng vị trí mong muốn mà đạt sai số nhỏ nhất về cả vị trí và hướng. Nếu ta thiết lập hai thông số này nhỏ thì robot sẽ đến chính xác vị trí hơn nhưng có thể robot sẽ không thể tự nhận là nó đã đến vị trí mong muốn vì có thể có sai số từ cảm biến, độ trượt của xe,... Ngược lại, nếu ta thiết lập thông số này quá lớn thì robot chưa đến được đích thì nó đã tự nhận rằng nó đã đến đích và dừng lại.

3.2.3.4. Thiết lập `move_base` để điều hướng cho xe

Nhằm để thiết lập một phần quan trọng trong Navigation stack, thì tác giả Eitan Marder-Eppstein đã cung cấp cho ta một package tên là ***move_base*** để thiết lập, tổng hợp các dữ liệu, các node cần thiết cho quá trình điều hướng.

Như mục 2.1.3 đã cho ta cái nhìn tổng quan về Navigation stack. Sau đây, chúng ta sẽ thiết lập vài thông số cần thiết để *move_base* hoạt động được:

```
<rosparam file="$(find navigation)/config/costmap_common_params.yaml"
command="load" ns="global_costmap" />

<rosparam file="$(find navigation)/config/costmap_common_params.yaml"
command="load" ns="local_costmap" />

  <rosparam file="$(find navigation)/config/local_costmap_params.yaml"
command="load" />

<rosparam file="$(find navigation)/config/global_costmap_params.yaml"
command="load" />

<rosparam file="$(find navigation)/config/dwa_local_planner_params.yaml"
command="load" />

<param name="controller_frequency" value="10"/>

<remap from="odom" to="/zed/odom"/>

<remap from="scan" to="/scan"/>

<param name="base_local_planner" value="dwa_local_planner/DWAPlanner-
ROS"/>
```

Bảng 3.17 Các thông số cấu hình cho move_base

Đầu tiên, chúng ta sẽ cần phải tải lên các thông số cấu hình cần thiết cho *global_costmap*, *local_costmap*, *global_planner* và *local_planner* hoạt động. Các thông số này được lưu trong những file *yaml* ở mục 3.2.3.3 ở trên.

Biến *controller_frequency* dùng để chọn tần số cập nhật để gửi giá trị vận tốc xuống cho *Mobile base*. Thông số này nên chọn phụ thuộc vào cấu hình của máy tính vì nếu chọn tần số cập nhật quá lớn so với khả năng tính toán của hệ thống thì nó sẽ cảnh báo và hoạt động không như mong muốn.

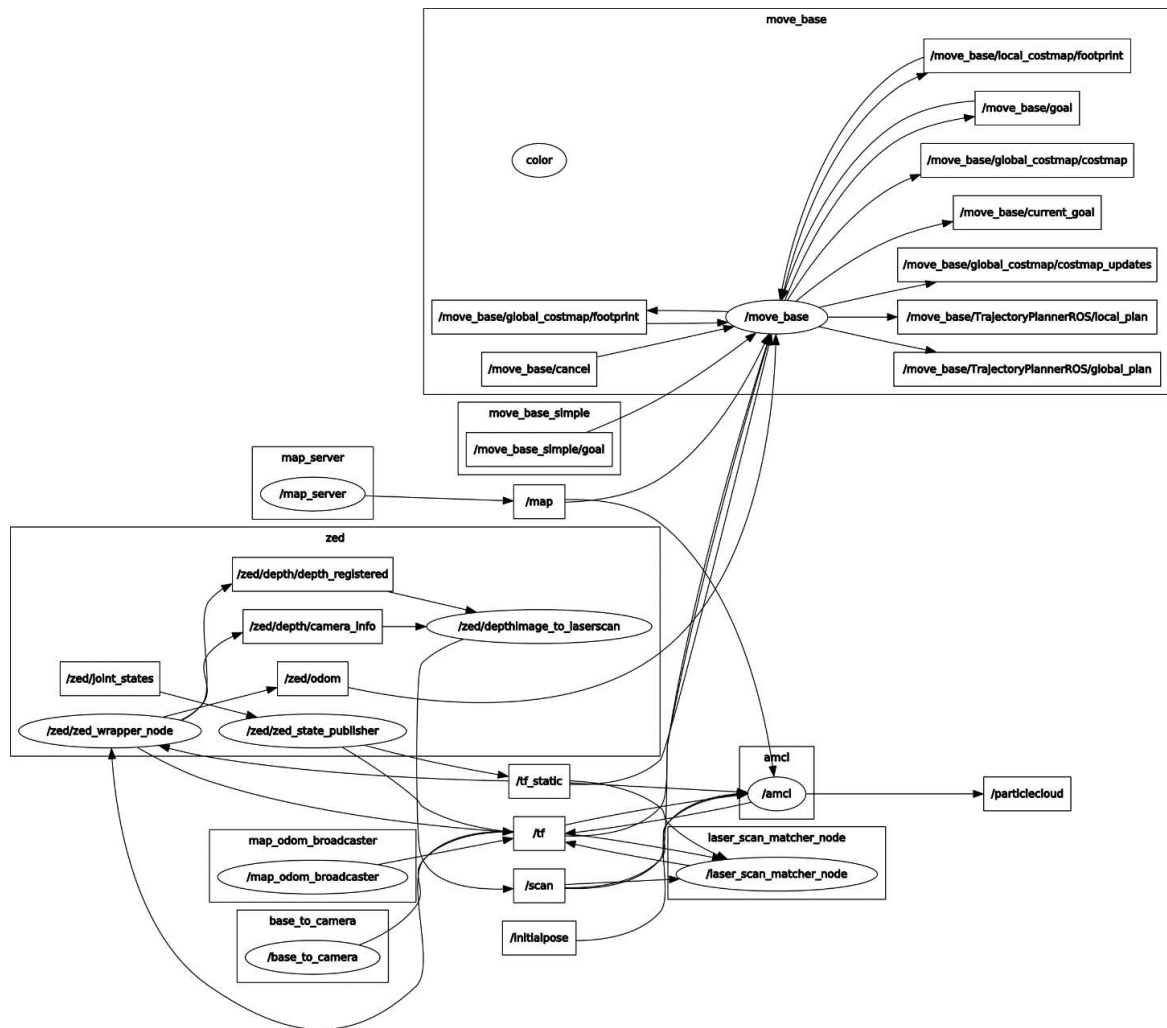
Biến *odom* dùng để khai báo rằng topic nào đang chứa dữ liệu về odom của hệ thống. Tương tự như laser scan.

