ROS2 SLAM implementation on Mecanum robot.

Installation process

ROS distro: foxy

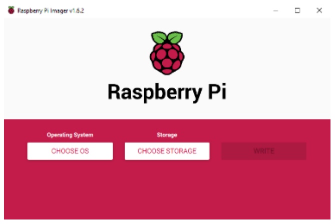
Although there is now ROS on windows, I have used a Windows machine running a Linux VM for this project, and it turned out quite well.

RPI 3B

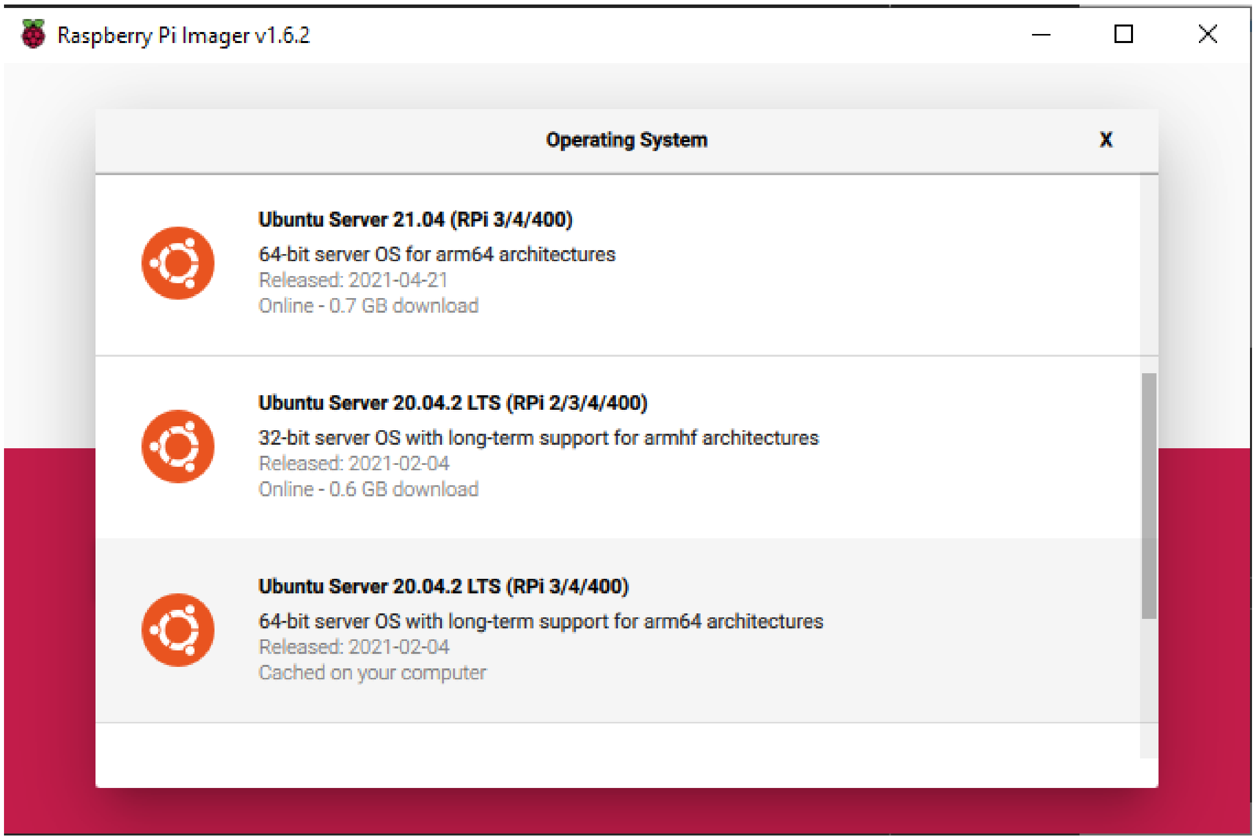
-Prepare a micro-SD >= 8Gb

-Installing OS, Ubuntu Server 20.04.2 LTS

Raspberry Pi Imager <v.1.6.2> at the time of writing



CHOOSE STORAGE -> {your micro-SD drive}
CHOOSE OS -> Other General-purpose OS -> ubuntu -> Ubuntu Server 20.04.2 LTS – 64bit



Raspberry Pi Imager v1.6.2

Operating System

Ubuntu Server 21.04 (RPi 3/4/400)
64-bit server OS for arm64 architectures
Released: 2021-04-21
Online - 0.7 GB download

Ubuntu Server 20.04.2 LTS (RPi 2/3/4/400)
32-bit server OS with long-term support for armhf architectures
Released: 2021-02-04
Online - 0.6 GB download

Ubuntu Server 20.04.2 LTS (RPi 3/4/400)
64-bit server OS with long-term support for arm64 architectures
Released: 2021-02-04
Cached on your computer

WRITE! *This will erase content in the SD*

-Configure RPi network

The default network configuration is stored in the file “50-cloud-init.yaml”

If you use a linux OS on PC (or VM), go to the SD card directory and edit this file.

```
cd /media/$USER/writable/etc/netplan
sudo nano 50-cloud-init.yaml
```

For those using Windows, plug in a keyboard and a monitor to the RPi, login using default username: “pi” passwords: “raspberry”.
Then,

```
sudo nano /etc/netplan/50-cloud-init.yaml
```

In the **50-cloud-init.yaml…**

The wlan0/eth0 are the names of the network interfaces, usually, these are the default values for the RPi model 3. If yours differ, `ifconfig` in the RPi should shows the available network interfaces.

```
network:
    ethernets:
        eth0:
            dhcp4: true
            optional: true
    version: 2
    renderer: NetworkManager
    wifis:
        wlan0:
            optional: true
            dhcp4: yes
            dhcp6: yes
            access-points:
               "__SSID__":
                   password: "wifipassword"
```

```
network:
    ethernets:
        eth0:
            dhcp4: true
            optional: true
    version: 2
    renderer: NetworkManager
    wifis:
        wlan0:
            optional: false
            dhcp4: no
            dhcp6: no
            access-points:
               "__SSID__":
                   band: 2.4GHz
                   password: "wifipassword"
            addresses: [static_ip/subnet_prefix]
            gateway4: 192.168.1.1
            nameservers:
               addresses: [1.1.1.1, 8.8.8.8, 4.4.4.4]
```

