

MRT Assignment Report

Anushka Verma

July 2023

0.1 move_base

The move_base package provides an implementation of an action (see the actionlib package) that, given a goal in the world, will attempt to reach it with a mobile base. The move_base node links together a global and local planner to accomplish its global navigation task. This package provides the move_base ROS Node which is a major component of the navigation stack. The move_base node also maintains two costmaps, one for the global planner, and one for a local planner that are used to accomplish navigation tasks. The move_base package lets you move a robot to desired positions using the navigation stack. This package provides the move_base ROS Node which is a major component of the navigation stack.

The move_base node provides a ROS interface for configuring, running, and interacting with the navigation stack on a robot.

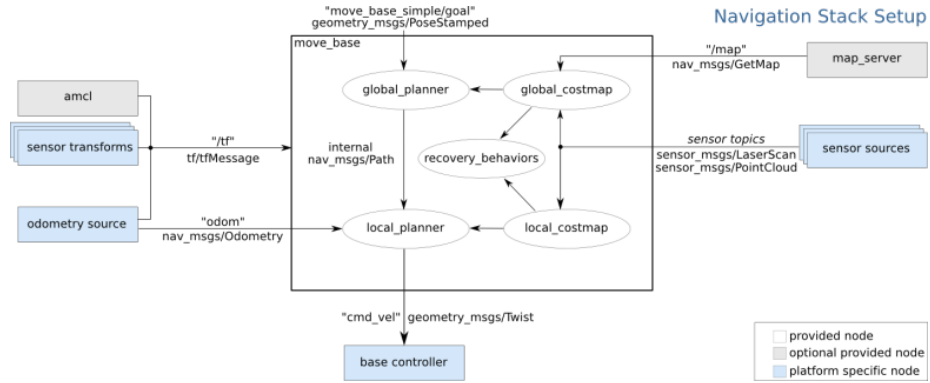


Figure 1: Navigation stack setup

0.1.1 Component APIs

The move_base node contains components that have their own ROS APIs. These components may vary based on the values of the `base_global_planner`, `base_local_planner`, and `recovery_behaviors` respectively.

1. costmap_2d

This package provides an implementation of a 2D costmap that takes in sensor data from the world, builds a 2D or 3D occupancy grid of the data (depending on whether a voxel based implementation is used), and inflates costs in a 2D costmap based on the occupancy grid and a user specified inflation radius.

2. nav_core

This package provides common interfaces for navigation specific robot actions. Currently, this package provides the BaseGlobalPlanner, BaseLocalPlanner, and RecoveryBehavior interfaces

3. base_local_planner

This package provides implementations of the Trajectory Rollout and Dynamic Window approaches to local robot navigation on a plane. Given a plan to follow and a costmap, the controller produces velocity commands to send to a mobile base.

4. navfn

navfn provides a fast interpolated navigation function that can be used to create plans for a mobile base.

5. clear_costmap_recovery

This package provides a recovery behavior for the navigation stack that attempts to clear space by reverting the costmaps used by the navigation stack to the static map outside of a given area.

6. rotate_recovery

This package provides a recovery behavior for the navigation stack that attempts to clear space by performing a 360 degree rotation of the robot.

0.2 Quaternions

Unit quaternions, known as versors, provide a convenient mathematical notation for representing spatial orientations and rotations of elements in three dimensional space. Specifically, they encode information about an axis-angle rotation about an arbitrary axis.

$$\mathbf{q} = e^{\frac{\theta}{2}(u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k})} = \cos \frac{\theta}{2} + (u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k}) \sin \frac{\theta}{2} = \cos \frac{\theta}{2} + \mathbf{u} \sin \frac{\theta}{2}$$

The desired rotation can be applied to an ordinary vector $\mathbf{p} = (p_x, p_y, p_z) = p_x\mathbf{i} + p_y\mathbf{j} + p_z\mathbf{k}$ in 3-dimensional space

Rotated vector = $\mathbf{q} \mathbf{p} \mathbf{q}^{-1}$

Here vector \mathbf{p} is rotated about the unit vector \mathbf{u} , by an angle θ

ROS uses quaternions to track and apply rotations. A quaternion has 4 components (x,y,z,w).

