ROS2導入

目次

1 ROS2 について

1.1 ROS とは

Robot Operating System の頭文字から名前が来ている. 名前の通りロボット用の OS のようなもの. 実際には OS ではなくロボットの制御をする際に必要な機能があらかじめ実装されているフレームワーク. ROS には ROS と ROS2 が存在する.ROS2 は ROS の後継であり,ROS のサポートは現在されておらず,ROS2 を使用することが一般的である. ROS2 にはディストリビューションが複数存在し, 現在長期サポートされているものは Jazzy Jalisco で 2029 年 5 月 31 日までサポートされている.

1.2 ROS2 でできること

ROS2 はフレームワークなので, 外部プログラムと組み合わせれば何でもできるがここではほぼ標準でできることについて紹介する.

ROS2 でできること:

- プログラム間通信 (一番重要)
- 並列処理が簡単
- 共通の定義済みの型が使用できる
- 他人の作ったプログラムの導入が楽
- 高度な数学知識なしで様々なことが行える (ちょっとはわからないと何もできない)

上記のことについて詳しく説明する.

1.3 プログラム間の通信について

通常プログラム間で通信を行うとなるとローカルかネットワークは関係なく UDP Socket や TCP Socket などを作成し, そこにデータをいれて送受信することになる. これをプログラム毎に記述するとなると大変である.
また, プログラム毎にポートを割り振ったり,IP を合わせたりしないと通信ができないのでめんどくさい. しかし,ROS2 を使用すれば, 標準でプログラム間での通信が可能である.

ROS2 ではプログラムのことをノードと呼ぶ (厳密には違う). データのやり取りをするハブを topic と呼ぶ. 通信の方法には PublishSubscribe 通信と Service 通信の 2 つの通信方法がある. PublishSubscribe 通信とはデータを topic に送信ノードが Publish(送信) して受信ノードが Subscribe(受信) する通信方法. この通信の良いところは 1: 複数通信が可能であるところ. データを topic に流しさえすれば設定なしで, 複数のノードからその topic の内容を Subscribe することができる. この通信が便利なのでこの通信を使う目的で使うというところが大きい. この 1: 複数通信するシステム (分散システム) のプロトコルのことを DDS(Data Distribution Service) と呼ぶ.
通信の図を図 1 に示す.

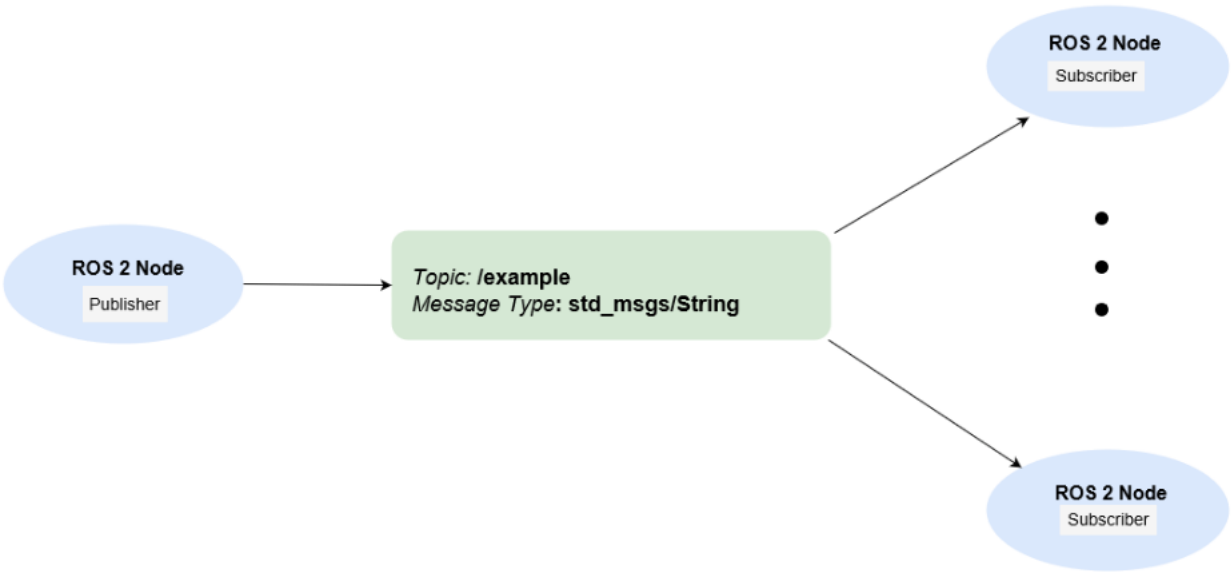


図 1: PubSub 通信の図 [1]

図 1 を見れば 1: 複数通信のイメージが掴めたと思う.Publish は高頻度で行うため, リアルタイム処理に適している. そのため, センサのデータを送信したりすることに使用される.
次に,Service 通信の図を図 2 に示す.

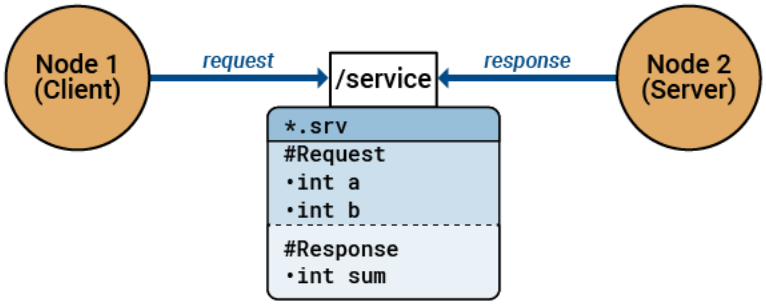


図 2: Service 通信の図 [2]

図 2 よりサービスはノードがクライアントでサービスがサーバーという関係になっている (厳密には違うがそう考えて問題ない). リクエストを送信し, サービスで処理をしてレスポンスを返すのでリアルタイム性には欠ける.

1.4　並列処理が簡単

並列処理を行う場合通常はスレッドを作成し, 管理する必要があるが,ROS2 フレームワークを使用するとそれも自動化できる.

1.5　共通の定義済みの型が使用できる

ROS2 では標準で文字列やセンサ, 環境マップやカメラ映像などを保持する配列などをまとめた構造体が定義されている.
例えば,string なら std_msgs::msg::String で定義されている,IMU センサなら sensor_msgs::msg::Imu のように標準で用意されている.
これの何が嬉しいかというと, 共同で開発するときに, 通信する際の型が決まってなければうまく通信ができない. 人数が多くなるとそれも崩れてくるので標準で用意されているのはありがたい.

1.6　他人の作ったプログラムの導入が楽

型などが決まっているので他人の作ったコードを簡単に統合することができる. 研究者などが ROS2 用にパッケージを作ってくれている場合もあるが, 完全に独自実装の場合もある.
使いたいプログラムが独自実装の場合は ROS2 で使えるように自分で変更しないと行けないがパッケージならたいていコンパイルすればそのまま動く. またこれは研究者が作ったものにとどまらず, 部活の中で誰かに処理を任せていた部分のノードができた際に簡単に統合ができる. これができるから ROS2 は強いし導入したい.

1.7　高度な数学知識なしで様々なことが行える

先程の内容と被るが, 高度なアルゴリズムが実装されているパッケージが github などに転がっているのでそれを使えばそのパッケージでできることは簡単に知識なしでできる.
例えば,lidar を使用してロボットの自己位置推定と環境マップの作成 (これらを合わせて SLAM と呼ぶ) が既存パッケージで行うことができる. slam を一から実装しようとすると一般的な微積分, 線形代数, 確率統計, 最適化についての知識が必要 (これが理解できたら大抵のことはできる). これらの知識なしで, 簡単にそれができるのは素晴らしいことである.

2　ROS2 を試す

ROS2 で一番の障壁になるのが環境である.ROS2 は linux(特に ubuntu) 向けに作成されている.windows にネイティブで入れて動かすこともできるらしいが, やったことはない.

ROS2 を使うためだけに **OS を変えろ**や**デュアルブートにしろ**などは言えないので開発環境も揃うので Docker を使って開発をすることとする.

2.1 Docker とは

Docker はコンテナ技術を用いて, ネイティブの os の上で別の os を動かすことができるものである.Docker を使えば windows の上で ubuntu を動かすことができる (windows の Docker は wsl 上で動くので wsl と環境はあまり変わらない). また,Dockerfile という, 入れるアプリやライブラリなどを記述したものをビルドすればイメージというものができ, そのイメージは同じ Dockerfile をビルドしたものなら環境がどのパソコンでも同じになる.
Docker は開発環境を完全に揃えることができるので一つ環境を整備すればそれを共有することができる. これも楽. 以上のことより,ROS2 を使用する際には Docker を使用することにする.
しかし, メリットばかりではない. ラズパイなどに乗せる場合はネイティブで入れた方が Docker を噛まさずに動かせるので負荷が軽減される.
また,windows 上で Docker を動かして ROS2 が動くか試してみたが, 動きはするものの,ROS2 での DDS 通信 (外部から) がコンテナまで届かなかった. これは windows に送られてきて wsl に転送する際にマルチキャスト udp パケットが全て弾かれているからだと考えられるが解決方法はわからない. つまり,windows で開発する人はロボットで処理が完結しないような設計ならば, ロボットと通信が一切できないということになる. これはどうにかしないといけない. 通信はできなくてもプログラムの開発はできるので, シミュレーションなどを使用して確認すれば一応は開発できる.

