Navigation 2

Release 1.0.0

Various

Aug 22, 2023

CONTENTS

CHAPTER

ONE

OUR SPONSORS

DEXORY

POLYMATH ROBOTICS

STEREOLABS

CONFIDENTIAL

Powered By

Open Navigation LLC

CHAPTER

TWO

OVERVIEW

Nav2 is the professionally supported spiritual successor of the ROS Navigation Stack. This project seeks to find a safe way to have a mobile robot move to complete complex tasks through many types of environments and classes of robot kinematics. Not only can it move from Point A to Point B, but it can have intermediary poses, and represent other types of tasks like object following and more. Nav2 is a production-grade and high-quality navigation framework trusted by 50+ companies worldwide.

It provides perception, planning, control, localization, visualization, and much more to build highly reliable autonomous systems. This will complete environmental modeling from sensor data, dynamic path planning, compute velocities for motors, avoid obstacles, represent semantic regions and objects, and structure higher-level robot behaviors. To learn more about this project, such as related projects, robots using, ROS1 comparison, and maintainers, see *About and Contact*. To learn more about navigation and ROS concepts, see *Navigation Concepts*.

Nav2 uses behavior trees to create customized and intelligent navigation behavior via orchestrating many independent modular servers. A task server can be used to compute a path, control effort, recovery, or any other navigation related task. These separate servers communicate with the behavior tree (BT) over a ROS interface such as an action server or service. A robot may utilize potentially many different behavior trees to allow a robot to perform many types of unique tasks.

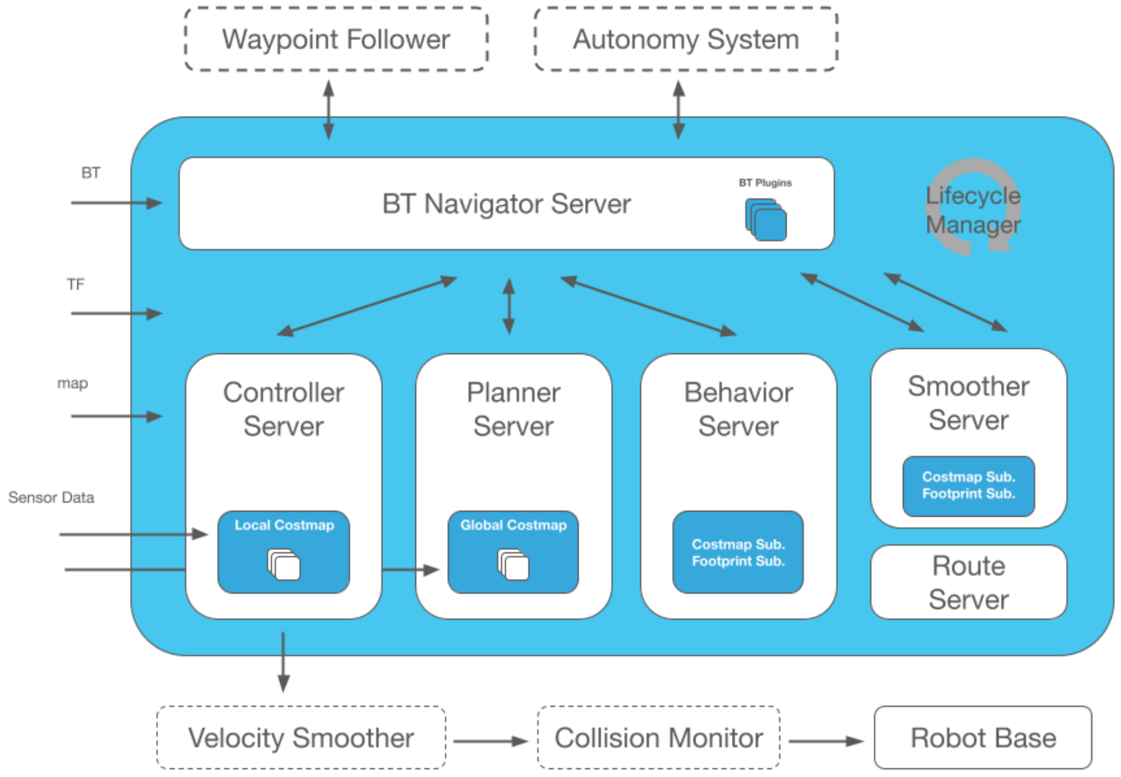
The diagram below will give you a good first-look at the structure of Nav2. Note: It is possible to have multiple plugins for controllers, planners, and recoveries in each of their servers with matching BT plugins. This can be used to create contextual navigation behaviors. If you would like to see a comparison between this project and ROS (1) Navigation, see *ROS to ROS 2 Navigation*.

The expected inputs to Nav2 are TF transformations conforming to REP-105, a map source if utilizing the Static Costmap Layer, a BT XML file, and any relevant sensor data sources. It will then provide valid velocity commands for the motors of a holonomic or non-holonomic robot to follow. We currently support all of the major robot types: holonomic, differential-drive, legged, and ackermann (car-like) base types! We support them uniquely with both circular and arbitrarily-shaped robots for SE2 collision checking.

It has tools to:

- Load, serve, and store maps (Map Server)
- Localize the robot on the map (AMCL)
- Plan a path from A to B around obstacles (Nav2 Planner)
- Control the robot as it follows the path (Nav2 Controller)
- Smooth path plans to be more continuous and feasible (Nav2 Smoother)
- Convert sensor data into a costmap representation of the world (Nav2 Costmap 2D)
- Build complicated robot behaviors using behavior trees (Nav2 Behavior Trees and BT Navigator)
- Compute recovery behaviors in case of failure (Nav2 Recoveries)
- Follow sequential waypoints (Nav2 Waypoint Follower)

- Manage the lifecycle and watchdog for the servers (Nav2 Lifecycle Manager)
- Plugins to enable your own custom algorithms and behaviors (Nav2 Core)
- Monitor raw sensor data for imminent collision or dangerous situation (Collision Monitor)
- Python3 API to interact with Nav2 in a pythonic manner (Simple Commander)
- A smoother on output velocities to guarantee dynamic feasibility of commands (Velocity Smoother)



We also provide a set of starting plugins to get you going. A list of all plugins can be found on *Navigation Plugins* - but they include algorithms for the spanning cross section of common behaviors and robot platform types.

CHAPTER

THREE

CITATIONS

If you use the navigation framework, an algorithm from this repository, or ideas from it please cite this work in your papers!

S. Macenski, F. Martín, R. White, J. Clavero. The Marathon 2: A Navigation System. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2020.

IROS 2020 talk on Nav2 Marathon Experiments:

```
@InProceedings{macenski2020marathon2,
author = {Macenski, Steven and Martin, Francisco and White, Ruffin and Ginés Clavero,␣
↪Jonatan},
title = {The Marathon 2: A Navigation System},
booktitle = {2020 IEEE/RSJ International Conference on Intelligent Robots and Systems␣
↪(IROS)},
year = {2020}
}
```

If you use **any** of the algorithms in Nav2 or the analysis of the algorithms in your work, please cite this work in your papers!

S. Macenski, T. Moore, DV Lu, A. Merzlyakov, M. Ferguson, From the desks of ROS maintainers: A survey of modern & capable mobile robotics algorithms in the robot operating system 2, Robotics and Autonomous Systems, 2023

```
@article{macenski2023survey,
      title={From the desks of ROS maintainers: A survey of modern & capable mobile␣
↪robotics algorithms in the robot operating system 2},
      author={S. Macenski, T. Moore, DV Lu, A. Merzlyakov, M. Ferguson},
      year={2023},
      journal = {Robotics and Autonomous Systems}
}
```

If you use the Regulated Pure Pursuit Controller algorithm or software from this repository, please cite this work in your papers!

S. Macenski, S. Singh, F. Martin, J. Gines, Regulated Pure Pursuit for Robot Path Tracking, Autonomous Robots, 2023.

```
@article{macenski2023regulated,
      title={Regulated Pure Pursuit for Robot Path Tracking},
      author={Steve Macenski and Shrijit Singh and Francisco Martin and Jonatan Gines}
↪,
      year={2023},
      journal = {Autonomous Robots}
}
```

If you use our work on VSLAM and formal comparisons for service robot needs, please cite the paper:

A. Merzlyakov, S. Macenski. A Comparison of Modern General-Purpose Visual SLAM Approaches. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2021.

```
@InProceedings{vslamComparison2021,
author = {Merzlyakov, Alexey and Macenski, Steven},
title = {A Comparison of Modern General-Purpose Visual SLAM Approaches},
booktitle = {2021 IEEE/RSJ International Conference on Intelligent Robots and Systems␣
↪(IROS)},
year = {2021}
}
```

CHAPTER

FOUR

EXAMPLE

Below is an example of the TB3 navigating in a small lounge.

4.1 Getting Started

This document will take you through the process of installing the Nav2 binaries and navigating a simulated Turtlebot 3 in the Gazebo simulator.

Note: See the *Build and Install* for other situations such as building from source or working with other types of robots.

Warning: This is a simplified version of the Turtlebot 3 instructions. We highly recommend you follow the official Turtlebot 3 manual if you intend to continue working with this robot beyond the minimal example provided here.

4.1.1 Installation

1. Install the ROS 2 binary packages as described in the official docs
2. Install the Nav2 packages using your operating system's package manager:

```
sudo apt install ros-<ros2-distro>-navigation2
sudo apt install ros-<ros2-distro>-nav2-bringup
```

3. Install the Turtlebot 3 packages (Humble and older):

```
sudo apt install ros-<ros2-distro>-turtlebot3-gazebo
```

4.1.2 Running the Example

1. Start a terminal in your GUI
2. Set key environment variables:

```
source /opt/ros/<ros2-distro>/setup.bash
export TURTLEBOT3_MODEL=waffle
export GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:/opt/ros/<ros2-distro>/share/
↪turtlebot3_gazebo/models
```

3. In the same terminal, run:

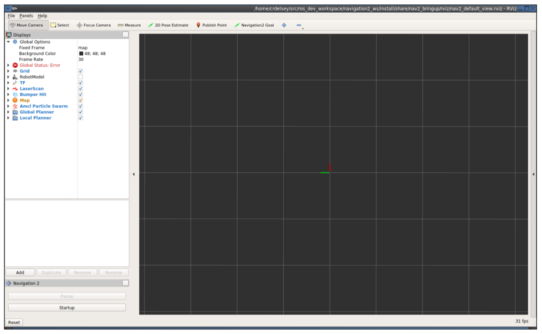
```
ros2 launch nav2_bringup tb3_simulation_launch.py headless:=False
```

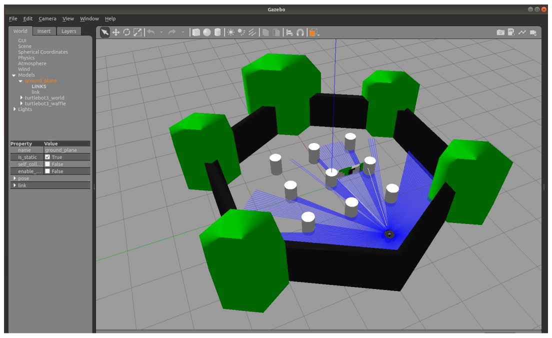
Note: For `ROS 2 Dashing Diademata` or earlier, use `nav2_simulation_launch.py`. However, it is recommended to use the most recent ROS 2 LTS distribution for improved stablity and feature completeness.

`headless` defaults to true; if not set to false, gzclient (the 3d view) is not started.

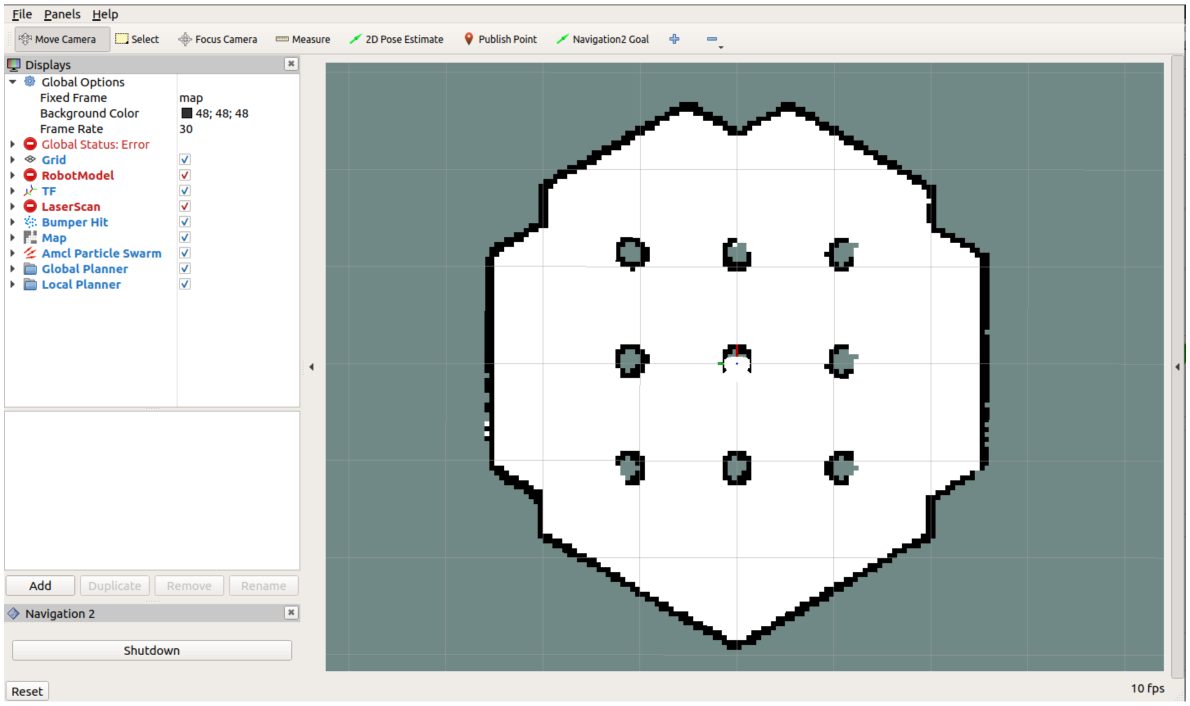
This launch file will launch Nav2 with the AMCL localizer in the `turtlebot3_world` world. It will also launch the robot state publisher to provide transforms, a Gazebo instance with the Turtlebot3 URDF, and RVIZ.