0.2.1 Usage of Quaternions in ROS

Quaternions are used in ROS (Robot Operating System) for representing orientations and rotations due to their mathematical properties and advantages over other representations like Euler angles. $q=w+xi+yj+zk$ where w,x,y,z are the components of the quaternion and i,j,k are the imaginary units.

The scalar component (w) represents the cosine of half the rotation angle, and the vector components (x, y, z) represent the axis of rotation scaled by the sine of half the rotation angle.

Together, the quaternion (w, x, y, z) describes the orientation or rotation in 3D space. It represents both the rotation angle and the axis of rotation, allowing for accurate and compact representation of rotations.

When using quaternions in ROS, these components are typically used to define the orientation of robots, sensors, or objects.

0.3 Visual Odometry

Localization is the main task for autonomous vehicles to be able to track their paths and properly detect and avoid obstacles. Vision-based odometry is one of the robust techniques used for vehicle localization. VO is the pose estimation process of an agent (e.g., vehicle, human, and robot) that involves the use of only a stream of images acquired from a single or from multiple cameras attached to it. VO provides an incremental online estimation of a vehicle's position by analyzing the image sequences captured by a camera. VO integrates pixel displacements between image frames over time, hence gets the name like wheel odometry where we estimate the motion of a vehicle incrementally by integrating the number of turns of its wheels over time. In visual localization, the computations involve several steps, namely, (1) acquisition of camera images, (2) extraction of several image features (edges, corners, lines, etc.), (3) matching between image frames, and (4) calculation of the position by calculating the pixel displacement between frames. VO is an inexpensive and alternative odometry technique that is more accurate than conventional techniques, such as GPS, INS, wheel odometry. . The main challenges in VO systems are mainly related to computational cost and light and imaging conditions

0.4 rtabmap_ros

rtabmap_ros is a ROS (Robot Operating System) package that provides an interface between the RTAB-Map (Real-Time Appearance-Based Mapping) library and ROS. RTAB-Map is a popular open-source mapping and localization framework designed for robotic applications.

rtabmap_ros package allows you to use RTAB-Map capabilities within the ROS ecosystem. It provides nodes and launch files to perform 3D mapping and localization tasks using sensors such as Intel RealSense.

0.5 actionlib

In ROS services, if the service takes a long time to execute, the user might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing. The actionlib package provides tools to create servers that execute long-running goals that can be preempted. It also provides a client interface in order to send requests to the server.

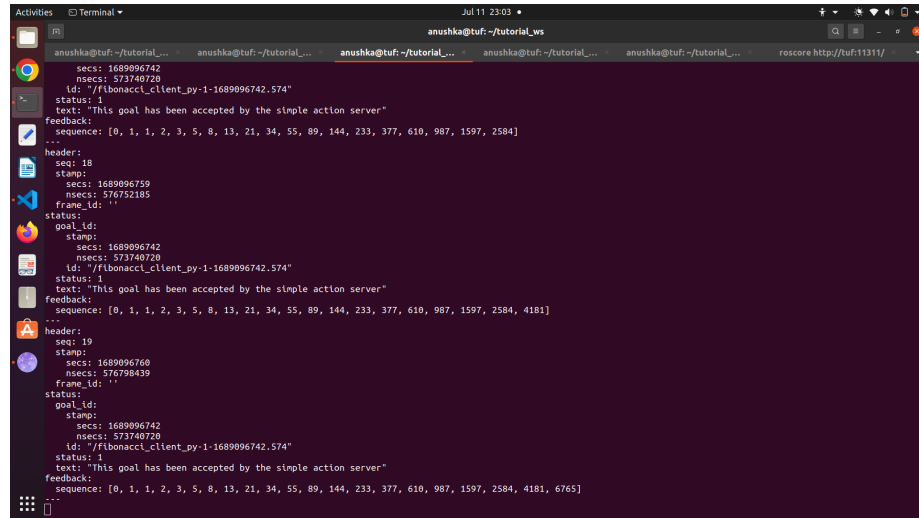
The ActionClient and ActionServer communicate via a "ROS Action Protocol", which is built on top of ROS messages. The client and server then provide a simple API for users to request goals (on the client side) or to execute goals (on the server side) via function calls and callbacks.