2.2 Docker を使う

先程説明した Docker を使って環境を構築する. まずは github からレポジトリ [3] をクローンする. ターミナルで以下のコマンドを実行する.

```
git clone https://github.com/rudolfkalman/ros2.git
cd ros2
```

上記のコマンドを実行すれば環境やサンプルプログラムがまとまったディレクトリができる. 次に Docker をインストールする. 今回は Docker Compose というものを使うのでそれも一緒にインストールする必要がある. Docker Desktop(windows) の場合は多分 Docker Compose もまとまってるらしい.
https://www.docker.com/ja-jp/[4] このリンクからインストーラがダウンロードできる. インストールができたらターミナルで以下をうちインストールできているか確認をする.

```
sudo docker # windowsの場合はsudoがないのでdockerだけでよい
```

以下 windows の場合は sudo をなくしたものがコマンドとなる. 実行して docker コマンドがないと言われたらインストールが失敗しているか単にパスが通っていない. Usage が出れば

インストールは完了している. Usage にコマンドの使い方が乗っているがよくわからない場合は Docker の公式サイトや個人ブログなどでコマンドの使い方やコンテナなどの説明を見てみると良い.
次にコンテナ (仮想環境のパソコンみたいなもの) を起動する.

```
pwd
#ここで~~/ros2みたいに最後がros2になってなければクローンしたディレクトリに移動
sudo docker compose up -d
#エラーがでなければイメージのダウンロードとビルドが行われ,最終的に以下のような
    出力になるはず
[+] Running 3/3
 ✓ Container ros2-jazzy              Started           0.2s
 ✓ Container microros-serial-agent   Started           0.2s
 ✓ Container microros-udp-agent      Started
```

成功しなければエラーが出るはずなので調べるなり,ai に聞くなりして解決すること.
次に下記のコマンドを実行する. 成功していれば以下のような出力が得られる.

```
sudo docker ps
#成功していれば以下の出力が出る
CONTAINER ID   IMAGE                                  COMMAND
    CREATED          STATUS          PORTS     NAMES
809e2785a1c1   microros/micro-ros-agent:kilted   "/bin/sh /micro-ros_…"   6
    minutes ago   Up 6 minutes              microros-udp-agent
efb6532539c8   microros/micro-ros-agent:kilted   "/bin/sh /micro-ros_…"   6
    minutes ago   Up 6 minutes              microros-serial-agent
f0a7fef9a34a   ros-jazzy:latest                  "/ros_entrypoint.sh …"   6
    minutes ago   Up 6 minutes              ros2-jazzy
```

次に, コンテナに入る. 通常コードを書くのは各々の環境 (windows なら windows 上の開発環境) で, その書いたコードを実行させるのはコンテナの中である.

```
sudo docker exec -it ros2-jazzy /bin/bash
#以下の出力が得られればコンテナに入ることができた.
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ThinkPad:~/project$
```

linux なら scripts/start_all.sh というスクリプトがあるのでそれを実行すればコンテナ起動とコンテナに入る動作が実行される. ros2 コンテナに入りたければ scpripts/entry_ros.sh があるのでそれを実行すればコンテナに簡単に入れる.

2.3 ROS2を使う

ここまでくればros2が使用できる. 使用できるか確認するためにテストプログラムを実行する. まず, データの送受信ができるかを確認する. 送信用のプログラムと受信用のプログラムを実行する必要があるのでターミナルを2つ開き, そのどちらでもros2-jazzyコンテナに入る必要がある. 以下のコマンドをそれぞれ実行する.

```
ros2 run demo_node_cpp talker # データ送信プログラム
#以下は別ターミナルで実行
ros2 run demo_node_cpp listener # データ受信プログラム
```

実行するとそれぞれのターミナルで以下のような出力が得られる.
talker:

```
[INFO] [1765719701.941659861] [talker]: Publishing: 'Hello World: 1'
[INFO] [1765719702.941704577] [talker]: Publishing: 'Hello World: 2'
[INFO] [1765719703.941698784] [talker]: Publishing: 'Hello World: 3'
[INFO] [1765719704.941699958] [talker]: Publishing: 'Hello World: 4'
[INFO] [1765719705.941701814] [talker]: Publishing: 'Hello World: 5'
[INFO] [1765719706.941681656] [talker]: Publishing: 'Hello World: 6'
[INFO] [1765719707.941702407] [talker]: Publishing: 'Hello World: 7'
[INFO] [1765719708.941700724] [talker]: Publishing: 'Hello World: 8'
[INFO] [1765719709.941704403] [talker]: Publishing: 'Hello World: 9'
[INFO] [1765719710.941699826] [talker]: Publishing: 'Hello World: 10'
```

listener:

```
[INFO] [1765719701.942007096] [listener]: I heard: [Hello World: 1]
[INFO] [1765719702.942022689] [listener]: I heard: [Hello World: 2]
[INFO] [1765719703.941951667] [listener]: I heard: [Hello World: 3]
[INFO] [1765719704.942079597] [listener]: I heard: [Hello World: 4]
[INFO] [1765719705.942024256] [listener]: I heard: [Hello World: 5]
[INFO] [1765719706.942051098] [listener]: I heard: [Hello World: 6]
[INFO] [1765719707.942059837] [listener]: I heard: [Hello World: 7]
[INFO] [1765719708.942085740] [listener]: I heard: [Hello World: 8]
[INFO] [1765719709.942066932] [listener]: I heard: [Hello World: 9]
[INFO] [1765719710.942149233] [listener]: I heard: [Hello World: 10]
```

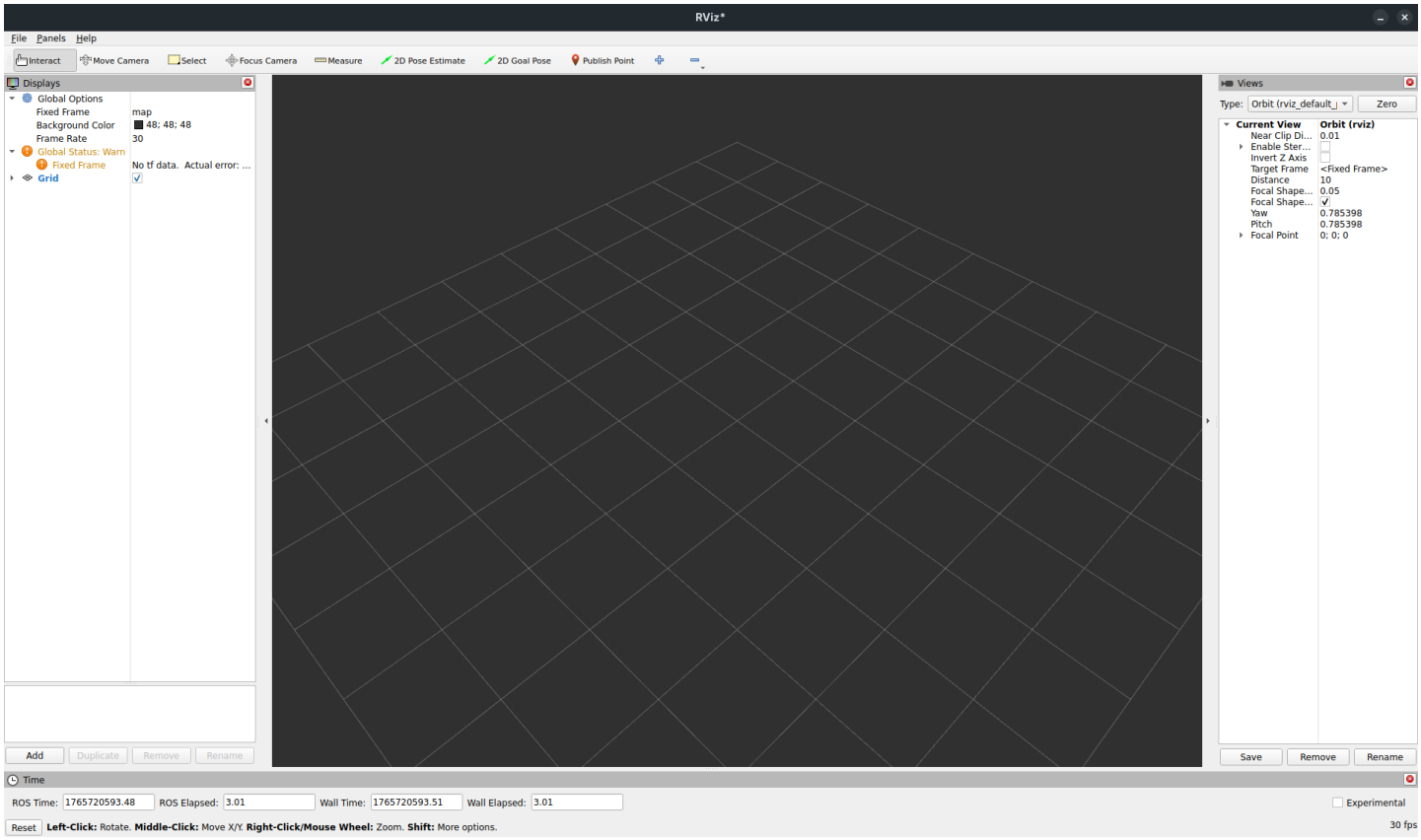
上記の出力が得られればROS2が動くことがわかる. 次にROS2のツールであるrviz2を使用する. linuxはそのままできるはずだが,windowsはVcXsrvというものを入れなければならない. 以下のリンクからダウンロードできる.
https://sourceforge.net/projects/vcxsrv/[5] インストーラでの選択はデフォルトで良かったはず.