Một thông số khá quan trọng là *base_local_planner*, thông số này thông báo với hệ thống là đang dùng thuật toán nào để hoạch định đường đi cục bộ. Nếu ta

không chỉnh lại thông số này thì mặc định của nó là `base_local_planner/TrajectoryPlannerROS`.

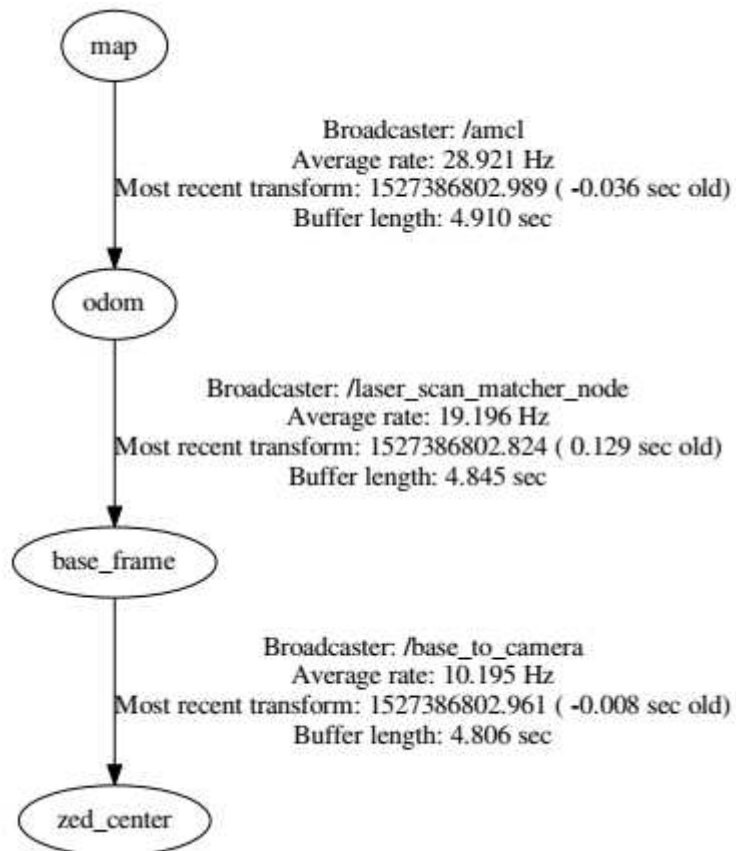
3.2.3.5. Tổng hợp lại các node được dùng trong điều hướng

Sau quá trình cấu hình Navigation stack, ta cần phải xem lại tính đúng đắn của cả hệ thống về luồng dữ liệu và sơ đồ cấu trúc frame.



Hình 3.26 Các node trong move_base

Theo như sơ đồ ở trên, **amcl** package subscribe hai topic chính là scan và map như đúng cấu hình. Tương tự, **move_base** package cũng đã subscribe các topic cần thiết để có thể thực hiện các tác vụ của mình để đưa ra giá trị vận tốc hợp lý theo thuật toán của nó.



Hình 3.27 Sơ đồ frame trong move_base

Qua thành các thành phần cấu nên cấu trúc khung cho hệ thống, ta thấy các node đã publish đúng các dữ liệu thể hiện mối quan hệ giữa các frame với nhau.

Chương 4. KẾT QUẢ THỰC NGHIỆM

Sau quá trình xây dựng mô hình Mobile base, thiết lập môi trường hệ điều hành ROS trên board nhúng, cấu hình và lập trình các node cần thiết cho quá trình trong hệ thống thì chúng sẽ kiểm chứng thực nghiệm để cân chỉnh các thông số và tìm ra các ưu điểm, khuyết điểm của hệ thống.

4.1. Lấy các dữ liệu cần thiết từ camera trên hệ điều hành ROS

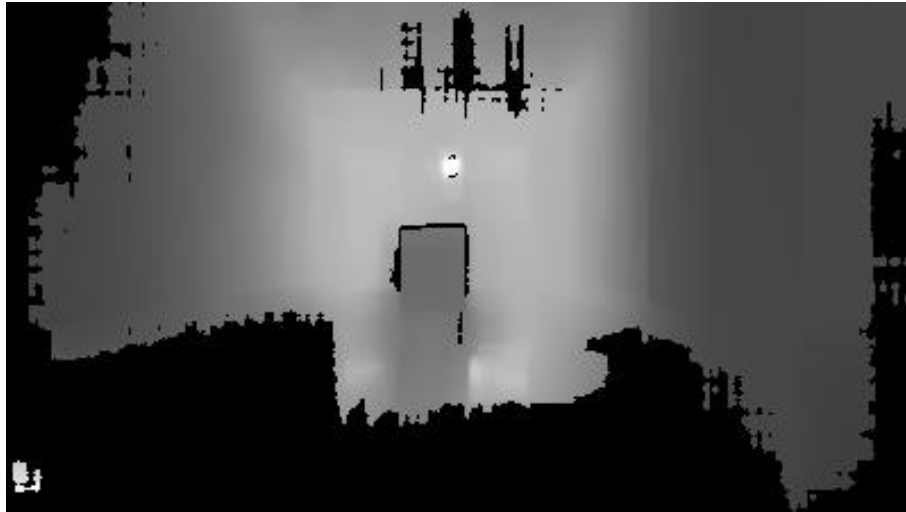
Để thực hiện quá trình, chúng ta sẽ lấy các giá trị từ các topic mà *zed_ros_wrapper* publish lên môi trường hệ điều hành ROS.