In order for the client and server to communicate, we need to define a few messages on which they communicate. This is with an action specification. This defines the Goal, Feedback, and Result messages.

The action specification is defined using a .action file. The .action file has the goal definition, followed by the result definition, followed by the feedback definition, with each section separated by 3 hyphens (—). Based on this .action file, 6 messages need to be generated in order for the client and server to communicate. This generation can be automatically triggered during the make process.

0.5.1 Tutorial

In the tutorial to generate a Fibonacci sequence, the goal is the order of the sequence, the feedback is the sequence as it is computed, and the result is the final sequence.



```
anushka@tuf: ~/tutorial_ws
secs: 1689996742
nsecs: 573740720
id: "/fibonacci_client_py-1-1689996742.574"
status: 1
text: "This goal has been accepted by the simple action server"
feedback:
sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584]
...
header:
seq: 18
stamp:
secs: 1689996759
nsecs: 576752185
frame_id: ''
status:
goal_id:
stamp:
secs: 1689996742
nsecs: 573740720
id: "/fibonacci_client_py-1-1689996742.574"
status: 1
text: "This goal has been accepted by the simple action server"
feedback:
sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]
...
header:
seq: 19
stamp:
secs: 1689996769
nsecs: 576798439
frame_id: ''
status:
goal_id:
stamp:
secs: 1689996742
nsecs: 573740720
id: "/fibonacci_client_py-1-1689996742.574"
status: 1
text: "This goal has been accepted by the simple action server"
feedback:
sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
```