2.4 rviz2

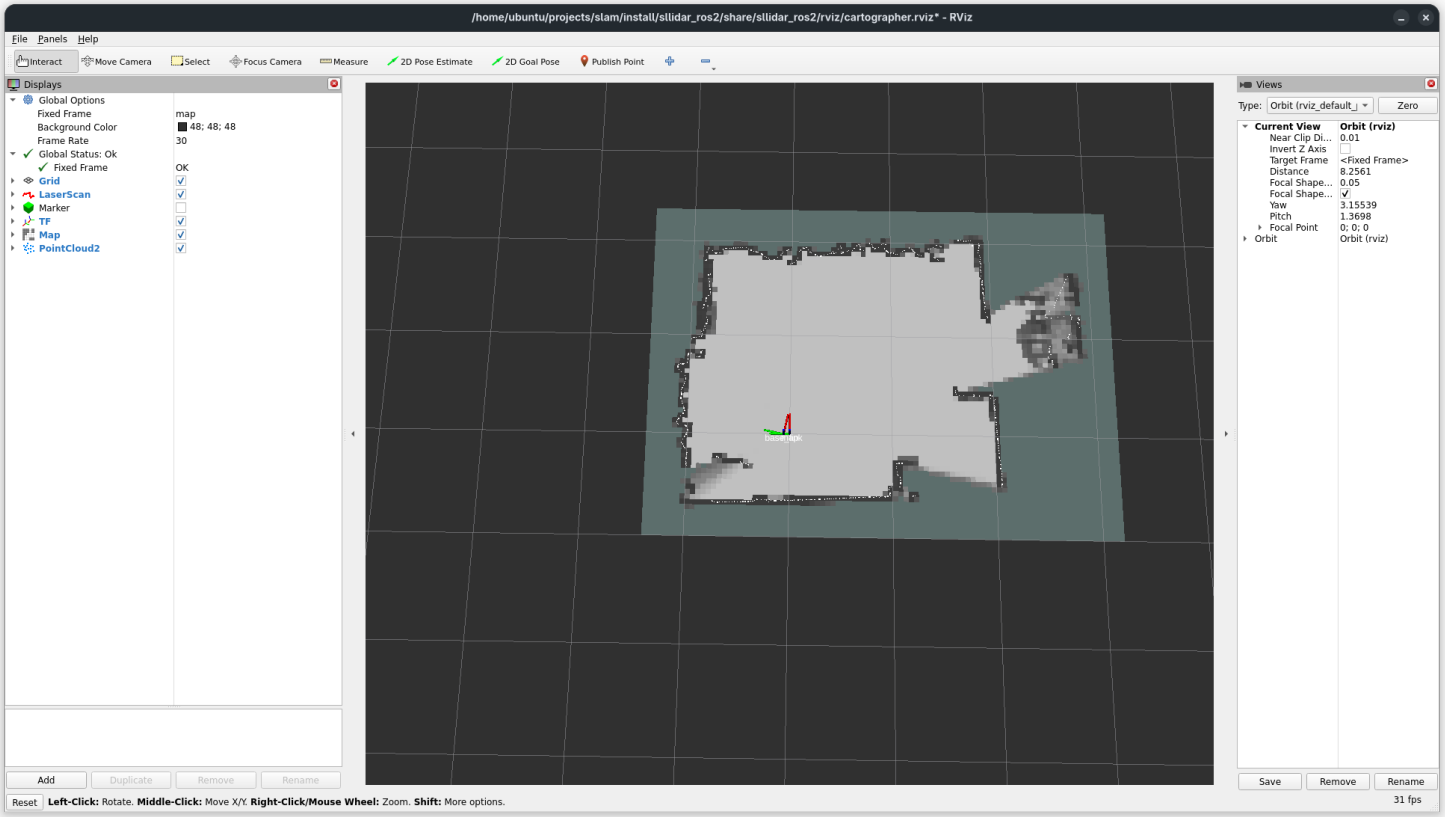
rviz2は下記コマンドで実行できる.

```
rviz2
```

起動したら以下のような画面が立ち上がる.



これが rviz2 の window である. ここではセンサから作ったマップを表示したりカメラ映像を表示したり, ロボットの状態を表示することができる. rviz2 を使用してカメラ映像を表示するのは実機では多分遅いのでカメラ映像の転送は gpu 圧縮などをして ros2 を介さないように工夫しなければならないかもしれない.
参考画像として rviz2 を用いて slam を行った結果を示す.



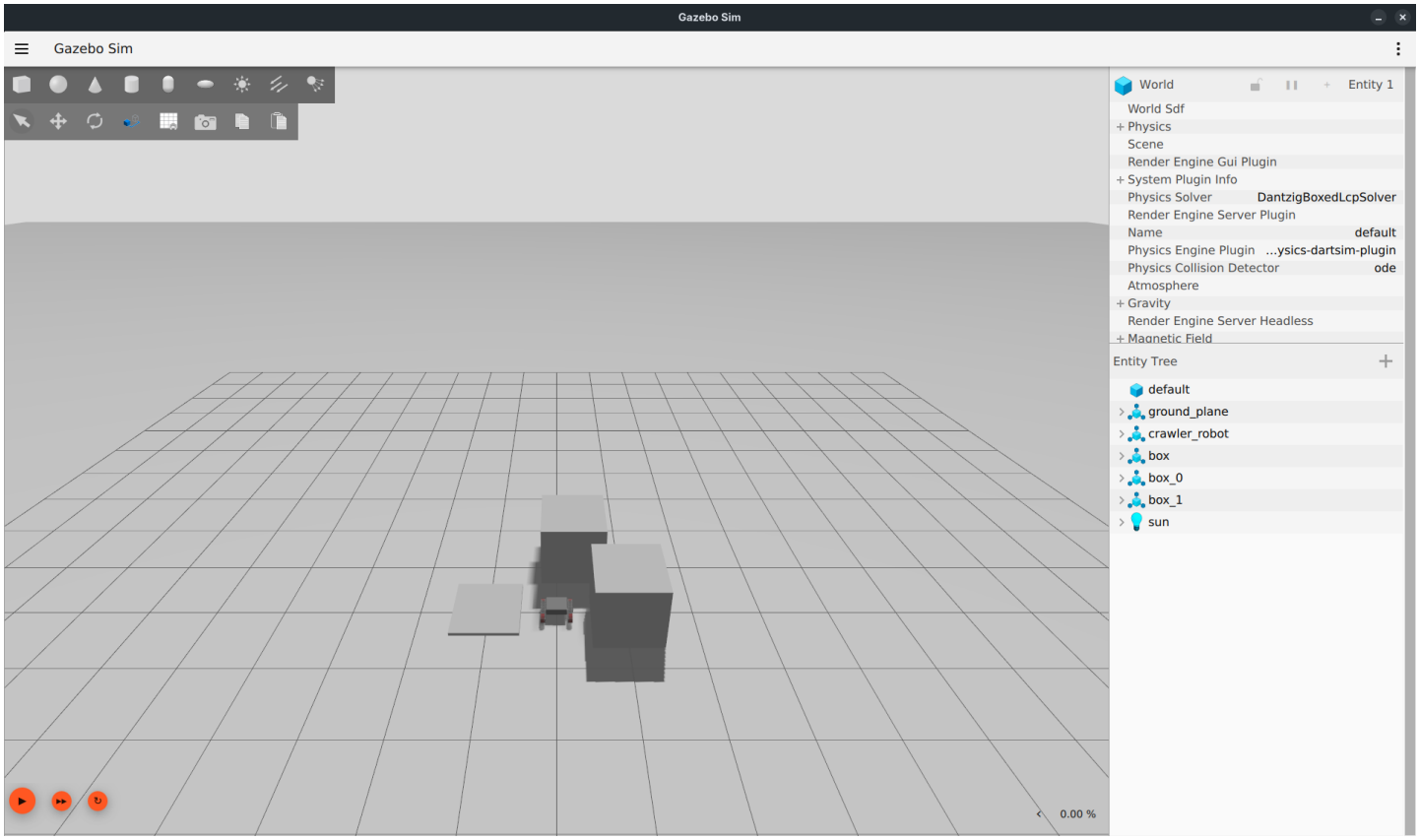
このように ROS2 を使えば簡単に高度なことができる (知識は必要).

2.5 シミュレータ

ROS2 には ROS2 と非常に親和性が高いシミュレータが存在する.gazebo や isaac sim,unity などがある. ここでは gazebo を使用する. gazebo はバージョン変更やディストリビューションが別れたりなどの理由でドキュメントがゴミすぎるが, 軽いので大抵のパソコンで動作するという理由で使用する. このコンテナにははじめから gazebo が入っているので何もする必要はない.
gazebo を起動するには以下のコマンドを実行する.

```
#今回はクローラーロボットを実装したシミュレーションファイルを起動する
cd gz
gz sim crawler_sim.xml
```