Hình 4.1 Hình ảnh từ camera trái



Hình 4.2 Hình ảnh từ camera phải

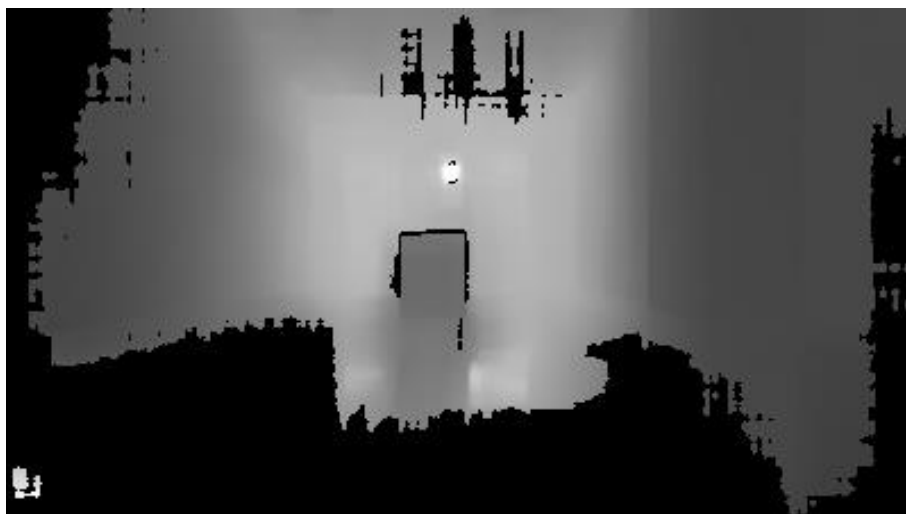


Hình 4.3 Hình ảnh độ sâu tính toán được

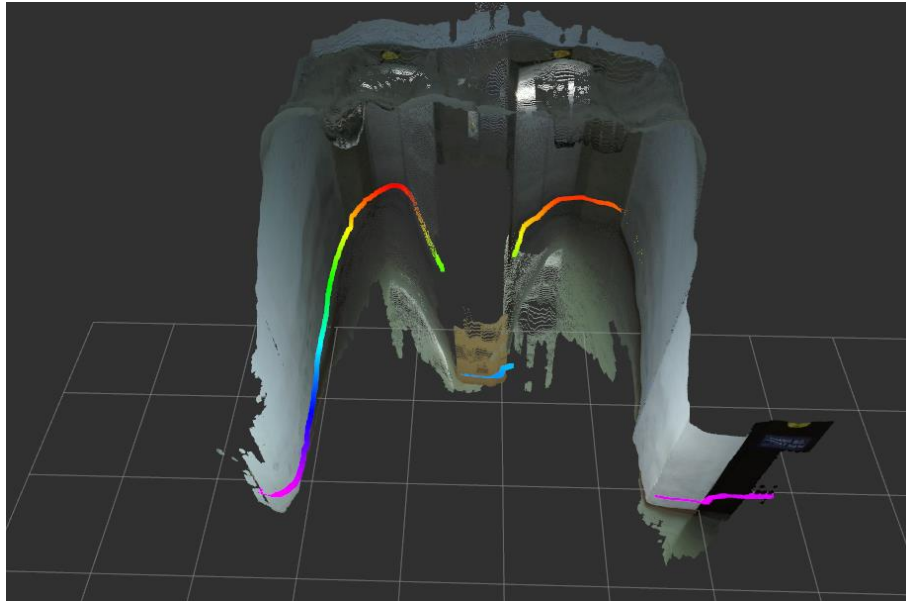
Nhờ sự tiện dụng và linh hoạt của package mà chúng ta có thể dễ dàng có được dữ liệu cần thiết cho hệ thống chúng ta.

4.2. Chuyển ảnh độ sâu thành laser scan:

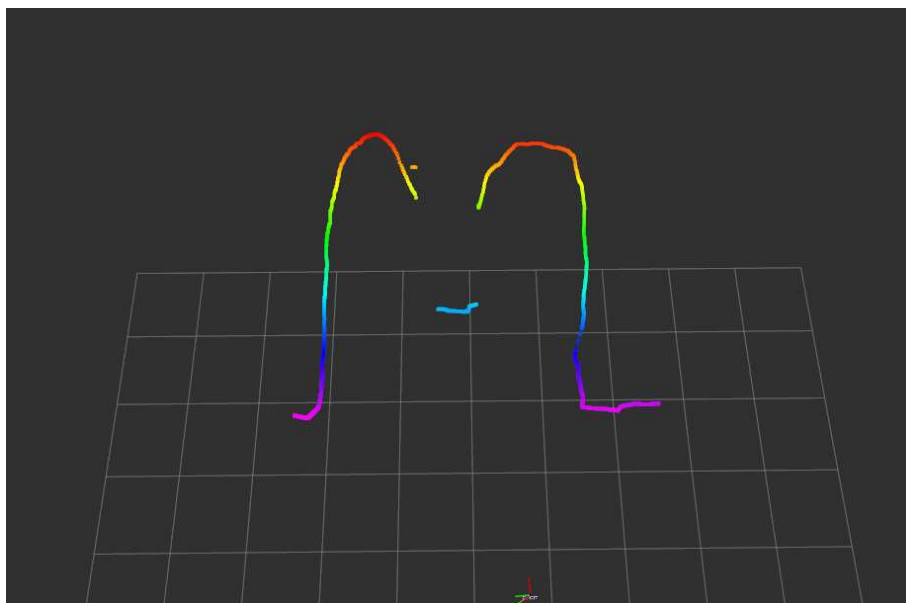
Như đã đề cập, chúng ta cần kiểu dữ liệu laser scan cho cả đề tài nên việc chuyển đổi này là cần thiết và quan trọng. Việc *depthimage_to_laserscan* cho ta dữ liệu laser scan từ camera cho ta kết quả có thể chấp nhận được vì một số đặc tính của camera.



