ROS 筆記

luu

Contents

Part I
ros2 入門 CLI 操作

接下來的資料來源於 ros 官方文件，可以上去看看。ros2

1 CLI 設定

在運行 ROS 2 的指令時需要開啟環境，所以要先使用 source 來啟用。

```
source /opt/ros/humble/setup.zsh
```

但是每次都要執行會有點多餘，所以可以寫在 shell 的設定檔內。

```
echo "source /opt/ros/humble/setup.zsh" >> ~/.zshrc
```

2 練習用的模擬器

Turtlesim 是一個用於學習 ROS 2 的輕量級模擬器。它說明了 ROS 2 在最基本層面上的功能，讓您了解稍後將使用真實機器人或機器人模擬做什麼。

ROS 2 工具是使用者管理、反思以及與 ROS 系統互動的方式。它支援針對系統及其操作的不同方面的多個命令。人們可以使用它來啟動節點、設定參數、監聽主題等等。ROS 2 工具是核心 ROS 2 安裝的一部分。

安裝

安裝東西的時候記得要更新來源。

```
sudo apt update
sudo apt install ros-humble-turtlesim
sudo apt install ~nros-humble-rqt"*"
```

檢查軟體包是否已安裝：

```
ros2 pkg executables turtlesim
#--------------------
turtlesim draw_square
turtlesim mimic
turtlesim turtle_teleop_key
turtlesim turtlesim_node
```

運行

烏龜模擬器

```
ros2 run turtlesim turtlesim_node
ros2 pkg executables turtlesim
```

rqt 是 ROS 2 的圖形使用者介面 (GUI) 工具。在 rqt 中完成的所有操作都可以在命令列上完成，但 rqt 提供了一種更用戶友好的方式來操作 ROS 2 元素。

```
rqt
```

3 節點

ROS 中的每個節點都應負責單一的模組化目的，例如控制車輪馬達或發布來自雷射測距儀的感測器資料。每個節點都可以透過主題、服務、操作或參數從其他節點發送和接收資料。

完整的機器人系統由許多協同工作的節點組成。在 ROS 2 中，單一可執行檔（C++ 程式、Python 程式等）可以包含一個或多個節點。

3.1 ros2 直行方法

run 的使用方式如下

```
ros2 run <package_name> <executable_name>
-----------------------------------------
usage: ros2 run [-h] [--prefix PREFIX]
package_name executable_name ...

Run a package specific executable

positional arguments:
package_name     Name of the ROS package
executable_name  Name of the executable
argv             Pass arbitrary arguments to the executable

options:
-h, --help       show this help message and exit
--prefix PREFIX  Prefix command, which should go before the
executable. Command must be wrapped in quotes
if it contains spaces (e.g. --prefix 'gdb -ex
run --args').
```

操作流程如下

1. 開啟第一個 terminal
2. 開啟 turtlesim turtlesim_node
3. 開啟第二個 terminal
4. 檢查節點裝況

開啟模擬視窗

```
ros2 run <package_name> <executable_name>
ros2 run turtlesim turtlesim_node
```

檢查節點

```
ros2 node list
```

開啟控制器

```
ros2 run turtlesim turtle_teleop_key
```

3.2 重新映射

重新映射可讓您將預設節點屬性（例如節點名稱、主題名稱、服務名稱等）重新指派給自訂值。

```
ros2 run turtlesim turtlesim_node --ros-args --remap \
__node:=my_turtle
```

這時後會看到我們映射的節點

```
/my_turtle
/turtlesim
/teleop_turtle
```

3.3 節點訊息

可以用 info 來查詢節點的資料，方法如下。

```
ros2 node info <node_name>
ros2 node info /my_turtle
```

可以看到如下的連線資料

1. Subscribers: 訂閱者
2. Publishers: 發布者
3. Service Servers: 服務伺服器
4. Service Clients: 服務客戶端
5. Action Servers: 動作伺服器
6. Action Clients: 動作客戶端

3.3.1 訂閱者（Subscribers）：

訂閱者是 ROS 中的一個元件，它可以接收特定主題（Topic）的訊息。主題是 ROS 中一種訊息傳遞機制，允許節點（ROS 中的程式）透過發佈和訂閱訊息進行通訊。

3.3.2 發佈者（Publishers）：

發佈者是 ROS 中的一個元件，它能夠向特定主題發佈訊息。發佈者節點生成訊息並將其發送到相應的主題，使其他訂閱者節點能夠接收這些訊息。

3.3.3 服務伺服器（Service Servers）：

服務伺服器提供了一種不同的通訊機制，稱為服務（Service），允許節點提供某種特定的功能或服務。當一個節點請求服務時，服務伺服器執行相應的任務，並返回結果。

3.3.4 服務客戶端（Service Clients）：

服務客戶端是一個節點，它能夠向服務伺服器發送請求，並等待伺服器返回結果。這種通訊模式通常用於需要特定服務或功能的節點之間的交互。

3.3.5 動作伺服器（Action Servers）：

動作伺服器是 ROS 中處理長時間執行任務的一種機制。它允許異步執行，並提供反饋。與服務不同，動作可以週期性地提供反饋，而不僅僅是單一的請求和回應。

3.3.6 動作客戶端（Action Clients）：

動作客戶端是一個節點，它可以向動作伺服器發送請求，並接收反饋和結果。這通常用於需要處理較長時間執行的任務，而服務模型則適用於簡短的請求和回應。

4 主題

4.1 topic 用法

主題是在節點之間以及系統不同部分之間移動數據的主要方式之一。
可以用-h 來查詢 topic 可以有哪些操作如下。

1. bw : 顯示主題使用的頻寬
2. delay : 從標題中的時間戳顯示主題的延遲
3. echo : 從主題輸出訊息
4. find : 輸出給定類型的可用主題列表
5. hz : 將平均發布平率列印到螢幕上
6. info : 列印主題的訊息
7. list : 輸出可用主題的列表
8. pub : 向主題發布訊息
9. type : 列印主題的類型

4.2 list

在新終端機中執行該命令將傳回系統中目前活動的所有主題的清單：

```
ros2 topic list
----------------
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

4.3 echo

若要查看某個主題上發布的數據，請使用：

```
ros2 topic echo <topic_name>
ros2 topic echo /turtle1/cmd_vel
---------------------------------
linear:
x: 2.0
y: 0.0
z: 0.0
angular:
x: 0.0
y: 0.0
z: 0.0
---
```

4.4 info

可以查詢主題的訂閱與發布資訊。

```
ros2 topic info /turtle1/cmd_vel
--------------------------------
Type: geometry_msgs/msg/Twist
Publisher count: 1
Subscription count: 2
```

4.5 interface

節點使用訊息透過主題發送資料。發布者和訂閱者必須發送和接收相同類型的訊息才能進行通訊。
我們可以透過 info 去查詢 topic 的 tyep，在由 interface show 來查詢格式。

```
ros2 interface show geometry_msgs/msg/Twist
-----------------------------------------
# This expresses velocity in free space broken
into its linear and angular parts.

Vector3  linear
float64 x
float64 y
float64 z
Vector3  angular
float64 x
float64 y
float64 z
```

4.6 pub

現在您已經有了訊息結構，用以下命令直接從命令列將資料發佈到主題：
'<args>' 是您將傳遞到主題的實際數據，採用您在上一節中剛剛發現的結構。

```
ros2 topic pub <topic_name> <msg_type> '<args>'
ros2 topic pub --once /turtle1/cmd_vel \
    geometry_msgs/msg/Twist \
    "{linear: {x: 2.0, y: 0.0, z: 0.0},\
    angular: {x: 0.0, y: 0.0, z: 1.8}}"
```

以上我們我們了解如何查尋主題的發布的內容，與如何對主題輸入資料。

5 參數

參數是節點的配置值。您可以將參數視為節點設定。節點可以將參數儲存為整數、浮點數、布林值、字串和列表。在 ROS 2 中，每個節點維護自己的參數。有關參數的更多背景信息，請參閱概念文件。

5.1 param 可以使用的指令

1. delete : 刪除參數
2. describe : 顯示有關聲明參數的描述信息
3. dump : 以 YAML 檔案格式顯示節點的所有參數
4. get : 取得參數
5. list : 輸出可用參數列表
6. load : 載入節點的參數文件
7. set : 設定參數

5.2 list

可以用 list 來看不同節的參數

```
ros2 param list
-------------------------
/teleop_turtle:
    qos_overrides./parameter_events.publisher.depth
    qos_overrides./parameter_events.publisher.durability
    qos_overrides./parameter_events.publisher.history
    qos_overrides./parameter_events.publisher.reliability
    scale_angular
    scale_linear
use_sim_time
/turtlesim:
```

```
    background_b
    background_g
    background_r
    qos_overrides./parameter_events.publisher.depth
    qos_overrides./parameter_events.publisher.durability
    qos_overrides./parameter_events.publisher.history
    qos_overrides./parameter_events.publisher.reliability
    use_sim_time
```

5.3 get

若要顯示參數的類型和目前值，請使用下列命令：
<node_name> 與 <parameter_name> 可以用 list 來查詢

```
ros2 param get <node_name> <parameter_name>
ros2 param get /turtlesim background_g
---------------------------------------
Integer value is: 86
```

5.4 set

我們也可以修改參數的數值

```
ros2 param set <node_name> <parameter_name> <value>
ros2 param set /turtlesim background_r 150
---------------------------------------
Set parameter successful
```

5.5 dump

您可以使用以下命令查看節點目前的所有參數值：

```
ros2 param dump <node_name>
ros2 param dump /turtlesim > turtlesim.yaml
-----------------------------------------
/turtlesim:
ros__parameters:
    background_b: 255
    background_g: 86
    background_r: 150
    qos_overrides:
    /parameter_events:
        publisher:
        depth: 1000
        durability: volatile
        history: keep_last
        reliability: reliable
    use_sim_time: false
```

上面的範例有 > turtlesim.yaml，這樣會輸出檔案如果要看內容可以用 cat。

5.6 load

您可以使用以下命令將參數從檔案載入到目前正在運行的節點：

```
ros2 param load <node_name> <parameter_file>
ros2 param load /turtlesim turtlesim.yaml
```

5.7 使用時就加入參數

若要使用已儲存的參數值啟動相同節點，請使用：

```
ros2 run <package_name> <executable_name>\
    --ros-args --params-file <file_name>

ros2 run turtlesim turtlesim_node \
    --ros-args --params-file turtlesim.yaml
```

以上我們學會如何查詢參數與如何修改參數。

6 動作

動作是 ROS 2 中的通訊類型之一，適用於長時間運行的任務。
它們由三部分組成：

1. 目標
2. 回饋
3. 結果

行動建立在 **主題**和**服務**的基礎上。它們的功能與服務類似，只是可以取消操作。它們還提供穩定的回饋，而不是返回單一回應的服務。

6.1 使用動作

操作過程如下

1. 啟動兩個 turtlesim 節點
2. 使用 turtle_teleop_key 來控制烏龜
3. 觀察 teleop_turtle 的資訊

6.1.1 開啟模擬

```
ros2 run turtlesim turtlesim_node
ros2 run turtlesim turtle_teleop_key
```

6.1.2 控制並觀察

根據提示去抄做烏龜的轉向。

```
    Use arrow keys to move the turtle.
Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations.
'F' to cancel a rotation.
```

如果完成就會回傳動作完成

```
[INFO] [turtlesim]: Rotation goal completed successfully
```

當動作被中止也會有終止訊息

```
[INFO] [turtlesim]: Rotation goal canceled
```

過程中修改母標的提示

```
[WARN] [turtlesim]: Rotation goal received before a
    previous goal finished.
    Aborting previous goal
```

6.2 list

我們可以用 list 來查詢 action 有哪些

```
ros2 action list
ros2 action list -t
```

6.3 info

用 info 來查詢動作的訊息

```
ros2 action info /turtle1/rotate_absolute
----------------------------------------
Action: /turtle1/rotate_absolute
Action clients: 1
    /teleop_turtle
Action servers: 1
    /turtlesim
```

6.4 interface

使用 interface show 來查詢相關資料格式。

```
ros2 interface show turtlesim/action/RotateAbsolute
---------------------------------------------------
# The desired heading in radians
float32 theta
---
# The angular displacement in radians to the starting position
float32 delta
---
# The remaining rotation in radians
float32 remaining
```

6.5 send_goal

我們可以用 send_goal 把目標傳道 action。

```
ros2 action send_goal <action_name> <action_type> <values>
ros2 action send_goal /turtle1/rotate_absolute \
    turtlesim/action/RotateAbsolute "{theta: 1.57}"
```

7 rqt console 查看日誌

rqt_console 是一個 GUI 工具，用於在 ROS 2 中內省日誌訊息。通常，日誌訊息會顯示在您的終端機中。使用 rqt_console，您可以隨著時間的推移收集這些訊息，以更有條理的方式仔細查看它們，過濾它們，保存它們，甚至重新加載保存的文件以在不同時間進行反思。

1. 開啟 rqt
2. 開啟節點
3. 控制烏龜
4. 查看或是存檔

7.1 開啟 rqt

我們操作之前可以先開 rqt，一邊做可以一邊檢查。

```
ros2 run rqt_console rqt_console
```

7.2 開啟節點並動作

開啟節點後用另外一個 terminal 來控制。

```
ros2 topic pub -r 1 /turtle1/cmd_vel \
    geometry_msgs/msg/Twist \
    "{linear: {x: 2.0, y: 0.0, z: 0.0}, \
    angular: {x: 0.0,y: 0.0,z: 0.0}}"
```

7.3 訊息級別

ROS 2 的記錄器等級依嚴重性排序：

1. Fatal: 訊息表明系統將終止以嘗試保護自身免受損害。
2. Error: 訊息表明存在重大問題會阻止其正常運作。
3. Warn: 訊息顯示意外的活動或不理想的結果，可能代表更深層的問題。
4. Info: 訊息指示事件和狀態更新。
5. Debug: 息詳細說明了系統執行的整個逐步過程。

7.4 設定預設記錄器級別

您可以在首次使用重新映射運行節點時設定預設記錄器等級/turtlesim。在終端機中輸入以下命令：

```
ros2 run turtlesim turtlesim_node --ros-args --log-level WARN
```

8 啟動節點

使用命令列工具一次啟動多個節點。

在大多數介紹性教學中，您一直在為運行的每個新節點開啟新終端。當您建立越來越複雜的系統並同時運行越來越多的節點時，打開終端機並重新輸入配置詳細資訊變得乏味。

啟動檔案可讓您同時啟動和設定多個包含 ROS 2 節點的可執行檔。

使用該命令運行單一啟動檔案將立即啟動整個系統 - 所有節點及其配置。ros2 launch

8.1 運行啟動文件

我們可以用 multisim.launch.py 來啟動。