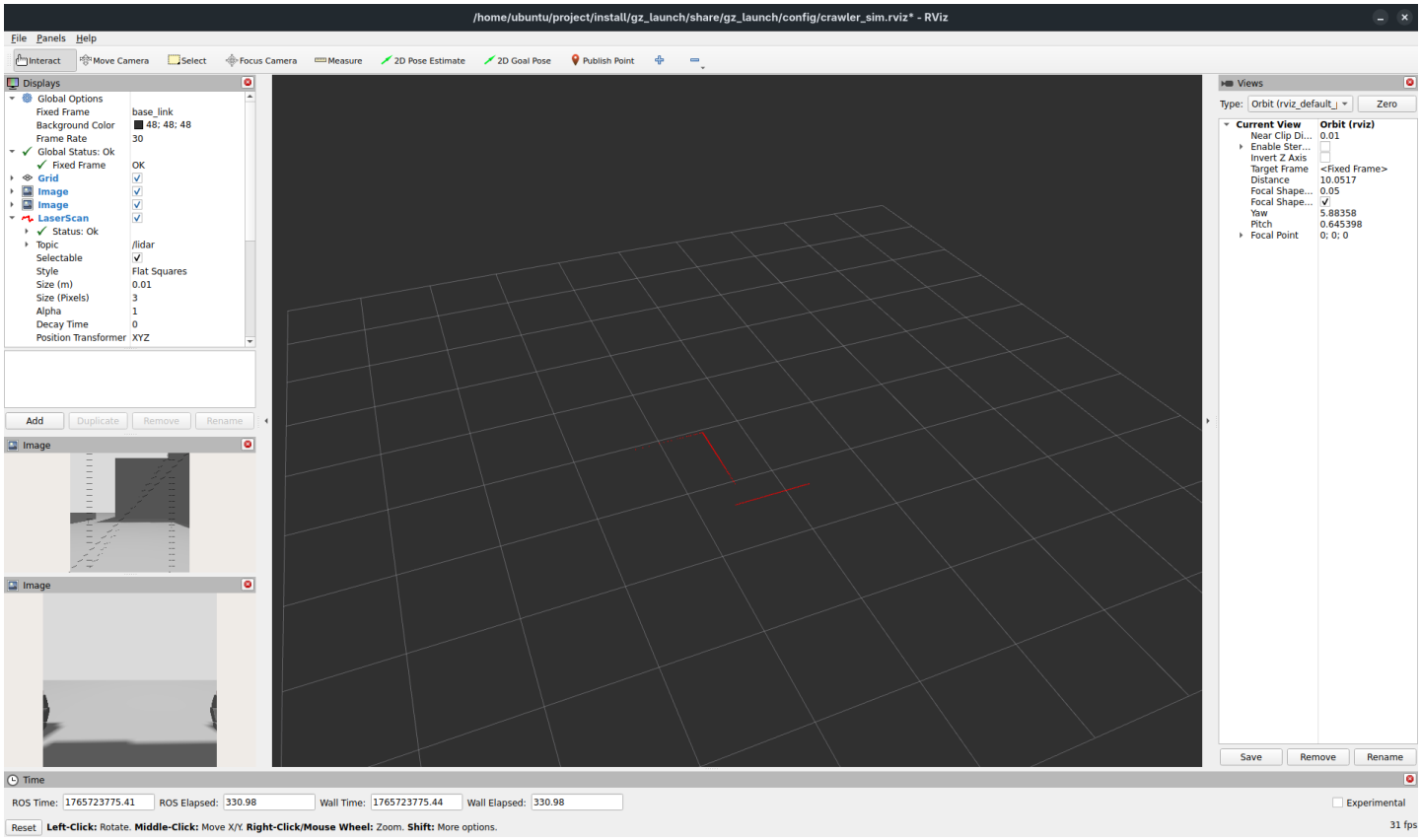
上記を実行すると以下のような画面が表示される.



これがシミュレータの画面. 操作などは gazebo tutorial などと調べて各自でやってみること. このシミュレーションファイルには操作可能なクローラが記述されており, コントローラを接続すれば操作できるようになっている. 操作ができるようにするには以下のコマンドを実行する必要がある.

```
#まずプロジェクトのルートに移動
colcon build # プログラムのビルド
source install/setup.bash # ビルドしたプログラムをインストール
ros2 launch gz_launch gz_launch.launch.py # gazebo用のプログラムをまとめた起動
    スクリプトを実行
```

実行すると rviz2 が起動する.
gazebo の window でシミュレーションのスタートボタンを押すと rviz2 が以下のような画面になる.



ロボットには 2d lidar(z 軸中心に回転し, x,y 平面で極座標形式で点群を返すセンサ) と, カメラ*2(前, 後ろ) がついている. rviz2 画面真ん中に出ている赤い点はロボットを中心として物体までの距離を可視化したもの (ここでロボット中心=グリッド中心). 左下のカメラ (下) がロボット前方のカメラ, 上はロボット後方のカメラになっている.
コントローラを接続すればL スティックで前後,R スティックで回転ができる.

実はこのクローラは完全にクローラの動作を模しているのではなく, クローラにタイヤをいくつもつけ, 接点を擬似的に再現しているので物理的な挙動は実機とは絶対に違う.
そのため, 動作自体を本当にシミュレーションしたければ工夫しなければならない.
しかし, カメラやプログラムの動作の確認をすることに使う分には全く問題はない.

以上のことより,gazebo はセンサやプログラムのシミュレーションが主な目的である. 経路計画などのシミュレーションやクローラ自体の摩擦計算などの数値計算などをしたければ gazebo で行うよりもプログラムで行うほうが良い.

3　ROS2 プログラミングの基礎

ここまで ros2 で使うツールについて説明した.
ここからはどのようにノードを作ったり, 通信の実装をするのかについて説明する. 以下のリンクに ROS2 公式のチュートリアルがある.
https://docs.ros.org/en/jazzy/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Publisher-And-Subscriber.html[6]
このチュートリアルを参考にしながら進めていく

3.1　ノードとは

ROS2 では処理の単位をノードと呼ぶ. ノードは 1 つの役割を持った塊であり, 実際にはプログラムとして実行されることが多い.

3.2　ノードの作成方法

ROS2 ではプログラムの主な役割ごとにパッケージを作成する. 一つのパッケージには関連するノードや設定ファイル,launch ファイルなどをまとめる. なぜパッケージに分けるのかについては管理しやすく、再利用や拡張が容易になるためである.
そのため, チュートリアル用として tutorial というパッケージを作成し, そこでノードの作り方などについて解説する. 主に言語は C++ を用いるが Python でも基本的なことは変わらない. パッケージはプロジェクトの src/下に通常置く. そこに tutorial というパッケージを作成する.
以下のコマンドを実行することで作成することができる.

```
cd src
ros2 pkg create tutorial --build-type ament_cmake --dependencies rclcpp
```

一般的な linux コマンドについては説明を省略する.
https://zenn.dev/kimushun1101/articles/ros2-beginners-ubuntu-terminal[7] このリンクで説明されてるコマンドがわかれば大抵は困らない.
上記のコマンドについて解説する.
ros2 pkg create は ROS2 のパッケージ作成コマンドであり, tutorial という名前のパッケージを生成する.
–build-type は使用するビルドシステムを指定するオプションであり, ament_cmake は C++ 向けのビルド方式である.
–dependencies rclcpp は, このパッケージが rclcpp に依存していることを指定し, ビルドや実行時に必要な設定を自動で行うためのものである.
パッケージを作った後 tutorial に移動し tree で以下のようになるか確認する.

```
~/project/src/tutorial$ tree
.
├── CMakeLists.txt
├── include
│   └── tutorial
├── package.xml
```

```
└── src
```

パッケージが作成できたら次はパッケージの中の src に移動する.
ros2/src/tutorial/src にいれば正解.
ここにプログラムを記述するファイルを作成する. これはコンテナからしなくても良い.
tutorial_pub.cpp と tutorial_sub.cpp というプログラムファイルを作成することとする.
ターミナルなら以下の通り.

```
touch tutorial\_pub.cpp tutorial\_sub.cpp
```

作成できたら tutorial_pub.cpp と tutorial_sub.cpp にコードを書く.
この tutorial_pub.cpp と tutorial_sub.cpp が publish 用のノードと subscribe 用のノードのプログラムだと考えれば良い.

4 Topic 通信の実装

プログラムを書く準備ができたのでここからは Topic 通信を実装してく.

4.1 Publisher の実装

Publisher 用のコードは tutorial_pub.cpp に記述する.
tutorial_pub.cpp のコードは以下とする.

```
#include <chrono>
#include <memory>
#include <string>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;

/* This example creates a subclass of Node and uses a fancy C++11 lambda
* function to shorten the callback syntax, at the expense of making the
* code somewhat more difficult to understand at first glance. */

class TutorialPublisher : public rclcpp::Node
{
public:
  TutorialPublisher()
  : Node("tutorial_publisher"), count_(0)
  {
    publisher_ = this->create_publisher<std_msgs::msg::String>("
   tutorial_message", 10);
    auto timer_callback =
      [this]() -> void {
        auto message = std_msgs::msg::String();
        message.data = "Hello, world! " + std::to_string(this->count_++);
        RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str
   ());
```

```
        this->publisher_->publish(message);
      };
    timer_ = this->create_wall_timer(500ms, timer_callback);
  }

private:
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
  size_t count_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<TutorialPublisher>());
  rclcpp::shutdown();
  return 0;
}
```

上記のプログラムはROS2のチュートリアルのプログラムの一部を変更したものなので, 公式からコピーして変更するほうが良いだろう.
プログラムの流れを説明する. まずプログラムを記述するために必要なヘッダのインクルード.

- `#include "rclcpp/rclcpp.hpp"`
- `#include "std_msgs/msg/string.hpp"`

この2つが特に重要で, これらがROS2のヘッダになっており C++で開発する場合はこのrclcpp.hppが必要.
下のstd_msgs/msg/string.hppは文字でやり取りをするために定義されているクラス.
他のプログラムを作る際はsensor_msgsやgeometry_msgsなどの用途にあったデータの型を使う.
次に,classの定義について.

```
class TutorialPublisher : public rclcpp::Node
```

ここではROS2のNodeクラスを継承したTutorialPublisherを作成している. このNodeクラスを継承することでROS2の設計に準拠したNodeを作成することができる.

```
class TutorialPublisher : public rclcpp::Node
{
public:
  TutorialPublisher()
  : Node("tutorial_publisher"), count_(0)
  {
    publisher_ = this->create_publisher<std_msgs::msg::String>("
   tutorial_message", 10);
    auto timer_callback =
      [this]() -> void {
        auto message = std_msgs::msg::String();
        message.data = "Hello, world! " + std::to_string(this->count_++);
```

```
        RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str
    ());
        this->publisher_->publish(message);
      };
    timer_ = this->create_wall_timer(500ms, timer_callback);
  }

private:
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
  size_t count_;
};
```

ここの部分がクラス TutorialPublisher. このクラスの publisher がデータをトピックである tutorial_message に定期的に送信するようになっている.
もう少し詳しく説明する.TutorialPublisher のインスタンスが作成されたときに, Node(tutorial_publisher という名前) が作成される. またこのとき同時に count_という変数が 0 で初期化される. そして, そのノードが持つ機能が以下に記述されている.

```
  {
    publisher_ = this->create_publisher<std_msgs::msg::String>("
   tutorial_message", 10);
    auto timer_callback =
      [this]() -> void {
        auto message = std_msgs::msg::String();
        message.data = "Hello, world! " + std::to_string(this->count_++);
        RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str
    ());
        this->publisher_->publish(message);
      };
    timer_ = this->create_wall_timer(500ms, timer_callback);
  }
```

Node が持つ publisher は publisher_という名前で,Node クラスの create_publisher というメンバ関数によって作成される.