Hình 4.4 Ảnh độ sâu thu được từ stereo camera



Hình 4.5 So sánh độ chính xác của laser scan so với point cloud



Hình 4.6 Kết quả laser scan thu được

4.3. Quá trình mapping và kết quả bản đồ

Để có một bản đồ dựng sẵn cung cấp cho việc điều hướng thì chúng ta nên phải dựng bản đồ từ môi trường đó. Việc này sẽ cho kết quả chính xác hơn vì nếu ta sử dụng những cảm biến không phải có nguồn gốc thật sự là laser range-finder thì đặc tính của cảm biến sẽ là yếu tố trực tiếp để robot chúng ta cảm nhận môi trường như thế nào. Chúng ta cũng có thể tự vẽ lại một bản đồ theo đo đạc thực tế để cung

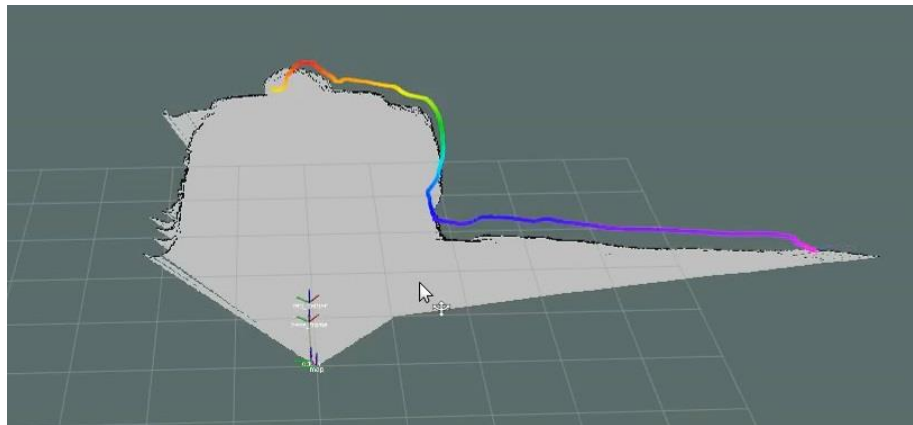
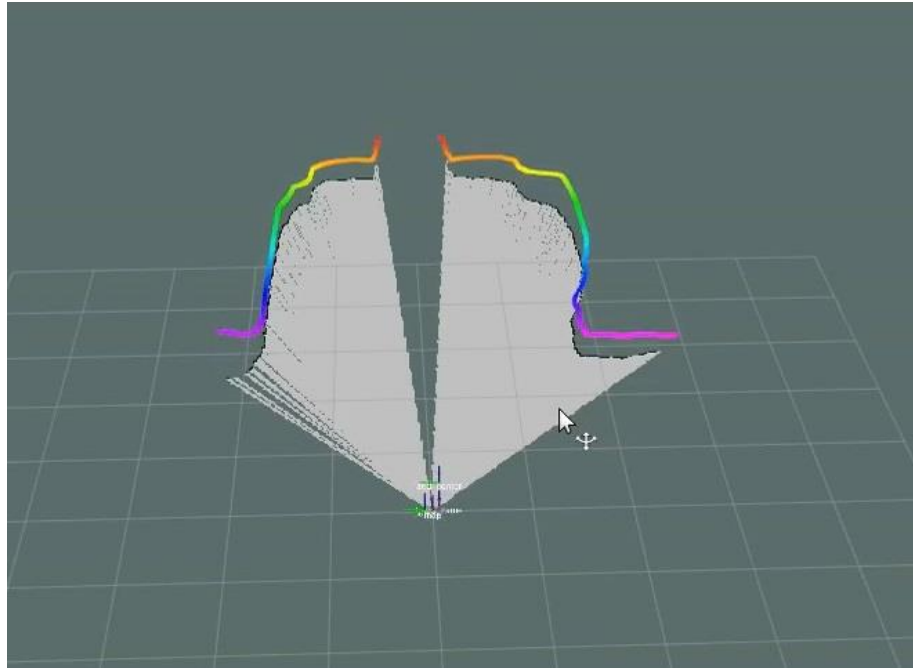
cấp cho Navigation stack, nhưng có thể nó sẽ không hoạt động chính xác như ta mong đợi.

Đây là hình ảnh môi trường thực tế để xây dựng bản đồ:



Hình 4.7 Không gian thực tế để dựng bản đồ

Một vài hình ảnh trong quá trình dựng bản đồ:



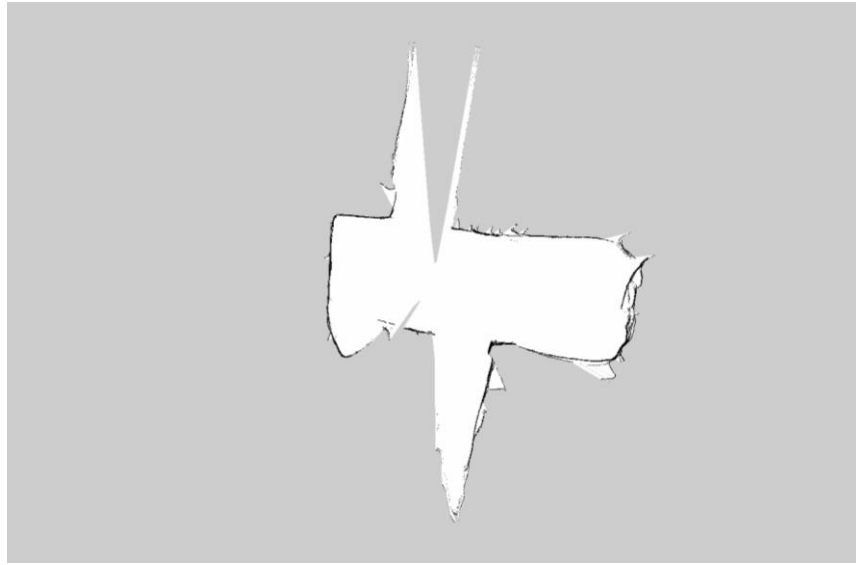
Hình 4.8 Hình thành bản đồ khi di chuyển Mobile base

Quá trình dựng bản đồ hoàn tất, chúng ta sẽ phải lưu lại kết quả dựa vào *map_server* package:

```
silent@silent-GL553VD:~$ rosrund map_server map_saver -f map-record-1
[ INFO] [1524646789.044782251]: Waiting for the map
[ INFO] [1524646789.281535662]: Received a 1984 X 1984 map @ 0.020 m/pix
[ INFO] [1524646789.281641190]: Writing map occupancy data to map-record-1.pgm
[ INFO] [1524646789.389281920]: Writing map occupancy data to map-record-1.yaml
[ INFO] [1524646789.389775705]: Done
```