Since my home router shares both bands with the same SSID, I was having connection trouble when remoting to RPi using just dhcp. (I spent days trying to solve this issue, don't be like me)

Where,

__SSID__ is your WiFi name
dhcp4: and dhcp6: is the ip address auto configuration.
band: the preferred WiFi band: 2.4GHz or 5GHz
channel: the WiFi channel.
addresses: the device ip address followed by / subnet prefix typically /24 (255.255.255.0)
gateway4: your network gateway, this should point to your router's ip address
nameservers: addresses: this is the DNS server addresses.

For more details, visit https://netplan.io/examples/

Check if the settings are correct with

```
sudo nano netplan -d generate
```

And, to apply the settings

```
sudo nano netplan apply
```

Test the connection with ping, to the RPi and to a website (like www.google.com).

~OK~ Now we should be able to install ROS2 on the RPi
The ROS website provides a good installation tutorial to follow. ->
https://docs.ros.org/en/foxy/Installation/Ubuntu-Install-Debians.html

Basically, you need to add ROS repo to the sources list, and install the base version of ROS.
(for newer hardware like RPi-4 using ubuntu desktop, the ROS desktop could work as well)
Since Ubuntu Server does not have GUI.

Now, the ROS server is installed on both your PC and RPi

-Test ROS connection.

On RPi

Install ros2 demo_nodes_cpp dmo_nodes_py, these are ROS demo packages.

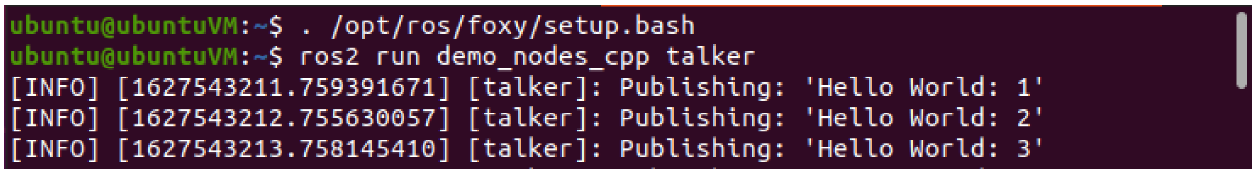
```
sudo apt install ros-<rosdistro>-demo-nodes-cpp
sudo apt install ros-foxy-demo-nodes-py
```

The demo node package is already installed for ROS desktop.
Now you can follow the ROS tutorial website for running the demo_nodes
Be sure to source ROS and set same ROS_DOMAIN_ID, on both devices.

```
. /opt/ros/<rosdistro>/setup.bash
export ROS_DOMAIN_ID=30
```

Now, run

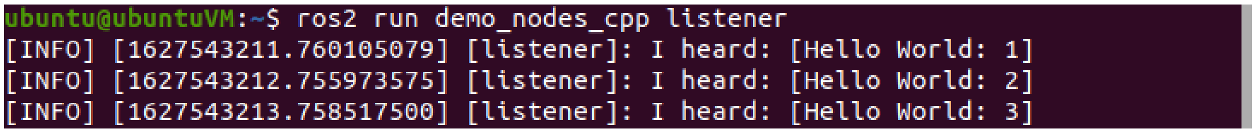
```
ros2 run demo_nodes_cpp talker
```



```
ubuntu@ubuntuVM:~$ . /opt/ros/foxy/setup.bash
ubuntu@ubuntuVM:~$ ros2 run demo_nodes_cpp talker
[INFO] [1627543211.759391671] [talker]: Publishing: 'Hello World: 1'
[INFO] [1627543212.755630057] [talker]: Publishing: 'Hello World: 2'
[INFO] [1627543213.758145410] [talker]: Publishing: 'Hello World: 3'
```

```
On another terminal.
```

```
ros2 run demo_nodes_cpp listener
```