```
ros2 launch turtlesim multisim.launch.py
```

8.2 用 list 來查詢

我們可以用 node list 與 topic list 來查訊完整的名稱。

之後我們就可以用 pub 或其他的方式來給訊息。

```
ros2 topic pub  /turtlesim1/turtle1/cmd_vel \
    geometry_msgs/msg/Twist \
    "{linear: {x: 2.0, y: 0.0, z: 0.0}, \
    angular: {x: 0.0, y: 0.0, z: 1.8}}"

ros2 topic pub  /turtlesim2/turtle1/cmd_vel \
    geometry_msgs/msg/Twist \
    "{linear: {x: 2.0, y: 0.0, z: 0.0}, \
    angular: {x: 0.0, y: 0.0, z: -1.8}}"
```

9 記錄和回放數據

記錄關於某個主題發布的數據，以便您可以隨時重播和檢查它。

9.1 啟動節點

記的用不同的 terminal 啟動節點

```
ros2 run turtlesim turtlesim_node
ros2 run turtlesim turtle_teleop_key
```

我們也建立一個新目錄來儲檔

9.2 記入的方法

要注意 topic 的名稱要正確，可以用 topic list 來查詢。

```
ros2 bag record <topic_name>
ros2 bag record /turtle1/cmd_vel
```

檔案存的地方會是運行指令的地方這要注意

9.3 info

一樣我們可以用 info 來查詢錄的檔案。

```
ros2 bag info <bag_file_name>
```

9.4 play

可以用 play 來重現錄製時做的動作。

```
ros2 bag play <bag_file_name>
```

10 指令

10.1 ros 指令

ros 的指令可用下面的功能

1. action - 用於處理與動作相關的操作。
2. bag - 用於處理與 rosbag 相關的操作。
3. component - 用於處理與組件相關的操作。
4. control - 用於處理與控制相關的操作。
5. daemon - 用於處理與守護程序相關的操作。
6. doctor - 用於檢查 ROS 設置和其他潛在問題。
7. interface - 顯示與 ROS 接口相關的信息。
8. launch - 用於運行一個 launch 文件。
9. lifecycle - 用於處理與生命周期相關的操作。
10. multicast - 用於處理與多播相關的操作。
11. node - 用於處理與節點相關的操作。
12. param - 用於處理與參數相關的操作。
13. pkg - 用於處理與包相關的操作。
14. run - 用於運行一個特定包的可執行文件。
15. security - 用於處理與安全相關的操作。
16. service - 用於處理與服務相關的操作。
17. topic - 用於處理與主題相關的操作。
18. wtf - 將 wtf 作為 doctor 的別名使用。

10.2 node

node 有兩個參數可以用

1. info Output information about a node
2. list Output a list of available nodes

10.2.1 info

ros2 node info [選項] 節點名稱

1. -h, –help：顯示幫助信息並退出。
2. –spin-time SPIN_TIME：在等待發現時的旋轉時間（僅在不使用已運行的守護程序時適用）。
3. -s, –use-sim-time：啟用 ROS 模擬時間。
4. –no-daemon：不要生成或使用已運行的守護程序。
5. –include-hidden：顯示隱藏的主題、服務和動作。

10.2.2 list

用法：ros2 node list [選項]

1. -h, –help：顯示幫助信息並退出。
2. –spin-time SPIN_TIME：在等待發現時的旋轉時間（僅在不使用已運行的守護程序時適用）。
3. -s, –use-sim-time：啟用 ROS 模擬時間。
4. –no-daemon：不要生成或使用已運行的守護程序。
5. -a, –all：顯示所有節點，包括隱藏的節點。
6. -c, –count-nodes：僅顯示發現的節點數量。

10.3 topic

1. bw：顯示主題使用的帶寬。
2. delay：顯示從標頭中的時間戳的主題延遲。
3. echo：從主題輸出消息。
4. find：輸出給定類型的可用主題列表。
5. hz：將平均發布速率打印到屏幕上。
6. info：打印有關主題的信息。
7. list：輸出可用主題的列表。
8. pub：向主題發布消息。
9. type：打印主題的類型。

10.3.1 bw

用法：ros2 topic bw [選項] 主題名稱

1. -h, –help：顯示幫助信息並退出。
2. –window WINDOW, -w WINDOW：計算速率的最大窗口大小，以消息數為單位（默認值：100）。
3. –spin-time SPIN_TIME：在等待發現時的旋轉時間（僅在不使用已運行的守護程序時適用）。
4. -s, –use-sim-time：啟用 ROS 模擬時間。

10.3.2 delay

用法：ros2 topic delay [選項] 主題名稱
選項：

1. -h, –help：顯示幫助信息並退出。
2. –window WINDOW, -w WINDOW：計算速率的窗口大小，以消息數為單位（默認值：10000）。
3. –spin-time SPIN_TIME：在等待發現時的旋轉時間（僅在不使用已運行的守護程序時適用）。
4. -s, –use-sim-time：啟用 ROS 模擬時間。

10.3.3 echo

用法：ros2 topic echo [選項] 主題名稱 [消息類型]

1. -h, –help：顯示幫助信息並退出。
2. –spin-time SPIN_TIME：在等待發現時的旋轉時間（僅在不使用已運行的守護程序時適用）。
3. -s, –use-sim-time：啟用 ROS 模擬時間。
4. –no-daemon：不要生成或使用已運行的守護程序。
5. –qos-profile preset_profile：訂閱時使用的質量服務預設配置。
6. –qos-depth N：訂閱時的隊列大小設置。
7. –qos-history history_setting：訂閱時的樣本歷史記錄設置。
8. –qos-reliability reliability_setting：訂閱時的可靠性設置。
9. –qos-durability durability_setting：訂閱時的耐久性設置。
10. –csv：將所有遞歸字段用逗號分隔輸出（例如，用於繪圖）。
11. –field FIELD：回顯消息的選定字段。使用'.' 來選擇子字段。
12. –full-length, -f：對於陣列、字節和長度 > '–truncate-length' 的字符串，輸出所有元素。
13. –truncate-length TRUNCATE_LENGTH, -l TRUNCATE_LENGTH：將陣列、字節和字符串截斷為指定長度。
14. –no-arr：不要打印消息的陣列字段。
15. –no-str：不要打印消息的字符串字段。
16. –flow-style：以塊樣式打印集合（與 csv 格式不可用）。
17. –lost-messages：已棄用，不起作用。
18. –no-lost-messages：不要報告消息丟失。
19. –raw：回顯原始二進制表示。
20. –filter FILTER_EXPR：用於過濾要打印的消息的 Python 表達式。
21. –once：僅打印收到的第一條消息，然後退出。

10.3.4 find

用法：ros2 topic find [選項] 主題類型

1. -h, –help：顯示幫助信息並退出。
2. –spin-time SPIN_TIME：在等待發現時的旋轉時間（僅在不使用已運行的守護程序時適用）。
3. -s, –use-sim-time：啟用 ROS 模擬時間。
4. –no-daemon：不要生成或使用已運行的守護程序。
5. -c, –count-topics：僅顯示發現的主題數量。
6. –include-hidden-topics：將隱藏的主題也考慮在內。

10.3.5 hz

用法：ros2 topic hz [選項] 主題名稱

1. -h, –help：顯示幫助信息並退出。
2. –window WINDOW, -w WINDOW：計算速率的窗口大小，以消息數為單位（默認值：10000）。
3. –filter EXPR：僅測量與指定 Python 表達式匹配的消息。
4. –wall-time：使用實際時間計算速率，在模擬期間未發佈時可以很有用。
5. –spin-time SPIN_TIME：在等待發現時的旋轉時間（僅在不使用已運行的守護程序時適用）。
6. -s, –use-sim-time：啟用 ROS 模擬時間。

10.3.6 info

用法：ros2 topic info [選項] 主題名稱

1. -h, –help：顯示幫助信息並退出。
2. –spin-time SPIN_TIME：在等待發現時的旋轉時間（僅在不使用已運行的守護程序時適用）。
3. -s, –use-sim-time：啟用 ROS 模擬時間。
4. –no-daemon：不要生成或使用已運行的守護程序。
5. –verbose, -v：打印詳細信息，如節點名稱、節點命名空間、主題類型、GUID 以及訂閱者和發布者的 QoS 配置文件。

10.4 service

用法：ros2 service [選項] < 命令 >
選項：

1. -h, –help：顯示幫助信息並退出。
2. –include-hidden-services：將隱藏的服務也考慮在內。

命令：

1. call：調用一個服務。
2. find：輸出給定類型的可用服務列表。
3. list：輸出可用服務的列表。
4. type：輸出服務的類型。

10.5 pkg

10.6 rum

Part II
程式設計: 客戶端庫使用

客戶端函式庫是允許使用者實作 ROS 2 程式碼的 API。使用客戶端程式庫，使用者可以存取 ROS 2 概念，例如節點、主題、服務等。

11 colcon 建立環境

colcon 是 ros 的建構工具。

11.1 安裝 colcon

我們安專 colcon 記的往後對於會影響 python 的環境都要小心

```
sudo apt install python3-colcon-common-extensions
```

11.2 建立工作目錄

ROS 工作空間是具有特定結構的目錄。通常有一個 src 子目錄。此子目錄內是 ROS 套件的源代碼所在的位置。通常目錄一開始是空的。

```
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws
---------------------------
.
|- src

1 directory, 0 files
```

11.3 下載範例程式

從 github 下載程式到 src 目錄中。

```
git clone https://github.com/ros2/examples \
    src/examples -b humble
```

11.4 編譯

之後就可以編譯。

```
colcon build
colcon test
```

11.5 環境啟動

編譯完成後就，就有運作環境可以使用。

```
source install/setup.zsh
```

11.6 測試

```
ros2 run examples_rclcpp_minimal_subscriber \
    subscriber_member_function

ros2 run examples_rclcpp_minimal_publisher \
    publisher_member_function
```

12 工作目錄

建立一個工作區並學習如何設定用於開發和測試的覆蓋層。

工作空間是包含 ROS 2 套件的目錄。在使用 ROS 2 之前，有必要在您計劃使用的終端機中取得 ROS 2 安裝工作區。這使得 ROS 2 的軟體包可供您在該終端中使用。

1. 建立工作的資料夾
2. 移動到 src 把程式下載在此
3. 回到工作區的根目錄
4. 檢查依賴
5. 建構程式

12.1 建構前的準備

在建構之前記的要先把程式下載好。

```
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
git clone https://github.com/ros/ros_tutorials.git -b humble
cd ../
rosdep install -i --from-path src --rosdistro humble -y
```

第一次的時後要記得初始化。

```
rosdep install -i --from-path src --rosdistro humble -y

ERROR: your rosdep installation has not been initialized
yet.  Please run:

    sudo rosdep init
    rosdep update
```

如國沒又問題就會有下面的訊息

```
All required rosdeps installed successfully
```

12.2 覆蓋工作區

覆蓋層中的包將覆蓋底層中的包。還可以有多層底層和覆蓋層，每個連續的覆蓋層都使用其父底層的包。

您可以 turtlesim 透過編輯 turtlesim 視窗上的標題列來修改覆蓋層。請 turtle_frame.cpp 在中找到該檔案 /ros2_ws/src/ros_tutorials/turtlesim/src。

turtle_frame.cpp 使用您喜歡的文字編輯器開啟。

1. 找到 turtle_frame.cpp
2. 修改 Title
3. 建構
4. 執行比較

```
cd ~/ros2_ws/src/ros_tutorials/turtlesim/src/turtlesim/
vi turtle_frame.cpp
# 修改記的存檔
colcon build
source install/local_setup.bash
ros2 run turtlesim turtlesim_node
```

13 編寫一個簡單的發布者和訂閱者 (Python)

建立節點，這些節點透過主題以字串訊息的形式相互傳遞訊息。這裡使用的例子是一個簡單的「說話者」和「傾聽者」系統；一個節點發布數據，另一個節點訂閱該主題，以便它可以接收該數據。

13.1 建構包

打開一個新終端並取得 ROS 2 安裝的源代碼，以便 ros2 命令可以運行。

```
ros2 pkg create --build-type ament_python --license Apache-2.0 py_pubsub
```

13.2 編寫發布者節點

建構完成後前往 py_pubsub/py_pubsub 寫程式，這裡用其他大大提供的原始碼。

```
wget https://raw.githubusercontent.com/ros2/\
    examples/humble/rclpy/topics/minimal_publisher/\
    examples_rclpy_minimal_publisher/\
    publisher_member_function.py

wget https://raw.githubusercontent.com/ros2/examples/\
    humble/rclpy/topics/minimal_subscriber/\
    examples_rclpy_minimal_subscriber/\
    subscriber_member_function.py
```

13.3 新增依賴

修改 package.xml，在裡面加入相依的套件。

```
<description>Examples of minimal publisher/subscriber using rclpy</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>

<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

13.4 新增入口點

入口點 (entry_points) talker 與 listener 是用使用時調用的名稱，等號右邊的是執行的程式檔名與要執行的 funtion。

```
    entry_points={
        'console_scripts': [
                'talker = py_pubsub.publisher_member_function:main',
                'listener = py_pubsub.subscriber_member_function:main',
        ],
    },
```

13.5 建構並運行

完成上面的設定之後就可以執行看看

```
    colcon build --packages-select py_pubsub
    source install/setup.bash
    ros2 run py_pubsub talker
    ros2 run py_pubsub listener
```

13.6 程式碼

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import String


class MinimalPublisher(Node):

    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic', 10)
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0

    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1


def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()


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

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import String


class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            'topic',
            self.listener_callback,
            10)
        self.subscription  # prevent unused variable warning

    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s"' % msg.data)


def main(args=None):
    rclpy.init(args=args)

    minimal_subscriber = MinimalSubscriber()

    rclpy.spin(minimal_subscriber)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()