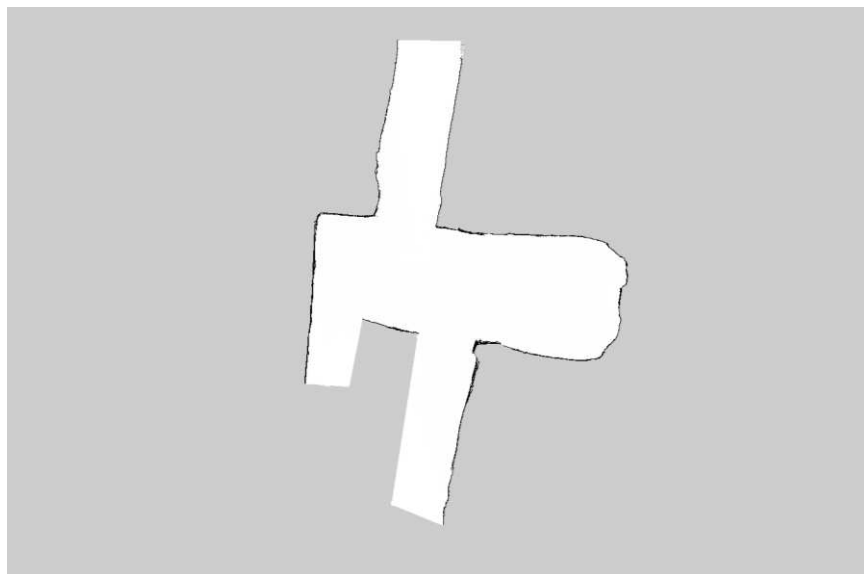
Hình 4.9 Lưu bản đồ đã dựng

Kết quả của bản đồ sau khi dựng:



Hình 4.10 Kết quả bản đồ sau khi dựng

Vì ta thấy nếu chuyển từ ảnh độ sâu thì dữ liệu laser scan sẽ không chính xác, đôi lúc khó phát hiện các góc cạnh nếu ít đặc trưng và sẽ có rất nhiều nhiễu. Để loại bỏ bớt một số nhiễu không mong muốn từ môi trường, ta có thể dùng một số công cụ chỉnh sửa ảnh (ví dụ như GIMP) để khắc phục lỗi trên. Nhưng lưu ý không nên chỉnh sửa quá nhiều sẽ dẫn đến dữ liệu cảm biến sẽ khác đi trên bản đồ làm cho hệ thống không phát hiện ra sự giống nhau.



Hình 4.11 Bản đồ sau khi loại bỏ nhiễu

Khi ta thử nghiệm ở môi trường có diện tích lớn hơn (hành lang có kích thước 42 x 2.5 mét) thì ta có được kết quả như sau:

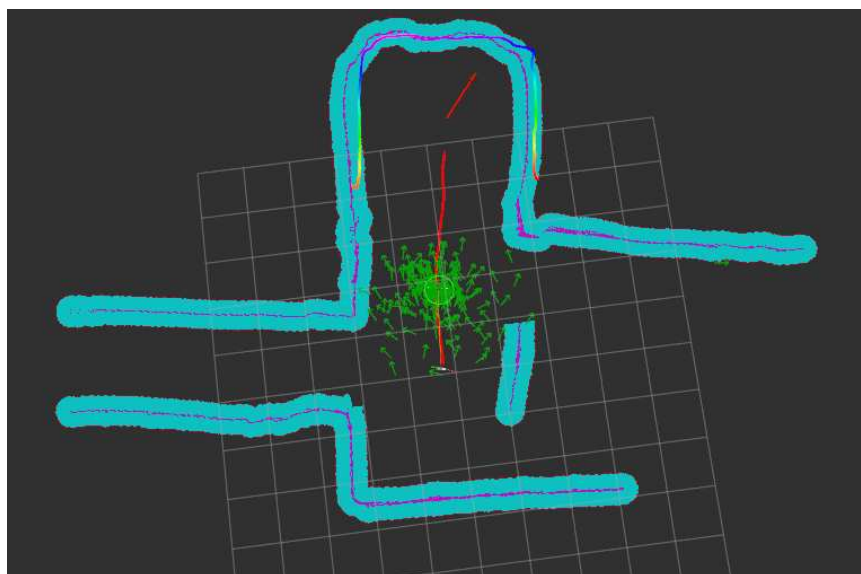


Hình 4.12 Bản đồ trong môi trường có diện tích lớn hơn

Khi robot đi trong môi trường rộng lớn hơn, ta thấy robot vẫn xây dựng được bản đồ và có kết quả tương đối tốt. Bên cạnh đó, như ta đã biết thì hành lang sẽ có hình dạng là đường thẳng nhưng kết quả cho ta thấy bản đồ có độ cong nhất định làm cho bản đồ xây dựng chưa chính xác. Nguyên nhân dẫn đến hiện tượng này là do giá trị odom ước lượng của hệ thống chưa chính xác.

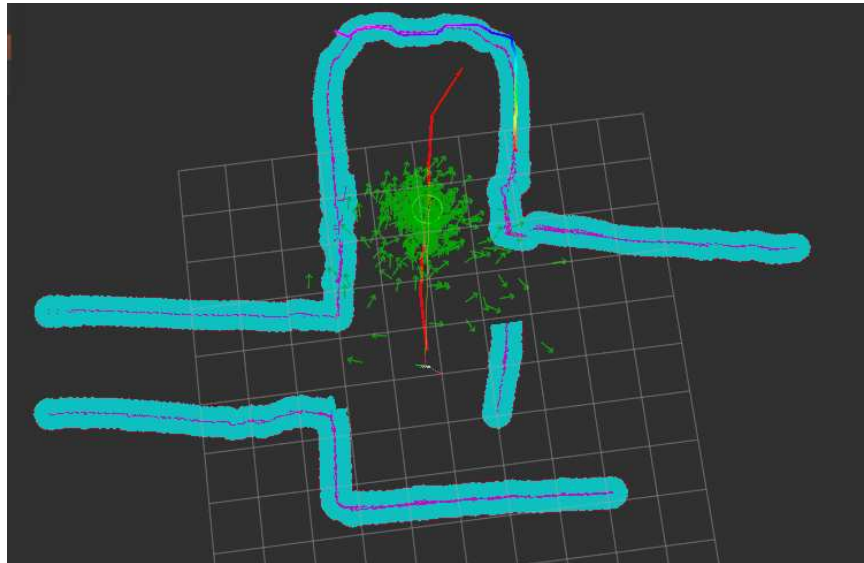
4.4. Quá trình điều hướng cho robot

Nếu môi trường giống ban đầu và robot đang nằm trong bản đồ đã dựng. Đầu tiên, ta phải khai báo cho nó được vị trí ban đầu. Sau đó, ta xác định cho nó biết được vị trí kết thúc của nó. Lúc này, `global_planner` sẽ hoạch định ra một đường đi từ vị trí bắt đầu đến vị trí đích của nó.