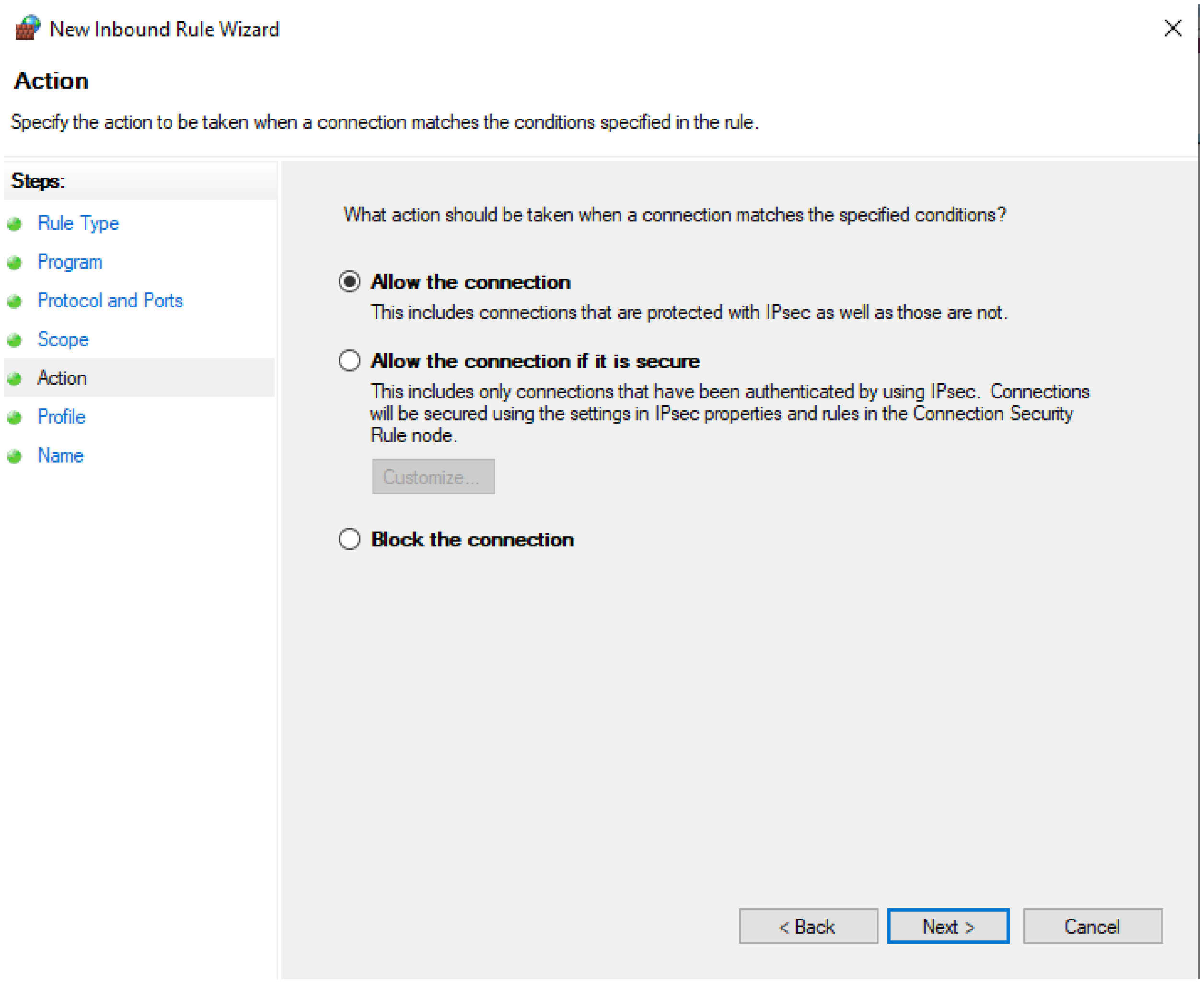
```
ubuntu@ubuntuVM:~$ ros2 run demo_nodes_cpp listener
[INFO] [1627543211.760105079] [listener]: I heard: [Hello World: 1]
[INFO] [1627543212.755973575] [listener]: I heard: [Hello World: 2]
[INFO] [1627543213.758517500] [listener]: I heard: [Hello World: 3]
```

If the device shows the following, ROS nodes can communicate to each other.
If not, there are three potential problems: ROS, network, or Firewall (for windows)

It may be necessary to add firewall rules for Windows devices to be able to receive the message.
Especially, if the PC can ping to the RPi, but not the other way.

To add firewall rule on windows for a specific IP

⊞ Win -> type ‘control panel‘-> Enter -> type ‘firewall’ -> Windows Defender Firewall -> Advances settings -> New Rule… -> Rule type -> Custom, Scope -> These IP addresses -> <Input RPi IP addresses in Remote IP addresses> -> Action -> Allow the connection. Select the appropriate Profile, Name the rule and enable it.



New Inbound Rule Wizard

Action

Specify the action to be taken when a connection matches the conditions specified in the rule.

Steps:
- Rule Type
- Program
- Protocol and Ports
- Scope
- Action
- Profile
- Name

What action should be taken when a connection matches the specified conditions?

[✓ **Allow the connection**]
This includes connections that are protected with IPsec as well as those are not.

Allow the connection if it is secure
This includes only connections that have been authenticated by using IPsec. Connections will be secured using the settings in IPsec properties and rules in the Connection Security Rule node.

Block the connection

For PC installation, follow
https://docs.ros.org/en/rolling/Installation/Ubuntu-Install-Debians.html

And install, navigation2, slam-tool-box, rosdep and colcon

```
sudo apt install ros-<distro>-navigation2 ros-<distro>-nav2-bringup
sudo apt install ros-<ros2-distro>-slam-toolbox
sudo apt install python3-rosdep2
sudo apt install python3-colcon-common-extensions
```

Running the package

Extract the install.tar.gz file to PC and install_aarch64.tar.gz to RPI.
Source ros2 on both machine

On PC
tar -xf install.tar.gz -C install
. install/local_setup.bash
ros2 launch bot_bringup pc.launch.py

On RPI
tar -xf install_aarch64.tar.gz install_aarch64
. install_aarch64/setup.bash
ros2 launch bot_bringup launch.py

Rebuilding the package

There are 2 main packages

Cross_compile_ws.tar.gz for PC and install_aarch64.tar.gz for RPI

Extract the archive into the desired directories and source them.

```
tar -xf install.tar.gz -C <destination_dir_name>
```

In PC, go into the extracted directory

```
cd <destination_dir_name>
rm -r build install log
colcon build
```

Wait for the program to finish building, warnings of AMENT_PREFIX_PATH and delta_2a_ros are normal.

Source the package (with in the cross_compile_ws)

```
. install/setup.bash
```

Launch the package on RPi and PC

RPi

```
export BOT_MODEL=prototype
ros2 launch bot_bringup launch.py
```

The valid models are A1 and PROTOTYPE, the two, so far, differs in cost calculation in navigation.

PC

```
ros2 launch bot_bringup pc.launch.py
```

On the PC screen you will see the program RVIZ launch with the robot in it.
The nav2 might not start sometimes, so just relaunch the script again.