この際に指定する引数は<>の中に publish するデータの型,() の中には Topic 名と QoS 設定を指定する. ここでは送信キューサイズを 10 としている. QoS の詳細については後述するが, 基本的な通信ではこの指定で問題ない.

大抵 ROS2 でメンバ変数を作る際は publisher_のようにアンダーバーをつける.
このプログラムではタイマーを用いて定期的に publish するようにしているので timer に対するコールバックを作成している.
処理の内容はメッセージを初期化. メッセージのデータ部分に Hello, world!と count_に保持されている数値をコールバックされる毎に加算して加算後の値を文字列にして Hello, world! に付け加えている.
その後 ROS2 の LOG を出力する機能を使用して Publishing: message.data.c_str() を出力している. これは作成した送信する message のデータ (ここでは Hello, world!に count_を加算したもの) を文字列にして並べて出力している.

最後に,publisher_の publish 関数を通して message を topic(tutorial_message) に publish している. その下で timer_が作成されている. この timer_は Node の標準機能である定期的に特定の関数を呼び出す timer である. 作る際の引数は周期 (ここでは 500ms) と呼び出す関数の 2 つ.

```
private:
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
  size_t count_;
};
```

上記の部分ではクラスのメンバ変数である timer_と publisher_と count_が定義されている.
先ほどと同様に<>の中には送信するデータの型を指定する.
今まで未定義だった publisher_や timer_などの変数はここで宣言されている.
先程の TutorialPublisher(コンストラクタ) のところで初期化などをしていた.

```
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<TutorialPublisher>());
  rclcpp::shutdown();
  return 0;
}
```

なにかを実行するには main 関数の中で実行しなければならない. そのため main 関数の中で ROS2 としてのプログラムの初期化,TutorialPublisher の Node の登録, その後プログラムが終わるように shutdown がされている. rclcpp::spin は,Node が終了するまでコールバック処理を実行し続ける関数である.

4.2 Subscriber の実装

publisher と同様に公式プログラムを変更したものを利用する.
tutorial_sub.cpp の内容は以下とする.

```
#include <memory>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

class TutorialSubscriber : public rclcpp::Node
{
public:
  TutorialSubscriber()
  : Node("tutorial_subscriber")
  {
    auto message_callback =
      [this](std_msgs::msg::String::UniquePtr msg) -> void {
        RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg->data.c_str());
      };
    subscription_ =
```

```
      this->create_subscription<std_msgs::msg::String>("tutorial_message", 10,
    message_callback);
  }

private:
  rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<TutorialSubscriber>());
  rclcpp::shutdown();
  return 0;
}
```

publisher の場合と同様ヘッダのインクルードと Node を継承した TutorialSubscriber クラスを作成している. この場合の Node の名前は tutorial_subscriber.

```
private:
  rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
};
```

ここで subscription_を宣言している.publisher のときと同様に<>の中には受け取る型を指定する.

```
  {
    auto message_callback =
      [this](std_msgs::msg::String::UniquePtr msg) -> void {
        RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg->data.c_str());
      };
    subscription_ =
      this->create_subscription<std_msgs::msg::String>("tutorial_message", 10,
    message_callback);
  }
```

上記の部分で message を受け取ったときの処理を書いている.publisher のときは timer に対する callback だったが今回は message に対する callback. message_callback の処理は, 受け取ったメッセージを文字列に変換して LOG に出力する. subscription を初期化する際の引数は Topic 名と Qos 設定,message を受け取った際に実行する関数の 3 つ.

```
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<TutorialSubscriber>());
  rclcpp::shutdown();
  return 0;
}
```

main の部分で初期化,Node を作成, 終了するまで動くようにする.
これで publisher と subscriber のプログラムが完成した.

5 Service通信の実装

本資料では Topic 通信を中心に扱うため, Service 通信の詳細は省略する. 次の ROS2 のビルドと実行に進む.

6 ROS2のビルドと実行

ここではC++のプログラムのビルド方法について説明する.

6.1 cmake

C++のプログラムをコンパイル, ビルドする方法は多数あるが,ROS2 では CMake が採用されている.
CMake を利用すると, プログラムの依存関係を簡単に記述でき, どの環境でも簡単にビルドができるようになる.
CMake を利用するには CMakeLists.txt を記述する必要がある.ROS2 においてパッケージ毎に 1 つの CMakeLists.txt を作成する. パッケージを作成した際の方法を使っていればros2/src/tutorial/に CMakeLists.txt が作成されている.
ROS2 以外でどのように使用するか簡単な例を使用して説明する. これは ROS2 には関係ない. 例として使用するプロジェクト構成は以下の通り.

```
% tree -L2
.
├── CMakeLists.txt
├── build
└── src
    └── test.cpp
```

test.cpp の内容は以下とする.

```
#include <iostream>

int main(){
  int a = 1;
  int b = 2;
  std::cout << a + b << std::endl;
  return 0;
}
```

このような簡単なプログラムなら簡単にコマンド一発でバイナリファイルが作成できるが, 今回は CMake を利用してバイナリファイルを作成する.
CMakeLists.txt の内容は以下の通り.

```
cmake_minimum_required(VERSION 3.8)
project(test)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()
```

```
add_executable(${PROJECT_NAME} src/test.cpp)
```

CMakeLists.txt について説明をする.
`cmake_minimum_required(VERSION 3.8)` は使用する CMake の最低 version の指定. 今回は 3.8. `project(test)` は project の名前. ここでは test.

```
if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()
```

上記の部分ではコンパイラの指定とコンパイルする際に使用するオプションの設定をしている.

```
add\_executable(${PROJECT_NAME} src/test.cpp)
```

この部分の意味は PROJECT_NAME という実行ファイルを src/test.cpp から作成するという宣言. 上記のように test とそのまま project 名を書かない理由は project 名が変わったら自動でその部分も変わって便利だから. この src/test.cpp はこの CMakeLists.txt が存在しているディレクトリからの相対パス.
次に, 以下のコマンドを実行する.

```
cd build
cmake ..
ls
```

すると以下のような結果が得られる.

```
% ls
CMakeCache.txt  CMakeFiles  Makefile  cmake_install.cmake
```

この時点ではバイナリファイルは生成されない. バイナリファイルを実行するために以下のコマンドを実行するとこのような出力が得られる.

```
% make
[ 50%] Building CXX object CMakeFiles/test.dir/src/test.cpp.o
[100%] Linking CXX executable test
[100%] Built target test
```

ここで `ls` をすると以下の出力が得られる.

```
% ls
CMakeCache.txt  CMakeFiles  Makefile  cmake_install.cmake  test
```

出力より,test というファイルが増えていることがわかる. このファイルがバイナリファイルで実行すると以下の結果が得られる.

```
% ./test
3
```

この 3 というのは a + b の値 (1 + 2) の値と一致しており, プログラムからきちんとバイナリファイルが得られたことがわかる.
このように一回 CMakeLists.txt を記述すれば単純なコマンドでプログラムをビルドすることができるので CMake が使用される.

6.2 colcon build

colcon というものは ROS2 のビルドツール.ROS2 では通常この colcon というコマンドを使ってすべてのパッケージを同時にビルドする.
じゃあなぜ CMake の存在について説明したか不思議に思うかもしれないが, 実はこの colcon は各パッケージの CMakeLists.txt を用いて内部的に CMake を実行し, 複数パッケージをまとめてビルドするためのツールである. そのため,CMake について説明した. ここからは colcon を使用して先程作成した Publisher と Subscriber のプログラムをビルドする. ROS2 でパッケージをビルドする際は通常プロジェクトのルートで行う.
プロジェクトルートに移動して以下のコマンドを実行する.

```
colcon build
```

実行するとビルドが開始され, プログラムがおかしければエラーが出てくる. その場合はエラーを読み, プログラムを修正する必要がある.
エラーがなければ以下のようなディレクトリ構成になっているはず.

```
% tree -L 1
.
├── Dockerfile
├── README.md
├── build
├── docker-compose.yml
├── gz
├── install
├── log
├── scripts
└── src
```

このように build, install, log というディレクトリが生成される. エラーが発生した場合でも build や log ディレクトリは生成されることが多い. log には build 時の error などの log が入っている.build に cmake などで作られる中間ファイルが配置される.install には生成後のバイナリや python のスクリプトファイルが入っている.

6.3 setup.bash の役割

build が成功しても先程の状態のままではまだ作成した tutorial というパッケージのプログラムは動かせない. 動かせるようにするには setup.bash を source する必要がある.
以下のコマンドを実行することで, 作成したパッケージを ROS2 から実行できるようになる.

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

一応,gazebo の説明のところでも出てきていたがここで説明する.
そもそも source コマンドとは:
スクリプトや設定ファイルを現在のシェルで実行するためのコマンド.
source を使うと, 新しいシェル (サブシェルではない) を生成せずに, 現在のシェル環境でコマンドが実行される.
そのため, 変数の定義や設定変更が即座に現在のシェルに反映される.
以上の説明で気づいたかもしれないが,build しただけではパスなどが環境変数に自動で追加

されるわけではないのでプログラムを実行できない (バイナリを直接実行すれば一応動くが). そのため,setup.bash は作成したパッケージの実行ファイルのパスや Python モジュールのパスなどを自動で環境変数に追加するためのもの. 具体的には setup.bash で,ros2 run で実行可能なノードのパスや, 共有ライブラリ,Python パッケージのパスなどが環境変数に追加される.
以上のことより build をしたら毎回 `source install/setup.bash` を実行して使えるようにしなければならない. これは実行したシェルだけに反映されるので複数シェルを開いている場合すべてのシェルで最新のものを使いたければすべてで `source install/setup.bash` をしなければならない.