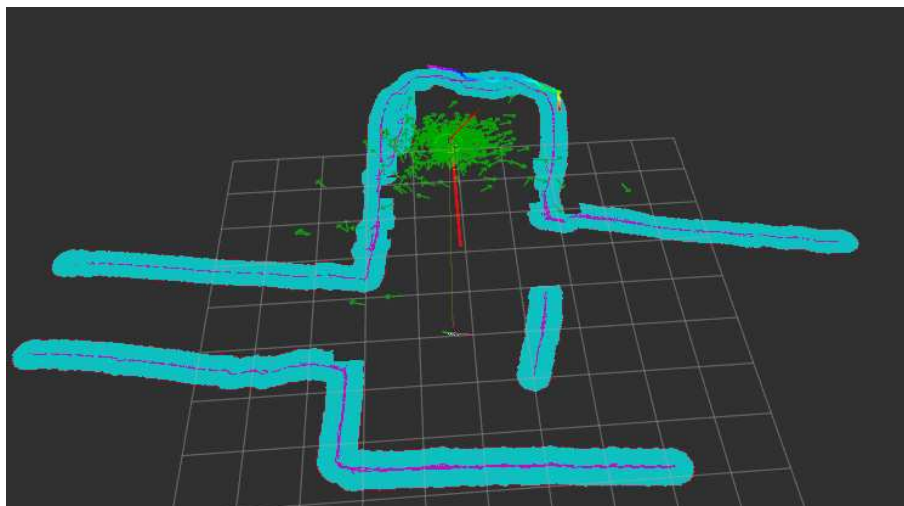
Hình 4.13 Hoạch định đường đi cho robot

Sau đó, nó sẽ tự di chuyển theo hoạch định đường đi mà nó tính toán được dựa vào `local_planner`.



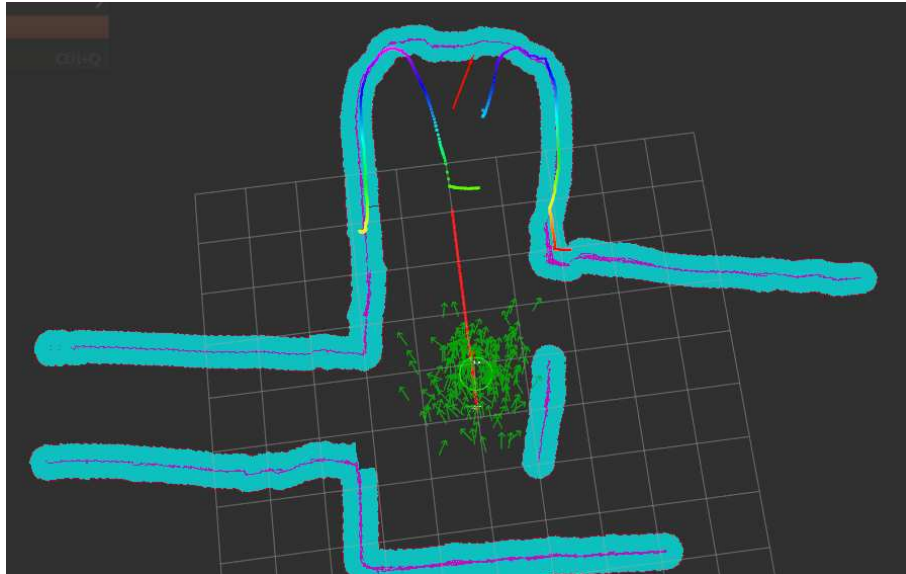
Hình 4.14 Robot đi theo đường đi đã hoạch định

Khi đến được đích, nó sẽ tự động dừng lại và báo với hệ thống rằng nó đã đến được vị trí đã được thiết lập.