If an error comes up and RVIZ2 failed to load,

To navigate the robot around use
Navigation2 Goal tool and place a waypoint on the map,

Or by publishing a ROS goal message

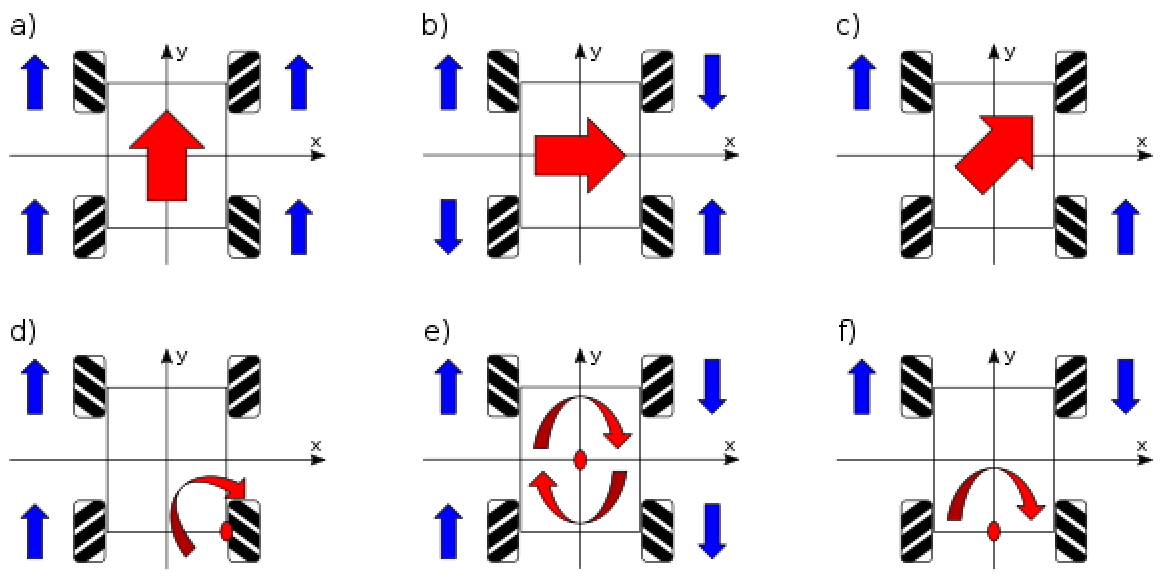
```
ros2 topic pub /goal_pose geometry_msgs/PoseStamped "{header: {stamp: {sec: 0}, frame_id: 'map'},
pose: {position: {x: 0.2, y: 0.0, z: 0.0}, orientation: {w: 1.0}}}"
```

and the robot should make an attempt to reach the published goal!

Also, the above script can be saved to a script and execute with crontab (crontab -e) to schedule the robot. Now, the robot will routinely navigate to the given position.

Hardware:

This robot platform has omnidirectional Mecanum wheels. So, it can traverse sideways unlike the radial wheels. This kind of wheel produces opposite forces when turning, the direction of the robot depends on the resultant force of the 4 wheels.



Here is the picture of how this Mecanum wheel configuration works.

Source: https://en.wikipedia.org/wiki/File:Mecanum_wheel_control_principle.svg

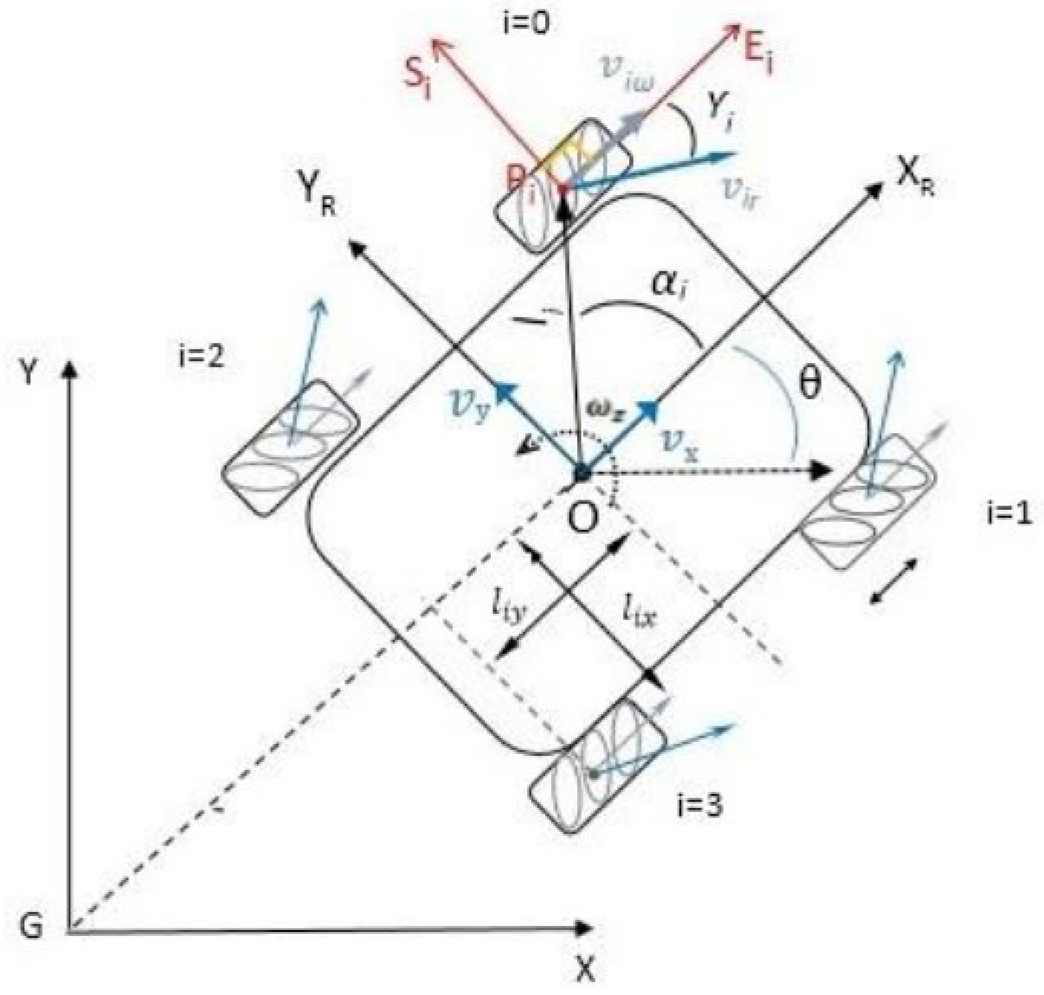
The goal here is to publish the odometry information to ROS2, so that other packages/nodes can use the odometry information for navigation, mapping, or magic. ROS2 has provided the class Odometry that holds this information specifically.