If everything has started correctly, you will see the RViz and Gazebo GUIs like this:





4. If not autostarting, click the “Startup” button in the bottom left corner of RViz. This will cause Nav2 to change to the Active state. It should change appearance to show the map.

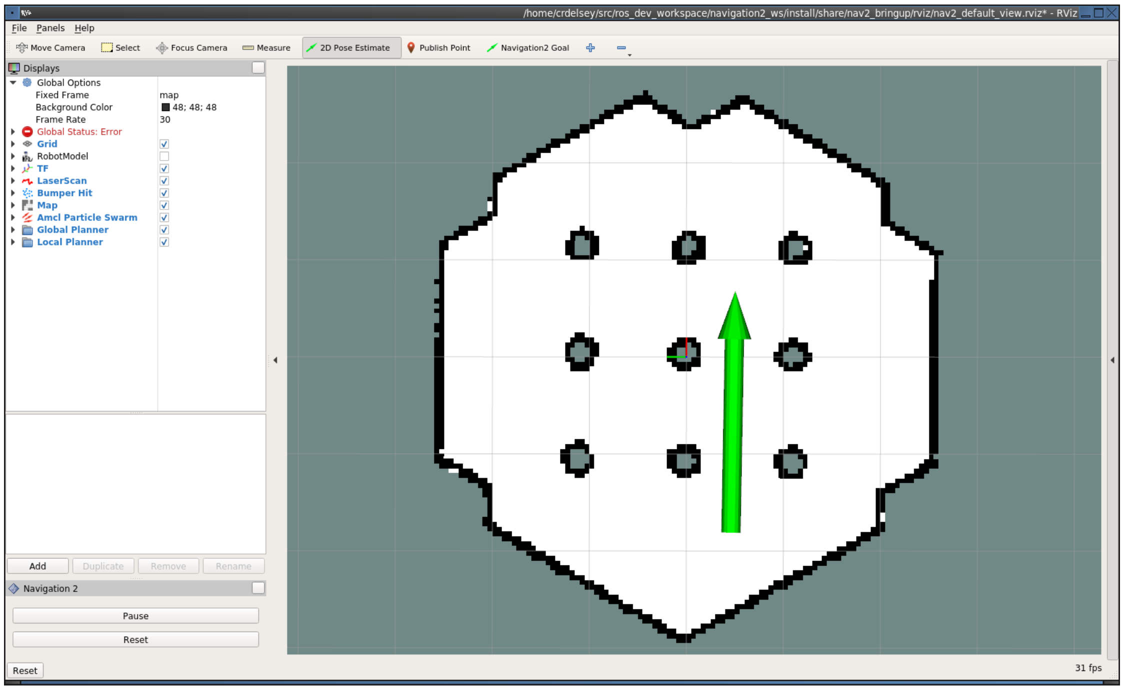


Global Options
Fixed Frame map
Background Color 48; 48; 48
Frame Rate 30
Global Status: Error
Grid ✓
RobotModel ✓
TF ✓
LaserScan ✓
Bumper Hit ✓
Map ✓
Amcl Particle Swarm ✓
Global Planner ✓
Local Planner ✓

4.1.3 Navigating

After starting, the robot initially has no idea where it is. By default, Nav2 waits for you to give it an approximate starting position. Take a look at where the robot is in the Gazebo world, and find that spot on the map. Set the initial pose by clicking the “2D Pose Estimate” button in RViz, and then down clicking on the map in that location. You set the orientation by dragging forward from the down click.

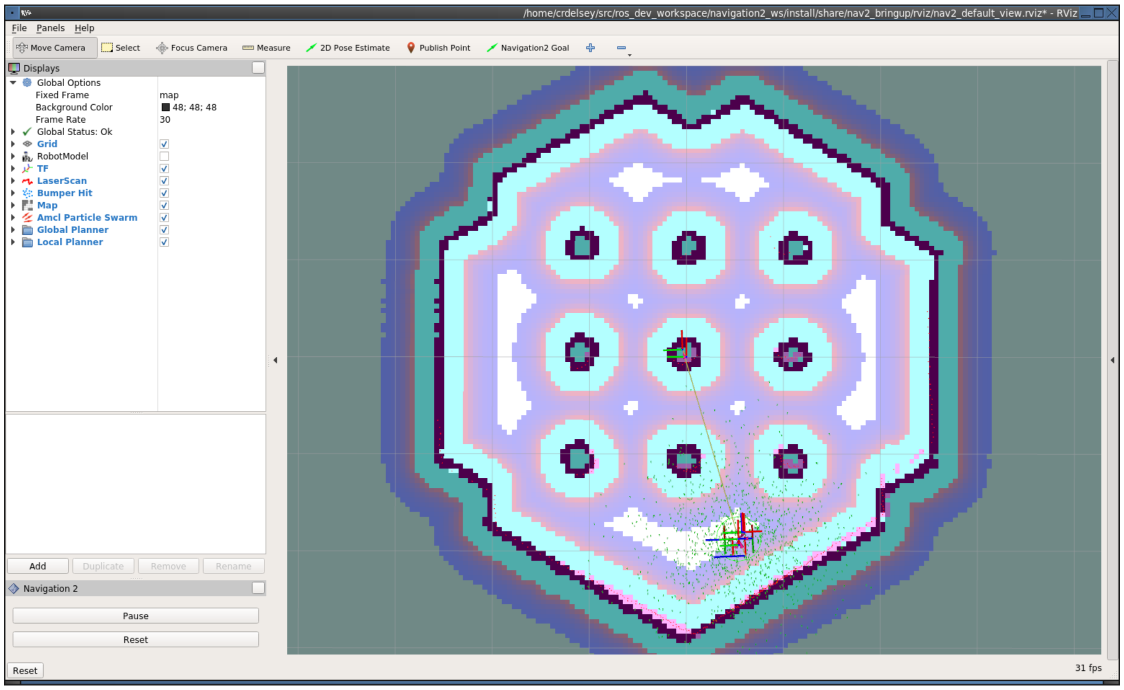
If you are using the defaults so far, the robot should look roughly like this.



Global Options
Fixed Frame map
Background Color 48; 48; 48
Frame Rate 30
Global Status: Error
Grid ✓
RobotModel
TF ✓
LaserScan ✓
Bumper Hit ✓
Map ✓
Amcl Particle Swarm ✓
Global Planner ✓
Local Planner ✓

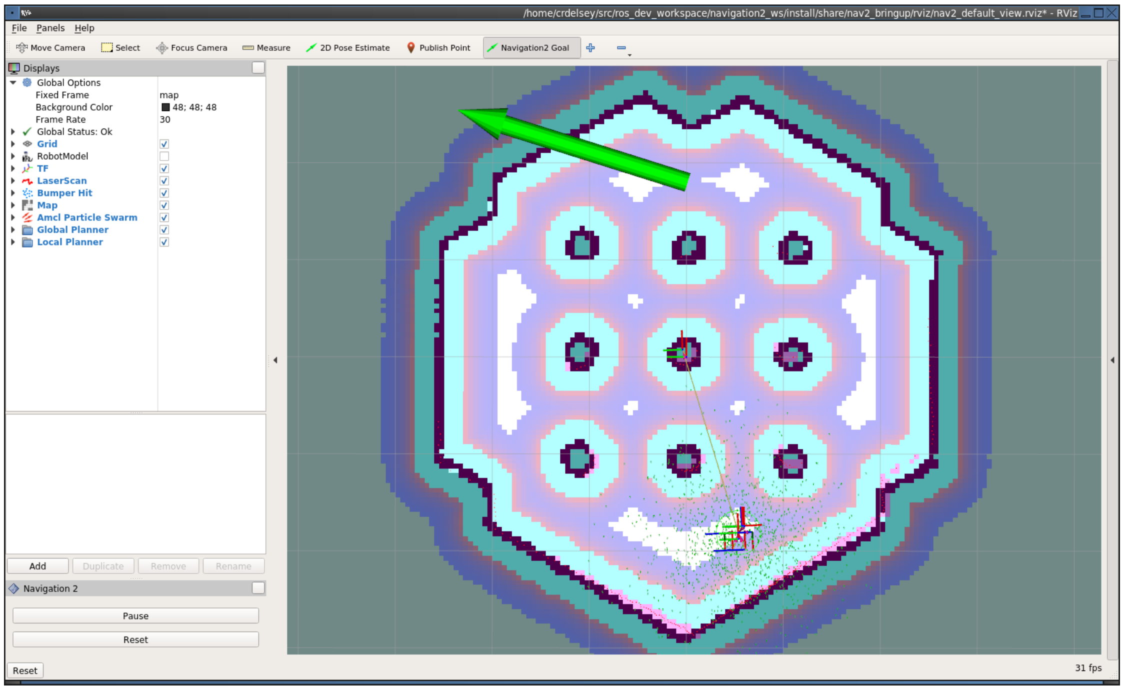
If you don't get the location exactly right, that's fine. Nav2 will refine the position as it navigates. You can also, click the "2D Pose Estimate" button and try again, if you prefer.

Once you've set the initial pose, the transform tree will be complete and Nav2 will be fully active and ready to go. You should see the robot and particle cloud now.


/home/crdelsey/src/ros_dev_workspace/navigation2_ws/install/share/nav2_bringup/rviz/nav2_default_view.rviz* - RViz

```
Global Options
  Fixed Frame        map
  Background Color   48; 48; 48
  Frame Rate         30
Global Status: Ok
```

Next, click the "Navigaton2 Goal" button and choose a destination. This will call the BT navigator to go to that goal through an action server. You can pause (cancel) or reset the action through the Nav2 rviz plugin shown.


/home/crdelsey/src/ros_dev_workspace/navigation2_ws/install/share/nav2_bringup/rviz/nav2_default_view.rviz* - RViz

```
Global Options
  Fixed Frame        map
  Background Color   48; 48; 48
  Frame Rate         30
Global Status: Ok
```

Now watch the robot go!

4.2 Development Guides

This section includes guides for developing Nav2, e.g. how to build from source, how to use dev containers, and how to get involved.

4.2.1 Build and Install

Install

Nav2 and its dependencies are released as binaries. You may install it via the following to get the latest stable released version:

```
source /opt/ros/<distro>/setup.bash
sudo apt install \
  ros-$ROS_DISTRO-navigation2 \
  ros-$ROS_DISTRO-nav2-bringup \
  ros-$ROS_DISTRO-turtlebot3*
```

Build

There are a few ways to build Nav2 using:

- Released Distribution Binaries
  - Build Nav2 using readily installable binary dependencies
- Rolling Development Source
  - Build Nav2 using custom or latest source dependencies
- Docker Container Images
  - Build Nav2 using cached images and templated Dockerfiles

Tip: For a *repeatable*, *reproducible* and *streamlined* development experience, check the Nav2 documentation on using *Dev Containers*!

Released Distribution Binaries

To build Nav2, you'll first need to build or install ROS 2 and related development tools, including: `colcon`, `rosdep` and `vcstool`.

See also:

For more information on building or installing ROS 2 distros, see the official documentation:

- ROS 2 Installation
- Install development tools and ROS tools

Once your environment is setup, clone the repo, install all dependencies, and build the workspace:

Attention: The branch naming schema for Nav2 is organized by ROS distro, while the default branch for Rolling is `main`.

```
source /opt/ros/<distro>/setup.bash
mkdir -p ~/nav2_ws/src && cd ~/nav2_ws
git clone https://github.com/ros-planning/navigation2.git --branch $ROS_DISTRO ./src/
↪navigation2
rosdep install -y \
  --from-paths ./src \
  --ignore-src
colcon build \
  --symlink-install
```

You can then `source ~/nav2_ws/install/setup.bash` to get ready for demonstrations!

Hint: For more examples on building Nav2 from released distribution binaries, checkout distro.Dockerfile.

Rolling Development Source

Building Nav2 using rolling development source is similar to building Nav2 from released distribution binaries, where instead you build dependencies from source using the main development branches for all ROS based packages.

See also:

For more information on building ROS 2 from source, see the official documentation:

- ROS 2 Building from source

Once your environment is setup, clone the repo, import all dependencies, and build the workspace:

Attention: Be sure to check that all dependencies you need are included and uncommented in the `.repos` file.

```
source <ros_ws>/install/setup.bash
mkdir -p ~/nav2_ws/src && cd ~/nav2_ws
git clone https://github.com/ros-planning/navigation2.git --branch $ROS_DISTRO ./src/
↪navigation2
vcs import ./src < ./src/navigation2/tools/underlay.repos
rosdep install -y \
  --from-paths ./src \
  --ignore-src
colcon build \
  --symlink-install
```

You can then `source ~/nav2_ws/install/setup.bash` to get ready for demonstrations!