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

Part III
分析 dblanding/diy-ROS-robot

這裡就針對 dblanding 發布在 github 上的文檔來做分析，雖然是 ros1 的版本但是可以借鑒。

14 機器人參數

```
    """
Publish robot parameters for various reasons:
    They might be needed by more than one package.
    It might be nice to be able to modify them at runtime.
    It is an obvious place to find them.

See doc at http://wiki.ros.org/rospy/Overview/Parameter%20Server
"""
import rospy

# Robot parameter values
ROBOT_TICKS_PER_REV = 686.4  # 24:1 worm, 26:10 spur, 11 pole encoder magnet
ROBOT_WHEEL_CIRCUMFERENCE = 0.213  # meters
```

```
ROBOT_TICKS_PER_METER = int(ROBOT_TICKS_PER_REV / ROBOT_WHEEL_CIRCUMFERENCE)
ROBOT_TRACK_WIDTH = .187  # Wheel Separation Distance (meters)
ROBOT_MIN_PWM_VAL = 80  # Minimum PWM value motors will turn reliably
ROBOT_MAX_PWM_VAL = 255  # Maximum allowable PWM value
ROBOT_MIN_X_VEL = 0.1  # Minimum x velocity robot can manage (m/s)
ROBOT_MAX_X_VEL = 0.25  # Maximum x velocity robot can manage (m/s)
ROBOT_MIN_Z_VEL = 0.8  # Minimum theta-z velocity robot can manage (rad/s)
ROBOT_MAX_Z_VEL = 3.0  # Maximum theta-z velocity robot can manage (rad/s)
ROBOT_MTR_KP = 0.5  # Proportional coeff
ROBOT_MTR_KD = 0.2  # Derivative coeff
ROBOT_MTR_MAX_PID_TRIM = 30  # Max allowable value for PID trim term

# end points of segments of piecewise linear curve in descending order
# where curve relates tick rate (tr) to motor speed (s)
ROBOT_TRS_CURVE = ((892, 240),
                   (578, 140),
                   (360, 100),
                   (142, 80))

param_dict = {
    'ROBOT_TICKS_PER_REV': ROBOT_TICKS_PER_REV,
    'ROBOT_WHEEL_CIRCUMFERENCE': ROBOT_WHEEL_CIRCUMFERENCE,
    'ROBOT_TICKS_PER_METER': ROBOT_TICKS_PER_METER,
    'ROBOT_TRACK_WIDTH': ROBOT_TRACK_WIDTH,
    'ROBOT_MIN_PWM_VAL': ROBOT_MIN_PWM_VAL,
    'ROBOT_MAX_PWM_VAL': ROBOT_MAX_PWM_VAL,
    'ROBOT_MIN_X_VEL': ROBOT_MIN_X_VEL,
    'ROBOT_MAX_X_VEL': ROBOT_MAX_X_VEL,
    'ROBOT_MIN_Z_VEL': ROBOT_MIN_Z_VEL,
    'ROBOT_MAX_Z_VEL': ROBOT_MAX_Z_VEL,
    'ROBOT_TRS_CURVE': ROBOT_TRS_CURVE,
    'ROBOT_MTR_KP': ROBOT_MTR_KP,
    'ROBOT_MTR_KD': ROBOT_MTR_KD,
    'ROBOT_MTR_MAX_PID_TRIM': ROBOT_MTR_MAX_PID_TRIM
    }

for key, val in param_dict.items():
    rospy.set_param(key, val)
```

Listing 1: publish_params.py

14.1 註解內容

將機器人參數發布出去有幾個原因：

1. 它們可能被多個軟體包需要。
2. 在運行時修改它們是一件好事。
3. 它是一個明顯的地方可以找到它們。

14.2 幾本內容

作者事先定義各個參數的數值，之後在用字典形式存起來之後在用 items() 來將 key 與 value 成對的存成元組，之後就可以用 rospy 的 set_param() 來將數值發布出去。

14.3 參數表

其中上面的參數含意如下表 1。

Table 1: 參數表

參數名稱	定義
ROBOT_TICKS_PER_REV	每轉的編碼器脈衝數（24:1 蠕蟲、26:10 齒輪、11 極磁鐵）
ROBOT_WHEEL_CIRCUMFERENCE	輪子周長（米）
ROBOT_TICKS_PER_METER	每米的編碼器脈衝數（整數）
ROBOT_TRACK_WIDTH	輪子間的距離（米）
ROBOT_MIN_PWM_VAL	馬達可靠轉動的最小 PWM 值
ROBOT_MAX_PWM_VAL	允許的最大 PWM 值
ROBOT_MIN_X_VEL	機器人可管理的最小 x 速度（米/秒）
ROBOT_MAX_X_VEL	機器人可管理的最大 x 速度（米/秒）
ROBOT_MIN_Z_VEL	機器人可管理的最小 theta-z 速度（弧度/秒）
ROBOT_MAX_Z_VEL	機器人可管理的最大 theta-z 速度（弧度/秒）
ROBOT_MTR_KP	馬達控制的比例係數
ROBOT_MTR_KD	馬達控制的微分係數
ROBOT_MTR_MAX_PID_TRIM	PID 調整項的最大允許值
ROBOT_TRS_CURVE	編碼器每秒產生的脈衝數與馬達速度的曲線描述

15 輪組控制

這部份作者寫了三個文件分別如下。

1. rotary_encoder.py 編碼器計算
2. motor_calibrator.py 校準馬達的參數
3. encoder_publisher.py 控制與發訊息

15.1 rotary_encoder.py 程式分析

這部份作者寫了轉向的判斷。

15.1.1 Decoder 物件建構

開頭的地方可以看到用到了 pigpio，這個是一個數梅派的 gpio 套件有了這個就可以使用 gpio 接腳。
首先是 class 的一 init 的建構內容可以看到會數入下面幾個參數。

1. pi：用於控制 GPIO 的 pigpio 對象。
2. gpioA 和 gpioB：要監聽的兩個 GPIO 腳。
3. callback：當 GPIO 腳狀態變化時調用的回調函數。

之後設定初始的數值。

1. levA 和 levB：記錄兩個 GPIO 腳的狀態（0 或 1）。
2. lastGpio：上一次觸發回調函數的 GPIO 腳。
3. pi.set_mode：設置 GPIO 腳的模式為輸入。
4. pi.set_pull_up_down：設置 GPIO 腳的上拉/下拉電阻。
5. pi.callback：設置 GPIO 腳的回調函數，當腳的電平變化時調用 _pulse 方法。

```
import pigpio
class Decoder:
   """Class to decode mechanical rotary encoder pulses."""
   def __init__(self, pi, gpioA, gpioB, callback):
      self.pi = pi
```

```
        self.gpioA = gpioA
        self.gpioB = gpioB
        self.callback = callback
        self.levA = 0
        self.levB = 0
        self.lastGpio = None
        self.pi.set_mode(gpioA, pigpio.INPUT)
        self.pi.set_mode(gpioB, pigpio.INPUT)
        self.pi.set_pull_up_down(gpioA, pigpio.PUD_UP)
        self.pi.set_pull_up_down(gpioB, pigpio.PUD_UP)
        self.cbA = self.pi.callback(gpioA, pigpio.EITHER_EDGE, self._pulse)
        self.cbB = self.pi.callback(gpioB, pigpio.EITHER_EDGE, self._pulse)
```

Listing 2: 開頭建立編碼器

15.1.2 轉向監測

這個方法 _pulse 是用來解碼旋轉編碼器的脈衝信號的。旋轉編碼器通過兩個 GPIO 腳（A 和 B）發送脈衝信號，這些脈衝信號會根據旋轉方向和速度變化。

輸入的變數如下。

1. gpio 是觸發回調的 GPIO 腳的編號。
2. level 是該 GPIO 腳的電平狀態，可以是 0 或 1。
3. tick 是回調被觸發時的時間戳記，用來測量時間間隔或其他時間相關的操作。

在地一個 if 是要函數中觸發的 GPIO 腳的編號來更新對應的狀態變數，再來的會是一個兩層的 if 結構，第一層 if 就是用於去除抖動也就是同一個電位多次觸發的狀況，再來一層就是判斷正反轉 (正轉 self.callback(1))。

```
   def _pulse(self, gpio, level, tick):
      """
      Decode the rotary encoder pulse.

                    +----------+          +----------+      0
                    |          |          |          |
         A          |          |          |          |
                    |          |          |          |
         +----------+          +----------+          +----- 1

              +----------+          +----------+            0
              |          |          |          |
         B    |          |          |          |
              |          |          |          |
         ----+          +----------+          +----------+  1
      """
      if gpio == self.gpioA:
         self.levA = level
      else:
         self.levB = level;

      if gpio != self.lastGpio: # debounce
         self.lastGpio = gpio
         if   gpio == self.gpioA and level == 1:
            if self.levB == 1:
               self.callback(1)
         elif gpio == self.gpioB and level == 1:
            if self.levA == 1:
               self.callback(-1)
```

Listing 3: _pulse 方法

15.1.3 關閉 gpio

這段程式碼是用來取消旋轉編碼器的解碼器。在這個方法中，它取消了對 GPIO 腳的監聽，從而停止解碼器的操作。通常在不再需要監聽旋轉編碼器的脈衝信號時調用這個方法，以節省資源和確保正確的操作。

```
   def cancel(self):
      """
      Cancel the rotary encoder decoder.
      """
      self.cbA.cancel()
      self.cbB.cancel()
```

Listing 4: 取消監控

15.1.4 測試程式

```
if __name__ == "__main__":
   import time
   import pigpio
   import rotary_encoder
   pos = 0
   def callback(way):
      global pos
      pos += way
      print("pos={}".format(pos))
   pi = pigpio.pi()
   decoder = rotary_encoder.Decoder(pi, 7, 8, callback)
   time.sleep(300)
   decoder.cancel()
   pi.stop()
```

Listing 5: 測試腳本

15.2 motor_calibrator.py 分析

15.2.1 引用的函數

1. rospy: 用於與 ROS 系統通信。
2. pigpio: 控制 Raspberry Pi 的 GPIO。
3. rotary_encoder.Decoder: 監控選轉方向

```
import rospy
import pigpio
from rotary\_encoder import Decoder
```

Listing 6: 引用函數庫

15.2.2 變數宣告

下表2是每個參數所代表的意義，基本上事先定義馬達控制 gpio，包含驅動訊號與霍爾傳感器的訊號，之後之後是設定馬達的狀態。

```
datafile = '/home/ubuntu/catkin_ws/src/my_robot/wheels/data/tr_spd.csv'
DWELL = 4  # seconds to dwell at each speed

# Set up gpio (Broadcom) pin aliases
left_mtr_spd_pin = 17
left_mtr_in1_pin = 27
left_mtr_in2_pin = 22

right_mtr_spd_pin = 11
right_mtr_in1_pin = 10
right_mtr_in2_pin = 9

left_enc_A_pin = 7
left_enc_B_pin = 8

```

```
right_enc_A_pin = 23
right_enc_B_pin = 24

L_mode = 'OFF'  # motor mode: 'FWD', 'REV', 'OFF'
R_mode = 'OFF'
```

Listing 7: 變數宣告

Table 2: 變數表

變數名稱	功能描述
datafile	存儲速度數據的文件路徑，可能是一個 CSV 文件。
DWELL	每個速度設定停留的時間（秒數）。
left_mtr_spd_pin	左馬達的速度控制引腳。
left_mtr_in1_pin	左馬達的方向控制引腳（1）。
left_mtr_in2_pin	左馬達的方向控制引腳（2）。
right_mtr_spd_pin	右馬達的速度控制引腳。
right_mtr_in1_pin	右馬達的方向控制引腳（1）。
right_mtr_in2_pin	右馬達的方向控制引腳（2）。
left_enc_A_pin	左編碼器的引腳 A。
left_enc_B_pin	左編碼器的引腳 B。
right_enc_A_pin	右編碼器的引腳 A。
right_enc_B_pin	右編碼器的引腳 B。
L_mode	左馬達的工作模式，可能是'FWD'（前進）、'REV'（後退）或'OFF'（停止）。
R_mode	右馬達的工作模式，可能是'FWD'（前進）、'REV'（後退）或'OFF'（停止）。

15.2.3 轉向控制

這段程式碼定義了一個名為 set_mtr_spd 的函數，用於驅動馬達使用 PWM 信號。它根據傳入的參數設置左右兩側馬達的速度和方向。

輸入到函數的變數有下面幾個

1. pi：pigpio 的實例，用於控制 GPIO。
2. L_PWM_val：左馬達的 PWM 值，用於控制馬達的速度。
3. R_PWM_val：右馬達的 PWM 值，同樣用於控制馬達的速度。
4. L_mode：左馬達的工作模式，可能是'FWD'（前進）、'REV'（後退）或'OFF'（停止）。
5. R_mode：右馬達的工作模式，同樣可能是'FWD'、'REV' 或'OFF'。

轉向的部份也會應為馬達的種類與接線影響，請確認接線方式與規範。

```
def set_mtr_spd(pi, L_PWM_val, R_PWM_val, L_mode, R_mode):
    """Drive motors using a PWM signal."""

    # Set motor direction pins appropriately
    if L_mode == 'FWD':
        pi.write(left_mtr_in1_pin, 0)
        pi.write(left_mtr_in2_pin, 1)
    elif L_mode == 'REV':
        pi.write(left_mtr_in1_pin, 1)
        pi.write(left_mtr_in2_pin, 0)
    else:  # Parked
        pi.write(left_mtr_in1_pin, 0)
        pi.write(left_mtr_in2_pin, 0)

    if R_mode == 'FWD':
        pi.write(right_mtr_in1_pin, 0)
        pi.write(right_mtr_in2_pin, 1)
    elif R_mode == 'REV':
```

```
        pi.write(right_mtr_in1_pin, 1)
        pi.write(right_mtr_in2_pin, 0)
    else:  # Parked
        pi.write(right_mtr_in1_pin, 0)
        pi.write(right_mtr_in2_pin, 0)

    # Send PWM values to the motors
    pi.set_PWM_dutycycle(left_mtr_spd_pin, L_PWM_val)
    pi.set_PWM_dutycycle(right_mtr_spd_pin, R_PWM_val)

left_pos = 0
```

Listing 8: 馬達動作定義

15.2.4 轉向紀錄

這段程式碼定義了一個回調函數 left_enc_callback，用於處理左編碼器的脈衝信號。每當編碼器產生一個脈衝時，該回調函數會被調用，並將 left_pos 變數的值根據脈衝的方向（+1 或 -1）進行增加或減少。

1. left_pos：用於追蹤左編碼器的位置或旋轉角度的變數，初始值為 0。
2. tick：表示一個脈衝的方向，可能是 +1（正方向）或 -1（負方向）。

```
  left_pos = 0
def left_enc_callback(tick):
    """Add 1 tick (either +1 or -1)"""

    global left_pos
    left_pos += tick
```