A peek into the docs.ros.org/nav_msgs/Odometry Message shows

```
# This represents an estimate of a position and velocity in free space.
# The pose in this message should be specified in the coordinate frame giv
en by header.frame_id.
# The twist in this message should be specified in the coordinate frame gi
ven by the child_frame_id
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

So, following the nav2 link to ROS1 odometry publisher file
http://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom/

Obtaining the twist and pose, which are the robot's speed and position



And there is this paper that carries out the calculation for this wheel configuration.
Source: https://research.ijcaonline.org/volume113/number3/pxc3901586.pdf

- This is the kinematics for the typical setup of the Mecanum found more common on the Internet. Make sure the wheel is aligned correctly.

So, here are the forward and inverse kinematics of the robot. (eq. 19, eq, 21)

According to equations (10) and (11) for *Forward* and *Inverse* kinematics there is:

$$\begin{bmatrix}\omega_1\\ \omega_2\\ \omega_3\\ \omega_4\end{bmatrix} = \frac{1}{r}\begin{bmatrix}1 & -1 & -(l_x+l_y)\\ 1 & 1 & (l_x+l_y)\\ 1 & 1 & -(l_x+l_y)\\ 1 & -1 & (l_x+l_y)\end{bmatrix}\begin{bmatrix}v_x\\ v_y\\ \omega_z\end{bmatrix}. \qquad \text{eq. 19}$$

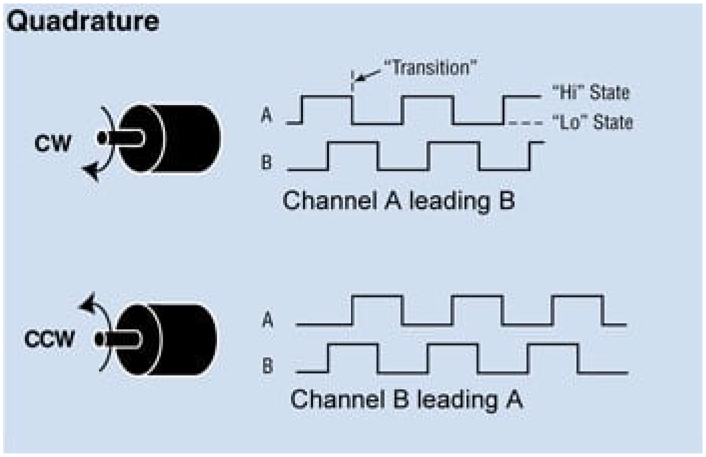
$$\begin{cases}\omega_1 = \frac{1}{r}(v_x - v_y - (l_x+l_y)\omega),\\ \omega_2 = \frac{1}{r}(v_x + v_y + (l_x+l_y)\omega),\\ \omega_3 = \frac{1}{r}(v_x + v_y - (l_x+l_y)\omega),\\ \omega_4 = \frac{1}{r}(v_x - v_y + (l_x+l_y)\omega).\end{cases} \qquad \text{eq. 20}$$

And

$$\begin{bmatrix}v_x\\ v_y\\ \omega_z\end{bmatrix} = \frac{r}{4}\begin{bmatrix}1 & 1 & 1 & 1\\ -1 & 1 & 1 & -1\\ -\frac{1}{(l_x+l_y)} & \frac{1}{(l_x+l_y)} & -\frac{1}{(l_x+l_y)} & \frac{1}{(l_x+l_y)}\end{bmatrix}\begin{bmatrix}\omega_1\\ \omega_2\\ \omega_3\\ \omega_4\end{bmatrix} \qquad \text{eq. 21}$$

The forward kinematics (eq. 19) will come in handy later. For now, we will be using the eq. 21.

Encoder



https://www.dynapar.com/hubfs/uploadedImages/_Site_Root/technology/encoder_basics/quadrature.jpg

The speed is obtained from measuring the number of pulses in a time window. And then convert it to angular speed. (Comparing with the number of pulses in one revolution (2pi rad).)

Ang_Speed(rad/s) = Pulse_speed(pulse/s) * (2π / pulses in 1 rev)

Thanks to GitHub user, nstansby, for providing a solid encoder library that handles signal bouncing very well.

*Encoder channel A, B will be swapped on one side of the robot, otherwise, the encoder will read the speed backwards.

Now that we have the angular speed of each wheel, we can use eq.19 to get the speeds.

The position is calculating multiplying the time step to the speed.

The LIDAR used here is Delta 2A Lidar from 3iRobotix, which is not supported on ROS2 LIDAR, there is only ROS1 code from the vendor's website. So, I had to port it to ROS2. The SDK and the publisher are included in the project file.
Be sure that the program has access to the LIDAR and provide to correct USB port in the launch file.

Check that RPi can 'see' the USB interface. The default port is /dev/ttyUSB0

Below commands will list the connected ttyUSB interfaces and add permission to the port /dev/ttyUSB0.

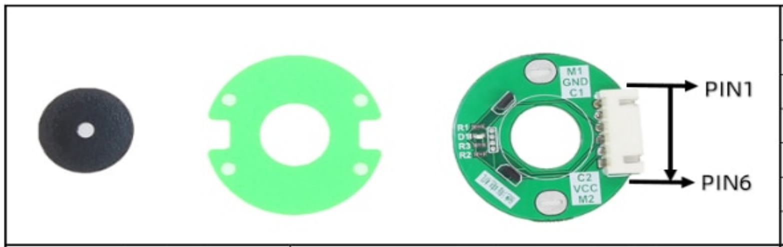
```
ls -l /dev/tty | grep USB
sudo chmod 666 /dev/ttyUSB0
```

The original SDK has not been found on the 3irobotics website at the time of writing, so an alternative from GitHub will be sourced here. Note that the original version of the node.cpp is for ROS1.

https://github.com/CWRU-AutonomousVehiclesLab/Delta-2B-Lidar-SDK

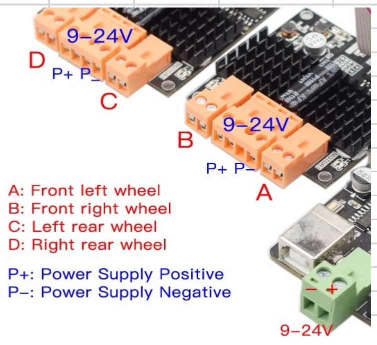
Wiring

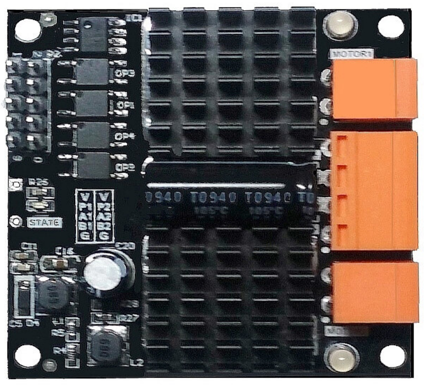
Wiring Diagram														
Motor/ Rpi	M1	MotorDrive	Rpi	M2	MotorDriv	Rpi	M3	MD2	Rpi	M4	MD2	Rpi		5V & Grou
A(ENCODER)	PIN-3		26	PIN-3		6	PIN-3		5	PIN-3		14		
B(ENCODER)	PIN-4		23	PIN-4		24	PIN-4		10	PIN-4		25		
pwm		P1	7		P2	8		P1	9		P2	11		
EN0		A1	20		A2	12		A1	19		A2	15		
EN1		B1	21		B2	16		B1	13		B2	18		

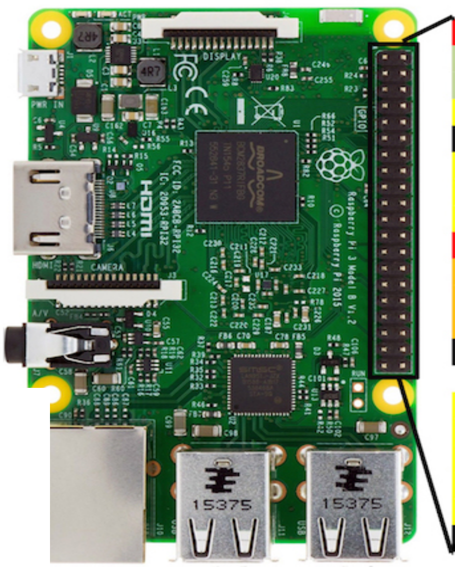


NO.	Interface description
PIN-1	M1-Motor input voltage
PIN-2	GND-Hall GND
PIN-3	C1-Hall sensor B Vout
PIN-4	C2-Hall sensor A Vout
PIN-5	VCC-Hall input voltage 3.3/5.0V
PIN-6	M2-Motor input voltage

N-S 共34极磁石	AB相双霍尔编码板，内置上拉电阻	MOEBIUS
N-S 34-pole magnet	AB phase dual Hall encoding board with built-in pull-up resistance	







	Pin No.		
3.3V	1	2	5V
GPIO2	3	4	5V
GPIO3	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3.3V	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7
DNC	27	28	DNC
GPIO5	29	30	GND
GPIO6	31	32	GPIO12
GPIO13	33	34	GND
GPIO19	35	36	GPIO16
GPIO26	37	38	GPIO20
GND	39	40	GPIO21

ROS nodes: turtlebot3_teleop, nav2, slam_toolbox, odom, state, Lidar

Now that the hardware is prepared, it is time to get the data and publish to ROS.

As stated in the documentation, the navigation stack uses robot transform(tf/tf2) and velocity. But the information is separated in to two topics. So, we need to publish two topics: transform and odometry.

There are many ways to write a publisher function, this is not the best one. I would recommend to follow the ROS2 tutorial and the navigation stack tutorial for more information.

Below is the modified python script ported from navigation tutorials
http://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom/

Be warned though, this is a code written by a monkey.
Putting a while loop in the init function will block other process, producing some weird behaviors.

The code below is not bug free though, but it will work for the time being.

