ROS2

**ROS Distributions**

A ROS distribution is **a versioned set of ROS packages**. These are akin to Linux distributions (e.g. Ubuntu). The purpose of the ROS distributions is to let developers work against a relatively stable codebase until they are ready to roll everything forward.

*Rolling Distribution*

**ROS 2 Rolling Ridley** is the rolling development distribution of ROS 2. It is described in **REP 2002** and was first introduced in June 2020.

The Rolling distribution of ROS 2 serves two purposes:

1. it is a staging area for future stable distributions of ROS 2, and
2. it is a collection of the most recent development releases.

**ROS packages**

**ROS** organizes the program using **packages**. A package comprises all the files that make up a particular ROS program.

Nodes:

**Nodes** are the process that perform computation. Each ROS node is written using ROS client libraries such as *roscpp* and *rospy*. Each node can send and receive data to other nodes via *topics, services, actions, or parameters*.

Topic:

ROS 2 breaks complex systems down into many modular nodes. **Topics** are a vital element of the ROS graph that act as a bus for nodes to exchange messages. Each message in ROS is transported using named buses called topics. When a node sends a message through a topic, then we can say the *node is publishing a topic*.

Message:

Nodes communicate with each other using messages. Messages are simply a data structure containing the typed field, which can hold a set of data and that can be sent to another node. There are standard primitive types (integer, floating point, Boolean, and so on) and these are supported by ROS messages. We can also build our own message types using these standard types

Services:

Services are another method of communication for nodes in the ROS graph. Services are based on a *call-and-response model*, versus topics’ *publisher-subscriber model*. While topics allow nodes to subscribe to data streams and get continual updates, services only provide data when they are specifically called by a client.In some robot applications, a publish/subscribe model will not be enough if it needs a request/response interaction. The publish/subscribe model is a kind of one-way transport system and when we work with a distributed system, we might need a request/response kind of interaction. ROS Services are used in these case. We can define a service definition that contains two parts; one is for requests and the other is for responses. Using ROS Services, we can write a server node and client node. The server node provides the service under a name, and when the client node sends a request message to this server, it will respond and send the result to the client. The client might need to wait until the server responds. The ROS service interaction is like a remote procedure call

Parameters:

A parameter is a configuration value of a node. You can think of parameters as node settings. A node can store parameters as integers, floats, booleans, strings, and lists. In ROS 2, each node maintains its own parameters. For more background on parameters,

Nodes have parameters to define their default configuration values. You can get and set parameter values from the command line. You can also save the parameter settings to a file to reload them in a future session.

Actions:

**Actions** are one of the communication types in ROS 2 and are intended for long running tasks. They consist of three parts: a **goal**, **feedback**, and a **result**.

**Actions** are built on topics and services. Their functionality is similar to services, except actions can be canceled. They also provide steady feedback, as opposed to services which return a single response.

**Actions** use a client-server model, similar to the publisher-subscriber model (described in the [topics tutorial](https://docs.ros.org/en/galactic/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html)). An “action client” node sends a goal to an “action server” node that acknowledges the goal and returns a stream of feedback and a result.

**Actions** are like services that allow you to execute long running tasks, provide regular feedback, and are cancelable.

A robot system would likely use actions for navigation. An action goal could tell a robot to travel to a position. While the robot navigates to the position, it can send updates along the way (i.e. feedback), and then a final result message once it’s reached its destination.

Recording and playing back data:

**Bags** are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, which can be difficult to collect but is necessary for developing and testing robot algorithms. Bags are very useful features when we work with complex robot mechanisms.

ros2 bag is a command line tool for recording data published on topics in your system. It accumulates the data passed on any number of topics and saves it in a database. You can then replay the data to reproduce the results of your tests and experiments. Recording topics is also a great way to share your work and allow others to recreate it.

ROS stack:

Packages can be organized into ROS *stacks*. Whereas the goal of packages is to create and compile minimal collections of code for easy *reuse*, the goal of stacks is to simplify the process of code *sharing*. Stacks are the primary mechanism in ROS for distributing software. Each stack has an associated version and can declare dependencies on other stacks. These dependencies also declare a version number, which provides greater stability in development.