Hint: For more examples on building Nav2 from rolling development source, checkout source.Dockerfile.

Docker Container Images

Building Nav2 using Docker container images provides a repeatable and reproducible environment to automate and self document the entire setup process. Instead of manually invoking the development tools as documented above, you can leverage the project's Dockerfiles to build and install Nav2 for various distributions.

See also:

For more information on installing Docker or leaning about Dockerfiles, see the official documentation:

- Docker Engine
- Dockerfile reference

Once your system is setup, you can build the Nav2 Dockerfile from the root of the repo:

```
export ROS_DISTRO=rolling
git clone https://github.com/ros-planning/navigation2.git --branch main
docker build --tag navigation2:$ROS_DISTRO \
  --build-arg FROM_IMAGE=ros:$ROS_DISTRO \
  --build-arg OVERLAY_MIXINS="release ccache lld" \
  --cache-from ghcr.io/ros-planning/navigation2:main \
  ./navigation2
```

The docker build command above creates a tagged image using the *Dockerfile* from the context specified using the path to the repo, where build-time variables are set using additional arguments, e.g. passing a set of colcon mixins to configure the workspace build. Check the `ARG` directives in the *Dockerfile* to discover all build-time variables available. The command also specifies an external cache source to pull the latest cached image from Nav2's Container Registry to speed up the build process.

Tip: The images cached from above are used for Nav2 CI, but can also be used with Nav2 *Dev Containers*!

Generate Doxygen

Run `doxygen` in the root of the Nav2 repository. It will generate a `/doc/*` directory containing the documentation. The documentation entrypoint in a browser is index.html.

Help

Build Troubleshooting Guide

Build Troubleshooting Guide

Common Nav2 Dependencies Build Failures

- Make sure that .bashrc file has no ROS environment variables in it. Open new terminals and try to build the packages again.
- Make sure to run rosdep for the correct ROS 2 distribution. `rosdep install -y -r -q --from-paths src --ignore-src --rosdistro <ros2-distro>`

- Make sure that the `setup.bash` is sourced in the ROS 2 installation or ROS 2 main build workspace, if applicable. Check if you can run talker and listener nodes.
- Make sure that the `setup.bash` in `nav2_depend_ws/install` is sourced.
- Check if you have the correct ROS version and distribution. `printenv | grep -i ROS`
- If you see a bunch of errors on startup about `map` or `odom` frame not existing, remember to activate drivers (or gazebo for simulation) and set an initial pose in `map` frame. Costmap2D will block activation until a full TF tree is available.
- Make sure you've activated the lifecycle nodes if you're not seeing transforms or servers running.
- Search GitHub Issues
- Make sure you're using the correct branch for your distribution. There is no cross support from branch for `DistroA` in `DistroB`. The main development branch uses the rolling distribution.

Still can't solve it? Let us know about your issue. Open a ticket.

Reporting Issue

- If you run into any issues when building Navigation2, you can use the search tool in the issues tab on GitHub and always feel free to open a ticket.

4.2.2 Dev Containers

You can use dev containers to build the project if you prefer a streamlined setup experience. This means you can use the same tools and dependencies as the rest of the team, including our Continuous Integration (CI) workflows, without worrying about installing dependencies on your host machine. Additionally, using Dev Containers makes it simple to switch between local or remote development environments, such as GitHub Codespaces.

See also:

More info on Dev Containers can be found here:

- Development Containers
  - An open specification for enriching containers with development specific content and settings
- Developing inside a Container
  - Learn how to use Visual Studio Code to develop inside a Docker container
- GitHub Codespaces overview
  - A development environment hosted in the cloud

Dev Container Guide

In this guide, we'll walk through the process of creating and using dev containers for the project. While included subsections will provide greater detail on the various aspects of the process, complete comprehension of the entire guide is not required to get started, but is recommended for those interested in how dev containers work, or how to customize and optimize them for their own personal workflows.

Creating Dev Containers

Before creating a dev container, you'll want to choose the exact configuration to use. By default the `.devcontainer/devcontainer.json` configuration is selected, however you can also choose any other `devcontainer.json` file in the `.devcontainer/` directory, where such configurations can be nested to provide greater customization: either by targeting different stages within different Dockerfiles, overriding any merged metadata or default properties, or inclusion of additional extensions and alternate commands.

See also:

The specification, reference, and schema for the `devcontainer.json` config file format can be found here:

- Specification
  - Development Container Specification
- Reference
  - Metadata and properties reference
- Schema
  - JSON schema for `devcontainer.json`

Building the image

When first creating Dev Containers, any supporting tool or service used will download and build the docker images needed to run the container. This includes pulling any parent images the project's Dockerfile builds `FROM`, as well as any tags or layers declared via `cacheFrom`, as specified in the chosen `devcontainer.json` config file. This can take a while, but only needs to be done once, or at least not again until such layers are updated and pushed to the image registry.

Specifically, for this project, the default `devcontainer.json` file targets the `dever` stage within the project's root Dockerfile, the stage that also includes handy tools for developing the project, such as bash auto completion. This stage is in turn built `FROM` the `builder` stage, the stage that only includes the dependencies needed for building the project, as reused by the project's CI. For example, the `dever` stage modifies `/etc/bash.bashrc` to automatically source `install/setup.bash` from the underlay workspace, ensuring all VS Code extensions are loaded with the correct environment, while avoiding any race conditions during installation and startup.

To speed up the initial build, images layers from this `builder` stage are cached by pulling the same image tag used by the project's CI, hosted from the image registry. This ensures your local dev container replicates our CI environment as close as possible, while benefiting from any cached work preemptively performed by the CI. Yet, this still allows you to customize the project's Dockerfile and rebuild the container, without needing to update CI images to reflect your local modifications.

See also:

More details on the project's CI and related docker image registry can be found here:

- Chronicles of Caching and Containerising CI for Nav2
  - Video presentation from ROS World 2021 - Ruffin White

Once the base image from the target stage is built, the supporting tool or service may then add additional layers to the image, such as installing additional features or customizations. For VS Code, this also includes some fancy file caching for any extensions to install later. Once this custom image is built, it is then used to start the dev container.

Starting the container

When first creating Dev Containers, any supporting tool or service will invoke a sequence of commands specified in the chosen `devcontainer.json` config file. This can take a while, but only needs to be done once, or at least not again until the container is rebuilt, triggered by either updating the Dockerfile, base image, or `.devcontainer/` config.

Specifically, for this project, the default `devcontainer.json` config executes the `onCreateCommand` to initially colcon cache, clean, and build the overlay workspace for the project. This ensures the workspace is precompiled and ready to use, while also ensuring any changes to the project's source code are reflected in the container. This is useful for:

- IntelliSense
  - Enables VS Code extensions to parse auto generated code
  - Applicable for ROS package defining messages and services files
  - Necessary for code modeling, navigation, and syntax highlighting
- Caching
  - Enables Codespace Prebuilds to cache the workspace artifacts
  - Applicable for reducing startup time when spawning new Codespaces
  - Necessary for limiting costs from CPU and storage usage

While the colcon workspace is being built, VS Code will simultaneously install any specified extensions and settings. Next the `updateContentCommand` is executed, which reruns whenever the container is started or restarted. Specifically, for this project, this command re-cleans and re-builds the same colcon workspace as before, but only for invalidated packages detected by colcon cache using the lockfiles initialized during the `onCreateCommand`. This caching behavior also replicates the project's CI workflow. This is useful for:

- Branching
  - Enables caching of workspace artifacts when switching between branches
  - Applicable for reviewing pull requests without rebuilding entire container
  - Necessary for reducing startup time when spawning new Codespaces

Hint: More documentation about these additional colcon verb extensions can be found here:

- colcon-cache
  - A colcon extension to cache the processing of packages
- colcon-clean
  - A colcon extension to clean package workspaces

Finally, the `postCreateCommand` is executed, which also reruns whenever the container is started or restarted. Specifically, for this project, this command makes a last few tweaks to the user's environment to improve the development experience.

To speed up subsequent startups, volumes that are mounted to the container store a persistent ccache and colcon workspace, while the environment is set to enable ccache via colcon mixins. These volumes are labeled using the `devcontainerId` variable, which uniquely identify the dev container on a Docker host, allowing us to refer to a common identifier that is unique to the dev container, while remaining stable across rebuilds. This is useful for:

- Caching

  - Enables colcon workspaces and ccache to persist between container rebuilds
  - Applicable for avoiding re-compilation when modifying dev container config files
  - Necessary for quickly customizing image or features without rebuilding from scratch

Tip: While these volumes are uniquely named, you could rename them locally to further organize or segment works-in-progress. E.g. appending branch names to the volume name to quickly switch between pull requests and cached colcon workspaces.

Additionally, the container can be granted privileged and non-default Linux capabilities, connected using the host network mode and IPC and PID spaces, with a relaxed security configuration and seccomp confinement for native debugging and external connectivity. This is useful for:

- Hybrid development
  - Enables connecting ROS nodes external to the container
  - Applicable for debugging or visualizing distributed systems
  - Necessary for DDS discovery and shared memory transport
- Device connectivity
  - Enables hardware forwarding from host machine to container
  - Applicable for ROS package using sensors and actuators
  - Necessary for some GPU drivers and USB devices

Attention: Such `runArgs` in the `devcontainer.json` config can be enabled or customized, either expanded or or narrowed in scope, to better suit your desired development environment. The default configuration merely comments out these parameters, to limit unintended side effects or cross talk between containers, but can be uncommented to accommodate the widest range of development use cases.

See also:

More details on using DDS, debuggers, or devices with Docker containers can be found here:

- How to Communicate Across Docker Containers Using the Host Driver
  - Using the `host` network driver to access all network interfaces of the host machine from the Docker container
- Communicate between two Docker containers using DDS and shared memory
  - Enabling containers to communicate with one another and with the host machine using interprocess communication (IPC)
- Debugging programs running inside Docker containers, in production
  - Using tools like strace, perf, gdb when debugging programs running inside containers

Using Dev Containers

Once the dev container has been created and setup completed, VS Code will open a new workspace directly from the project's root directory, which itself is mounted within the source directory in the overlay colcon workspace. From here you can build, test, and debug the project as you normally would, with the added benefit of having the project's dependencies, intellisense, linters, and other extensions pre-configured and ready to use. Simply open a new terminal (Crtl+Shift+\`), cd to the root of the colcon workspace, and run the usual colcon commands.

Tip: You can incorporate the same scripts used by the `devcontainer.json` config file to further automate your local development workflow.

Terminals

If you prefer using an alternate terminal emulator, rather than the built-in VS Code terminal, you can open a separate shell session by simply using the Dev Container CLI or directly using the Docker CLI via the `exec` subcommands.

- Dev Container CLI
  - `devcontainer exec --workspace-folder $NAV2_WS/src/navigation2 bash`
- docker exec
  - `docker exec -it <container-id> bash`

Attention: Shell sessions spawned directly via `docker exec` do not set the same environment that `devcontainer exec` does using `userEnvProbe`. Additional environment variables include `REMOTE_CONTAINERS_IPC`, `REMOTE_CONTAINERS_SOCKETS` and are used by vscode, ssh and X11.

Hint: The environment provided by `userEnvProbe` can be sourced manually. E.g. for the default `loginInteractiveShell` probe:

```
find /tmp -type f -path "*/devcontainers-*/env-loginInteractiveShell.json" -exec \
  jq -r 'to_entries | .[] | "\(.key)=\(.value | @sh)"' {} \; > .env
source .env
```

Lifecycle

While using the dev container, try and keep in mind the lifecycle of the container itself. Specifically, containers are ephemeral, meaning they are normally destroyed and recreated whenever the dev environment is rebuilt or updated. Subsequently, a best practice is to avoid storing any persistent data within the container, and instead utilize the project's source directory, or a separate mounted volume. When altering the development environment inside the container, try to remember to codify your changes into the Dockerfile, or the `devcontainer.json` config file, so that they can be easily reproduced and shared with others.

Important: This is particularly important when the host machine is inherently ephemeral as well, as the case may be when using cloud based environments such as Codespaces, so be sure to commit and push local changes to a remote repository:

- The codespace lifecycle

  - Maintain your data throughout the entire codespace lifecycle

---

Rebuilding

From time to time, you may need to rebuild the dev container, either because the base image, or `.devcontainer/` config was updated, or simply out of wanting a new fresh development environment. To do so, simply open the Command Palette (Ctrl+Shift+P) and select the `Remote-Containers: Rebuild Container` command.

Caution: Rebuilding the container will destroy any changes made to the container itself, such as installing additional packages, or modifying the environment. However, the project's source directory, and any mounted volumes, will remain unaffected.

For example, you may need to rebuild the dev container when:

- Pulling newer images from a container registry
  - specifically, image tags built `FROM` in the Dockerfile
  - or tags listed under `cacheFrom` in `devcontainer.json`
  - periodically done manually to ensure local environment reflects CI
- Updating the dev container configuration
  - specifically when modifying dependent stages in the `Dockerfile`
  - or when modifying `./devcontainer` files and commands
  - where build cache reuse correlates with severity of changes made

When necessary, you can also rebuild the container from scratch, e.i. without caching from docker, by selecting the `Remote-Containers: Rebuild Container Without Cache` command. This instead omits the `--cache-from` flag from the `docker buildx` command, while also adding the `--no-cache` and `--pull` flags to prevent caching from any existing image layers, using only the latest images from a container registry.