```python
#!/usr/bin/env python3
import time
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile
from rclpy.duration import Duration

from math import sin, cos
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Quaternion, Vector3, TransformStamped
from tf2_ros import TransformBroadcaster#, TransformStamped
from sensor_msgs.msg import JointState, Imu
from shared_param.common_param import WHEEL_PORTS
from encoder2 import Encoder2

class OdometryPublisher(Node):
    def __init__(self):
        super().__init__('bot_state_publisher')

        qos_profile = QoSProfile(depth=10)
        self.odom_pub = self.create_publisher(Odometry, 'odom', qos_profile)
        self.broadcaster = TransformBroadcaster(self, qos=qos_profile)

        # message declarations
        odom_trans = TransformStamped()
        odom_trans.header.frame_id = 'odom'
        odom_trans.child_frame_id = 'base_link' #'base_link'

        odom = Odometry()
        odom.header.frame_id = 'odom'
        odom.child_frame_id = 'base_link' #'base_link'
        loop_rate = self.create_rate(30) #Hz

        x = 0.0
        y = 0.0
        th = 0.0
        vx = 0.0
        vy = 0.0
        vth = 0.0

        current_time = 0
        last_time = Duration(seconds=0, nanoseconds=0)
try:
            while rclpy.ok():
                rclpy.spin_once(self)
                #update time
                current_time = self.get_clock().now()

                # compute odometry in a typical way given velocities
                dt = float((current_time - last_time).nanoseconds) / (10 ** 9) #
to sec