Listing 9: 轉向記錄

15.3 motor_calibrator 主循環

這段程式碼是一個主要的控制循環，用於驅動機器人以不同的速度前進並收集旋轉編碼器的數據。讓我們來分析一下：

15.3.1 初始化參數

這段程式碼的目的是初始化並設置機器人的控制參數，準備進行速度控制和數據收集。

1. pigpio.pi() 建立了一個 pigpio 實例 pi
2. Decoder 類建立了兩個編碼器解碼器
3. ROS 節點，並設置了初始值
4. 使用 set_mtr_spd(pi, 0, 0, L_mode, R_mode) 將左右馬達的速度設置為 0，以確保機器人起始時停止運動。
5. 速度列表 speeds，包含從 240 到 60 間隔為 -20

```
    pi = pigpio.pi()
    left_decoder = Decoder(pi, left_enc_A_pin, left_enc_B_pin,\
        left_enc_callback)
    right_decoder = Decoder(pi, right_enc_A_pin, \
        right_enc_B_pin, right_enc_callback)

    # Set up initial values
    rospy.init_node('wheels', anonymous=True)
    rate = rospy.Rate(10)
    prev_left_pos = 0
    prev_right_pos = 0
    set_mtr_spd(pi, 0, 0, L_mode, R_mode)
    speeds = range(240, 60, -20)  # list of speeds at which to drive
    data = ["Lft_TR, Rgt_TR, Avg_TR, spd\n",]  # Header
```

Listing 10: 初始化

15.3.2 while 轉速計算

這邊真的很長，主要是由一個 for 迴圈包著一個 while 回圈，for 迴圈跑得是速度，while 迴圈就是根據數度去測量轉創況

for 開始的就是設定向前轉，之後在歸零計時器這樣做的目的是為了在後續的迴圈中計算每次迭代所花費的時間，以便控制機器人以設定的速度行駛一段時間後，收集左右輪的實際速度數據。

```
    for spd in speeds:
        # set direction
        L_mode = 'FWD'
        R_mode = 'FWD'
        prev_time = rospy.Time.now().to_sec()
        start_time = prev_time
        delta_time = 0.0
```

Listing 11: whil 之前的準備

接下來的是 whil 的速度運算

- 左輪位置變化量 $delta\_left\_pos = left\_pos - prev\_left\_pos$
- 右輪位置變化量 $delta\_right\_pos = right\_pos - prev\_right\_pos$
- 時間間隔 $delta\_time = curr\_time - prev\_time$
- 左輪實際速度 $L\_atr = \frac{delta\_left\_pos}{delta\_time}$
- 右輪實際速度 $R\_atr = \frac{delta\_right\_pos}{delta\_time}$

```
while True:
        # Calculate actual tick rate for left & right wheels
        delta_left_pos = left_pos - prev_left_pos
        delta_right_pos = right_pos - prev_right_pos
        prev_left_pos = left_pos
        prev_right_pos = right_pos
        curr_time = rospy.Time.now().to_sec()
        delta_time = curr_time - prev_time
        prev_time = curr_time
        L_atr = delta_left_pos / delta_time
        R_atr = delta_right_pos / delta_time
```

Listing 12: 轉速運算

15.3.3 while 紀錄轉速

DWELL 的八分之一，則會清空左右輪的速度列表，以便穩定速度。否則，將實際速度值添加到左右輪的速度列表中，以進行後續數據收集。

這段程式碼的作用是確保在開始收集數據之前，機器人的速度已經穩定下來，以確保收集到的數據準確性。

```
    if curr_time - start_time < DWELL/8:
        L_atr_list = []
        R_atr_list = []

    # Collect data
    else:
        L_atr_list.append(L_atr)
        R_atr_list.append(R_atr)
```

Listing 13: 紀錄轉速

15.3.4 while 結束處理

結束時候就會計算單邊平均，與兩邊平均之後結束了。

```
    if curr_time - start_time > DWELL:
        print(f"Done with spd={spd}")
        # find average values of L_atr and R_atr
        L_avg = sum(L_atr_list)/len(L_atr_list)
        R_avg = sum(R_atr_list)/len(R_atr_list)
        combined = (L_avg + R_avg)/2
        data.append(f"{L_avg:.2f}, {R_avg:.2f},{combined:.2f}, {spd}\n")
        break
```

Listing 14: 紀錄轉速

15.3.5 輸出到 csv 檔

結束所有測量之後就會輸出到外頭的 csv 文件中。

```
print("Finished LOOP")
  with open(datafile, 'w') as f:
      for line in data:
          f.write(line)
```

Listing 15: 紀錄資料

15.3.6 結束處理

這裡就是將馬達停下，之後關閉 gpio 的接點。

```
    set_mtr_spd(pi, 0, 0, 'OFF', 'OFF')
    # Clean up & exit
    print("\nExiting")
    left_decoder.cancel()
    right_decoder.cancel()
    pi.set_servo_pulsewidth(left_mtr_spd_pin, 0)
    pi.set_servo_pulsewidth(right_mtr_spd_pin, 0)
    pi.stop()
```

Listing 16: 紀錄資料

15.4 encoder_publisher

程式那容大致分成下面幾個部份。

- 引入函數庫
- 設定變數
- listener() 監聽主題
- listener_callback(msg): 回傳線性 x 與角度 z
- 將 cmd vels 轉換為目標刻度率
- 計算速度
- 設定轉速
- 紀錄旋轉量
- 主循環

15.4.1 函數庫

- rospy：ros 的客戶端通訊程式庫
- geometry_msgs.msg: 幾何學的資料界面
- std_msgs.msg: 資料型態庫
- pigpio:gpio 控制庫
- rotary_encoder: 編碼器的訊號

15.4.2 參數設定

用到的參數如下表3

Table 3: 變數說明

變數	說明
MTR_DEBUG	啟用/停用打印 mtr pwm 值的開關
left_mtr_spd_pin	左馬達速度控制的 GPIO 腳位
left_mtr_in1_pin	左馬達控制輸入 1 的 GPIO 腳位
left_mtr_in2_pin	左馬達控制輸入 2 的 GPIO 腳位
right_mtr_spd_pin	右馬達速度控制的 GPIO 腳位
right_mtr_in1_pin	右馬達控制輸入 1 的 GPIO 腳位
right_mtr_in2_pin	右馬達控制輸入 2 的 GPIO 腳位
left_enc_A_pin	左編碼器 A 相的 GPIO 腳位
left_enc_B_pin	左編碼器 B 相的 GPIO 腳位
right_enc_A_pin	右編碼器 A 相的 GPIO 腳位
right_enc_B_pin	右編碼器 B 相的 GPIO 腳位
TRS_CURVE	TRS 曲線參數，從 rospy 取得的參數
TRS_COEFF	TRS 系數
TICKS_PER_REV	每圈編碼器脈衝數，從 rospy 取得的參數
TICKS_PER_METER	每米編碼器脈衝數，從 rospy 取得的參數
TRACK_WIDTH	車輪軸距，從 rospy 取得的參數
MIN_PWM_VAL	最小 PWM 值，從 rospy 取得的參數
MAX_PWM_VAL	最大 PWM 值，從 rospy 取得的參數
MIN_X_VEL	最小 X 軸速度，從 rospy 取得的參數
new_ttr	目標脈衝率值是否為新值的標誌
L_ttr	左輪目標脈衝率
R_ttr	右輪目標脈衝率
L_spd	左馬達速度（正 8 位元整數）用於 PWM 信號
R_spd	右馬達速度（正 8 位元整數）用於 PWM 信號
L_mode	馬達模式：'FWD'（前進）、'REV'（後退）、'OFF'（關閉）
R_mode	馬達模式：'FWD'（前進）、'REV'（後退）、'OFF'（關閉）
KP	PID 控制器比例參數，從 rospy 取得的參數
KD	PID 控制器微分參數，從 rospy 取得的參數
L_prev_err	左輪上一次誤差
R_prev_err	右輪上一次誤差
MAX_PID_TRIM	PID 修正值的最大值，從 rospy 取得的參數

15.4.3 監聽主題

topic 是 ros 常用的通訊方式，這邊一口氣會跑三個 function，主要是先監聽 topic 之後在由這裡取的，x 的移動數據與 z 的旋轉數據再來就開始計算。好的，讓我一行一行解釋這段程式碼：

global L_ttr, R_ttr, new_ttr：聲明要使用的全局變數。

vel_L_wheel = x - (theta * (TRACK_WIDTH / 2))：計算左輪的目標線速度，根據機器人的運動學模型，左輪速度為 x 減去角速度乘以車輪軸距的一半。

vel_R_wheel = x + (theta * (TRACK_WIDTH / 2))：計算右輪的目標線速度，右輪速度為 x 加上角速度乘以車輪軸距的一半。

L_rate = vel_L_wheel * TICKS_PER_METER：將左輪的目標線速度轉換為目標脈衝率，即將每秒米轉換為每秒脈衝數。

R_rate = vel_R_wheel * TICKS_PER_METER：將右輪的目標線速度轉換為目標脈衝率，同樣是將每秒米轉換為每秒脈衝數。

if L_ttr != L_rate:：檢查左輪的目標脈衝率是否與前一次計算的值不同。

L_ttr = L_rate：如果左輪的目標脈衝率與前一次計算的值不同，則更新左輪的目標脈衝率為新計算的值。

new_ttr = True：將 new_ttr 標誌設置為 True，表示左輪的目標脈衝率是新值。

if R_ttr != R_rate:：檢查右輪的目標脈衝率是否與前一次計算的值不同。

R_ttr = R_rate：如果右輪的目標脈衝率與前一次計算的值不同，則更新右輪的目標脈衝率為新計算的值。

new_ttr = True：將 new_ttr 標誌設置為 True，表示右輪的目標脈衝率是新值。

這段程式碼的作用是根據機器人的運動學模型計算出左右輪的目標脈衝率，並檢查是否有新的目標脈衝率值。

15.4.4 tr_to_spd 計算速度

這段程式碼是一個註釋，說明了一個功能，即將目標脈衝率轉換為速度值。根據註釋，目標脈衝率和傳送到馬達的速度信號之間的關係大致遵循一條拋物線曲線，在低速時斜率較緩。這種關係通常通過由一系列線性片段組成的分段線性曲線來近似。在算法中，將檢查脈衝率以確定應用於其的段，從斜率最陡（最高值）的段開始。然後使用適用的斜率和截距來計算速度值。這樣的轉換通常用於將脈衝率轉換為實際速度值，以便控制機器人達到特定的運動行為。

15.4.5 set_mtr_spd 設定馬達的轉速

這段程式碼是一個註釋，描述了一個功能，即根據左右輪的目標脈衝率來推斷馬達的速度和模式，並驅動馬達。目標脈衝率被轉換為" 最佳猜測" 的 PWM 值，以使用經驗曲線來驅動馬達。

註釋還指出，脈衝率可以是正數或負數，而馬達速度（spd）將始終是一個正的 8 位元整數（0-255）。因此，需要為馬達指定一個模式：FWD（前進）、REV（後退）或 OFF（關閉）。

在低速時，一個非常小的 PWM 信號很難使馬達克服一個本質上無法預測的摩擦量。這使得機器人非常難以準確地跟隨命令速度。

為了提高機器人準確跟隨命令速度的能力（特別是在進行緩慢的原地轉向時尤為明顯），使用 PID 反饋來最小化目標脈衝率和實際脈衝率之間的差異（誤差）。

15.4.6 主函數

這段程式碼的作用是訂閱 /cmd_vel 主題以獲取機器人的速度指令，並發布左右輪的編碼器數據和機器人的實際速度。同時，它也設置了一個迴圈來處理速度控制和數據發布。具體來說，這段程式碼執行以下操作：

監聽 /cmd_vel 主題，並調用 listener_callback 函數來處理速度指令。
設置發布左右輪編碼器數據的發布者，以及發布機器人實際速度的發布者。
計算左右輪的實際脈衝率，並根據脈衝率計算機器人的實際速度。
發布左右輪的編碼器數據和機器人的實際速度。
設置馬達速度以控制機器人運動。
在迴圈中使用 rate.sleep() 來控制迴圈頻率，以達到每秒 10 次的發布頻率。
在程式執行結束時，進行清理操作，包括取消訂閱編碼器數據、停止馬達控制信號的發送以及關閉 pigpio。
這樣的程式碼結構通常用於機器人控制系統中，用於實現速度控制和速度反饋。

16 里程計通訊

這部份會去監聽輪組控制時發布的三個主題。

1. /right_ticks 主題，用於接收右輪的編碼器數據。
2. /left_ticks 主題，用於接收左輪的編碼器數據。
3. /act_vel 主題，用於接收機器人的實際速度信息。

16.1 用到的套件

這部份用到的套件如下表4

16.2 變數初始化

這段程式碼是用來初始化 ROS 节点的，並且從 ROS 參數伺服器中獲取了兩個重要的參數值：每米的編碼器脈衝數和車輪間的距離。接著定義了一些變量，包括左輪和右輪的編碼器脈衝數、機器人的線速度（米/秒）和角速度（弧度/秒）。這些變量將用於計算機器人的里程計數據。

16.3 回調函數

這段程式碼定義了兩個回調函數 left_tick_callback 和 right_tick_callback，分別用於接收來自左輪和右輪編碼器的訊息。這兩個函數將分別更新全局變量 left_ticks 和 right_ticks 的值，以便後續計算里程。接著，定義了兩個函數 left_tick_listener 和 right_tick_listener，用於訂閱 ROS 訊息主題 /left_ticks 和 /right_ticks，並將它們與對應的回調函數關聯起來，以便接收編碼器訊息。

回調函數 act_vel_callback，用於從 Twist 訊息中提取 linear.x 和 angular.z 的值。這兩個值將被存儲在全局變量 vx 和 vth 中，以便後續計算里程。接著，定義了一個函數 act_vel_listener，用於訂閱 ROS 訊息主題 /act_vel，並將其與 act_vel_callback 回調函數關聯起來，以接收實際速度訊息。

Table 4: 模組/類別及其用途

模組/類別	用途
rospy	創建 ROS 節點、訂閱和發佈主題等
tf2_ros	處理轉換關係
Odometry	定義機器人的里程計信息
Point	定義機器人的位置信息
Pose	定義機器人的姿態信息
Quaternion	定義機器人的旋轉信息（用四元數表示）
Twist	定義機器人的速度信息
Vector3	定義三維向量
TransformStamped	定義轉換關係的消息類型
Int32	定義整數的消息類型
Float32	定義浮點數的消息類型
asin	計算反正弦值
sin	計算正弦值
cos	計算餘弦值
pi	值