Caution: Rebuilding the container without cache may likely pull newer images from a container registry or install newer packages, as is common when developing for ROS 2 Rolling. You may then want to clean your overlay volume to avoid ABI incompatibilities or stale artifacts.

Rebuilding without caching may be necessary when:

- Needing to update the base image
  - specifically if dev container configurations remain unmodified
  - to forcefully rerun a `RUN` directive in the Dockerfile
  - such as unchanged `apt upgrade` or `rosdep update` commands

Specifically, for this project, volumes remain unaffected by this rebuilding process: i.e. those used to mount the ccache directory or colcon workspace. While volume management is left to the user's discretion, other projects may of course handle this differently, so be sure to check the `./devcontainer` configuration to inspect how various container resources may be managed.

Tip: Docker volume management can be done via the Docker CLI, or the VS Code Docker extension:

- Docker Volume CLI
  - Manage volumes using subcommands to create, inspect, list, remove, or prune volumes
- VS Code Docker extension
  - Makes it easy to create, manage, and debug containerized applications

What, Why, How?

Lets briefly explain what dev containers are, why you should use them, and how they work.

Hint: Here we'll assume the use of VS Code, but still applies to alternative tools and services, including other CLIs, IDEs, etc. such as:

- Dev Container CLI
  - A reference implementation for the open specification
- JetBrains Space | Develop in Dev Environment
  - Using Dev Containers with JetBrain based products
- Supporting tools
  - List of tools and services supporting the development container specification

What is a Dev Container?

A dev container is a Docker container that has all the tools and dependencies you need to develop the project. It runs in a self-contained environment and is isolated from other containers and your host machine. This lets you reliably develop for the project anywhere, notably for linux distributions targeted by ROS, regardless of your host machine's operating system.

Why use a Dev Container?

A dev container provides a common and consistent development environment. It ensures that everyone on the team is using the same tools and dependencies. It also makes it easy to switch between projects because each project can use a different container. This is especially useful if you work on multiple projects that use different versions of the same tools and dependencies, such as different versions of ROS.

How do Dev Containers work?

When you open the project in VS Code, VS Code checks for the dev container configuration nested within the `.devcontainer` folder under the project's root directory. If it finds one, it can prompt you to reopen the project in a container. If you choose to do so, it launches the container, connects to it, and mounts your project folder inside the container. You can then use VS Code in the container just as you would locally. While setting up the container, VS Code can also attempt to passthrough useful aspects of your local environment, such as git user configurations, X11 sockets, and more.

This is quite similar to earlier tools used to customize and run docker containers for development:

- rocker | ROS + Docker

  - A tool to run docker containers with overlays and convenient options for things like GUIs etc.
  - Developed by Open Robotics
- ADE Development Environment
  - A modular Docker-based tool to ensure developers have a common, consistent development environment
  - Developed by Apex.AI

Prerequisites

To use dev containers, you'll need the following:

- Docker Engine installed and running on the host machine
- Visual Studio Code installed on any remote machine
- Dev Containers extension installed in VS Code

Note: Alternatively, you could use GitHub Codespaces directly from the project repo, or any other remote host machine:

- Creating a codespace for a repository
  - How to create a codespace for repository via GitHub CLI, VS Code, or Web browser
- Develop on a remote Docker host
  - How to connect VS Code to a remote Docker host using SSH tunnels or TCP sockets

Getting started

Getting started using dev containers is as simple as opening the project in VS Code by either: following the notification prompt to reopen the project in a container, or explicitly opening the command palette (Crtl+Shift+P) and selecting `Remote-Containers: Reopen in Container`. This will create a new container, install any extensions specified in the project's default `.devcontainer/devcontainer.json` config file, and mount the project's root directory as the workspace folder. Once the container is created, VS Code will connect to it and you can start developing.

Tip: Clicking the `Starting Dev Container (show log)` notification in VS Code allows you to observe in live time how the sausage is made, while typing `Dev Containers: Show Log` into the command palette will list all the available commands to review and revisit these log files later.

While waiting for the initial setup, feel free to stretch your legs, grab a coffee, or continue to read the following guides to learn more about creating and using dev containers, or how to visualize and leverage graphical user interfaces from a headless development environment.

- ***Dev Container Guide***
  - How to develop Nav2 using dev containers and supporting tools

Security

Caution: Ensure you trust the authors and contents of workspaces before launching derived dev containers.

A word of caution when using dev containers: they are powerful tools, but can be a security concern, as the capability of arbitrary code execution facilitated by IDE extensions to enable such automation and convenience remains inherently dual use. Before launching a dev container, ensure you trust the workspaces and authors. For example, when reviewing a pull request, verify patches remain benign and do not introduce any malicious code. Although such vigilance is merited whenever compiling and running patched code, using containers with either elevated privileges or filesystem access renders this diligence even more prudent.

See also:

More info on trusting workspaces and extensions in general can be found here:

- Workspace Trust
  - VS Code user guid on trusting and configure workspaces

4.2.3 Getting Involved

As an open-source project, we welcome and encourage the community to submit patches directly to the Nav2. In our collaborative open source environment, standards and methods for submitting changes help reduce the chaos that can result from an active development community.

This document explains how to participate in project conversations, log and track bugs and enhancement requests, and submit patches to the project so your patch will be accepted quickly in the codebase.

Getting Involved

If you're interested in getting involved in Navigation 2, first of all, welcome! We encourage everyone to get involved from students, to junior developers, to senior developers, and executives. There's something to do for everyone from bug fixes, to feature development, new algorithms, and refactoring.

All ROS 2 TSC Working Groups have their meetings on the working group calendar. Here, you can find the date and time of the Navigation2 working group meeting. Make sure you're checking in your local timezone. From this calendar, you can add yourself to the event so it will appear on your google calendar and get the event link to the call through Google Hangouts. We encourage everyone interested to come to the meeting to introduce yourself, your project, and see what everyone is working on.

Further, ROS Discourse is a good place to follow larger discussions happening in the community and announcements. This is **not** the correct place to post questions or ask for assistance. Please visit ROS Answers for Q&A.

Lastly, we have a Community Slack where we chat in real-time about topics in public channels or sidebar maintainers on individual projects via PMs. If you're interested in contributing to Nav2, this is a great place to join!

If you're looking to contribute code or bugs, please see the Process section below.

Over time, for developers that have an interest and have shown technical competence in an area of the stack, we elevate developers to a maintainers status. That allows push rights to our protected branches, first-reviewers rights, and getting your name on *About and Contact*. There currently is not a clear process for getting to be a maintainer, but if you've been involved and contributing over a duration of several months, you may be a good candidate and should email the project lead listed on *About and Contact*.

Process

After you've introduced yourself in a working group meeting (recommended, not required), you're ready to get started! We recommend a typical open-source project flow and value detail and transparency. If you commit to something and need to pull back, say so. We all know priorities change and appreciate the heads up so that task can go into the open queue of tasks.

The process is simple and is as follow:

1. Create a ticket for any issues or features you'd like to see. You are not required to fix / implement patches required, but it would be helpful. Reporting bugs is also a valuable contribution.
2. If this ticket, or another existing ticket, is something you would like to work on, comment in the ticket claiming ownership over it. It would be helpful at this time if you declared a strategy and a timeline for planning purposes of other folks working around you. Over time, update the ticket with progress of key markers and engage in any constructive feedback maintainers or other users may have.
3. Once you've completed the task you set out to complete, submit a PR! Please fill out the PR template in complete to ensure that we have a full understanding of your work. At that point, 1-2 reviewers will take a look at your work and give it some feedback to be merged into the codebase. For trivial changes, a single maintainer may merge it after review if they're happy with it, up to their discretion. Any substantial changes should be approved by at least 1 maintainer and 1 other community member.

Note: We take code quality seriously and strive for high-quality and consistent code. We make use of the linting and static analysis tools provided in ROS 2 (`ament_cpplint`, `ament_uncrustify`, `ament_cppcheck`, etc). All PRs are built in CI with the appropriate ROS distributions and run through a set of unit and system level tests including static analysis. You can see the results of these tests in the pull request. It is expected for feature development for tests to cover this work to be added. If any documentation must be updated due to your changes, that should be included in your pull request.

Licensing

Licensing is very important to open source projects. It helps ensure the software continues to be available under the terms that the author desired.

Because much of the source code is ported from other ROS 1 projects, each package has it's own license. Contributions should be made under the predominant license of that package. Entirely new packages should be made available under the Apache 2.0 license.

A license tells you what rights you have as a developer, as provided by the copyright holder. It is important that the contributor fully understands the licensing rights and agrees to them. Sometimes the copyright holder isn't the contributor, such as when the contributor is doing work on behalf of a company.

If for some reason Apache 2.0 or BSD licenses are not appropriate for your work, please get in contact with a project maintainer and discuss your concerns or requirements. We may consider special exceptions for exceptional work, within reason (we will not accept any licenses that makes it unsuitable for commercial use).

Developer Certification of Origin (DCO)

To make a good faith effort to ensure licensing criteria are met, Nav2 encourages the Developer Certificate of Origin (DCO) process to be followed.

The DCO is an attestation attached to every contribution made by a developer. In the commit message of the contribution, (described more fully later in this document), the developer simply adds a `Signed-off-by` statement and thereby agrees to the DCO.

In practice, its easier to just `git commit -s -m "commit messsage."`. Where `-s` adds this automatically. If you forgot to add this to a commit, it is easy to append via: `git commit --amend -s`.

When a developer submits a patch, it is a commitment that the contributor has the right to submit the patch per the license. The DCO agreement is shown below and at http://developercertificate.org/.

```
Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the
    best of my knowledge, is covered under an appropriate open
    source license and I have the right under that license to
    submit that work with modifications, whether created in whole
    or in part by me, under the same open source license (unless
    I am permitted to submit under a different license), as
    Indicated in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including
    all personal information I submit with it, including my
    sign-off) is maintained indefinitely and may be redistributed
    consistent with this project or the open source license(s)
    involved.
```

4.3 Navigation Concepts

This page is to help familiarize new roboticists to the concepts of mobile robot navigation, in particular, with the concepts required to appreciating and working with this project.

4.3.1 ROS 2

ROS 2 is the core middleware used for Nav2. If you are unfamiliar with this, please visit the ROS 2 documentation before continuing.

Action Server

Just as in ROS, action servers are a common way to control long running tasks like navigation. This stack makes more extensive use of actions, and in some cases, without an easy topic interface. It is more important to understand action servers as a developer in ROS 2. Some simple CLI examples can be found in the ROS 2 documentation.

Action servers are similar to a canonical service server. A client will request some task to be completed, except, this task may take a long time. An example would be moving the shovel up from a bulldozer or ask a robot to travel 10 meters to the right.

In this situation, action servers and clients allow us to call a long-running task in another process or thread and return a future to its result. It is permissible at this point to block until the action is complete, however, you may want to occasionally check if the action is complete and continue to process work in the client thread. Since it is long-running, action servers will also provide feedback to their clients. This feedback can be anything and is defined in the ROS `.action` along with the request and result types. In the bulldozer example, a request may be an angle, a feedback may be the angle remaining to be moved, and the result is a success or fail boolean with the end angle. In the navigation example, a request may be a position, a feedback may be the time its been navigating for and the distance to the goal, and the result a boolean for success.

Feedback and results can be gathered synchronously by registering callbacks with the action client. They may also be gathered by asynchronously requesting information from the shared future objects. Both require spinning the client node to process callback groups.

Action servers are used in this stack to communicate with the highest level Behavior Tree (BT) navigator through a `NavigateToPose` action message. They are also used for the BT navigator to communicate with the subsequent smaller action servers to compute plans, control efforts, and recoveries. Each will have their own unique `.action` type in `nav2_msgs` for interacting with the servers.

Lifecycle Nodes and Bond

Lifecycle (or Managed, more correctly) nodes are unique to ROS 2. More information can be found here. They are nodes that contain state machine transitions for bringup and teardown of ROS 2 servers. This helps in deterministic behavior of ROS systems in startup and shutdown. It also helps users structure their programs in reasonable ways for commercial uses and debugging.

When a node is started, it is in the unconfigured state, only processing the node's constructor which should **not** contain any ROS networking setup or parameter reading. By the launch system, or the supplied lifecycle manager, the nodes need to be transitioned to inactive by configuring. After, it is possible to activate the node by transitioning through the activating stage.

This state will allow the node to process information and be fully setup to run. The configuration stage, triggering the `on_configure()` method, will setup all parameters, ROS networking interfaces, and for safety systems, all dynamically allocated memory. The activation stage, triggering the `on_activate()` method, will active the ROS networking interfaces and set any states in the program to start processing information.

To shutdown, we transition into deactivating, cleaning up, shutting down and end in the finalized state. The networking interfaces are deactivated and stop processing, deallocate memory, exit cleanly, in those stages, respectively.

The lifecycle node framework is used extensively through out this project and all servers utilize it. It is best convention for all ROS systems to use lifecycle nodes if it is possible.

Within Nav2, we use a wrapper of LifecycleNodes, `nav2_util LifecycleNode`. This wrapper wraps much of the complexities of LifecycleNodes for typical applications. It also includes a `bond` connection for the lifecycle manager to ensure that after a server transitions up, it also remains active. If a server crashes, it lets the lifecycle manager know and transition down the system to prevent a critical failure. See *Eloquent to Foxy* for details.