                #get velocities
                vx, vy, vth = en_all.cal_r_velocity(dt * 1000) # convert back to
miliseconds

                ## debug
                ##print("x: {} y: {} z{}".format(vx, vy, vth), end='\r')
                delta_x = (vx * cos(th) - vy * sin(th)) * dt
                delta_y = (vx * sin(th) + vy * cos(th)) * dt
                delta_th = vth * dt

                x += delta_x
                y += delta_y
                th += delta_th

                #update odom
                odom.header.stamp = current_time.to_msg()
                odom.pose.pose.position.x = x
                odom.pose.pose.position.y = y
                odom.pose.pose.position.z = 0.0

                odom.pose.pose.orientation = \
                    euler_to_quaternion(0, 0, th)

                odom.twist.twist.linear.x = vx
                odom.twist.twist.linear.y = vy
                odom.twist.twist.angular.z = vth

                self.odom_pub.publish(odom)

                #update transform
                odom_trans.header.stamp = current_time.to_msg()
                odom_trans.transform.translation.x = x
                odom_trans.transform.translation.y = y
                odom_trans.transform.translation.z = 0.0

                odom_trans.transform.rotation = \
                    euler_to_quaternion(0, 0, th)


                self.broadcaster.sendTransform(odom_trans)
                joint_state.header.stamp = current_time.to_msg()
                joint_state.position = \
                    en_all.get_encoder_rad()

                self.joint_state_pub.publish(joint_state)


                #updating lasttime
                last_time = current_time
                # This will adjust as needed per iteration
                loop_rate.sleep()

        except KeyboardInterrupt:
            self.destroy_node()
            rclpy.shutdown()

def euler_to_quaternion(roll, pitch, yaw):
    qx = sin(roll/2) * cos(pitch/2) * cos(yaw/2) - cos(roll/2) * sin(pitch/2) * s
in(yaw/2)
    qy = cos(roll/2) * sin(pitch/2) * cos(yaw/2) + sin(roll/2) * cos(pitch/2) * s
in(yaw/2)
    qz = cos(roll/2) * cos(pitch/2) * sin(yaw/2) - sin(roll/2) * sin(pitch/2) * c
os(yaw/2)
    qw = cos(roll/2) * cos(pitch/2) * cos(yaw/2) + sin(roll/2) * sin(pitch/2) * s
in(yaw/2)
    return Quaternion(x=qx, y=qy, z=qz, w=qw)

def main(args = None):

    rclpy.init(args= args)
    OdometryPublisher()

if __name__ == '__main__':
    main()
```

Controlling the robot through ROS.

This is a ROS subscriber node to the topic ‘cmd_vel’, allowing other packages to control the robot.

The cmd_vel message specifies the speed in standard ROS convention (Vx, Vy, W), this where eq.19 translates those velocities into wheel speeds. The rest, is to pass that to the motor controller.

```python
    def cmd_vel_callback(self, msg):
        # Store velocity from teleop
        vx = msg.linear.x
        vy = msg.linear.y
        wz = msg.angular.z

        w1, w2, w3, w4 = mecanum_IK(vx, vy, wz)

        motors[0].simple_linear_speed_control(w1)
        motors[1].simple_linear_speed_control(w2)
        motors[2].simple_linear_speed_control(w3)
        motors[3].simple_linear_speed_control(w4)

def mecanum_IK(vx, vy, wz, \
    wheelbase = WHEELBASE_, track = TRACK_, wheel_radius = WHEELRADIUS_):
    # calculate wheel speed (rps) from given speeds (m/s, rps)
    lxly = (track + wheelbase) / 2
    fwradius = 1/wheel_radius
    w1 = fwradius * (vx - vy - lxly * wz)
    w2 = fwradius * (vx + vy + lxly * wz)
    w3 = fwradius * (vx + vy - lxly * wz)
    w4 = fwradius * (vx - vy + lxly * wz)

    return w1, w2, w3, w4
```

And below script just commands a H-Bridge connected to the motor.

The plot duty cycle vs angular speed shows a somewhat linear relationship. (I tried PID control as well, but it did not work as well) so, the conversion is simply:

duty_cycle = m * w(rad/s), where m is (max duty cycle / max angular speed)

I also added a low threshold, because the motors will not rotate at a low duty cycle. I suspect my puffy batteries do not have an adequate current.

```python
    def drive(self, dc):
        self.pwm.ChangeDutyCycle(dc)
        self.duty_cycles = dc

    def simple_linear_speed_control(self, velocity):
        pwm_dc = linear_profiler(abs(velocity))

        if (velocity > 0):
            # positive -> forward mode 01 on H bridge
            self.change_mode('01')
        elif (velocity < 0):
            self.change_mode('10')
        else:
            self.change_mode('00')
        #print("duty cycle {}".format(pwm_dc))
        self.drive(pwm_dc)

def bound(x, out_min = 0, out_max = 100):
    #r =  (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min
    r = x
            # check constrain
    if r > out_max:
        r = out_max
    elif r < out_min:
        r = out_min
    return r

def linear_profiler(x, m = PWM_LINEAR_PROFILE, min = 12 , max = 90):
    # rps to pwm
    pwm = x * m
    # no speed change in rps
    if pwm < min:
        return pwm
    elif pwm > max:
        pwm = 100
    # with
    return bound(pwm)
```

Modify turtlebot3 keyboard teleoperation script

Added Y-axis speed to the message.