16.4 主迴圈

這段程式碼包含了主要的里程計計算和發佈邏輯。在一個循環中，它首先計算了兩個輪子的行進距離，然後根據兩輪之間的差異計算機器人的轉向角度。接著，它使用這些數據更新機器人的位置和姿勢。最後，它發佈了里程計訊息和 TF2 的 Transform。

需要注意的是，在計算轉向角度時，程式碼使用了 asin 函數。但是，這可能會產生 ValueError，因為當 cycle_diff / TRACK_WIDTH 的絕對值大於 1 時，asin 的參數必須在 -1 和 1 之間。當發生這種情況時，程式碼將 cycle_angle 設置為 0，並打印出一條消息，並繼續運行。

此外，程式碼還設置了里程計和速度訊息的協方差矩陣。這些值通常是根據機器人的實際性能和環境來調整和優化的。最後，它使用 publish 方法發佈里程計訊息，並使用 r.sleep() 使循環按照設置的頻率運行。

程式碼是一個循環，持續運行直到 ROS 被關閉。在每個循環中，它計算了自上一次循環以來的時間差，並根據左右輪的編碼器讀數計算了機器人的行進距離和角度變化。然後，它使用這些數據來更新機器人的位姿（位置和方向），並將該信息發布到 ROS 中的/odom 主題上，以便其他節點可以使用。最後，它更新了上一次循環的時間，並通過 r.sleep() 等待下一個循環。這個循環使機器人能夠持續跟踪自身的運動狀態，並在 ROS 中提供里程計信息。

17 導航設定

這部份作者沒寫什麼原始檔，基本上就設定 ros 變數以及啟動檔。表 5

17.1 param(變數) 設定

Table 5: 參數檔

檔名	功能
base_local_planner_params.yaml	局部路徑規劃器參數設定檔。
costmap_common_params.yaml	成本地圖共用參數設定檔。
dwa_local_planner_params.yaml	動態窗口法局部路徑規劃器參數設定
global_costmap_params.yaml	全局成本地圖參數設定檔
global_planner_params.yaml	全局路徑規劃器參數設定檔。
local_costmap_params.yaml	局部成本地圖參數設定檔。
move_base_params.yaml	Move Base 功能包參數設定檔。
navfn_global_planner_params.yaml	navfn 全局路徑規劃器參數設定檔。

17.1.1 base_local_planner_params.yaml

配置檔定義了 TrajectoryPlannerROS 的設定，主要用於 move_base 套件中的軌跡規劃器。以下是各參數的簡要說明：

- max_vel_x：最大直線速度。

- min_vel_x：最小直線速度。
- holonomic_robot：是否為全向移動機器人。
- meter_scoring：是否對距離進行計分。
- xy_goal_tolerance：目標點的 XY 距離容忍度。
- yaw_goal_tolerance：目標點的 Yaw（偏航角）容忍度。

17.1.2 costmap_common_params.yaml

- obstacle_range: 障礙物檢測範圍為 3.0 米。
- raytrace_range: 射線追踪範圍為 3.5 米。
- footprint: 機器人的車輪軌跡形狀，由四個座標點組成，表示機器人在地圖上的輪廓。
- map_topic: 使用的地圖主題。
- robot_base_frame: 機器人底盤的坐標系。
- update_frequency: 更新頻率為每秒 10 次。
- publish_frequency: 發布頻率為每秒 10 次。
- transform_tolerance: 變換的容忍度為 0.3。

插件：

static_layer: 靜態圖層，用於處理靜態障礙物的成本。obstacle_layer: 障礙物圖層，用於接收雷射掃描器的觀測數據，並更新成本地圖。inflation_layer: 膨脹圖層，用於在成本地圖中將障礙物周圍的區域擴大，以確保機器人周圍有足夠的安全空間。這段配置文件是用於設置 ROS 的 move_base 套件中的 costmap 部分，用於在導航過程中生成和更新地圖的成本地圖。這些參數設置了機器人在地圖上的輪廓、障礙物檢測範圍、射線追踪範圍以及地圖的更新頻率和發布頻率等。插件部分定義了靜態圖層、障礙物圖層和膨脹圖層，這些圖層用於處理靜態障礙物的成本、接收雷射掃描器的觀測數據以及在成本地圖中擴大障礙物周圍的區域。

17.1.3 dwa_local_planner_params.yaml

這段配置文件是用於設置 DWAPlannerROS，這是 ROS 中用於差動驅動機器人的動態窗口法 (DWA) 軌跡規劃器的參數。以下是各部分的說明：

- Robot Configuration Parameters: 設置機器人的速度限制和加速度限制，以及機器人的運動方式（例如差動驅動）。
- Goal Tolerance Parameters: 設置目標姿勢的容忍度，包括位置和姿勢的容忍度。
- Forward Simulation Parameters: 設置前向模擬的參數，包括模擬時間和速度取樣。
- Trajectory Scoring Parameters: 設置軌跡評分的參數，包括對全局路徑計劃的依附程度、到達目標的優先度和避開障礙物的權重。
- Oscillation Prevention Parameters: 設置防止機器人在同一位置徘徊的參數。
- Debugging: 設置是否發佈軌跡和成本網格的點雲，以及全局坐標系的設置。

這些參數用於配置 DWAPlannerROS，以便生成機器人的軌跡，並確保機器人能夠安全且有效地移動到目標位置。

17.1.4 global_costmap_params.yaml

這段配置文件是用於設置全局代價地圖（global costmap）的參數。全局代價地圖是機器人在全局範圍內進行路徑規劃時考慮的地圖，通常用於避免障礙物和計算到達目標的路徑。

- global_frame: 設置全局代價地圖的坐標系。在這種情況下，地圖使用”map” 坐標系。
- resolution: 設置地圖的解析度，即每個單元格的大小。在這種情況下，解析度為 0.1 米。
- rolling_window: 指示是否使用滾動窗口方式來管理全局代價地圖。如果設置為 false，則表示使用固定窗口方式。滾動窗口方式可以在機器人移動時動態調整地圖，而固定窗口方式則不會。

17.1.5 global_planner_params.yaml

於設置全局規劃器（Global Planner）的參數。全局規劃器用於計算機器人在全局地圖上的路徑，以達到目標位置。

- old_navfn_behavior: 是否完全模仿 navfn 的行為，使用其他布林參數的默認值，默認為 false。
- use_quadratic: 是否使用潛在場的二次近似。否則，使用一種更簡單的計算方法，默認為 true。
- use_dijkstra: 是否使用 Dijkstra 算法。否則，使用 A* 算法，默認為 true。
- use_grid_path: 是否創建一條沿著網格邊界的路徑。否則，使用梯度下降法，默認為 false。
- allow_unknown: 允許規劃器通過未知空間進行規劃，默認為 true。需要在障礙物/體素層中設置 track_unknown_space: true 才能正常工作。
- planner_window_x, planner_window_y: 規劃器視窗的大小，默認為 0.0。
- default_tolerance: 如果目標在障礙物中，則計劃到半徑 default_tolerance 內的最近點，默認為 0.0。
- publish_scale: 發布的潛在場尺度，默認為 100。
- planner_costmap_publish_frequency: 規劃器代價地圖的發布頻率，默認為 0.0。
- lethal_cost: 致命代價，默認為 253。
- neutral_cost: 中性代價，默認為 50。
- cost_factor: 將每個代價從代價地圖中的每個代價乘以的因子，默認為 3.0。
- publish_potential: 是否發布潛在場地圖（這不像 navfn pointcloud2 的潛在場地圖），默認為 true。

17.1.6 local_costmap_params.yaml

這段配置文件是用於設置局部代價地圖（Local Costmap）的參數。局部代價地圖用於在機器人周圍創建一個代價地圖，以便計劃器可以避免障礙物。

- rolling_window: 是否使用滾動窗口模式，默認為 true。滾動窗口模式意味著只保留機器人周圍的地圖部分，隨著機器人的移動而更新地圖。
- resolution: 代價地圖的解析度，即每個單元格的大小，默認為 0.1 米。
- inflation_radius: 充氣半徑，用於將障礙物周圍的代價值增加，使機器人有更大的安全距離，默認為 0.2 米。
- width, height: 代價地圖的寬度和高度，以米為單位，默認為 2.0 米。
- plugins: 代價地圖使用的插件列表。在這個配置中，只有一個插件 obstacle_layer，用於處理障礙物層的配置。
- obstacle_layer: 障礙物層的配置，包括觀測源和雷射掃描傳感器的相關信息。

17.1.7 move_base_params.yaml

這段配置文件是用於 Move Base 的參數設置，Move Base 是 ROS 中用於機器人導航的重要組件之一。以下是這些參數的解釋：

- shutdown_costmaps: 是否在 Move Base 結束時關閉代價地圖，默認為 false。
- controller_frequency: 控制器更新頻率，默認為 10.0 Hz。
- controller_patience: 控制器等待時間，超過這個時間仍然沒有達到目標，將視為失敗，默認為 15.0 秒。
- planner_frequency: 規劃器更新頻率，默認為 5.0 Hz。
- planner_patience: 規劃器等待時間，超過這個時間仍然沒有找到計劃，將視為失敗，默認為 5.0 秒。
- oscillation_timeout: 搖擺避障超時時間，即機器人在同一地點來回搖擺的最大時間，默認為 10.0 秒。
- oscillation_distance: 搖擺避障距離，即機器人視為搖擺的最小移動距離，默認為 0.2 米。

- conservative_reset_dist: 保守重置距離，當機器人與目標之間的距離超過此距離時，將重置規劃器，默認為 3.0 米。
- base_local_planner: 使用的本地規劃器，默認為"dwa_local_planner/DWAPlannerROS"，即動態窗口法（DWA）規劃器。
- base_global_planner: 使用的全局規劃器，默認為"navfn/NavfnROS"，即導航函數（Navfn）規劃器，也可以選擇其他全局規劃器，

17.1.8 navfn_global_planner_params.yaml

這是用於 NavfnROS 全局路徑規劃器的參數配置。NavfnROS 是 ROS 中的一個基於 A* 算法的全局路徑規劃器。以下是這些參數的含義：

- visualize_potential: 是否將潛在場景視覺化為點雲，在 RViz 中顯示，默認為 false。
- allow_unknown: 是否允許規劃器在未知空間中創建計劃，默認為 false。需要在代價地圖參數中將 track_unknown_space 設置為 true 才能生效。
- planner_window_x 和 planner_window_y: 限制規劃範圍的窗口大小，默認為 0.0，表示不限制範圍。
- default_tolerance: 如果目標在障礙物內，規劃器將規劃到最接近目標的半徑內的點，默認為 0.0。這個值越大，搜索範圍越廣，計算速度可能變慢。

17.2 launch(啟動) 檔

Table 6: 參數檔

檔名	功能
gmapping.launch	啟動 GMapping SLAM 算法的 launch 檔案。
map.launch	發佈地圖的 ROS launch 檔案。
move_base.launch	啟動 move_base 導航節點的 launch 檔案。
navigation.launch	啟動導航相關節點和服務的 launch 檔案。

17.2.1 gmapping.launch

程式碼是用來啟動一個叫做 slam_gmapping 的 ROS 節點，這個節點用於將激光掃描數據轉換成地圖，並實現機器人的同時定位和地圖構建（SLAM）功能。以下是這段程式碼中的一些重要參數和功能：

- scan_topic 參數用來指定激光掃描的 ROS 主題名稱，預設為"scan"，可以通過這個參數來動態改變節點訂閱的激光掃描主題。
- odom_frame、base_frame 和 map_frame 分別是機器人里程計、基礎座標和地圖座標的框架名稱。
- maxUrange 和 maxRange 分別是激光傳感器的最大可用範圍和最大範圍。
- transform_publish_period 是 tf 廣播的時間間隔，用於發布機器人的位姿。
- linearUpdate 和 angularUpdate 分別是機器人平移和旋轉的最小距離，用於確定何時處理一次新的掃描。
- particles 是粒子濾波器中的粒子數量，用於估計機器人的姿勢。
- xmin、ymin、xmax 和 ymax 定義了地圖的初始大小。
- 其他參數如 sigma、kernelSize、lstep、astep、iterations 等用於控制地圖建構過程中的不同參數和算法。

17.2.2 map.launch

這段程式碼設定了兩個 ROS 節點，用於地圖轉換和地圖服務器。

- static_transform_publisher 用於發佈靜態的座標變換，即發佈 map 到 odom 座標系之間的關係。參數 args="0 0 0 0 0 0 1 map odom" 指定了變換的平移和旋轉關係，這對於機器人的定位至關重要。
- map_server 節點是一個地圖服務器，用於發佈地圖數據。它會讀取指定的地圖文件（參數 $(arg map_file)），並將地圖數據發佈到 /map 和 /map_metadata 主題上，供其他節點使用。

17.2.3 move_base.launch

這段程式碼是一個包含多個節點的 ROS launch 文件，用於建立一個包含地圖、定位、導航和移動基地等功能的機器人系統。

- static_transform_publisher 節點用於發佈 map 到 odom 座標系之間的靜態座標變換關係。
- map_server 節點負責讀取地圖文件並發佈地圖數據到 /map 和 /map_metadata 主題。
- amcl 是用於機器人定位的節點，包含在 amcl_diff.launch 文件中，設置了訂閱 /scan、/tf、/initialpose 和 /map 主題，並發佈 /amcl_pose、/particlecloud 和 /tf 主題。
- move_base 節點是導航功能的核心，接收目標位置信息，通過 base_local_planner 執行運動控制，並通過 global_costmap 和 local_costmap 規劃全局和局部地圖。

這個 launch 文件集成了機器人的地圖、定位、導航和移動基地功能，使得機器人能夠在已知地圖中進行自主導航。

17.2.4 navigation.launch

這個 launch 文件配置了機器人的地圖、定位、導航和移動基地功能。

- static_transform_publisher 節點用於發佈 map 到 odom 座標系之間的靜態座標變換關係。
- map_server 節點負責讀取地圖文件並發佈地圖數據到 /map 和 /map_metadata 主題。
- amcl 是用於機器人定位的節點，包含在 amcl_diff.launch 文件中，設置了訂閱 /scan、/tf、/initialpose 和 /map 主題，並發佈 /amcl_pose、/particlecloud 和 /tf 主題。
- move_base 節點是導航功能的核心，接收目標位置信息，通過 dwa_local_planner 和 global_planner 執行運動控制，並通過 global_costmap 和 local_costmap 規劃全局和局部地圖。