4.3.2 Behavior Trees

Behavior trees (BT) are becoming increasingly common in complex robotics tasks. They are a tree structure of tasks to be completed. It creates a more scalable and human-understandable framework for defining multi-step or many state applications. This is opposed to a finite state machine (FSM) which may have dozens of states and hundreds of transitions. An example would be a soccer-playing robot. Embedding the logic of soccer game play into a FSM would be challenging and error prone with many possible states and rules. Additionally, modeling choices like to shoot at the goal from the left, right, or center, is particularly unclear. With a BT, basic primitives, like "kick", "walk", "go to ball", can be created and reused for many behaviors. More information can be found in this book. I **strongly** recommend reading chapters 1-3 to get a good understanding of the nomenclature and workflow. It should only take about 30 minutes.

Behavior Trees provide a formal structure for navigation logic which can be both used to create complex systems but also be verifiable and validated as provenly correct using advanced tools. Having the application logic centralized in the behavior tree and with independent task servers (which only communicate data over the tree) allows for formal analysis.

For this project, we use BehaviorTree CPP V3 as the behavior tree library. We create node plugins which can be constructed into a tree, inside the `BT Navigator`. The node plugins are loaded into the BT and when the XML file of the tree is parsed, the registered names are associated. At this point, we can march through the behavior tree to navigate.

One reason this library is used is its ability to load subtrees. This means that the Nav2 behavior tree can be loaded into another higher-level BT to use this project as node plugin. An example would be in soccer play, using the Nav2 behavior tree as the "go to ball" node with a ball detection as part of a larger task. Additionally, we supply a `NavigateToPoseAction` plugin (among others) for BT so the Nav2 stack can be called from a client application through the usual action interface.

Other systems could be used to design complex autonomous behavior, namely Hierarchical FSMs (HFSM). Behavior Trees were selected due to popularity across the robotics and related industries and by largely user demand. However, due to the independent task server nature of Nav2, it is not difficult to offer a `nav2_hfsm_navigator` package in the future, pending interest and contribution.

4.3.3 Navigation Servers

Planners and controllers are at the heart of a navigation task. Recoveries are used to get the robot out of a bad situation or attempt to deal with various forms of issues to make the system fault-tolerant. Smoothers can be used for additional quality improvements of the planned path. In this section, the general concepts around them and their uses in this project are analyzed.

Planner, Controller, Smoother and Recovery Servers

Four of the action servers in this project are the planner, behavior, smoother and controller servers.

These action servers are used to host a map of algorithm plugins to complete various tasks. They also host the environmental representation used by the algorithm plugins to compute their outputs.

The planner, smoother and controller servers will be configured at runtime with the names (aliases) and types of algorithms to use. These types are the pluginlib names that have been registered and the names are the aliases for the task. An example would be the DWB controller used with name `FollowPath`, as it follows a reference path. In this case, then all parameters for DWB would be placed in that namespace, e.g. `FollowPath.<param>`.

These three servers then expose an action interface corresponding to their task. When the behavior tree ticks the corresponding BT node, it will call the action server to process its task. The action server callback inside the server will call the chosen algorithm by its name (e.g. `FollowPath`) that maps to a specific algorithm. This allows a user to abstract the algorithm used in the behavior tree to classes of algorithms. For instance, you can have `N` plugin controllers to follow paths, dock with charger, avoid dynamic obstacles, or interface with a tool. Having all of these plugins in the same server allows the user to make use of a single environmental representation object, which is costly to duplicate.

For the behavior server, each of the behaviors also contains their own name, however, each plugin will also expose its own special action server. This is done because of the wide variety of behavior actions that may be created which cannot have a single simple interface to share. The behavior server also contains a costmap subscriber to the local costmap, receiving real-time updates from the controller server, to compute its tasks. We do this to avoid having multiple instances of the local costmap which are computationally expensive to duplicate.

Alternatively, since the BT nodes are trivial plugins calling an action, new BT nodes can be created to call other action servers with other action types. It is advisable to use the provided servers if possible at all times. If, due to the plugin or action interfaces, a new server is needed, that can be sustained with the framework. The new server should use the new type and plugin interface, similar to the provided servers. A new BT node plugin will need to be created to call the new action server – however no forking or modification is required in the Nav2 repo itself by making extensive use of servers and plugins.

If you find that you require a new interface to the pluginlib definition or action type, please file a ticket and see if we can rectify that in the same interfaces.

Planners

The task of a planner is to compute a path to complete some objective function. The path can also be known as a route, depending on the nomenclature and algorithm selected. Two canonical examples are computing a plan to a goal (e.g. from current position to a goal) or complete coverage (e.g. plan to cover all free space). The planner will have access to a global environmental representation and sensor data buffered into it. Planners can be written to:

- Compute shortest path
- Compute complete coverage path
- Compute paths along sparse or predefined routes

The general task in Nav2 for the planner is to compute a valid, and potentially optimal, path from the current pose to a goal pose. However, many classes of plans and routes exist which are supported.

Controllers

Controllers, also known as local planners in ROS 1, are the way we follow the globally computed path or complete a local task. The controller will have access to a local environment representation to attempt to compute feasible control efforts for the base to follow. Many controller will project the robot forward in space and compute a locally feasible path at each update iteration. Controllers can be written to:

- Follow a path
- Dock with a charging station using detectors in the odometric frame
- Board an elevator
- Interface with a tool

The general task in Nav2 for a controller is to compute a valid control effort to follow the global plan. However, many classes of controllers and local planners exist. It is the goal of this project that all controller algorithms can be plugins in this server for common research and industrial tasks.

Behaviors

Recovery behaviors are a mainstay of fault-tolerant systems. The goal of recoveries are to deal with unknown or failure conditions of the system and autonomously handle them. Examples may include faults in the perception system resulting in the environmental representation being full of fake obstacles. The clear costmap recovery would then be triggered to allow the robot to move.

Another example would be if the robot was stuck due to dynamic obstacles or poor control. Backing up or spinning in place, if permissible, allow the robot to move from a poor location into free space it may navigate successfully.

Finally, in the case of a total failure, a recovery may be implemented to call an operator's attention for help. This can be done via email, SMS, Slack, Matrix, etc.

It is important to note that the behavior server can hold any behavior to share access to expensive resources like costmaps or TF buffers, not just recovery behaviors. Each may have its own API.

Smoothers

As criteria for optimality of the path searched by a planner are usually reduced compared to reality, additional path refinement is often beneficial. Smoothers have been introduced for this purpose, typically responsible for reducing path raggedness and smoothing abrupt rotations, but also for increasing distance from obstacles and high-cost areas as the smoothers have access to a global environmental representation.

Use of a separate smoother over one that is included as part of a planner is advantageous when combining different planners with different smoothers or when a specific control over smoothing is required, e.g. smoothing only a specific part of the path.

The general task in Nav2 for a smoother is to receive a path and return its improved version. However, for different input paths, criteria of the improvements and methods of acquiring them exist, creating space for a multitude of smoothers that can be registered in this server.

Waypoint Following

Waypoint following is a basic feature of a navigation system. It tells our system how to use navigation to get to multiple destinations.

The `nav2_waypoint_follower` contains a waypoint following program with a plugin interface for specific task executors. This is useful if you need to go to a given location and complete a specific task like take a picture, pick up a box, or wait for user input. It is a nice demo application for how to use Nav2 in a sample application.

However, it could be used for more than just a sample application. There are 2 schools of thoughts for fleet managers / dispatchers:

- Dumb robot; smart centralized dispatcher
- Smart robot; dumb centralized dispatcher

In the first, the `nav2_waypoint_follower` is fully sufficient to create a production-grade on-robot solution. Since the autonomy system / dispatcher is taking into account things like the robot's pose, battery level, current task, and more when assigning tasks, the application on the robot just needs to worry about the task at hand and not the other complexities of the system to complete the requested task. In this situation, you should think of a request to the waypoint follower as 1 unit of work (e.g. 1 pick in a warehouse, 1 security patrole loop, 1 aisle, etc) to do a task and then return to the dispatcher for the next task or request to recharge. In this school of thought, the waypoint following application is just one step above navigation and below the system autonomy application.

In the second, the `nav2_waypoint_follower` is a nice sample application / proof of concept, but you really need your waypoint following / autonomy system on the robot to carry more weight in making a robust solution. In this case, you should use the `nav2_behavior_tree` package to create a custom application-level behavior tree using navigation to complete the task. This can include subtrees like checking for the charge status mid-task for returning to dock or handling more than 1 unit of work in a more complex task. Soon, there will be a `nav2_bt_waypoint_follower` (name subject to adjustment) that will allow you to create this application more easily. In this school of thought, the waypoint following application is more closely tied to the system autonomy, or in many cases, is the system autonomy.

Neither is better than the other, it highly depends on the tasks your robot(s) are completing, in what type of environment, and with what cloud resources available. Often this distinction is very clear for a given business case.

4.3.4 State Estimation

Within the navigation project, there are 2 major transformations that need to be provided, according to community standards. The `map` to `odom` transform is provided by a positioning system (localization, mapping, SLAM) and `odom` to `base_link` by an odometry system.

Note: There is **no** requirement on using a LIDAR on your robot to use the navigation system. There is no requirement to use lidar-based collision avoidance, localization, or SLAM. However, we do provide instructions and support tried and true implementations of these things using lidars. You can be equally as successful using a vision or depth based positioning system and using other sensors for collision avoidance. The only requirement is that you follow the standards below with your choice of implementation.

Standards

REP 105 defines the frames and conventions required for navigation and the larger ROS ecosystem. These conventions should be followed at all times to make use of the rich positioning, odometry, and SLAM projects available in the community.

In a nutshell, REP-105 says that you must, at minimum, build a TF tree that contains a full `map` -> `odom` -> `base_link` -> `[sensor frames]` for your robot. TF2 is the time-variant transformation library in ROS 2 we use to represent and obtain time synchronized transformations. It is the job of the global positioning system (GPS, SLAM, Motion Capture) to, at minimum, provide the `map` -> `odom` transformation. It is then the role of the odometry system to provide the `odom` -> `base_link` transformation. The remainder of the transformations relative to `base_link` should be static and defined in your URDF.

Global Positioning: Localization and SLAM

It is the job of the global positioning system (GPS, SLAM, Motion Capture) to, at minimum, provide the `map` -> `odom` transformation. We provide `amcl` which is an Adaptive Monte-Carlo Localization technique based on a particle filter for localization in a static map. We also provide SLAM Toolbox as the default SLAM algorithm for use to position and generate a static map.

These methods may also produce other output including position topics, maps, or other metadata, but they must provide that transformation to be valid. Multiple positioning methods can be fused together using robot localization, discussed more below.

Odometry

It is the role of the odometry system to provide the `odom` -> `base_link` transformation. Odometry can come from many sources including LIDAR, RADAR, wheel encoders, VIO, and IMUs. The goal of the odometry is to provide a smooth and continuous local frame based on robot motion. The global positioning system will update the transformation relative to the global frame to account for the odometric drift.

Robot Localization is typically used for this fusion. It will take in `N` sensors of various types and provide a continuous and smooth odometry to TF and to a topic. A typical mobile robotics setup may have odometry from wheel encoders, IMUs, and vision fused in this manner.

The smooth output can be used then for dead-reckoning for precise motion and updating the position of the robot accurately between global position updates.

4.3.5 Environmental Representation

The environmental representation is the way the robot perceives its environment. It also acts as the central localization for various algorithms and data sources to combine their information into a single space. This space is then used by the controllers, planners, and recoveries to compute their tasks safely and efficiently.

Costmaps and Layers

The current environmental representation is a costmap. A costmap is a regular 2D grid of cells containing a cost from unknown, free, occupied, or inflated cost. This costmap is then searched to compute a global plan or sampled to compute local control efforts.

Various costmap layers are implemented as pluginlib plugins to buffer information into the costmap. This includes information from LIDAR, RADAR, sonar, depth images, etc. It may be wise to process sensor data before inputting it into the costmap layer, but that is up to the developer.

Costmap layers can be created to detect and track obstacles in the scene for collision avoidance using camera or depth sensors. Additionally, layers can be created to algorithmically change the underlying costmap based on some rule or heuristic. Finally, they may be used to buffer live data into the 2D or 3D world for binary obstacle marking.

Costmap Filters

Imagine, you're annotating a map file (or any image file) in order to have a specific action occur based on the location in the annotated map. Examples of marking/annotating might be keep out zones to avoid planning inside, or have pixels belong to maximum speeds in marked areas. This annotated map is called "filter mask". Just like a mask overlaid on a surface, it can or cannot be same size, pose and scale as a main map. The main goal of filter mask - is to provide the ability of marking areas on maps with some additional features or behavioral changes.

Costmap filters are a costmap layer-based approach of applying spatial-dependent behavioral changes, annotated in filter masks, into the Nav2 stack. Costmap filters are implemented as costmap plugins. These plugins are called "filters" as they are filtering a costmap by spatial annotations marked on filter masks. In order to make a filtered costmap and change a robot's behavior in annotated areas, the filter plugin reads the data coming from the filter mask. This data is being linearly transformed into a feature map in a filter space. Having this transformed feature map along with a map/costmap, any sensor data and current robot coordinate filters can update the underlying costmap and change the behavior of the robot depending on where it is. For example, the following functionality could be made by use of costmap filters:

- Keep-out/safety zones where robots will never enter.
- Speed restriction areas. Maximum speed of robots going inside those areas will be limited.
- Preferred lanes for robots moving in industrial environments and warehouses.