Stacks collect packages that collectively provide functionality, such as a navigation stack or a manipulation stack.

LIDAR

היא טכנולוגיית מדידת מרחק על ידי הארת המטרה בקרן לייזר, ומדידת הזמן שלוקח לקרן האור לחזור למקלט.

Planning requires a 3D representation of objects

Some example sensing modalities that satisfy this:

● LiDAR

● RaDAR

● Stereo Cameras

● Depth Cameras

● Structure from motion + scale estimator

● Deep learning

● Some combination of the above

Deep Learning

● Cool

● Cutting edge performance

● Requires lots of labeled data

● Generally requires GPU for any

hope of handling live data

Classical

● Boring

● Easy to understand

● Tuning parameters

● Pretty simple

Why Classical 3D Perception?

● Simple, easy to understand

● Easier to implement

● Fewer (no) dependencies

● More robust to adversarial attacks

● Lower computational overhead

(potentially)

LiDAR Preprocessing, A Problem Statement

We want the minimum amount of information needed to

produce the correct results:

● Remove useless data

● Remove problematic/bad data

● Remove redundant data

● Produce a single, consistent input

Range-Based Filtering

Angle-Based Filtering

Downsampling

Fusing Point Clouds

Preprocessing gives you a single, lightweight

representation needed by downstream algorithms

• Remove noisy data from problematic areas:

• Range/angle filters

• Remove redundant data:

• Voxel grids, other downsampling

• Create a single consistent representation:

• Static transforms into common frame

• Fuse into a single point cloud

• Ego-motion is required for high speed use

cases to correct for slewing

The Problem of Ground Filtering

● Primary motivation for object detection is collision

detection

● You can’t hit things on the ground

● We want the minimal sufficient information passed into

our algorithm

● Ignore or filter useless\* ground points

Identifying ground points, a general strategy

1. Look at curvature/normals -> Moosman et al

2. Fit a big fat plane to the scene -> RANSAC

3. Look at rays/columns in depth image ->

Petrovskaya & Thrun, Bogoslavskyi, Tier4, Cho

et al

4. Other approaches (factor graphs, voxels, etc)

(Standard) Ray Ground Filtering

Petrovskaya & Thrun, Bogoslavskyi:

1. Build range image or otherwise bin into angle

slices

2. Know where the ground is local to the sensor

3. Scan through points in a ray with increasing

distance:

a. If it’s pretty flat, ground

b. If there’s a big change in angle, non-ground from here out

(Standard) Ray Ground Filtering

The Problem of Object Detection

Why?

● Input nonground point cloud is cumbersome alone

● Need to discriminate between separate objects

○ e.g. segment nonground point cloud

● Partition point cloud into objects

● Remove some noise

How?

● Fundamentally group point together

○ Somehow

○ Based on some metric

An overview on shape extraction techniques

● Why?

● Different ways to compute bounding

boxes:

○ Rotating calipers

○ Eigenvalues

○ Optimization

● Other shape extraction techniques

Why not use point blobs directly?

Recall sufficient statistics:

● We only want the minimum of information to

produce the right results

● Point blobs are big and unbounded

○ Additional communication and memory overhead

○ Additional computational overhead for collision

checking, other use cases

Shape extraction and the collision detection use case

Object detection is generally to

inform collision detection

The final object representation

should:

● be easy to collision check

● be easy to compute

● reasonably represent the

actual object

Can instead explicitly solve an

optimization problem

● Shen et al

○ Box and partition that best fits L-shape

○ ...but relies on a single scanning LiDAR

● Zhang et al

○ Relaxes assumption of single scanning LiDAR

○ ...but introduces discretization error

Bounding Boxes in Autoware. Auto Use approach from Shen et al, but:

● Compute principal component of blob

● Sort points along this axis

● Extra work, but:

○ No discretization error

○ Works on arbitrary point clouds

Further Approaches

Many other ways to do shape extraction

● Convex hulls

○ Only loosely bounded memory complexity

● Minimum Volume Bounding Ellipsoids

○ e.g. Kumar and Yıldırım

○ Collision detection requires solving an optimization problem

● Super quadratics (i.e. Pascoal et al):

○ Handles concave shapes

○ Requires solving an optimization problem