這個 launch 文件集成了機器人的地圖、定位、導航和移動基地功能，使得機器人能夠在已知地圖中進行自主導航。

18 總啟動檔

這個 launch 文件配置了一個機器人系統，包括了以下功能：

- robot_params 節點用於發佈機器人的參數。
- mtr_encdr 節點用於發佈輪子編碼器的信息。
- ros_imu_bno055_node 節點用於處理 IMU 數據，提供了一系列參數用於配置 IMU。
- rplidarNode 節點用於控制 RPLidar 激光雷達，提供了一系列參數用於配置激光雷達。
- tf_base_link_imu 和 tf_base_link_laser 節點用於發佈靜態坐標變換關係，將 base_link 與 imu_link、laser 坐標系連接起來。
- odometer 節點用於發佈里程計信息。
- robot_pose_ekf 節點用於執行擴展卡爾曼濾波器，整合里程計、IMU 和視覺里程計等信息，提供機器人的位姿信息。
- rplidar_motor_control 節點用於自動控制 RPLidar 激光雷達的馬達，當不需要時自動關閉。

這個 launch 文件整合了機器人系統中的多個功能節點，使得機器人能夠正常運行並提供所需的信息。

Part IV
slam tool box 分解

Part V
專題製作

19 操作方式

使用方式有幾個東西要啟動

1. 機器啟動
    - 光學雷達
    - 馬達區動
    - urdf 與 tf 樹
2. 導航算法
    - slam
    - nav 套件

19.1 機器啟動

依序下這幾個指令來請動機器人。

```
ros2 launch sllidar_ros2 sllidar_a1_launch.py
ros2 launch project display.launch.py
ros2 launch project wheel_launch.py
```

19.1.1 注意事項

起動時要注意 ttyUSB 的設備，這跟 linux 的設備管理有關西，先接入的設備是 USB0 再來是 USB1，所以要注意設備連接的順序。

這些程式要在 ros2 設備端來啟動。

19.2 導航算法

啟動完機器設備後就可以開始啟動算法。

```
ros2 launch nav2_bringup navigation_launch.py
ros2 launch slam_toolbox online_async_launch.py
```

19.2.1 注意事項

要注意啟動算法之錢必須確定 tf 樹的設定，這個會影響計算的結果且要注意是否有正確指向。

20 工作大綱

我們的專題目前訂成 ros2 的自走車，最基本的功能就是要作到可以在地圖上指定一個點他就會自行前往。

要達到這個功能，就要用到nav2的功能，我原先參考的資料有下面。

- turtlebot3
- diy ros

我原先是打算用 turtlebot3 的套件但是，價格真的不是很低而且他們為了後續的升級基本上用到的零件都超出需要的性能，所以我嘗試找有沒有土炮的版本，就被我找到 dblanding 大大的板本，我算了一下架個大約會在 1w 上下，但是有些零件實驗室應該會有。

所以我們就先以完成 dblanding 大大的版本為一個小目標之後看時間，如果我們做太慢，就這樣就好。

20.1 任務

根據 dblanding 的 github 上的資料，他把整體分成下面幾個部份。

1. wheels：輪子控制相關程序
2. odom_pub：里程計數據發布程序
3. robot_nav：機器人導航程序和配置
4. robot_params：機器人參數配置
5. bringup：啟動配置和程序管理

單然這要有 ros 的概念，很不幸的他是用 ros1 做的而且在 2025ros1 就不再會有更新，所以我們會用 ros2 的版本。

20.2 先備知識

要有以下的概念會比較好。

- linux 的指令操作 (ssh)
- ros2 的通訊概念
- python

21 外觀電路設計

組件	價格
ELEGOO Tumbller 自平衡機器人車套件	$85
100 RPM 齒輪馬達（2）	$30
Raspberry Pi 3 B+	$55
32GB EVO Plus Class 10 微型 SDHC	$10
Geekworm Raspberry Pi UPS HAT	$29
3.7V 18650 鋰電池（2）	$15
L298N 馬達驅動控制板	$3
Slamtec RPLIDAR A1M8	$100
BNO055 慣性測量單元	$40
USB 轉 TTL 轉接器	$12
GPIO 分線板和排線	$10
1/4 英寸 x 4 英寸 x 3 英尺 S4S 橡木板	$5
滾珠輪	$4
30C 500mAh 3S 11.1V 鋰聚合物電池	$7

BNO055	CP2104 Friend
Vin	5V
GND	GND
SDA	RXD
SCL	TXD

Table 7: 連接 BNO055 和 CP2104 Friend 的表格

21.1 電路接法

如圖1是驅動器的接線方式，如圖2是霍爾編碼器的接腳，圖3是 arduino 與編碼器跟驅動器的接線圖。gpio 的接腳可以由程式定義，基本上根據定義來接線，IMU 的接法如表7接法一樣之後在把 CP2104 接到數梅派的 USB 就可以用 UART 通訊讀取資料。

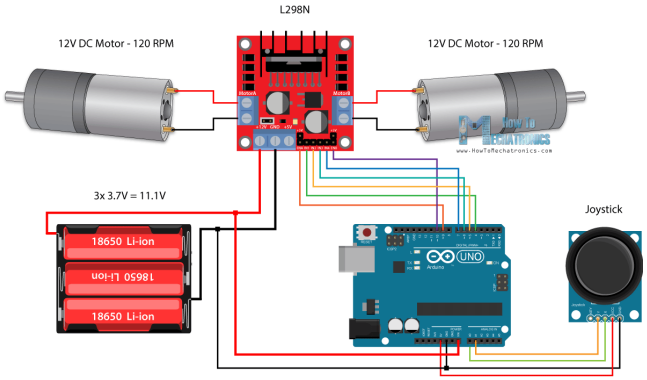
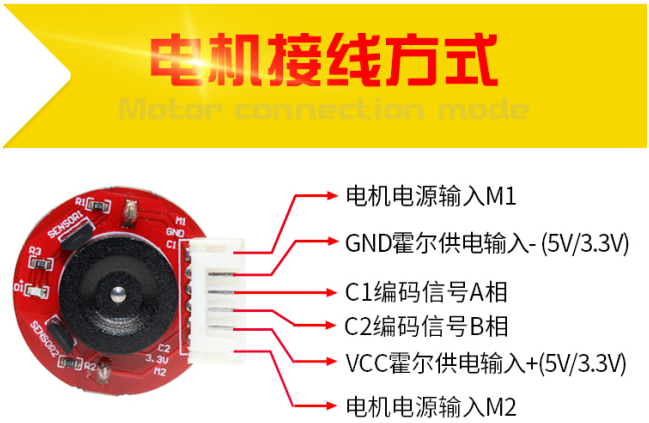


Figure 1: 馬達驅動器接線圖



编码器参数

类型	AB双相增量式磁性霍尔编码器
线速	基础脉冲11 PPR×齿轮减速比
供电电压	DC 3.3V / DC5.0
基本功能	自带上拉整形电阻，单片机直连
接口类型	PH2.0（标配连接线）
输出信号类型	方波 AB相
响应频率	100KHz
基础脉冲数	11 PPR
磁环触发极数	22极（11对极）

Figure 2: 霍爾傳感器

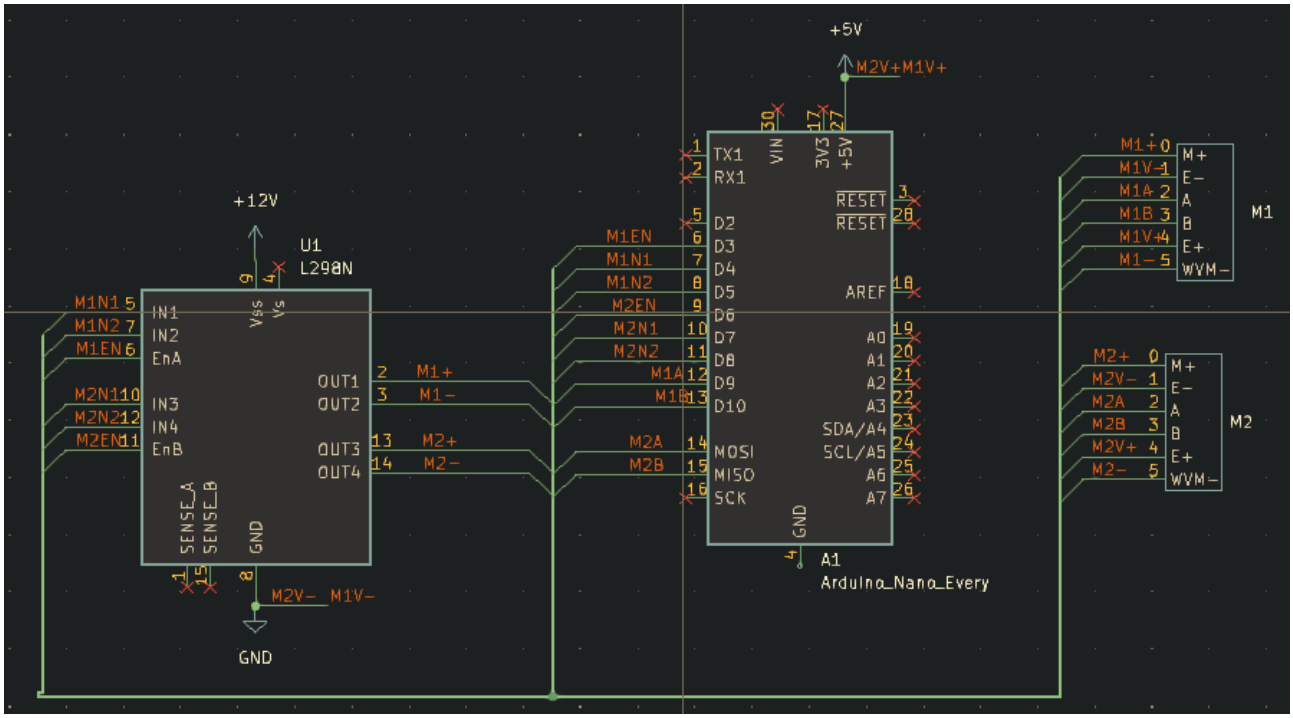


Figure 3: 電路接線圖

22 輪子控制

控制馬達我們要注意的幾個部份

1. 控制訊息的格式
2. arduino 與 ros 之間的通訊
3. PID 控制器
4. 編碼器計算
5. 里程計計算

22.1 控制訊息格式

下面有兩個本次在輪組控制時用到的格式，一個 ros 內建的用於表示機器在空間中的狀態，如線性方向的變化 (x,y,z) 與旋轉變化的 (x,y,z)，另外一個是我自訂意的變數格式用於表示兩個輪子的轉動速度 (rps)

- geometry_msgs/msg/Twist
- diff_robot_description/msg/Wheel

```
ros2 interface show diff_robot_description/msg/Wheel
---------------------------------------------------------
float64 left_wheel_speed   # 左輪轉速
float64 right_wheel_speed  # 右輪轉速
```

```
ros2 interface show geometry_msgs/msg/Twist
---------------------------------------------------------------------------
# This expresses velocity in free space broken into its linear and angular parts.
Vector3  linear
float64 x
float64 y
float64 z
Vector3  angular
float64 x
float64 y
float64 z
```

22.2 arduino 與 ros 通訊

我們這次是用 uart 通訊來做 arduino 與 ros 之間的數據交換。ros 可以用 python 的 pyserial 套件來做 urat 通訊，arduino 用 serial 就可以了。

- ros 給 arduino 目標轉速
- arduino 給 ros 測量轉速

22.3 PID

PID 馬達控制透過比例、積分、微分控制器調節馬達輸出，使實際速度接近目標速度。比例控制速度誤差、積分控制積累誤差、微分控制速度變化率，綜合調整可達到精準控制效果。可以參考 PID 馬達控制

本次使用 arduino pid 的套件

- Setpoint: 目標轉速
- Input: 測量轉速
- Output:pwm 輸出量

23 輪子控制程式

輪子本次我們決定用 arduino 來寫控制程式，其中用 uart 根 ros2 通訊，節點如下。

1. control_node 用於訂閱 cmd 的訊息來轉換成輪子轉速發布給 Wheel
2. Serial 與 arduino 通訊並把轉速傳給 diffodom
3. 接收實際轉速換成里程位置

23.1 control node

這是一個狀態資訊轉成差速器的節點。Wheel 資料是自訂意的資料型態，內容是兩輪的轉速。

- 訂閱 cmd_vel
- 發布 Wheel
- 將 Twist 資料轉成 Wheel

23.2 control node python

- $W_r = V + \frac{\omega \times L}{2}$
- $W_l = V - \frac{\omega \times L}{2}$
  - W_l 左輪轉速
  - W_r 右輪轉速
  - L 輪距
  - V 前進速度
  - ω 轉彎角速度

```python
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
from std_msgs.msg import Float32
from diff_robot_description.msg import Wheel

class ControlNode(Node):
    def __init__(self):
        super().__init__('control_node')
        self.get_logger().info('control node ....')
        self.subscription = self.create_subscription(
            Twist,
            'cmd_vel',
            self.cmd_vel_callback,
            10)
        self.Wheel = self.create_publisher(
            Wheel,
            'Wheel',
            10)
        self.get_logger().info('control node start')

    def cmd_vel_callback(self, msg):
        linear_vel = msg.linear.x
        angular_vel = msg.angular.z
        speed = Wheel()
        L = 0.5
        speed.right_wheel_speed = linear_vel+angular_vel*L/2
        speed.left_wheel_speed  = linear_vel-angular_vel*L/2
        self.get_logger().info('whell:\t%f'%speed.right_wheel_speed)
        self.get_logger().info('left:\t%f'%speed.left_wheel_speed)
        self.Wheel.publish(speed)

def main(args=None):
```

```
    rclpy.init(args=args)
    control_node = ControlNode()
    rclpy.spin(control_node)
    control_node.destroy_node()
    rclpy.shutdown()

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

Listing 17: control node

23.3 serial node(未完成)

與 arduino 做通訊目前尚未完工