Other Forms

Various other forms of environmental representations exist. These include:

- gradient maps, which are similar to costmaps but represent surface gradients to check traversibility over
- 3D costmaps, which represent the space in 3D, but then also requires 3D planning and collision checking
- Mesh maps, which are similar to gradient maps but with surface meshes at many angles
- "Vector space", taking in sensor information and using machine learning to detect individual items and locations to track rather than buffering discrete points.

4.3.6 Nav2 Academic Overview

4.4 First-Time Robot Setup Guide

This section is a collection of guides that aims to provide readers a good resource for setting up Nav2. The objectives for this section are as follows:

- Help new users with setting up Navigation2 with a new robot
- Help people with custom built robots to properly set up their robots to be used in ROS/Navigation2
- Act as a checklist, template or boilerplate reference for more experienced readers
- Provide examples which can be run on simulators/tools like Gazebo or RViz to guide readers on the Nav2 setup process even without a physical robot.
- Broad strokes, tips, and tricks for configuring certain packages and integrating different components of the robot platform (sensors, odometry, etc.)

To guide you through the first-time setup of your robot, we will be tackling the following topics:

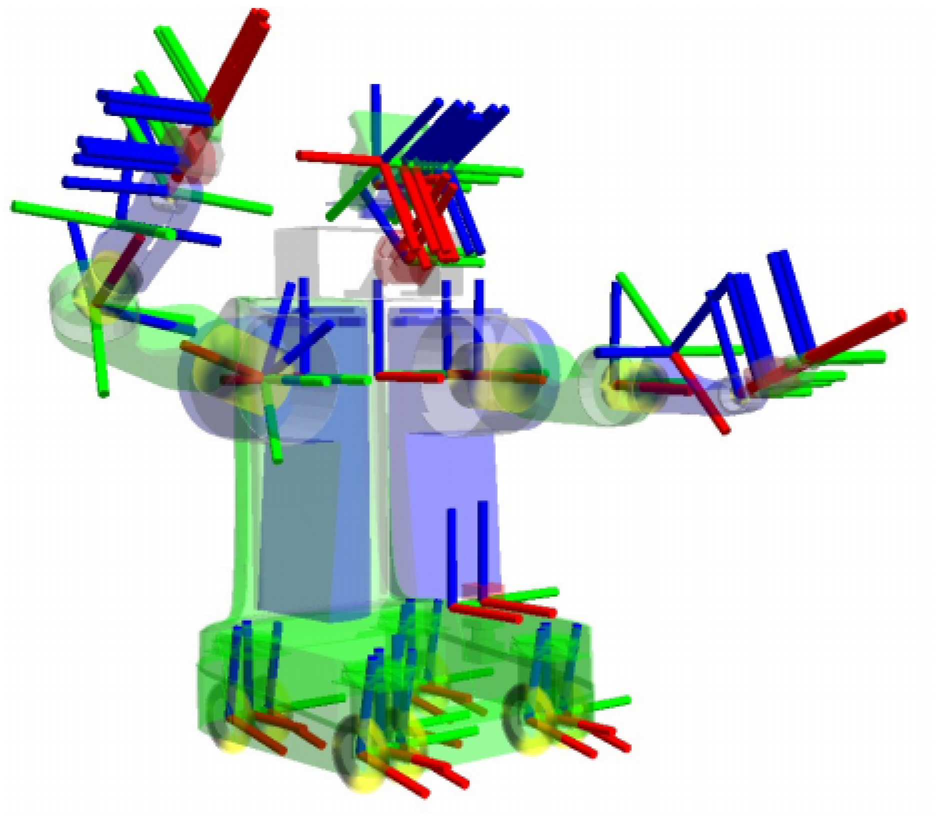
- Introduce TF2 and setup your robot URDF
- Setup sensor sources for robot odometry
- Setup sensor sources for perception
- Configure round or arbitrary shaped footprints for your robot
- Select and set up planner and controller navigation plugins for your robot's navigation tasks
- Lifecycle node management for easy bringup of other related sensors or nodes

Note: These tutorials are not meant to be full tuning and configuration guides since they only aim to help you get your robot up and running with a basic configuration. For more detailed discussions and guides on how to customize and tune Nav2 for your robot, head on to the *Configuration Guide* section.

Table of Contents:

4.4.1 Setting Up Transformations

In this guide, we will be looking at the necessary transforms required by Nav2. These transforms allow Nav2 to interpret information coming in from various sources, such as sensors and odometry, by transforming them to the coordinate frames for use. Below is what a full transform tree for a robot looks like but we'll start with something much more simpler.



For this tutorial, we will first provide a brief introduction to transforms in ROS. Second, we will be working on a simple command-line demo of a TF2 static publisher to see it in action. Lastly, we will outline the necessary transforms that need to be published for Nav2 to function.

Transforms Introduction

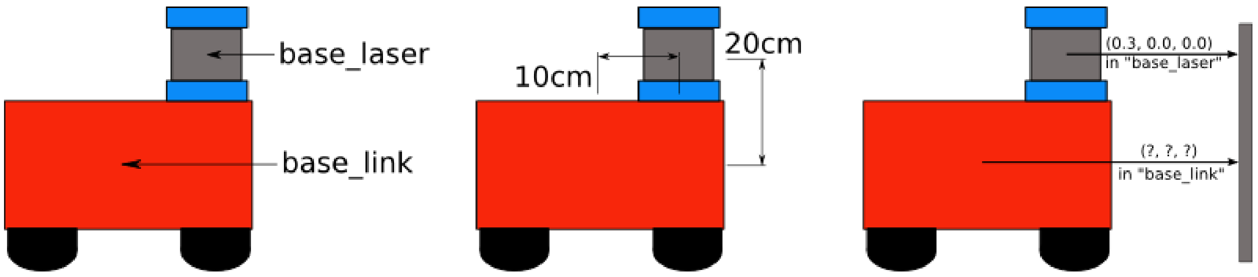
Note: This section of this guide has been adapted from the Setting Up You Robot using tf tutorial in the ROS (1) Navigation documentation.

Many ROS packages require the transform tree of a robot to be published using the TF2 ROS package. A transformation tree defines the relations between different coordinate systems, in terms of translation, rotation, and relative motion. To make this more concrete, let us apply an example of a simple robot that has a mobile base with a single laser sensor mounted on top of it.

This robot has two defined coordinate frames: one corresponding to the center point of the mobile base of the robot, and one for the center point of the laser that is mounted on top of the base. We'll call the coordinate frame attached to the mobile base `base_link` and we'll call the coordinate frame attached to the laser `base_laser`. Note that will be talking more about the naming and conventions of these coordinate frames in the next section.

At this point, let's assume that we have some data from the laser in the form of distance measurements from the laser's center point. In other words, we have some data in the `base_laser` coordinate frame.

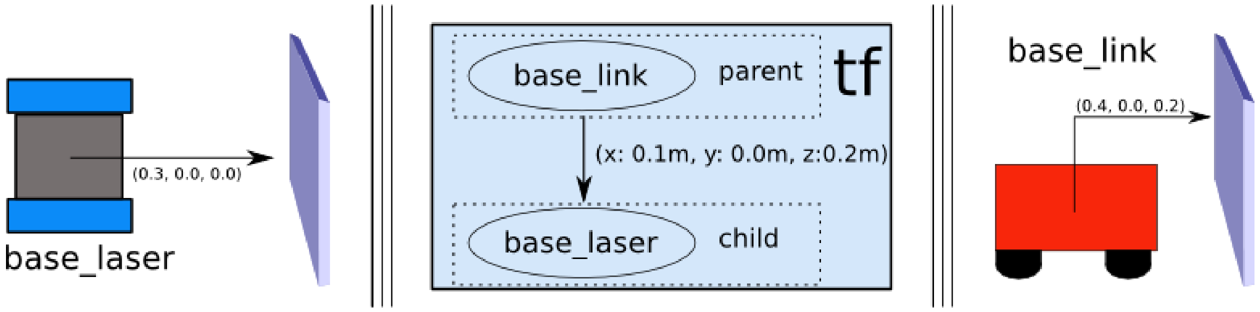
Now, suppose we want to take this data and use it to help the mobile base avoid obstacles in the world. To do this successfully, we need a way to transform the laser scan we've received from the `base_laser` frame to the `base_link` frame. In essence, we need to define a relationship between the `base_laser` and `base_link` coordinate frames.



In defining this relationship, let us assume that the only data we have is that the laser is mounted 10cm forward and 20cm above the center point of the mobile base. This gives us a translational offset that relates the `base_link` frame to the `base_laser` frame. Specifically, we know that to get data from the `base_link` frame to the `base_laser` frame, we must apply a translation of (x: 0.1m, y: 0.0m, z: 0.2m), and transversely, to get data from the `base_laser` frame to the `base_link` frame, we must apply the opposite translation (x: -0.1m, y: 0.0m, z: -0.20m).

We could choose to manage this relationship ourselves, meaning to store and apply the appropriate translations between the frames when necessary, but this becomes a real pain as the number of coordinate frames increases. Luckily, we don't have to do this work ourselves. Instead, we'll define the relationship between `base_link` and `base_laser` once using TF2 and let it manage the transformation between the two coordinate frames for us. This is especially useful when working with non-static transformations, such as a set of frames that are moving relative to each other, like a robot base frame in a map frame.

To define and store the relationship between the `base_link` and `base_laser` frames using TF2, we need to add them to a transform tree. Conceptually, each node in the transform tree corresponds to a coordinate frame, and each edge corresponds to the transform that needs to be applied to move from the current node to its child. TF2 uses a tree structure to guarantee that there is only a single traversal that links any two coordinate frames together, and assumes that all edges in the tree are directed from parent to child nodes.



To create a transform tree for our simple example, we'll create two nodes: one for the `base_link` coordinate frame and one for the `base_laser` coordinate frame. To create the edge between them, we first need to decide which node will be the parent and which will be the child. Remember — this distinction is important because TF2 assumes that all transforms move from parent to child.

Let's choose the `base_link` coordinate frame as the parent because when other pieces/sensors are added to the robot, it will make the most sense for them to relate to the `base_laser` frame by traversing through the `base_link` frame. This means that the transform associated with the edge connecting `base_link` and `base_laser` should be (x: 0.1m, y: 0.0m, z: 0.2m).

With this transform tree set up, converting the laser scan received in the `base_laser` frame to the `base_link` frame is as simple as making a call to the TF2 library. Our robot can now use this information to reason about laser scans in the `base_link` frame and safely plan around obstacles in its environment.

Static Transform Publisher Demo

Warning: If you are new to ROS 2 or do not have a working environment yet, then please take some time to properly setup your machine using the resources in the official ROS 2 Installation Documentation

Now let's try publishing a very simple transform using the static_transform_publisher tool provided by TF2. We will be publishing a transformation from the link `base_link` to the link `base_laser` with a translation of (x: 0.1m, y: 0.0m, z: 0.2m). Note that we will be building the transform from the diagram earlier in this tutorial.

Open up your command line and execute the following command:

```
ros2 run tf2_ros static_transform_publisher 0.1 0 0.2 0 0 0 base_link base_laser
```

With this, we are now sucessfully publishing our `base_link` to `base_laser` transform in TF2. Let us now check if it is working properly through `tf2_echo`. Open up a separate command line window and execute the following:

```
ros2 run tf2_ros tf2_echo base_link base_laser
```

You should be able to observe a repeated output simiar to the one below.

```
At time 0.0
- Translation: [0.100, 0.000, 0.200]
- Rotation: in Quaternion [0.000, 0.000, 0.000, 1.000]
```

And that's it for this short demo - we were able to successfully publish a transform from `base_link` to `base_laser` using the TF2 library. Note that we do not recommend using the above demo in publishing transforms for your actual robotics projects, it is just a quick demo to see TF2 in action. For a real robot system, we would create a URDF file which embeds this information and more about your robot for use of the robot_state_publisher rather than the static_transform_publisher. There are more suitable and practical ways to go about this which will be discussed in the *Setting Up The URDF* tutorial.

See also:

If you would like to learn more about TF2 and how to create your own transform publishers, head onto the official TF2 Documentation

Transforms in Navigation2

There are two important ROS REPs which we highly suggest for you to check out. These documents detail some standards set about by the ROS community to ensure proper operation across different packages. Nav2 also adheres to these standards and conventions.

1. REP 105 - Coordinate Frames for Mobile Platforms
2. REP 103 - Standard Units of Measure and Coordinate Conventions

To quickly summarize REP 105, this document specifies the naming conventions and semantic meanings of the different coordinate frames used in ROS. Of interest to this tutorial are the `base_link`, `odom` and `map` coordinate frames. The `base_link` is a coordinate frame that is attached to a fixed position on the robot, typically at its main chassis and its rotational center. The `odom` coordinate frame is a fixed frame relative to the robot's starting position and is mainly used for locally-consistent representations of distances. Lastly, the `map` coordinate frame is a world fixed frame that is used for globally-consistent representations of distances.

REP 103, on the other hand, discusses some standard units of measure and other related conventions to keep integration issues between different ROS packages to a minimum. The basic overview is that frames are defined using the right hand rule, with Z up and X forward, and units should be standard SI units.

Now let's move on to some specifics for the Navigation2 package to function correctly. Nav2 requires the following transformations to be published in ROS:

1. `map` => `odom`
2. `odom` => `base_link`
3. `base_link` => `base_laser` (sensor base frames)

Note: The `base_laser` coordinate frame is not included in the REP 105 standard. For this guide, we will be using this name to refer to the coordinate frame for a laser sensor on our robot platform. If there are multiple sensor base frames (e.g. camera_link, base_laser2, lidar_link etc.), then a transformation back to `base_link` for each one is required.