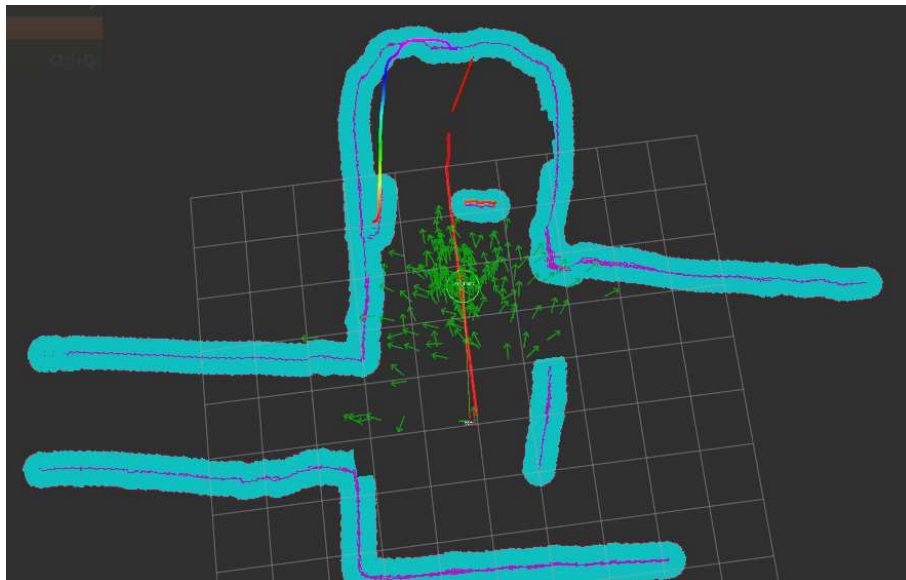


Hình 4.15 Khi robot đến đích

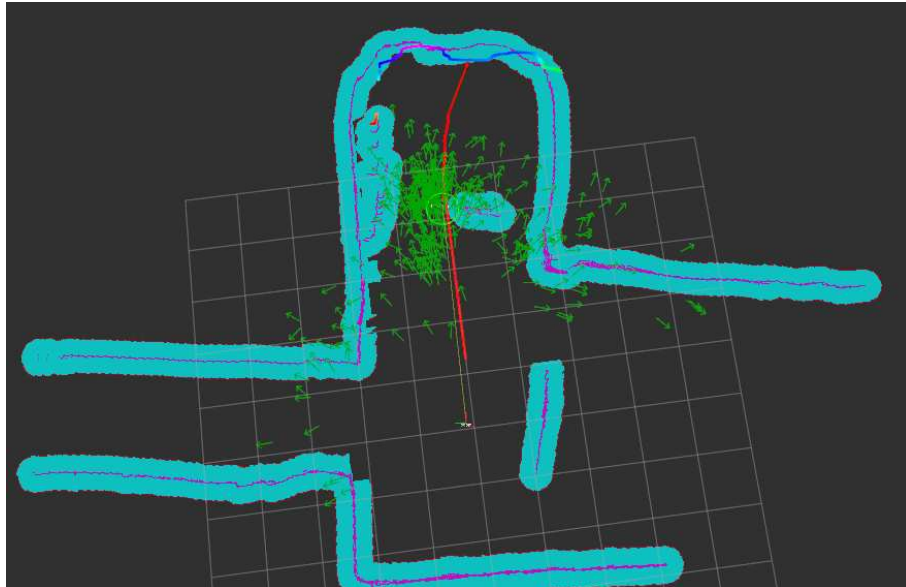
Nếu môi trường có vật cản thì nó sẽ cập nhật vật cản vào trong bản đồ chi phí và hoạch định lại đường đi nếu đường đi cũ băng qua vật cản.



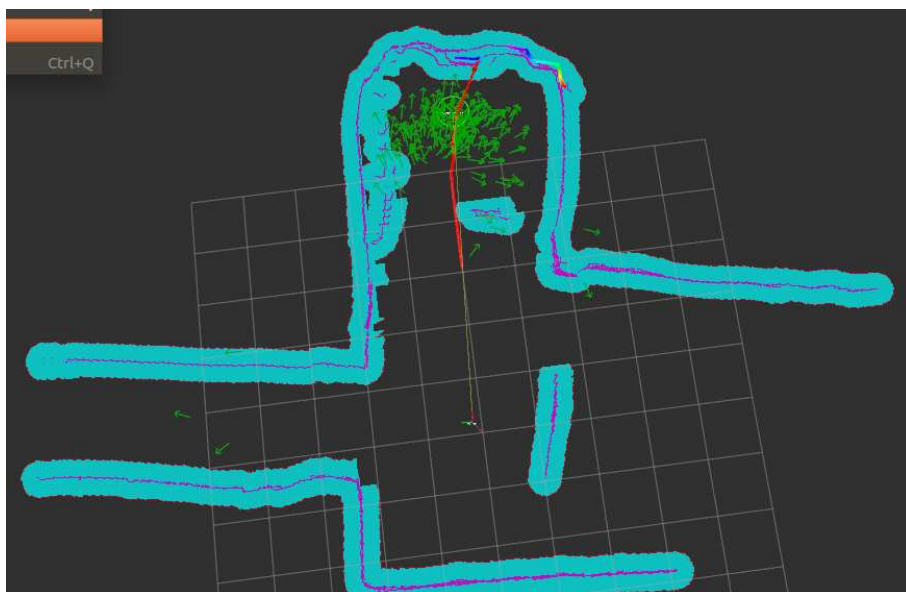
Hình 4.16 Môi trường có vật cản



Hình 4.17 Robot phát hiện vật cản và cập nhật vào bản đồ

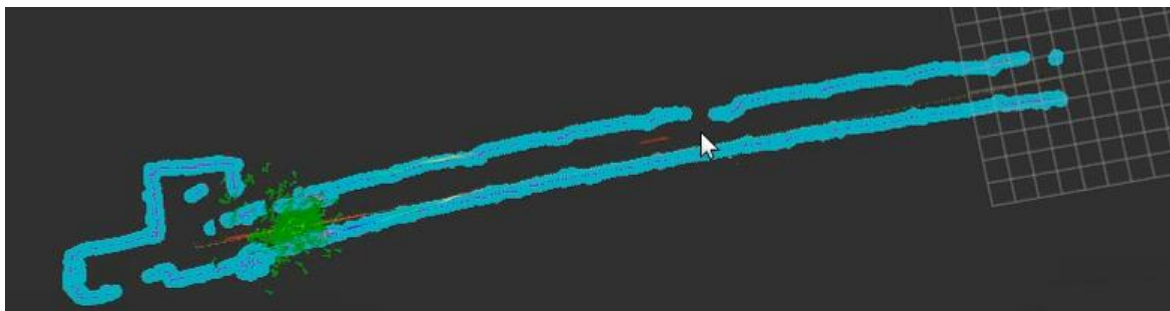


Hình 4.18 Hoạch lại đường đi cho phù hợp với vật cản vừa mới nhận diện được

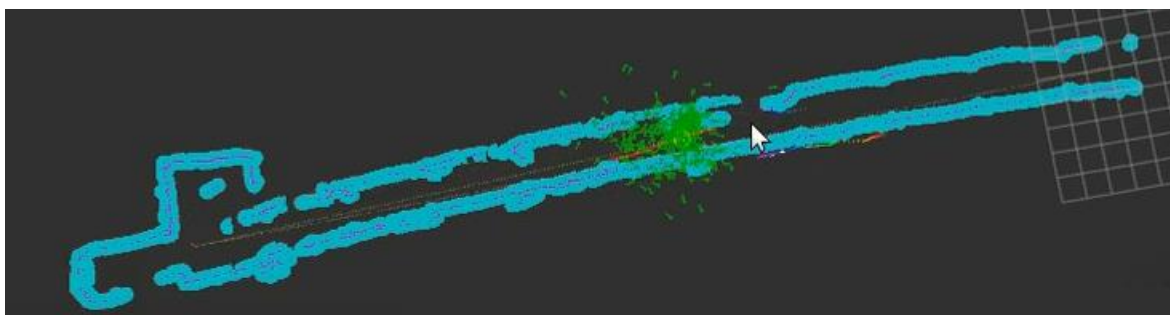


Hình 4.19 Robot đến đích theo đường đi đã hoạch định khi có vật cản

Khi ta thử nghiệm ở môi trường lớn hơn với điều kiện sáng phức tạp hơn từ bản đồ đã dựng (trong môi trường 42 x 2.5 mét) ở trên thì ta có kết quả điều hướng khá tốt.