```
import serial
import rclpy
from rclpy.node import Node
from diff_robot_description.msg import Wheel
class SerialNode(Node):

    def __init__(self):
        super().__init__('serial_node')
        self.publisher_ = self.create_publisher(Wheel, 'Wheel_speed', 10)
        self.subscription = self.create_subscription(
            Wheel,
            'Wheel',
            self.wheel_callback,
            10)
        self.get_logger().info('serial node start')

    def  wheel_callback(self,msg):

        self.get_logger().info(str(msg))
        self.get_logger().info('get data:')
        self.publisher_.publish(msg)
        pass
    def serial_get(self):
        Wheel_speed = Wheel
        Wheel_speed.left_wheel_speed = 0.85
        Wheel_speed.right_wheel_speed =0.88

def main(args=None):
    rclpy.init(args=args)
    Serial_Node = SerialNode()
    rclpy.spin(Serial_Node)
    rclpy.shutdown()

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

Listing 18: control node

23.4 diffodom 里程計計算

```
from os import walk
import rclpy
from rclpy.node import Node
from diff_robot_description.msg import Wheel
from geometry_msgs.msg import Twist, Vector3
from math import cos, sin
from geometry_msgs.msg import TransformStamped
import tf2_ros
import geometry_msgs.msg

```

```python
class OdomCalculator(Node):
    def __init__(self):
        super().__init__('odom_calculator')
        self.subscription = self.create_subscription(
            Wheel,
            'Wheel_speed',
            self.wheel_speed_callback,
            10)
        self.publisher = self.create_publisher(
        Twist,
        'odom',
        10)
        self.current_pose = [0.0,0.0,0.0]
        self.tf_broadcaster = tf2_ros.StaticTransformBroadcaster(self)
    def wheel_speed_callback(self,msg):
        left_speed = msg.left_wheel_speed
        right_speed= msg.right_wheel_speed
        wheel_base = 0.5

        linear_vel = (left_speed + right_speed) / 2
        angular_vel = (right_speed - left_speed) / wheel_base
        self.update_odometry(linear_vel, angular_vel)


    def update_odometry(self, linear_vel, angular_vel):
        time_step = 0.1  # 假設時間間隔為0.1秒
        self.current_pose[0] += linear_vel * cos(self.current_pose[2]) * time_step
        self.current_pose[1] += linear_vel * sin(self.current_pose[2]) * time_step
        self.current_pose[2] += angular_vel * time_step
        twist_msg = Twist(
            linear=Vector3(x=self.current_pose[0], y=self.current_pose[1], z=0.0),
            angular=Vector3(x=0.0, y=0.0, z=self.current_pose[2]))
        self.publisher.publish(twist_msg)
        transform_stamped = geometry_msgs.msg.TransformStamped()
        transform_stamped.header.stamp = self.get_clock().now().to_msg()
        transform_stamped.header.frame_id = 'odom'
        transform_stamped.child_frame_id = 'base_link'
        transform_stamped.transform.translation.x = self.current_pose[0]
        transform_stamped.transform.translation.y = self.current_pose[1]
        transform_stamped.transform.rotation.w = cos(self.current_pose[2] / 2)
        transform_stamped.transform.rotation.x = 0.0
        transform_stamped.transform.rotation.y = 0.0
        transform_stamped.transform.rotation.z = sin(self.current_pose[2] / 2)

        self.tf_broadcaster.sendTransform(transform_stamped)


def main(args=None):
    rclpy.init(args=args)
    odom_calculator = OdomCalculator()
    rclpy.spin(odom_calculator)
    odom_calculator.destroy_node()
    rclpy.shutdown()


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

Listing 19: control node

24 里程記設定

再有編碼器的情況下我們就可以計算我們的移動距離與轉向。相關的之料可以提供給導航模組去使用。

25 導航設定

這部份我還沒研究完，我看 dblanding 大大這部份只有寫參數設定，根啟動檔，這邊應該除了 dblanding 的 github 還要在查看其他資料。

25.1 導航參數文件

檔案名稱	用途
base_local_planner_params.yaml	配置基礎局部規劃器（local planner）的參數
costmap_common_params.yaml	配置共用的地圖（costmap）參數，用於全局和局部 costmap
dwa_local_planner_params.yaml	配置動態窗口局部規劃器（Dynamic Window Approach）的參數
global_costmap_params.yaml	配置全局 costmap 的參數
global_planner_params.yaml	配置全局規劃器（global planner）的參數
local_costmap_params.yaml	配置局部 costmap 的參數
move_base_params.yaml	配置 move_base 模組的參數，用於整合全局和局部規劃器以及 costmap
navfn_global_planner_params.yaml	配置基於 Dijkstra 或 A* 算法的全局規劃器（navfn）的參數

26 參數設定

參數如下

27 啟動檔

28 SLAM Toolbox

SLAM Toolbox：這是一個用於 ROS 2 的 SLAM 解決方案，用於構建機器人周圍環境的地圖並同時估計機器人的位置。SLAM Toolbox 通常會生成一個地圖，描述了機器人周圍的環境，並用於後續的導航任務。

```
ros2 pkg executables slam_toolbox
---------------------------------
slam_toolbox async_slam_toolbox_node
slam_toolbox localization_slam_toolbox_node
slam_toolbox map_and_localization_slam_toolbox_node
slam_toolbox merge_maps_kinematic
slam_toolbox sync_slam_toolbox_node
```

1. async_slam_toolbox_node：非同步 SLAM 節點，用於接收激光雷達數據並建構地圖，同時估計機器人的位置。這個節點可以在 SLAM 過程中非同步地處理數據，提高了效率。
2. localization_slam_toolbox_node：定位節點，用於在已知地圖的情況下估計機器人的位置。這個節點通常用於在已經建立好地圖的環境中進行機器人定位。
3. map_and_localization_slam_toolbox_node：地圖和定位節點，結合了地圖建構和定位功能。這個節點可以同時建構地圖並估計機器人的位置，適用於需要即時定位和地圖更新的場景。
4. merge_maps_kinematic：地圖合併工具，用於將多個地圖合併成一個更大的地圖。這個工具可以用於將不同時間或不同地點生成的地圖合併成一個整體地圖。
5. sync_slam_toolbox_node：同步 SLAM 節點，類似於非同步 SLAM 節點，用於接收激光雷達數據並建構地圖，但在處理數據時採用同步方式，可以更精確地估計機器人的位置。

28.1 slam toolbox 節點訊息

async_slam_toolbox_node 只有一個節點 slam toolbox，他的訊息如下

- 訂閱者（Subscribers）：
  - /map：接收地圖資料 (nav_msgs/msg/OccupancyGrid)。
  - /parameter_events：接收參數事件 (rcl_interfaces/msg/ParameterEvent)。
  - /scan：接收雷射掃描資料 (sensor_msgs/msg/LaserScan)。
  - /slam_toolbox/feedback：接收互動標記反饋 (visualization_msgs/msg/InteractiveMarkerFeedback)。
- 發布者（Publishers）：
  - /map：發布地圖資料 (nav_msgs/msg/OccupancyGrid)。
  - /map_metadata：發布地圖的元資料 (nav_msgs/msg/MapMetaData)。
  - /parameter_events：發布參數事件 (rcl_interfaces/msg/ParameterEvent)。
  - /pose：發布具有協方差的姿態資訊 (geometry_msgs/msg/PoseWithCovarianceStamped)。
  - /rosout：用於 ROS 輸出 (rcl_interfaces/msg/Log)。
  - /slam_toolbox/graph_visualization：發布圖形可視化資料 (visualization_msgs/msg/MarkerArray)。
  - /slam_toolbox/scan_visualization：發布雷射掃描資料的可視化 (sensor_msgs/msg/LaserScan)。
  - /slam_toolbox/update：用於更新互動標記的訊息 (visualization_msgs/msg/InteractiveMarkerUpdate)。

  - /tf：發布 TF 訊息 (tf2_msgs/msg/TFMessage)。
- 服務伺服器（Service Servers）：
  - /slam_toolbox/clear_changes：用於清除變更的服務 (slam_toolbox/srv/Clear)。
  - /slam_toolbox/describe_parameters：用於描述參數的服務 (rcl_interfaces/srv/DescribeParameters)。
  - /slam_toolbox/deserialize_map：用於反序列化地圖的服務 (slam_toolbox/srv/DeserializePoseGraph)。
  - /slam_toolbox/dynamic_map：用於獲取動態地圖的服務 (nav_msgs/srv/GetMap)。
  - /slam_toolbox/get_interactive_markers：用於獲取互動標記的服務 (visualization_msgs/srv/GetInteractiveMarkers)。
  - /slam_toolbox/get_parameter_types：用於獲取參數類型的服務 (rcl_interfaces/srv/GetParameterTypes)。
  - /slam_toolbox/get_parameters：用於獲取參數的服務 (rcl_interfaces/srv/GetParameters)。
  - /slam_toolbox/list_parameters：用於列出參數的服務 (rcl_interfaces/srv/ListParameters)。
  - /slam_toolbox/manual_loop_closure：用於手動閉環的服務 (slam_toolbox/srv/LoopClosure)。
  - /slam_toolbox/pause_new_measurements：用於暫停新測量的服務 (slam_toolbox/srv/Pause)。
  - /slam_toolbox/save_map：用於保存地圖的服務 (slam_toolbox/srv/SaveMap)。
  - /slam_toolbox/serialize_map：用於序列化地圖的服務 (slam_toolbox/srv/SerializePoseGraph)。
  - /slam_toolbox/set_parameters：用於設置參數的服務 (rcl_interfaces/srv/SetParameters)。
  - /slam_toolbox/set_parameters_atomically：用於原子設置參數的服務 (rcl_interfaces/srv/SetParametersAtomically)。
  - /slam_toolbox/toggle_interactive_mode：用於切換互動模式的服務 (slam_toolbox/srv/ToggleInteractive)。

提供一系列服務，包括清除變化、描述參數、反序列化地圖、獲取動態地圖、獲取互動式標記等。

28.1.1 topic 節點

上面的節點就可以看得出來他的 topic 很多。

- /joint_states：發布機器人各個關節的狀態，包括位置、速度等信息，通常由機器人的控制器發布。
- /map：SLAM（同步定位與地圖構建）算法生成的地圖，通常是一個佔用格地圖（Occupancy Grid Map），描述了機器人周圍的環境。
- /map_metadata：地圖的元數據，包含了地圖的尺寸、分辨率等信息。
- /parameter_events：發布參數的變更事件，用於通知參數發生了變化。
- /pose：發布機器人的姿態信息，包括位置和方向。
- /robot_description：包含了機器人的 URDF（Unified Robot Description Format）描述，用於描述機器人的結構和幾何形狀。
- /rosout：ROS 系統的輸出信息，包括日誌和調試信息。
- /scan：激光雷達的掃描數據，用於感測機器人周圍的障礙物。
- /slam_toolbox/feedback：互動標記的反饋信息，用於顯示和操作地圖上的互動標記。
- /slam_toolbox/graph_visualization：圖形可視化數據，用於在地圖上顯示標記或軌跡等信息。
- /slam_toolbox/scan_visualization：激光雷達數據的可視化，用於在 RViz 等工具中顯示激光雷達掃描。
- /slam_toolbox/update：更新互動標記的消息，用於更新地圖上的互動標記。
- /tf：用於傳遞坐標變換信息，描述了不同坐標系之間的關係。
- /tf_static：用於傳遞靜態的坐標變換信息，這些變換在運行時不會改變。

28.1.2 參數

- angle_variance_penalty: 角度變異懲罰。
- base_frame: 機器人的基礎座標系。
- ceres_dogleg_type: Ceres 解算器的 Dogleg 類型。
- ceres_linear_solver: Ceres 解算器的線性求解器。
- ceres_loss_function: Ceres 解算器的損失函數。
- ceres_preconditioner: Ceres 解算器的預處理器。
- ceres_trust_strategy: Ceres 解算器的信任策略。
- coarse_angle_resolution: 粗糙角度解析度。
- coarse_search_angle_offset: 粗糙搜索角度偏移量。
- correlation_search_space_dimension: 相關性搜索空間維度。
- correlation_search_space_resolution: 相關性搜索空間解析度。
- correlation_search_space_smear_deviation: 相關性搜索空間模糊偏差。
- debug_logging: 調試日誌記錄。
- distance_variance_penalty: 距離方差懲罰。
- do_loop_closing: 是否執行循環閉合。
- enable_interactive_mode: 是否啟用互動模式。
- fine_search_angle_offset: 精細搜索角度偏移量。
- interactive_mode: 互動模式。
- link_match_minimum_response_fine: 連接匹配最小響應（細）。
- link_scan_maximum_distance: 連接掃描最大距離。
- loop_match_maximum_variance_coarse: 循環匹配最大方差（粗）。
- loop_match_minimum_chain_size: 循環匹配最小鏈尺寸。
- loop_match_minimum_response_coarse: 循環匹配最小響應（粗）。
- loop_match_minimum_response_fine: 循環匹配最小響應（細）。
- loop_search_maximum_distance: 循環搜索最大距離。
- loop_search_space_dimension: 循環搜索空間維度。
- loop_search_space_resolution: 循環搜索空間解析度。
- loop_search_space_smear_deviation: 循環搜索空間模糊偏差。
- map_file_name: 地圖檔案名稱。
- map_frame: 地圖座標系。
- map_name: 地圖名稱。
- map_start_at_dock: 地圖是否從碼頭開始。
- map_start_pose: 地圖的起始姿態。
- map_update_interval: 地圖更新間隔。
- minimum_angle_penalty: 最小角度懲罰。
- minimum_distance_penalty: 最小距離懲罰。

- minimum_time_interval: 最小時間間隔。
- minimum_travel_distance: 最小行駛距離。
- minimum_travel_heading: 最小行駛方向。
- mode: 模式（地圖構建或定位）。
- odom_frame: 里程計座標系。
- paused_new_measurements: 暫停新測量。
- paused_processing: 暫停處理。
- position_covariance_scale: 位置協方差比例。
- qos_overrides: QoS 覆蓋。
- resolution: 解析度。
- scan_buffer_maximum_scan_distance: 掃描緩衝區最大掃描距離。
- scan_buffer_size: 掃描緩衝區大小。
- scan_queue_size: 掃描隊列大小。
- scan_topic: 掃描主題。
- solver_plugin: 解算器插件。
- tf_buffer_duration: TF 緩衝區持續時間。
- throttle_scans: 掃描節流。
- transform_publish_period: 變換發布週期。
- transform_timeout: 變換超時。
- use_map_saver: 是否使用地圖保存器。
- use_response_expansion: 是否使用響應擴展。
- use_scan_barycenter: 是否使用掃描重心。
- use_scan_matching: 是否使用掃描匹配。
- use_sim_time: 是否使用模擬時間。
- yaw_covariance_scale: 偏航協方差比例。

29 lidar

29.1 操作方法

本次使的 lidar 是 RPLIDAR-A1，再有 ros2 的套件輔助下我們可以教簡單使用。

1. 使用 git 把原始碼安裝在工作目錄
2. 編譯原始碼
3. 執行測試

29.1.1 安裝編譯原始碼

動作流程如下。