The first transform `map => odom` is usually provided by a different ROS package dealing with localization and mapping such as AMCL. This transform updates live in use so we don't set static values for this in our robot's TF tree. Further detail about how to set this up may be pretty complex, so we highly suggest to have a look at the documentation of the mapping or localization package you are using for your platform. All ROS compliant SLAM and localization packages will provide you with this transformation automatically on launch.

The `odom` => `base_link` is usually published by our odometry system using sensors such as wheel encoders. This is typically computed via sensor fusion of odometry sensors (IMU, wheel encoders, VIO, etc) using the `robot_localization` package.

All other statically defined transforms (e.g. `base_link` => `base_laser`, `base_link` => `wheels`, `wheels` => `IMU`, etc) is what we will be talking about for the rest of this guide. This transformation tree is used by Nav2 to properly relate the information from sensors or other frame of interest to the rest of the robot. The transformation between these two coordinate frames is usually provided to Nav2 through the Robot State Publisher and the Universal Robot Descriptor File (URDF). In cases where there are more sensor coordinate frames on your platform, then a transform tree from `base_link` to each sensor coordinate frame needs to be published.

See also:

For a more in-depth discussion on the usage of transforms and how these are used to estimate the current state of your robot, we highly recommend having a look at the State Estimation topic in *Navigation Concepts*.

Conclusion

In this tutorial, we have discussed about the concept of transforms and how they are used in Nav2.

In the last section, we have also explored using the static_transform_publisher of TF2 to publish our transforms. You may use this to set up your transforms for Nav2, but this is generally not the best way to do it. In most robotics projects, we make use of the Robot State Publisher since it is much easier to use and scales well as our robot gets more complex. We will be talking about the Robot State Publisher, URDF, and how to set it up in the next tutorial on *Setting Up The URDF*.

Lastly, we also discussed the three published transform requirements of Nav2 and the neccessary REPs to keep in mind when setting them up.

4.4.2 Setting Up The URDF

For this guide, we will be creating the Unified Robot Description Format (URDF) file for a simple differential drive robot to give you hands-on experience on working with URDF. We will also setup the robot state publisher and visualize our model in RVIZ. Lastly, we will be adding some kinematic properties to our robot URDF to prepare it for simulation purposes. These steps are necessary to represent all the sensor, hardware, and robot transforms of your robot for use in navigation.

See also:

The complete source code in this tutorial can be found in navigation2_tutorials repository under the `sam_bot_description` package. Note that the repository contains the full code after accomplishing all the tutorials in this guide.

URDF and the Robot State Publisher

As discussed in the previous tutorial, one of the requirements for Navigation2 is the transformation from `base_link` to the various sensors and reference frames. This transformation tree can range from a simple tree with only one link from the `base_link` to `laser_link` or a tree comprised of multiple sensors located in different locations, each having their own coordinate frame. Creating multiple publishers to handle all of these coordinate frame transformations may become tedious. Therefore, we will be making use of the Robot State Publisher package to publish our transforms.

The Robot State Publisher is a package of ROS 2 that interacts with the tf2 package to publish all of the necessary transforms that can be directly inferred from the geometry and structure of the robot. We need to provide it with the correct URDF and it will automatically handle publishing the transforms. This is very useful for complex transformations but it is still recommended for simpler transform trees.

The Unified Robot Description Format (URDF) is an XML file that represents a robot model. In this tutorial, it will mainly be used to build transformations trees related with the robot geometry, but it also has other uses. One example is how it can be used in visualizing your robot model in RVIZ, a 3D Visualization tool for ROS, by defining visual components such as materials and meshes. Another example is how the URDF can be used to define the physical properties of the robot. These properties are then used in physics simulators such as Gazebo to simulate how your robot will interact in an environment.

Another major feature of URDF is that it also supports Xacro (XML Macros) to help you create a shorter and readable XML to help in defining complex robots. We can use these macros to eliminate the need for repeating blocks of XML in our URDF. Xacro is also useful in defining configuration constants which can be reused throughout the URDF.

See also:

If you want to learn more about the URDF and the Robot State Publisher, we encourage you to have a look at the official URDF Documentation and Robot State Publisher Documentation

Setting Up the Environment

In this guide, we are assuming that you are already familiar with ROS 2 and how to setup your development environment, so we'll breeze through the steps in this section.

Let's begin by installing some additional ROS 2 packages that we will be using during this tutorial.

```
sudo apt install ros-<ros2-distro>-joint-state-publisher-gui
sudo apt install ros-<ros2-distro>-xacro
```

Next, create a directory for your project, initialize a ROS 2 workspace and give your robot a name. For ours, we'll be calling it `sam_bot`.

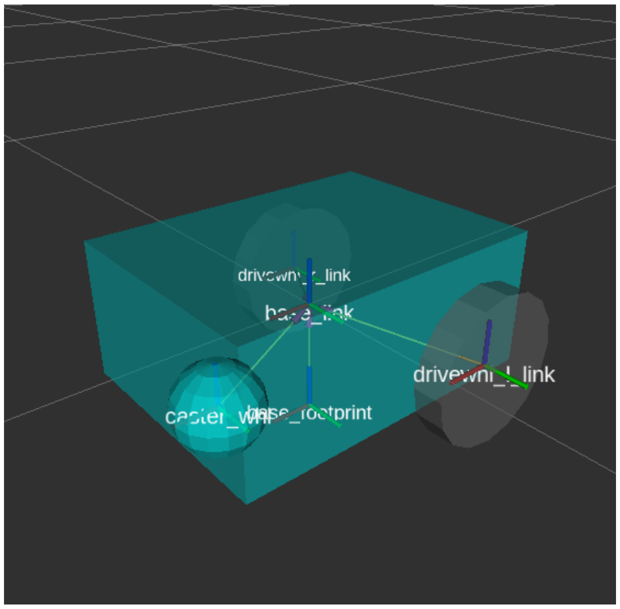
```
ros2 pkg create --build-type ament_cmake sam_bot_description
```

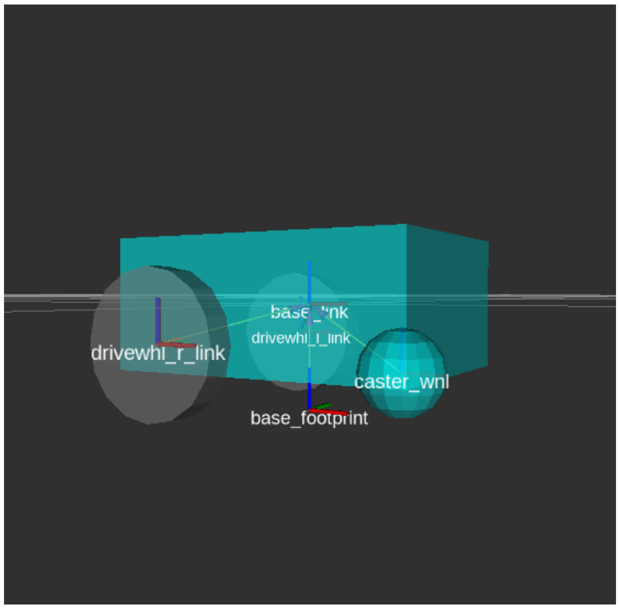
Writing the URDF

See also:

This section aims to provide you with a beginner-friendly introduction to building URDFs for your robot. If you would like to learn more about URDF and XAcro, we suggest for you to have a look at the official URDF Documentation

Now that we have our project workspace set up, let's dive straight into writing the URDF. Below is an image of the robot we will be trying to build.





To get started, create a file named sam_bot_description.urdf under src/description and input the following as the initial contents of the file.

```
<?xml version="1.0"?>
<robot name="sam_bot" xmlns:xacro="http://ros.org/wiki/xacro">



</robot>
```

Note: The following code snippets should be placed within the <robot> tags. We suggest to add them in the same order as introduced in this tutorial. We have also included some line numbers to give you a rough idea on where to input the code. This may differ from the actual file you are writing depending on your usage of whitespaces. Also note that the line numbers assume that you are putting in code as they appear in this guide.

Next, let us define some constants using XAcro properties that will be reused throughout the URDF.

```
4    <!-- Define robot constants -->
5    <xacro:property name="base_width" value="0.31"/>
6    <xacro:property name="base_length" value="0.42"/>
7    <xacro:property name="base_height" value="0.18"/>
8
9    <xacro:property name="wheel_radius" value="0.10"/>
10   <xacro:property name="wheel_width" value="0.04"/>
11   <xacro:property name="wheel_ygap" value="0.025"/>
12   <xacro:property name="wheel_zoff" value="0.05"/>
13   <xacro:property name="wheel_xoff" value="0.12"/>
14
15   <xacro:property name="caster_xoff" value="0.14"/>
```

Here is a brief discussion on what these properties will represent in our urdf. The base_* properties all define the size of the robot's main chassis. The wheel_radius and wheel_width define the shape of the robot's two back wheels. The wheel_ygap adjusts the gap between the wheel and the chassis along the y-axis whilst wheel_zoff and wheel_xoff position the back wheels along the z-axis and x-axis appropriately. Lastly, the caster_xoff positions the front caster wheel along the x-axis.

Let us then define our base_link - this link will be a large box and will act as the main chassis of our robot. In URDF, a link element describes a rigid part or component of our robot. The robot state publisher then utilizes these definitions to determine coordinate frames for each link and publish the transformations between them.

We will also be defining some of the link's visual properties which can be used by tools such as Gazebo and Rviz to show us a 3D model of our robot. Amongst these properties are <geometry> which describes the link's shape and <material> which describes it's color.

For the code block block below, we access the base properties from the robot constants sections we defined before using the ${property} syntax. In addition, we also set the material color of the main chassis to Cyan. Note that we set these parameters under the <visual> tag so they will only be applied as visual parameters which dont affect any collision or physical properties.

```
17   <!-- Robot Base -->
18   <link name="base_link">
19     <visual>
20       <geometry>
21         <box size="${base_length} ${base_width} ${base_height}"/>
22       </geometry>
23       <material name="Cyan">
```

(continues on next page)

(continued from previous page)

```
      <color rgba="0 1.0 1.0 1.0"/>
    </material>
  </visual>
</link>
```

Next, let us define a `base_footprint` link. The `base_footprint` link is a virtual (non-physical) link which has no dimensions or collision areas. Its primary purpose is to enable various packages determine the center of a robot projected to the ground. For example, Navigation2 uses this link to determine the center of a circular footprint used in its obstacle avoidance algorithms. Again, we set this link with no dimensions and to which position the robot's center is in when it is projected to the ground plane.

After defining our base_link, we then add a joint to connect it to `base_link`. In URDF, a `joint` element describes the kinematic and dynamic properties between coordinate frames. For this case, we will be defining a `fixed` joint with the appropriate offsets to place our `base_footprint` link in the proper location based on the description above. Remember that we want to set our base_footprint to be at the ground plane when projected from the center of the main chassis, hence we get the sum of the `wheel_radius` and the `wheel_zoff` to get the appropriate location along the z-axis.

```
<!-- Robot Footprint -->
<link name="base_footprint"/>

<joint name="base_joint" type="fixed">
  <parent link="base_link"/>
  <child link="base_footprint"/>
  <origin xyz="0.0 0.0 ${-(wheel_radius+wheel_zoff)}" rpy="0 0 0"/>
</joint>
```

Now, we will be adding two large drive wheels to our robot. To make our code cleaner and avoid repetition, we will make use of macros to define a block of code that will be repeated with differing parameters. Our macro will have 3 params: `prefix` which simply adds a prefix to our link and joint names, and `x_reflect` and `y_reflect` which allows us to flip the positions of our wheels with respect to the x and y axis respectively. Within this macro, we can also define the visual properties of a single wheel. Lastly, we will also define a `continuous` joint to allow our wheels to freely rotate about an axis. This joint also connects our wheel to the `base_link` at the appropriate location.

At the end of this code block, we will be instantiating two wheels using the macro we just made through the `xacro:wheel` tags. Note that we also define the parameters to have one wheel on both sides at the back of our robot.

```
<!-- Wheels -->
<xacro:macro name="wheel" params="prefix x_reflect y_reflect">
  <link name="${prefix}_link">
    <visual>
      <origin xyz="0 0 0" rpy="${pi/2} 0 0"/>
      <geometry>
          <cylinder radius="${wheel_radius}" length="${wheel_width}"/>
      </geometry>
      <material name="Gray">
        <color rgba="0.5 0.5 0.5 1.0"/>
      </material>
    </visual>
  </link>

  <joint name="${prefix}_joint" type="continuous">
    <parent link="base_link"/>
    <child link="${prefix}_link"/>
    <origin xyz="${x_reflect*wheel_xoff} ${y_reflect*(base_width/2+wheel_ygap)} ${-
↪wheel_zoff}" rpy="0 0 0"/>
```

(continues on next page)

(continued from previous page)

```
      <axis xyz="0 1 0"/>
    </joint>
  </xacro:macro>

  <xacro:wheel prefix="drivewhl_l" x_reflect="-1" y_reflect="1" />
  <xacro:wheel prefix="drivewhl_r" x_reflect="-1" y_reflect="-1" />
```

Next, we will be adding a caster wheel at the front of our robot. We will be modelling this wheel as a sphere to keep things simple. Again, we define the wheel's geometry, material and the joint to connect it to `base_link` at the appropriate location.

```
  <!-- Caster Wheel -->
  <link name="front_caster">
    <visual>
      <geometry>
        <sphere radius="${(wheel_radius+wheel_zoff-(base_height/2))}"/>
      </geometry>
      <material name="Cyan">
        <color rgba="0 1.0 1.0 1.0"/>
      </material>
    </visual>
  </link>

  <joint name="caster_joint" type="fixed">
    <parent link="base_link"/>
    <child link="front_caster"/>
    <origin xyz="${caster_xoff} 0.0 ${-(base_height/2)}" rpy="0 0 0"/>
  </joint>
```

And that's it! We have built a URDF for a simple differential drive robot. In the next section, we will focus on building the ROS Package containing our URDF, launching the robot state publisher, and visualizing the robot in RVIz.