Hình 4.20 Robot di chuyển theo đường đi đã hoạch định



Hình 4.21 Khi robot đến đích đã được thiết lập trước đó

```
nvidia@tegra-ubuntu: ~
  nsecs: 886347504
  id: /move_base-1-1528616416.886347504
  status: 3
  text: Goal reached.
---
header:
  seq: 862
  stamp:
    secs: 1528616539
    nsecs: 69498992
  frame_id: ''
status_list:
-
  goal_id:
    stamp:
      secs: 1528616416
      nsecs: 886347504
    id: /move_base-1-1528616416.886347504
    status: 3
    text: Goal reached.
```

Hình 4.22 Terminal thông báo là robot đã đến đích

Chương 5. KẾT LUẬN, ĐÁNH GIÁ, HƯỚNG PHÁT TRIỂN

5.1. Đánh giá

Việc dựng bản đồ (mapping) và triển khai điều hướng (navigation) trên bản đồ đã dựng khá thành công. Với thông số sai số cho phép là robot sẽ đến đích trong phạm vi đường tròn bán kính 20cm và góc lệch là 0.5 radian. Robot có thể đi đến đích mà không va vào vật cản chưa được xây dựng trước trong bản đồ.

Việc dựng bản đồ từ stereo camera cho ta kết quả có thể chấp nhận được ở mức độ khá tốt. Vì đặc tính của phương pháp là từ hai ảnh của camera cho ra ảnh độ sâu thì môi trường phải có nhiều đặc trưng (ví dụ như đặc trưng góc cạnh) để có thể ra chính xác hơn; bên cạnh đó, vì đây là xử lý ảnh nên chất lượng ảnh cũng như độ sáng của môi trường cũng ảnh hưởng đến chất lượng của ảnh độ sâu.

Khi chuyển, robot có thể phát hiện vật cản cách robot 3m và vật cản phải cao hơn 30cm.

Trong một bản đồ dựng trước, nếu ta chọn một điểm đến cho robot thì chương trình sẽ tìm ra đường đi ngắn nhất và tối ưu nhất. Nếu trong quá trình di chuyển, nó phát hiện có vật cản ảnh hưởng đến đường đi của nó thì nó sẽ tự động hoạch định lại đường đi mới để có thể đến đích an toàn.

5.2. Kết luận

Qua bước đầu tìm hiểu và nghiên cứu, mặc dù gặp nhiều khó khăn trong việc nghiên cứu, làm quen và tìm hiểu đề tài, nhưng ta thấy được tính khả thi là có và có thể thực hiện được với thời gian và điều kiện cho phép.

Những công việc mà báo cáo đã thực hiện được:

- Tìm hiểu và ứng dụng được thuật toán điều khiển điều hướng cho robot trên nền tảng hệ điều ROS.
- Ứng dụng được stereo camera vào trong quá trình xây dựng bản đồ, điều hướng cho robot và tránh được vật cản trong quá trình di chuyển.

- Tích hợp được máy tính nhúng lên xe nhằm giảm kích thước cho robot và tăng tính linh động cho nó.
- Ứng dụng được các kiến thức đã học vào trong quá trình xây dựng và thiết lập Mobile base cho đề tài.

Bên cạnh đó, có những công việc mà báo cáo chưa hoàn thành tốt:

- Giá trị ước lượng vị trí của robot chưa tốt và chưa thật sự ổn định vì phải lấy dữ liệu từ một nguồn chưa ổn định.
- Chưa tích hợp được quá trình tự localization vào trong hệ thống.

5.3. Hướng phát triển

Với những nghiên cứu hiện tại là nền tảng cũng như tiền đề để xây dựng và phát triển thành công đề tài ở mức độ cao hơn. Hoàn thiện hệ thống một cách tổng quát và hoạt động ổn định hơn trong nhiều môi trường.

Để tăng tính ổn định cho vị trí ước lượng robot, ta có thể tích hợp thêm nhiều nguồn cảm biến để chất lượng ngõ ra chính xác hơn như cảm biến IMU, hay đơn giản hơn là thêm encoder vào trong hệ thống để có thể tính toán được vị trí chính xác hơn.

TÀI LIỆU THAM KHẢO

- [1] ROS, "Robot Operating System," [Online]. Available:
<http://www.ros.org/is-ros-for-me/>.
- [2] R. D. Wiki, "Robot Operating System Wiki," Distributions, [Online].
Available: <http://wiki.ros.org/Distributions>.
- [3] R. N. Wiki, "Robot Operating System Wiki," Navigation, [Online].
Available: <http://wiki.ros.org/navigation/Tutorials/RobotSetup>.
- [4] W. C. ROS, "Robot Operating System Wiki," Costmap 2D, [Online].
Available: http://wiki.ros.org/costmap_2d.
- [5] H. W. G. D. R. D. S. P. Manuel Vázquez-Arellano, "3-D Imaging Systems for Agricultural Applications," p. 3, April 2016.
- [6] A. I. M. F. A. K. Manaf A. Mahammed, "Object Distance Measurement by Stereo VISION," p. 1, March 2013.
- [7] D. A. Wikipedia, "Wikipedia," [Online]. Available:
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.
- [8] A. Tomović, "Path Planning Algorithms For The Robot Operating System," p. 5.
- [9] W. G. p. ROS, "Robot Operating System Wiki," Global planner, [Online].
Available: http://wiki.ros.org/global_planner.
- [10] D. F. D. F. Global planner, "The Dynamic Window Approach to Collision Avoidance," pp. 10-12.