6.4 作成したノードを動かす

ここまでで ROS2 から作成したプログラムを実行できるようになったので, 実際にノードを動かしてみる. まず Publisher と Subscriber を始め動かしてみたときのようにターミナルを 2 つ開き, 同じコンテナに入る.
そして以下のコマンドをそれぞれ別のターミナルから実行する.

```
ros2 run tutorial talker
ros2 run tutorial listener
```

talker は Publisher ノード,listener は Subscriber ノードである.
実行したら, それぞれ以下のような出力が得られる.
talker:

```
[INFO] [1765815087.515010013] [tutorial_publisher]: Publishing: 'Hello, world!
    0'
[INFO] [1765815088.014763818] [tutorial_publisher]: Publishing: 'Hello, world!
    1'
[INFO] [1765815088.514730152] [tutorial_publisher]: Publishing: 'Hello, world!
    2'
[INFO] [1765815089.015141308] [tutorial_publisher]: Publishing: 'Hello, world!
    3'
[INFO] [1765815089.515356349] [tutorial_publisher]: Publishing: 'Hello, world!
    4'
[INFO] [1765815090.015015450] [tutorial_publisher]: Publishing: 'Hello, world!
    5'
[INFO] [1765815090.515059799] [tutorial_publisher]: Publishing: 'Hello, world!
    6'
[INFO] [1765815091.014742787] [tutorial_publisher]: Publishing: 'Hello, world!
    7'
[INFO] [1765815091.515010979] [tutorial_publisher]: Publishing: 'Hello, world!
    8'
[INFO] [1765815092.015020198] [tutorial_publisher]: Publishing: 'Hello, world!
    9'
[INFO] [1765815092.515100727] [tutorial_publisher]: Publishing: 'Hello, world!
    10'
```

listener:

```
[INFO] [1765815087.515696138] [tutorial_subscriber]: I heard: 'Hello, world! 0
    '
```

```
[INFO] [1765815088.015312564] [tutorial_subscriber]: I heard: 'Hello, world! 1
    '
[INFO] [1765815088.515312213] [tutorial_subscriber]: I heard: 'Hello, world! 2
    '
[INFO] [1765815089.015680695] [tutorial_subscriber]: I heard: 'Hello, world! 3
    '
[INFO] [1765815089.515957407] [tutorial_subscriber]: I heard: 'Hello, world! 4
    '
[INFO] [1765815090.015484088] [tutorial_subscriber]: I heard: 'Hello, world! 5
    '
[INFO] [1765815090.515496589] [tutorial_subscriber]: I heard: 'Hello, world! 6
    '
[INFO] [1765815091.015052395] [tutorial_subscriber]: I heard: 'Hello, world! 7
    '
[INFO] [1765815091.515617554] [tutorial_subscriber]: I heard: 'Hello, world! 8
    '
[INFO] [1765815092.015587452] [tutorial_subscriber]: I heard: 'Hello, world! 9
    '
[INFO] [1765815092.515867658] [tutorial_subscriber]: I heard: 'Hello, world!
   10'
```

このような出力が得られればパッケージの作成とノードの作成に成功している.
詳しく見るために以下のコマンドを talker と listener を実行した状態で別ターミナルから実行すると以下の出力が得られる.

```
$ ros2 topic list
/parameter_events
/rosout
/tutorial_message
```

`ros2 topic list` で現在発行されている topic の一覧を見ることができる. /parameter_events と /rosout は通常発行されているトピックで今回作成したトピックは tutorial_message であり, 出力からきちんと発行できていることがわかる.
通信はこの tutorial_message というトピックを通して talker と listener が行っていることもわかるだろう.
以下のコマンドを実行すればわかりやすい.

```
$ ros2 node list
/tutorial_publisher
/tutorial_subscriber
```

`ros2 node list` で現在動いている ROS2 の Node がわかる. ここでは今回作成した,tutorial_publisher と tutorial_subscriber が動いている.
今は publish されている内容を listener で見ているが, コマンドから見ることもできる.

```
$ ros2 topic echo /tutorial_message
data: Hello, world! 19
---
data: Hello, world! 20
---
data: Hello, world! 21
---
```

```
data: Hello, world! 22
---
data: Hello, world! 23
---
data: Hello, world! 24
---
data: Hello, world! 25
---
data: Hello, world! 26
---
data: Hello, world! 27
---
```

ros2 topic echo トピック名とすることで特定のトピックに流れているデータの内容を見ることができる. また, そのデータの型も確認することができる.

```
$ ros2 topic info /tutorial_message
Type: std_msgs/msg/String
Publisher count: 1
Subscription count: 1
```

上記の出力からわかることは以下の通り.

- データの型
- そのトピックに publish しているノードの数
- そのトピックを subscribe しているノードの数

以下のコマンドを実行すれば std_msgs/msg/String などの型の情報も見ることができる.

```
ros2 interface list #メッセージの型などの一覧が見れる
$ ros2 interface show std_msgs/msg/String
# This was originally provided as an example message.
# It is deprecated as of Foxy
# It is recommended to create your own semantically meaningful message.
# However if you would like to continue using this please use the equivalent
   in example_msgs.

string data
```

ros2 interface show std_msgs/msg/String の出力を見ると, この型は string 型の data というものを持っていることがわかる.

7 Launch ファイル

ここまででターミナルを複数開いて実行することは面倒くさかったと思う. ROS2 では Launch ファイルという起動スクリプトを書くことができる. ここでは talker と listener を同時に起動するスクリプトを記述する. 通常 launch ファイルはそのパッケージにまとめるので tutorial パッケージの中に launch ディレクトリを作成し, その中に tutorial.launch.py を作成する. 作成後の tutorial パッケージのディレクトリ構造は以下の通り.

```
% tree -L 2
.
├── CMakeLists.txt
├── include
│   └── tutorial
├── launch
│   └── tutorial.launch.py
├── package.xml
└── src
    ├── tutorial_pub.cpp
    └── tutorial_sub.cpp
```

この launch/tutorial.launch.py に記述する.
以下の内容を記述する.

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='tutorial',
            executable='talker',
            name='tutorial_publisher'
        ),
        Node(
            package='tutorial',
            executable='listener',
            name='tutorial_subscriber'
        )
    ])
```

次に,CMakeLists.txt に listener の install の記述の下に追加する.

```
install(
  DIRECTORY launch
  DESTINATION share/${PROJECT_NAME}/
)
```

CMakeLists.txt に追加することにより,launch ディレクトリの内容が正しい場所にインストールされるようになる. これは ros2 launch が install/share 以下に配置された launch ファイルを参照するためである.
次に,launch ファイルの説明を行う. LaunchDescription は, 起動時に実行する内容をまとめるためのクラスである.
Node は,ROS2 のノードを起動するためのアクションである.
`def generate_launch_description()`: この関数は launchfile では必ず呼び出される. ここで返した LaunchDescription の内容が実行される.

```
Node(
    package='tutorial',
    executable='talker',
    name='tutorial_publisher'
)
```

package : 起動するノードが含まれるパッケージ名
executable : ros2 run tutorial talker の talker に相当
name : 実行時のノード名（省略するとプログラム内の名前が使われる）
listener 側も同様
　これらを記述した後に, 再度ビルドして source をする.
その後以下のコマンドを他に何もノードが動いていない状態で実行すると以下のようになる.

```
ros2 launch tutorial tutorial.launch.py
        02:03 [5/5]
[INFO] [launch]: All log files can be found below /home/ubuntu/.ros/log
   /2025-12-15-17-03-05-331129-ThinkPad-1584
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [talker-1]: process started with pid [1587]
[INFO] [listener-2]: process started with pid [1588]
[talker-1] [INFO] [1765818185.947110929] [tutorial_publisher]: Publishing: '
   Hello, world! 0'
[listener-2] [INFO] [1765818185.947690405] [tutorial_subscriber]: I heard: '
   Hello, world! 0'
[talker-1] [INFO] [1765818186.447395077] [tutorial_publisher]: Publishing: '
   Hello, world! 1'
[listener-2] [INFO] [1765818186.447822437] [tutorial_subscriber]: I heard: '
   Hello, world! 1'
[talker-1] [INFO] [1765818186.947141793] [tutorial_publisher]: Publishing: '
   Hello, world! 2'
[listener-2] [INFO] [1765818186.947760310] [tutorial_subscriber]: I heard: '
   Hello, world! 2'
[talker-1] [INFO] [1765818187.447433554] [tutorial_publisher]: Publishing: '
   Hello, world! 3'
[listener-2] [INFO] [1765818187.448010376] [tutorial_subscriber]: I heard: '
   Hello, world! 3'
[talker-1] [INFO] [1765818187.947437847] [tutorial_publisher]: Publishing: '
   Hello, world! 4'
[listener-2] [INFO] [1765818187.947918497] [tutorial_subscriber]: I heard: '
   Hello, world! 4'
[talker-1] [INFO] [1765818188.447254686] [tutorial_publisher]: Publishing: '
   Hello, world! 5'
[listener-2] [INFO] [1765818188.447782968] [tutorial_subscriber]: I heard: '
   Hello, world! 5'
[talker-1] [INFO] [1765818188.947125164] [tutorial_publisher]: Publishing: '
   Hello, world! 6'
[listener-2] [INFO] [1765818188.947471229] [tutorial_subscriber]: I heard: '
   Hello, world! 6'
[talker-1] [INFO] [1765818189.447667099] [tutorial_publisher]: Publishing: '
   Hello, world! 7'
[listener-2] [INFO] [1765818189.448107101] [tutorial_subscriber]: I heard: '
   Hello, world! 7'
```

何が起きているのか.
`ros2 launch tutorial tutorial.launch.py` を実行することにより talker と listener の 2 つが同時に起動した.
この 2 つは LOG を出力するので発信した LOG と受け取った LOG が出力され, このような

出力になる. つまり, きちんと launch file で複数 Node を起動できたことがわかる.
ちなみに,[talker-1], [listener-2] のような表記は,launch によって起動された各ノードの識別子である.
このようにして複数のプログラムを同時に起動したりする.

8 Parameter

ROS2 では Parameter(パラメータ) と呼ばれる仕組みを用いて, ノードの動作を実行時に変更することができる.
Parameter は数値や文字列などの設定値をノードに与えるためのものであり, プログラムを書き換えずに挙動を変更できる点が特徴である.
例えば, ノードの実行周期や使用するトピック名, フレーム名などを Parameter として扱うことが多い. これにより, 同じノードを異なる設定で再利用することが可能になる.
ROS2 では Parameter は各ノードに紐付いて管理されており, ノード起動後に現在設定されている Parameter を確認したり, 値を取得したりすることができる.
以下のようなコマンドを用いることで,Parameter の一覧や内容を確認できる.