```
cd ~/ros2_ws/src
git clone https://github.com/Slamtec/sllidar_ros2
cd ../
rosdep install -i --from-path src --rosdistro humble -y
colcon build --packages-select sllidar_ros2
```

1. 移動到工作目錄的原始碼資料夾
2. 下載 lidar 的套件
3. 回到工作根目錄
4. 檢查相依程式
5. 開始編譯

29.1.2 測試與使用

編譯完原始碼的話基本上就可以使用了，使用方法如下

```
source install/setup.zsh
ros2 launch sllidar_ros2 sllidar_a1_launch.py
#可以選折下面有視覺化設定的
ros2 launch sllidar_ros2 view_sllidar_a1_launch.py
```

1. 啟用 ros2 環境
2. 開啟 lidar 的請動檔

這邊要注意使用的光達型號不同型號有對應的啟動檔，也可以直接使用可是化的啟動檔 view 開頭的那個。

29.2 檢視資料

接下來就以觀察這個套件有哪些要注意的地方。

- 開啟的節點
- 主題的發布與訂閱
- 資料的格式
- 節點參數

29.2.1 節點說明

這個光達啟動檔產生的節點為/sllidar_node，我們可以用 info 查詢相關訊息。

```
ros2 node info /sllidar\_node
```

這樣可以看到下面有關 node 的通訊狀況，包含他會訂閱/發布哪些主題與服務根動作。

- 訂閱器（Subscribers）：
  - /parameter_events：訂閱參數事件的消息，類型為 rcl_interfaces/msg/ParameterEvent。
- 發布器（Publishers）：
  - /parameter_events：發布參數事件的消息，類型為 rcl_interfaces/msg/ParameterEvent。
  - /rosout：發布 ROS 的日誌消息，類型為 rcl_interfaces/msg/Log。
  - /scan：發布激光雷達掃描消息，類型為 sensor_msgs/msg/LaserScan。

- 服務伺服器（Service Servers）：
  - /sllidar_node/describe_parameters：提供描述參數的服務，類型為 rcl_interfaces/srv/DescribeParameters。
  - /sllidar_node/get_parameter_types：提供獲取參數類型的服務，類型為 rcl_interfaces/srv/GetParameterTypes。
  - /sllidar_node/get_parameters：提供獲取參數值的服務，類型為 rcl_interfaces/srv/GetParameters。
  - /sllidar_node/list_parameters：提供列出參數的服務，類型為 rcl_interfaces/srv/ListParameters。
  - /sllidar_node/set_parameters：提供設置參數值的服務，類型為 rcl_interfaces/srv/SetParameters。
  - /sllidar_node/set_parameters_atomically：提供原子設置多個參數值的服務，類型為 rcl_interfaces/srv/SetParametersAtomically。
  - /start_motor：提供啟動馬達的服務，類型為 std_srvs/srv/Empty。
  - /stop_motor：提供停止馬達的服務，類型為 std_srvs/srv/Empty。
- 服務客戶端（Service Clients）：(未提供)
- 動作伺服器（Action Servers）：(未提供)
- 動作客戶端（Action Clients）：(未提供)

這些功能使得 sllidar_node 節點能夠與其他節點進行通訊和交互操作，從而實現控制和獲取激光雷達的功能。

29.2.2 topic 查詢

首先要確認/scan 的訊息。

```
    ros2 lidar info /scan
----------------------------------------
Type: sensor_msgs/msg/LaserScan
Publisher count: 1
Subscription count: 0
```

我們看得出來他的，訂閱與發不的狀況根他的數據界面。再來我們可以用 echo 來監聽這個主題。

```
    ros2 topic echo /scan --once
----------------------------------
    header:
  stamp:
    sec: 1710051588
    nanosec: 187130763
  frame_id: laser
angle_min: -3.1241390705108643
angle_max: 3.1415927410125732
angle_increment: 0.005806980188935995
time_increment: 0.0001362734183203429
scan_time: 0.1470390260219574
range_min: 0.15000000596046448
range_max: 12.0
ranges:
.......
intensities:
```

有關各個數值的意義如下

- header：包含時間戳記（stamp）和框架 ID（frame_id），用於確定訊息的時間和座標系。
- stamp：時間戳記，以秒（sec）和納秒（nanosec）表示。

- frame_id：訊息的座標系標識符，在這裡是 laser。
- angle_min：激光雷達掃描的最小角度（弧度）。
- angle_max：激光雷達掃描的最大角度（弧度）。
- angle_increment：每個激光束之間的角度增量（弧度）。
- time_increment：每個激光束的時間增量（秒）。
- scan_time：完成一次完整掃描所需的時間（秒）。
- range_min：激光雷達能夠測量的最小距離（公尺）。
- range_max：激光雷達能夠測量的最大距離（公尺）。
- ranges：每個激光束測量的距離值陣列。
- intensities：每個激光束的強度值陣列（有些激光雷達可以測量強度，但並非所有）。

29.2.3 參數查詢

接下來就是參數的訊息了，透過 param dump 來查看

```
    ros2 param dump /sllidar_node
-------------------------------------------
/sllidar_node:
  ros__parameters:
    angle_compensate: true
    channel_type: serial
    frame_id: laser
    inverted: false
    qos_overrides:
      /parameter_events:
        publisher:
          depth: 1000
          durability: volatile
          history: keep_last
          reliability: reliable
    scan_frequency: 10.0
    scan_mode: ''
    serial_baudrate: 115200
    serial_port: /dev/ttyUSB0
    tcp_ip: 192.168.0.7
    tcp_port: 20108
    udp_ip: 192.168.11.2
    udp_port: 8089
    use_sim_time: false
```

這是 sllidar_node 節點的參數設定。讓我們逐個解釋：

1. angle_compensate：角度補償，設置為 true 表示啟用角度補償。
2. channel_type：通道類型，這裡設置為 serial。
3. frame_id：訊息的座標系標識符，在這裡是 laser。
4. inverted：是否反向，設置為 false 表示不反向。
5. qos_overrides：QoS 覆蓋設定，這裡設置了一些 QoS 參數。
6. scan_frequency：掃描頻率，設置為 10.0 Hz。
7. serial_baudrate：串口波特率，設置為 115200。
8. serial_port：串口端口，設置為/dev/ttyUSB0。

9. tcp_ip：TCP IP 地址，設置為 192.168.0.7。
10. tcp_port：TCP 端口，設置為 20108。
11. udp_ip：UDP IP 地址，設置為 192.168.11.2。
12. udp_port：UDP 端口，設置為 8089。
13. use_sim_time：使用模擬時間，設置為 false 表示不使用模擬時間。

這些參數描述了 sllidar_node 節點的配置，包括通訊方式、座標系、掃描頻率等，我們是走 serial 通訊的，所以要注意 port 的參數未來接上的設定。

29.3 資料視覺化

這裡我們用 rviz 可以更清楚的看到光達所掃掉的資料。

1. 打開終端機
2. 輸入 rviz2
3. 設定 fixed frame = laser
4. 在 add 加入 laser scan 的資料

30 yolov8 辨識

ros2 有 yolov8 的套件可以用 yolov8_ros。

30.1 安裝方法

流程如下，就照指令做就好了。

```
cd ~/ros2_ws/src
git clone https://github.com/mgonzs13/yolov8_ros.git
pip3 install -r yolov8_ros/requirements.txt
cd ~/ros2_ws
rosdep install --from-paths src --ignore-src -r -y
colcon build
```

30.2 基本操作

```
$ ros2 launch yolov8_bringup yolov8.launch.py
```

30.3 主題與參數

- Topics
  - /yolo/detections: Objects detected by YOLO using the RGB images. Each object contains a bounding boxes and a class name. It may also include a mak or a list of keypoints.
  - /yolo/tracking: Objects detected and tracked from YOLO results. Each object is assigned a tracking ID.
  - /yolo/debug_image: Debug images showing the detected and tracked objects. They can be visualized with rviz2.
- Parameters
  - model: YOLOv8 model (default: yolov8m.pt)
  - tracker: Tracker file (default: bytetrack.yaml)
  - device: GPU/CUDA (default: cuda:0)
  - enable: Wether to start YOLOv8 enabled (default: True)

- threshold: Detection threshold (default: 0.5)
- input_image_topic: Camera topic of RGB images (default: /camera/rgb/image_raw)
- image_reliability: Reliability for the image topic: 0=system default, 1=Reliable, 2=Best Effort (default: 2)

30.4 注意事項

使用時他會去訂閱/camera/rgb/image_raw 的 topic 所以要把想要做辨識的圖發布到這裡。

30.5 程式分析

30.5.1 yolov8_node.py

這段程式碼是一個 ROS 2 節點，用於在圖像中偵測和追蹤物體，並且發布偵測到的物體資訊。節點使用 YOLOv8 模型進行物體偵測，並透過 ROS 2 訊息將偵測到的物體發布出去。

該節點繼承自 LifecycleNode 類別，實現了節點的生命週期方法（on_configure、on_activate、on_deactivate、on_cleanup），並實現了 enable_cb 方法來處理啟用/禁用節點的服務請求。節點還實現了一些輔助方法來解析 YOLOv8 的偵測結果，並將結果轉換為 ROS 2 訊息。

在 main 函數中，節點初始化後手動觸發了配置和啟用操作，並通過 rclpy.spin(node) 來啟動節點的訊息迴圈，等待訊息的到來和處理。src

- ___init___(self, **kwargs) None: 初始化函式，用於初始化節點的各種參數和屬性。
- on_configure(self, state: LifecycleState)
  TransitionCallbackReturn: 配置函式，用於配置節點在啟動前的相關設置。
- enable_cb(self, request, response): 啟用/禁用節點的服務回調函式。
- on_activate(self, state: LifecycleState)
  TransitionCallbackReturn: 啟動函式，用於啟動節點並開始執行相應的任務。
- on_deactivate(self, state: LifecycleState)
  TransitionCallbackReturn: 停用函式，用於停用節點並進行清理工作。
- on_cleanup(self, state: LifecycleState)
  TransitionCallbackReturn: 清理函式，用於在節點結束運行時進行清理工作。
- parse_hypothesis(self, results: Results)
  List[Dict]: 解析假設的函式，用於解析物體偵測的結果，返回一個假設列表。
- parse_boxes(self, results: Results)
  List[BoundingBox2D]: 解析框框的函式，用於解析物體偵測的結果，返回一個框框列表。
- parse_masks(self, results: Results)
  List[Mask]: 解析遮罩的函式，用於解析物體偵測的結果，返回一個遮罩列表。
- parse_keypoints(self, results: Results)
  List[KeyPoint2DArray]: 解析關鍵點的函式，用於解析物體偵測的結果，返回一個關鍵點列表。
- image_cb(self, msg: Image)
  None: 圖像回調函式，用於接收來自訂閱的圖像訊息並進行物體偵測處理

31 實做紀錄

31.1 2-25 到 3-02

本週拿到 jetson nano 了。

- jetson nano

31.1.1 完成任務

本週住要將 jection 的作業環境搭建完成，只是 nvidia 提供的版本是，ubuntu18.04 的版本這不是我們要的版本，所以提升到 22.04 但是應為套件的關西更新根升級的時間會花很久。

- 安裝 jetpack4.6 板
- 將內部作業系統升級 22.04
- usb 與 ssh 通訊測試
- 完成 3D 列硬的參數測試

31.1.2 遇到的困難

jetson nano 已經停產，所以 nvidia 沒有為 nano 做軟體的更新服務所以 jetpack 可以用的版本停在 4.6 板，再上去的版本沒有直接支援，但這部份只是光方沒寫支援實際我還沒測過，本次的作法是用 jetpack4.6 內的 ubuntu18.04 直接升級到 22.04，目前還沒遇到兼容性問題。

- nvidia 軟體支援差

31.2 3-3 到 3-9

這周零件陸陸續續的收到了，現在手邊的東西有。

- 光達
- 馬達驅動器
- jetson nano n nano 以今停產，
- 不斷電系統
- imu
- 航空電池
- usb to ttl
- 輪子

31.2.1 完成任務

這周主要收到光達，以測試光達為主。

31.2.2 遇到的困難

剛開始的時候雖然官方有提供驅動，所以輕鬆的可以看到/scan 的訊息，但是我在 rviz2 上只能在 laser 上看到訊號，在其他座標係就看不到了，這主要是我的 tf 觀念有問題要。

- 注意 tf id
- 靜態廣播
- urdf 設定

31.3 3-10 到 3-14

- if 樹設定
- urdf 模型設定

31.4 3-15 到 3-21

- 光學雷達里程計
- slam 製圖

31.5 3-22 到 3-28

- 確認馬達相關參數
- ros2 馬達控制節點
- ros2 馬達 serial 節點
- ros2 馬達里程節點

32 4 月實做紀錄

32.1 401 到 407

輪組的材料檢查

- 確認接線方法
- 設計個模組間的接線

32.2 408 到 414

完成了輪子編碼器設定，馬達的轉速測量

- 完成輪組控制電路
- 測量控制訊號
- 測量霍爾編碼器訊號

32.3 415 到 421

本週期中考基本上沒有進度，主裝材料改成用材料行的盒子。

- 完成 pid 調整
- 完成輪組控制鐵三角 (目標、控制、里程)
- 3d 列硬大失敗
- 修改外觀材料

32.4 422 到 428

以完成基本的框架，現在可以進行基本的控制 slam 與導航。

- 車子的前後左右的控制
- 光達的 slam 製圖
- nav2 導航

32.4.1 問題

還是 jetson 的兼容問題，主要是官方沒有提供 22.04 的版本所以我們從官方提供的最新 18.04 進行升級，但是還是有某些軟體不會跟著所以再把我筆點的程式與 ros2 的套件轉移到 jetson nano 來運作時就時常會有軟體相依性的問題。

再來可能是官方已經擺爛了所以 20.04 或是往後感覺他的驅動都可能會有問題，這次 arduino nano 在根 jetson nano 做 uart 的通訊時，也出現亂碼很明縣就是有大量的雜訊干擾雖然反覆的交叉比對之後最後有把通訊問題排除。

usb wifi 非常不穩定，我在用 ssh 登入 jetson 時很卡並且運算上感覺不是很好。