Figure 2: rostopic echo of the feedback

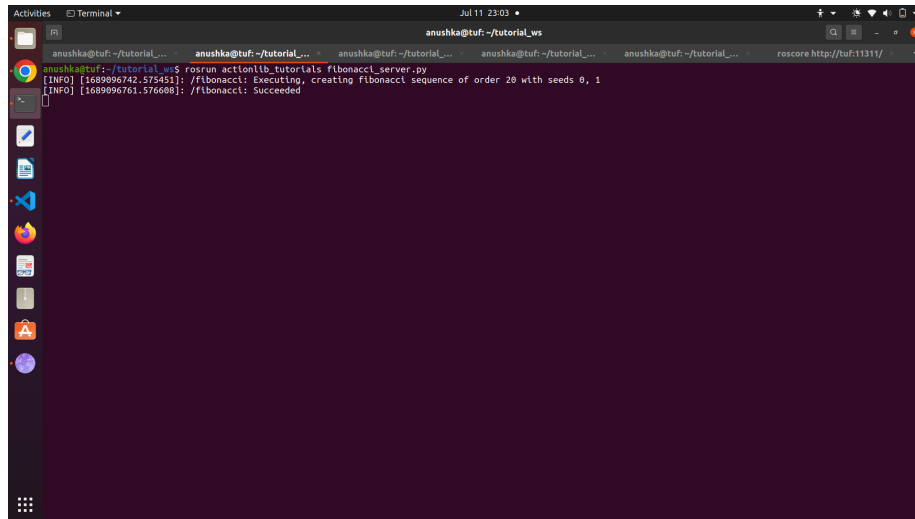


Figure 3: Server displaying succeeded action

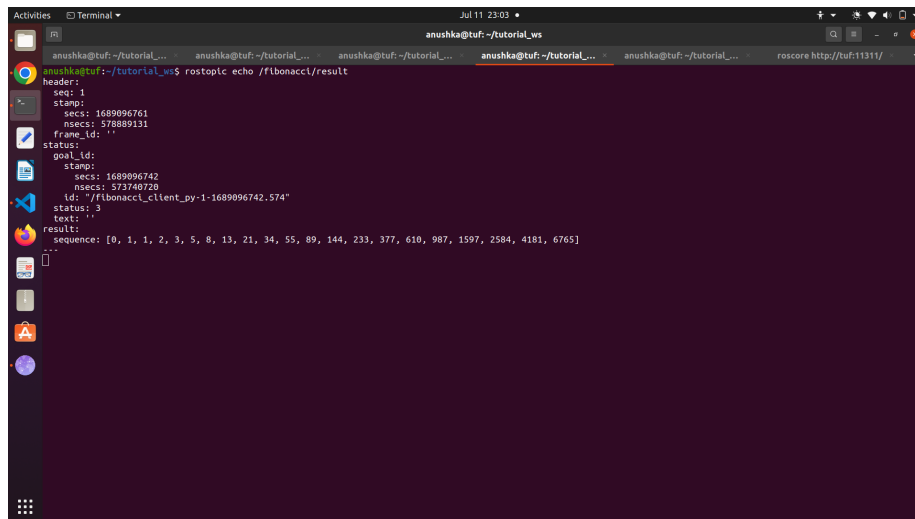
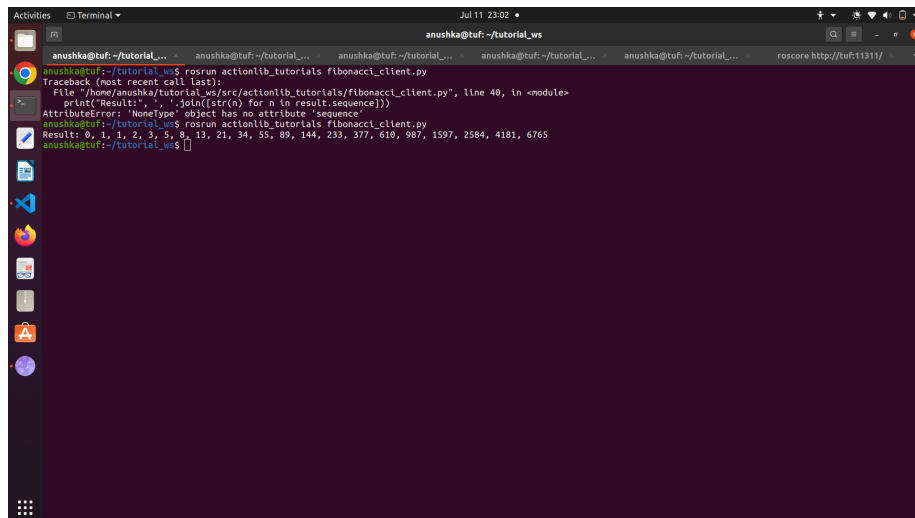


Figure 4: rostopic echo of the result



The image shows a terminal window with a dark background and light-colored text. The window title is "anushka@tuf: ~/tutorial_ws". The terminal output shows the execution of a ROS action, which results in a Fibonacci sequence. The sequence is displayed as a list of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765. The terminal also shows a traceback for an `AttributeError` that occurred during the execution of the action.

```
anushka@tuf:~/tutorial_ws$ roslaunch actionlib_tutorials fibonacci_client.py
Traceback (most recent call last):
  File "/home/anushka/tutorial_ws/src/actionlib_tutorials/fibonacci_client.py", line 40, in <module>
    print('Result:', ' '.join([str(n) for n in result.sequence]))
AttributeError: 'NoneType' object has no attribute 'sequence'
anushka@tuf:~/tutorial_ws$ roslaunch actionlib_tutorials fibonacci_client.py
Result: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765
anushka@tuf:~/tutorial_ws$
```

Figure 5: Result of the Action