```
ros2 param list
ros2 param get ノード名 パラメータ名
```

また,Parameter は launch ファイルからまとめて指定することもできる. この仕組みを利用することで, 起動時にノードの設定を一括で変更でき, 複数ノードを扱う場合でも管理が容易になる.
特に次章で説明する tf では, 座標系の名前 (frame_id) などを Parameter として扱うことが多い. そのため,Parameter は ROS2 を用いたロボット開発において重要な概念の一つである.

9 tf (座標系管理)

9.1 tf(Transform) とは

ROS2 において複数の座標系の関係を管理するための仕組みである. ロボットでは, 本体やセンサ, 地図などがそれぞれ異なる座標系を持っており, それらの関係を明確に定義しなければ正しい計算ができない. tf は, これらの座標系間の位置と姿勢の関係を統一的に扱うために用いられる.

そもそも座標系とは, 空間内の位置を数値として表現するための基準である. 三次元空間では, 原点と 3 本の直交する軸 (x, y, z) を定義することで座標系が定まる.

すべての物体 (剛体やリンク) には, その物体に固定された座標系を定義することができる. このような座標系は物体座標系と呼ばれ, 物体が移動したり回転したりすると, その座標系も物体とともに移動・回転する.

一方で, 空間全体の基準となる座標系をグローバル座標系 (ワールド座標系) と呼ぶ. グロー

バル座標系は通常固定されており, 時間が経過しても変化しない.

物体の姿勢とは, グローバル座標系に対して物体座標系がどの位置にあり, どの向きを向いているかを表したものである. この姿勢は, 位置 (平行移動) と向き (回転) によって表現される.

ROS2 では, これらの座標系を frame と呼び, frame 同士の位置・姿勢の関係を Transform として管理する. tf では, この Transform を時刻情報とともに保持し, 任意の時刻における座標変換を取得できるようになっている.

tf を用いることで,
「地図座標系から見たロボットの位置」や
「ロボット座標系から見たセンサの位置」
といった変換を簡単に計算することができる.

このように tf は, ロボットにおける座標系の関係を明確にし, 複数のセンサやアルゴリズムを正しく連携させるために不可欠な仕組みである.

ROS2 では姿勢の表現にクォータニオン (四元数) と呼ばれるものが使用される.
クォータニオンの前に虚数 i の性質について考える.

9.2 回転の記述

複素数は横軸に実部, 縦軸に虚部を取ることにより平面 (複素平面) として考えることができる.
複素平面上で $z = 1 + 0i$ を用いて複素数の積について考える.

$$
\begin{aligned}
z \cdot i &= i \\
i \cdot i &= -1 \\
-1 \cdot i &= -i \\
-i \cdot i &= 1
\end{aligned}
$$

以上の結果から z を位置ベクトルと見れば, 複素数は複素平面上で複素数の積は回転に対応していそうだとわかる. 一般の複素数で考える.$z_1 = r_1(a_1 + b_1 i), z_2 = r_2(a_2 + b_2 i)$ を用いる. これらの積は以下の通り.

$$
\begin{aligned}
z_1 z_2 &= r_1(a_1 + b_1 i)\, r_2(a_2 + b_2 i) \\
&= r_1 r_2 (a_1 + b_1 i)(a_2 + b_2 i) \\
&= r_1 r_2 \left(a_1 a_2 + a_1 b_2 i + b_1 a_2 i + b_1 b_2 i^2\right) \\
&= r_1 r_2 \left((a_1 a_2 - b_1 b_2) + (a_1 b_2 + b_1 a_2) i\right) \qquad (1)
\end{aligned}
$$

ここで,

$$
\begin{aligned}
z_1 &= r_1(\cos\theta_1 + i \sin\theta_1) \\
z_2 &= r_2(\cos\theta_2 + i \sin\theta_2)
\end{aligned}
$$

であるので, 上記の式を用いて (1) を以下のように記述できる.

$$\begin{aligned} z_1 z_2 &= r_1 r_2 \left((a_1 a_2 - b_1 b_2) + i(a_1 b_2 + b_1 a_2)\right) \\ &= r_1 r_2 \left((\cos\theta_1 \cos\theta_2 - \sin\theta_1 \sin\theta_2) + i(\cos\theta_1 \sin\theta_2 + \sin\theta_1 \cos\theta_2)\right) \\ &= r_1 r_2 \left((\cos(\theta_1 + \theta_2) + i(\sin(\theta_1 + \theta_2)\right) \end{aligned}$$

ここで $r_2 = 1$ であるならば,

$$z_1 z_2 = r_1(\cos(\theta_1 + \theta_2) + i\sin(\theta_1 + \theta_2)) \tag{2}$$

となり,z_1 の偏角 θ_1 に z_2 の偏角 θ_2 が加算されている. 故に, 複素平面上で, ある複素数 z とノルムが 1 である複素数 z_{rot} の積は z を回転させることがわかる.
また, 回転を記述する方法は, 他にもある. $\mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix}$ を原点を中心にして x,y 平面上を θ' 回転させる方法は, 行列を用いて以下のように表される.

$$\begin{aligned} \mathbf{x}_{\text{rot}} &= \begin{pmatrix} \cos\theta' & -\sin\theta' \\ \sin\theta' & \cos\theta' \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \\ &= \begin{pmatrix} \cos\theta' & -\sin\theta' \\ \sin\theta' & \cos\theta' \end{pmatrix} \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix} \end{aligned} \tag{3}$$

通常は (3) 式を用いて記述される.(3) 式を展開すると以下のようになる.

$$\begin{aligned} \begin{pmatrix} \cos\theta' & -\sin\theta' \\ \sin\theta' & \cos\theta' \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} &= \begin{pmatrix} \cos\theta\cos\theta' - \sin\theta\sin\theta' \\ \cos\theta\sin\theta' + \sin\theta\cos\theta' \end{pmatrix} \\ &= \begin{pmatrix} \cos(\theta + \theta') \\ \sin(\theta + \theta') \end{pmatrix} \end{aligned} \tag{4}$$

(2) 式と (4) 式より, 回転は加法定理を用いて表されることがわかる.

9.3 近似式

次に, ある無限回微分可能な関数 f(x) について考える.
f(x) について以下のことが成り立つ.

$$f(x) = \sum_{n=0}^{\infty} \frac{f^n(a)}{n!}(x-a)^n$$

上記の展開をテイラー展開と呼ぶ. 特に,$a = 0$ のときのテイラー展開をマクローリン展開と呼ぶ.
これはある無限回微分可能な関数 f(x) は多項式として表されるということ.
この展開は任意の n でやめて良い.
この展開を n=1 でやめた場合 1 次近似,n=2 でやめた場合 2 次近似と呼ぶ. この近似は物理や最適化などでよく出てくるので知っておくと良い.
$\sin x, \cos x, e^x$のマクローリン展開は以下で表される.

$$\begin{aligned}\sin x &= \frac{\sin(0)}{0!}x^0 + \frac{\sin'(0)}{1!}x^1 + \frac{\sin''(0)}{2!}x^2 + \frac{\sin'''(0)}{3!}x^3 + \dots \\ &= 0 + x + 0 - \frac{x^3}{3!} + \dots \\ &= x - \frac{x^3}{3!} + \dots \\ &= \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n+1}}{(2n+1)!} \end{aligned} \tag{5}$$

$$\begin{aligned}\cos x &= \frac{\cos(0)}{0!}x^0 + \frac{\cos'(0)}{1!}x^1 + \frac{\cos''(0)}{2!}x^2 + \frac{\cos'''(0)}{3!}x^3 + \dots \\ &= 1 + 0 - \frac{x^2}{2!} + 0 + \dots \\ &= 1 - \frac{x^2}{2!} + \dots \\ &= \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n}}{(2n)!} \end{aligned} \tag{6}$$

$$\begin{aligned}e^x &= \frac{e^0}{0!}x^0 + \frac{e^0}{1!}x^1 + \frac{e^0}{2!}x^2 + \frac{e^0}{3!}x^3 + \dots \\ &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \\ &= \sum_{n=0}^{\infty}\frac{x^n}{n!} \end{aligned} \tag{7}$$

上記の 3 つの式を眺めると e^x のマクローリン展開 ((7) 式) は $\sin x$ のマクローリン展開 ((5) 式) と $\cos x$ のマクローリン展開 ((6) 式) の和に似ている. 符号が入れ替わったりしているのでこの符号をどうにかして偶数項と奇数項を足して e^xのようなものを表したい.
ここで$i^2 = -1$を利用すると, 偶数乗ではi2n = (-1)^n, 奇数乗では $i2n+1 = (-1)^{n+1}$ となり,

$$e^{ix} = \sum_{n=0}^{\infty}\frac{(ix)^n}{n!} = \sum_{n=0}^{\infty}\frac{(ix)^{2n}}{(2n)!} + \sum_{n=0}^{\infty}\frac{(ix)^{2n+1}}{(2n+1)!} \tag{8}$$

が成立し, 偶数項が $\cos x$奇数項が$i\sin x$に対応することがわかる.

以上より,(8) 式は以下のように表せる.