```
            elif key == 'q':
                target_linear_y_velocity =\
                    check_linear_y_limit_velocity(target_linear_y_velocity + LIN_
VEL_STEP_SIZE)
                #if target_linear_y_velocity < BOT_LOW_BOUND and target_linear_y_
velocity > 0:
                #    target_linear_y_velocity = BOT_LOW_BOUND
                target_linear_y_velocity = low_bound(target_linear_y_velocity, BO
T_LOW_BOUND)
                status = status + 1
                print_vels(target_linear_velocity, target_angular_velocity, targe
t_linear_y_velocity)
            elif key == 'e':
                target_linear_y_velocity =\
                    check_linear_y_limit_velocity(target_linear_y_velocity - LIN_
VEL_STEP_SIZE)
                #if target_linear_y_velocity > - BOT_LOW_BOUND and target_linear_
y_velocity < 0:
                #    target_linear_y_velocity = BOT_LOW_BOUND
                target_linear_y_velocity = low_bound(target_linear_y_velocity, -
BOT_LOW_BOUND)
                status = status + 1
                print_vels(target_linear_y_velocity, target_angular_velocity, tar
get_linear_y_velocity)
```

Slam toolbox

This is the package responsible for generating a map from the lidar sensor. To use this package, firstly install it with

```
apt install ros-eloquent-slam-toolbox
```

To test run it.

```
ros2 launch slam_toolbox online_sync_launch.py
```

For customization, we will modify the config file from the original package to suit the robot settings.

The important parameters as stated by http://www.robotandchisel.com/2020/08/19/slam-in-ros2/ are

```
    # ROS Parameters
    odom_frame: odom
    map_frame: map
    base_frame: base_footprint #base_link
    scan_topic: /scan
```

Here the toolbox provides transform of map -> odom frame, and the uses robot's projection on the map frame, base_footprint. The scan topic should point to the LIDAR scan topic.

The slamtoolbox is called by a launch file.

```
    declare_slam_params_file_cmd = DeclareLaunchArgument(
        'slam_params_file',
        default_value=os.path.join(get_package_share_directory("bot_navigation2")
,
                                   'config', 'mapper_params_online_async.yaml'),
        description='Full path to the ROS2 parameters file to use for the slam_to
olbox node')
```

This points to the config file we have just edited.

And,

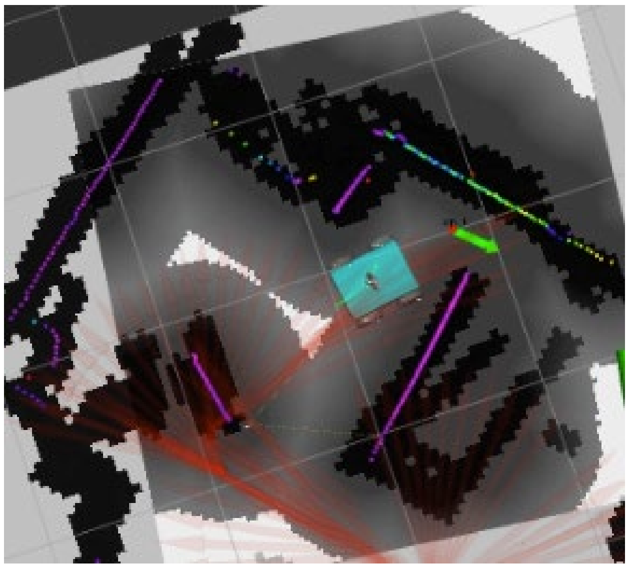
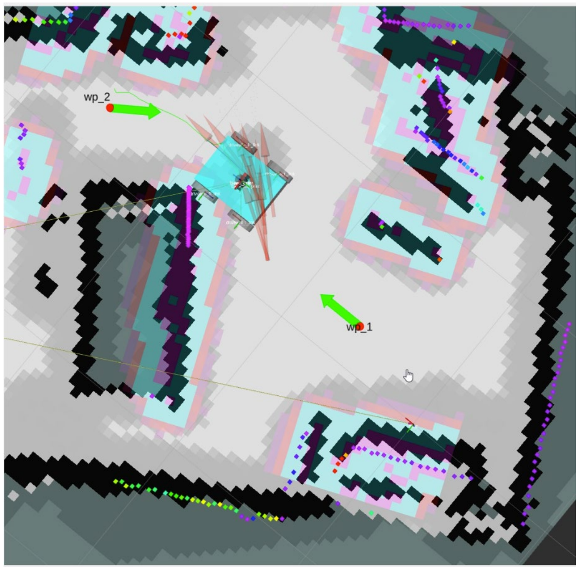
```
    start_async_slam_toolbox_node = Node(
        parameters=[
          slam_params_file,
          {'use_sim_time': use_sim_time}
        ],
        package='slam_toolbox',
        executable='async_slam_toolbox_node',
        name='slam_toolbox',
        output='screen')
```

Declares the node, finally they are added to the launch description of ROS launch file.

```
    ld = LaunchDescription()

    ld.add_action(declare_use_sim_time_argument)
    ld.add_action(declare_slam_params_file_cmd)
    ld.add_action(start_async_slam_toolbox_node)
    return ld
```

Nav2 cost map

Like slam_toolbox, the nav2 can be configured. The left and right pictures use different robot radius, inflation radius and cost scaling factor.
The navigation Costmap parameters are explained in
https://emanual.robotis.com/docs/en/platform/turtlebot3/navigation/ and
https://navigation.ros.org/configuration/packages/configuring-costmaps.html

```
 robot_model_type: "omnidirectional" #"differential"
```

Since, we are using Mecanum wheels, I have changed it to omnidirectional, but there is not much changes during the testing.

I also reduced the rotational speed of the robot, to allow the slam_toolbox to match up the map nicely. In the controller server DWB parameters and recovery server

```
controller_server:
  ros__parameters:
    # DWB parameters
    FollowPath:
      plugin: "dwb_core::DWBLocalPlanner"
      debug_trajectory_details: True
      min_vel_x: -0.22
      min_vel_y: -0.22 #0.0
      max_vel_x: 0.22
      max_vel_y: 0.22 #0.0
      max_vel_theta: 0.39
      min_speed_xy: -0.22 #0.0
      max_speed_xy: 0.22
      min_speed_theta: -0.39
```

```
recoveries_server:
  ros__parameters:
    max_rotational_vel: 0.44
    min_rotational_vel: 0.41
    rotational_acc_lim: 3.2
```