$$e^{ix} = \cos x + i\sin x \tag{9}$$

よく (9) の表記は出てくるのでこれも回転であると知っておくと良いだろう.

9.4　3 次元空間での回転

今までの回転はすべて x,y 平面やそれに似た複素平面だったが現実世界では次元が 1 上がり 3 次元になる．そのため平面の回転ではなく 3 次元空間上の回転を考える．目的はロボットの姿勢表すこと．どのようにすれば 3 次元空間での姿勢を表せるだろう．平面の回転を x,y,z 空間で見ると x,y 平面上での回転は z 軸周りの回転になっていることがわかる．この回転は z 座標が変わらずに x,y 座標のみ変化する. 状態変数 $\mathbf{x}$ と回転行列 $\mathrm{R}(\psi)$ を用いて表す.

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

$$\begin{aligned} \mathbf{x_{rot}} &= \begin{pmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \\ &= \begin{pmatrix} x\cos\psi - y\sin\psi + 0 \\ x\sin\psi + y\cos\psi + 0 \\ 0 + 0 + z \end{pmatrix} \\ &= \begin{pmatrix} x\cos\psi - y\sin\psi \\ x\sin\psi + y\cos\psi \\ z \end{pmatrix} \end{aligned} \tag{10}$$

(10) 式より ,x,y 成分は回転作用が加わっており ,z 成分はそのままで z 軸周りの回転がうまく表せていることがわかる．このように x 軸周り ,y 軸周り ,z 軸周りに対応する回転行列をそれぞれ, 以下の $\mathrm{R}(\psi)$,$\mathrm{R}(\theta)$,$\mathrm{R}(\phi)$ で定義する.

$$R(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{pmatrix} \tag{11}$$

$$R(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \tag{12}$$

$$R(\psi) = \begin{pmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{13}$$

3 次元空間での姿勢は x 軸周りに ϕ,y 軸周りに θ,z 軸周りに ψ 回転しているという情報の 3 つで表すことができる.
https://youtu.be/pQ24NtnaLl8[8]
このような動画を見るとイメージがしやすいだろう.
姿勢は表すことができたが, 問題がある.x->y->z 軸周りの順番で回転をさせるのと ,y->x->z 軸周りの順番で回転をさせるのでは姿勢が変わる．この表現では姿勢が一意に定まらない.

これの理由は x->y->z の際は $R(\phi)R(\theta)R(\psi)(x)$ によって回転後のベクトルが表されるのに対し, y->x->z の際は $R(\phi)R(\psi)R(\theta)(x)$ によって回転が表されるからである. 行列の積は非可換であるので $R(\theta)R(\psi)$ と $R(\psi)R(\theta)$ は等しくないため. 人によって (回転順によって) 姿勢が異なるというのは致命的である.

9.5 任意軸周りの回転

先程の定義だと回転の順序によって姿勢が変わってしまい, 姿勢を一意に定めることができなかった. しかし, 任意の軸とその軸周りの回転角として姿勢を定義すれば, 軸も角度も人によって変わらず, 姿勢を一意に定めることができる.

任意軸周りの回転は, ロドリゲスの回転公式によって以下のように表される. ここで,$\mathbf{n}$ は回転軸を表す単位ベクトル,θ は回転角である.

$$\mathbf{x}_{\mathrm{rot}} = \mathbf{x}\cos\theta + (\mathbf{n}\times\mathbf{x})\sin\theta + \mathbf{n}(\mathbf{n}\cdot\mathbf{x})(1-\cos\theta) \tag{14}$$

この式は, 回転軸 $\mathbf{n}$ に平行な成分と垂直な成分に分解して考えることで理解できる. 回転軸に平行な成分は回転によって変化せず, 回転軸に垂直な成分のみが角度 θ だけ回転する. ロドリゲスの回転公式は, この幾何的事実をそのまま数式として表現したものである.

9.6 クォータニオン演算

クォータニオン (Quaternion) は, 実数 1 成分と虚数 3 成分からなる数であり, 以下のように定義される.

$$q = w + xi + yj + zk$$

ここで w, x, y, z は実数であり ,i, j, k は虚数単位である. これらは以下の性質を持つ.

$$i^2 = j^2 = k^2 = ijk = -1$$

また, 積の順序に注意すると,

$$ij = k, \quad jk = i, \quad ki = j$$

$$ji = -k, \quad kj = -i, \quad ik = -j$$

となり, クォータニオンの積は可換ではない.

クォータニオンの和とスカラー倍は成分ごとに定義される.

$$q_1 + q_2 = (w_1 + w_2) + (x_1 + x_2)i + (y_1 + y_2)j + (z_1 + z_2)k$$

$$\lambda q = \lambda w + \lambda xi + \lambda yj + \lambda zk$$

一方で, クォータニオンの積は上記の虚数単位の積の規則を用いて計算される.

$$q_1 q_2 = (w_1 + \mathbf{v}_1)(w_2 + \mathbf{v}_2)$$

ここで $\mathbf{v} = xi + yj + zk$ を 3 次元ベクトルとみなすと,

$$q_1 q_2 = (w_1 w_2 - \mathbf{v}_1 \cdot \mathbf{v}_2) + (w_1 \mathbf{v}_2 + w_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2)$$

と書ける.

この形は内積と外積を同時に含んでおり,3 次元空間の回転と非常に相性が良い構造になっている. 実際, 単位ベクトル $\mathbf{n}$ 周りに角度 θ だけ回転するクォータニオンは,

$$q = \cos\frac{\theta}{2} + \mathbf{n}\sin\frac{\theta}{2}$$

と定義される.

回転させたいベクトル $\mathbf{x}$ を純虚クォータニオン

$$p = 0 + \mathbf{x}$$

として表すと, 回転後のベクトル p' は

$$p' = q\, p\, q^{-1}$$

によって与えられる.

この変換を計算すると, 前節で示したロドリゲスの回転公式と一致する. つまり, クォータニオンによる回転は, 任意軸周りの回転を自然に表現するために定義された演算であり, その計算規則自体が回転の性質を内包していることがわかる.

このように, クォータニオンは 3 次元空間の姿勢を一意に表現でき, 回転の合成も積として簡潔に記述できるため, ROS2 を含む多くのロボットシステムで姿勢表現として用いられている.

クォータニオンやロドリゲスの回転公式の導出や解説は以下のリンクがわかりやすい.
https://zenn.dev/mebiusbox/books/132b654aa02124/viewer/2966c7[9]

9.7 ROS2 でのクォータニオンの使用

ROS2 では,3 次元空間における姿勢の表現としてクォータニオンが広く用いられている. これは前節で述べたように, クォータニオンが回転を一意に表現でき, 回転の合成も安定して扱えるためである.

特に, 座標系間の関係を管理する tf では, 各座標変換は「並進」と「回転」の組として表される. このとき, 並進は 3 次元ベクトル, 回転はクォータニオンによって表現される.

例えば, センサ座標系からロボット本体の座標系への変換や, ロボット本体からワールド座標系への変換などは, tf を通してクォータニオンを含む形で管理されている. ユーザはこれらの座標変換を直接計算する必要はなく, tf を利用することで統一的に扱うことができる.

このように, クォータニオンは単なる数学的概念ではなく, ROS2 において実際に姿勢を扱うための基盤となる表現であり, tf をはじめとした多くの機能の内部で利用されている.

10　まとめ

本資料では、ROS2 の導入から基本的なプログラミング、そして座標系管理（tf）に必要な数学的背景について解説した。ROS2 は、ノード間通信を標準化し、既存パッケージの再利用を容易にすることで、ロボット開発の効率を向上させるフレームワークである。今後は、本資料の内容を基に、より複雑な制御や自律移動の実装へと進むことが期待される。

11　参考文献

参考文献

[1] Matlab, “ROS2 のパブリッシャーとサブスクライバーとのデータ交換”, https://jp.mathworks.com/help/ros/ug/exchange-data-with-ros-2-publishers-and-subscribers.html (2025 年 12 月アクセス)

[2] Matlab, “ROS 2 サービスを探索: サービス クライアントとサービス サーバー ガイド”, https://ww2.mathworks.cn/help/ros/gs/ros2-services.html (2025 年 12 月アクセス)

[3] rudolfkalman, “ros2 (GitHub Repository)”, https://github.com/rudolfkalman/ros2.git

[4] Docker Inc., “Docker 公式サイト”, https://www.docker.com/ja-jp/

[5] SourceForge, “VcXsrv Windows X Server”, https://sourceforge.net/projects/vcxsrv/

[6] Open Robotics, “Writing a simple publisher and subscriber (C++)”, https://docs.ros.org/en/jazzy/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Publisher-And-Subscriber.html

[7] kimushun1101, “ROS2 初心者のための Ubuntu ターミナル基本コマンド (Zenn)”, https://zenn.dev/kimushun1101/articles/ros2-beginners-ubuntu-terminal

[8] YouTube, “3D Rotation Visualizer”, https://youtu.be/pQ24NtnaLv8

[9] mebiusbox, “クォータニオンとロドリゲスの回転公式の解説 (Zenn)”, https://zenn.dev/mebiusbox/books/132b654aa02124/viewer/2966c7