Build and Launch

See also:

The launch files from this tutorial were adapted from the official URDF Tutorials for ROS 2

Let's start this section by adding some dependencies that will be required once we build this project. Open up the root of your project directory and add the following lines to your `package.xml` (preferably after the `<buildtool_depend>` tag)

```
<exec_depend>joint_state_publisher</exec_depend>
<exec_depend>joint_state_publisher_gui</exec_depend>
<exec_depend>robot_state_publisher</exec_depend>
<exec_depend>rviz</exec_depend>
<exec_depend>xacro</exec_depend>
```

Next, let us create our launch file. Launch files are used by ROS 2 to bring up the necessary nodes for our package. From the root of the project, create a directory named `launch` and a `display.launch.py` file within it. The launch file below launches a robot publisher node in ROS 2 that uses our URDF to publish the transforms for our robot. In addition, the launch file also automatically launches RVIZ so we can visualize our robot as defined by the URDF. Copy and paste the snippet below into your `display.launch.py` file.

```
import launch
from launch.substitutions import Command, LaunchConfiguration
import launch_ros
import os

def generate_launch_description():
    pkg_share = launch_ros.substitutions.FindPackageShare(package='sam_bot_description
↪').find('sam_bot_description')
    default_model_path = os.path.join(pkg_share, 'src/description/sam_bot_description.
↪urdf')
    default_rviz_config_path = os.path.join(pkg_share, 'rviz/urdf_config.rviz')

    robot_state_publisher_node = launch_ros.actions.Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        parameters=[{'robot_description': Command(['xacro ', LaunchConfiguration(
↪'model')])}]
    )
    joint_state_publisher_node = launch_ros.actions.Node(
        package='joint_state_publisher',
        executable='joint_state_publisher',
        name='joint_state_publisher',
        condition=launch.conditions.UnlessCondition(LaunchConfiguration('gui'))
    )
    joint_state_publisher_gui_node = launch_ros.actions.Node(
        package='joint_state_publisher_gui',
        executable='joint_state_publisher_gui',
        name='joint_state_publisher_gui',
        condition=launch.conditions.IfCondition(LaunchConfiguration('gui'))
    )
    rviz_node = launch_ros.actions.Node(
        package='rviz2',
        executable='rviz2',
        name='rviz2',
        output='screen',
        arguments=['-d', LaunchConfiguration('rvizconfig')],
    )

    return launch.LaunchDescription([
        launch.actions.DeclareLaunchArgument(name='gui', default_value='True',
                                            description='Flag to enable joint_state_
↪publisher_gui'),
        launch.actions.DeclareLaunchArgument(name='model', default_value=default_
↪model_path,
                                            description='Absolute path to robot urdf␣
↪file'),
        launch.actions.DeclareLaunchArgument(name='rvizconfig', default_value=default_
↪rviz_config_path,
                                            description='Absolute path to rviz config␣
↪file'),
        joint_state_publisher_node,
        joint_state_publisher_gui_node,
        robot_state_publisher_node,
        rviz_node
    ])
```

See also:

For more information regarding the launch system in ROS 2, you can have a look at the official ROS 2 Launch System Documentation

To keep things simpler when we get to visualization, we have provided an RVIz config file that will be loaded when we launch our package. This configuration file initializes RVIz with the proper settings so you can view the robot immediately once it launches. Create a directory named `rviz` in the root of your project and a file named `urdf_config.rviz` under it. Place the following as the contents of `urdf_config.rviz`

```
Panels:
  - Class: rviz_common/Displays
    Help Height: 78
    Name: Displays
    Property Tree Widget:
      Expanded:
        - /Global Options1
        - /Status1
        - /RobotModel1/Links1
        - /TF1
      Splitter Ratio: 0.5
    Tree Height: 557
Visualization Manager:
  Class: ""
  Displays:
    - Alpha: 0.5
      Cell Size: 1
      Class: rviz_default_plugins/Grid
      Color: 160; 160; 164
      Enabled: true
      Name: Grid
    - Alpha: 0.6
      Class: rviz_default_plugins/RobotModel
      Description Topic:
        Depth: 5
        Durability Policy: Volatile
        History Policy: Keep Last
        Reliability Policy: Reliable
        Value: /robot_description
      Enabled: true
      Name: RobotModel
      Visual Enabled: true
    - Class: rviz_default_plugins/TF
      Enabled: true
      Name: TF
      Marker Scale: 0.3
      Show Arrows: true
      Show Axes: true
      Show Names: true
  Enabled: true
  Global Options:
    Background Color: 48; 48; 48
    Fixed Frame: base_link
    Frame Rate: 30
  Name: root
  Tools:
    - Class: rviz_default_plugins/Interact
      Hide Inactive Objects: true
```

(continues on next page)

(continued from previous page)

```
    - Class: rviz_default_plugins/MoveCamera
    - Class: rviz_default_plugins/Select
    - Class: rviz_default_plugins/FocusCamera
    - Class: rviz_default_plugins/Measure
      Line color: 128; 128; 0
  Transformation:
    Current:
      Class: rviz_default_plugins/TF
  Value: true
  Views:
    Current:
      Class: rviz_default_plugins/Orbit
      Name: Current View
      Target Frame: <Fixed Frame>
      Value: Orbit (rviz)
    Saved: ~
```

Lastly, let us modify the `CMakeLists.txt` file in the project root directory to include the files we just created during the package installation process. Add the following snippet to `CMakeLists.txt` file preferrably above the `if(BUILD_TESTING)` line:

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

We are now ready to build our project using colcon. Navigate to the project root and execute the following commands.

```
colcon build
. install/setup.bash
```

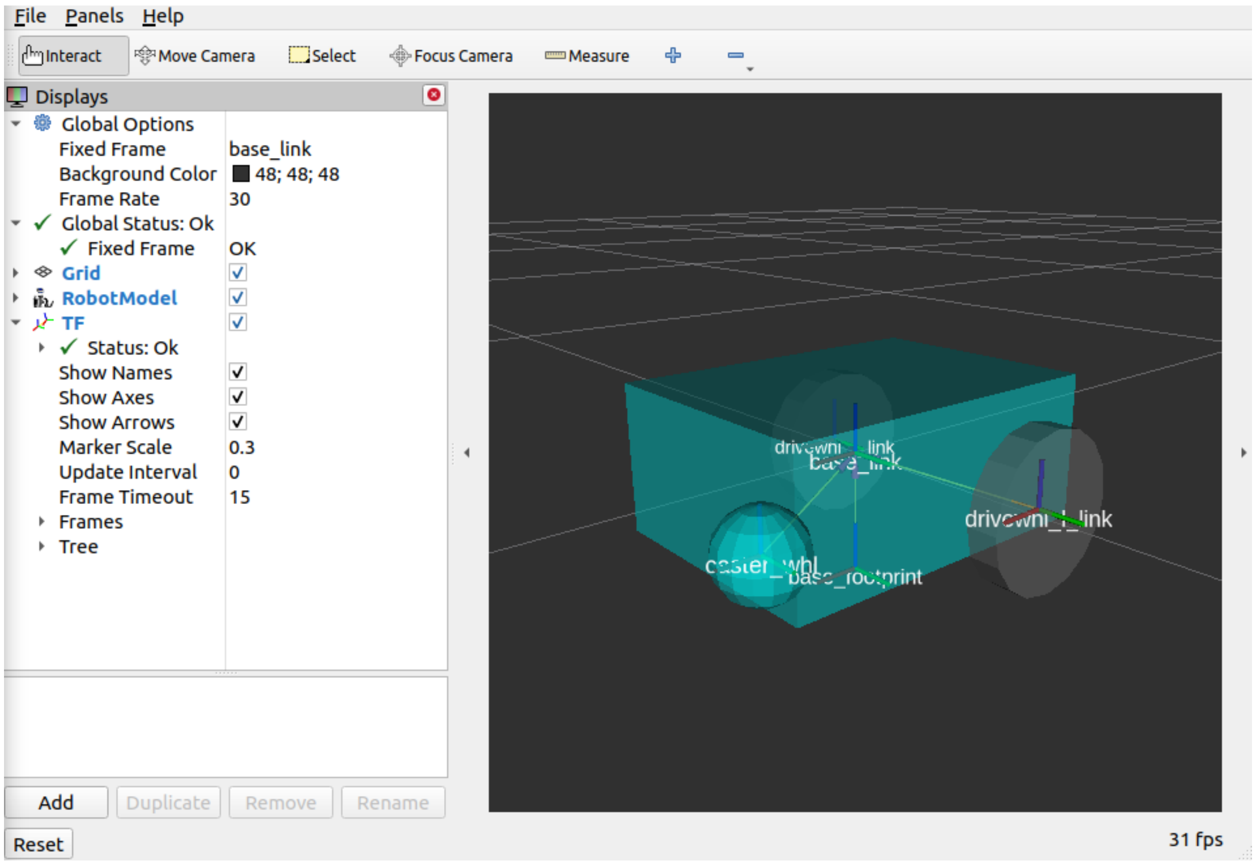
After a successful build, execute the following commands to install the ROS 2 package and launch our project.

```
ros2 launch sam_bot_description display.launch.py
```

ROS 2 should now launch a robot publisher node and start up RVIZ using our URDF. We'll be taking a look at our robot using RVIZ in the next section.

Visualization using RVIZ

RVIZ is a robot visualization tool that allows us to see a 3D model of our robot using its URDF. Upon a successful launch using the commands in the previous section, RVIZ should now be visible on your screen and should look like the image below. You may need to move around and manipulate the view to get a good look at your robot.



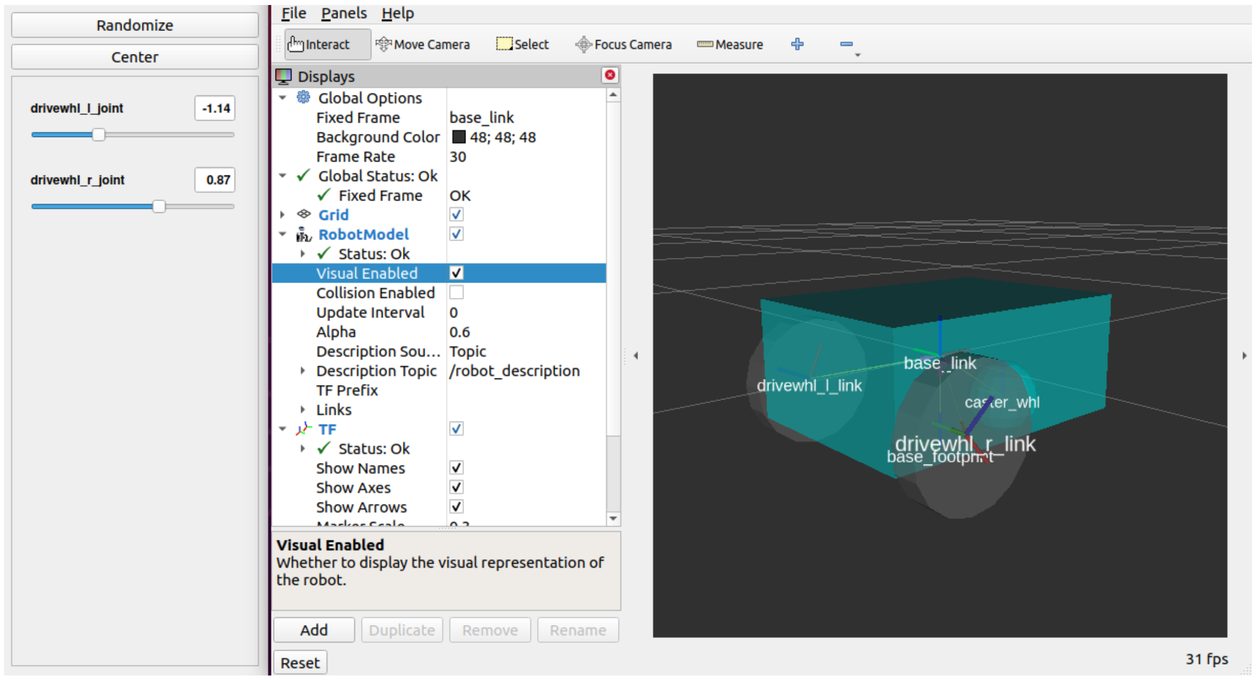
Global Options

Fixed Frame	base_link
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
TF	✓
✓ Status: Ok	
Show Names	✓
Show Axes	✓
Show Arrows	✓
Marker Scale	0.3
Update Interval	0
Frame Timeout	15
Frames	
Tree	

31 fps

As you can see, we have successfully created a simple differential drive robot and visualized it in RVIz. It is not necessary to visualize your robot in RVIz, but it's a good step in order to see if you have properly defined your URDF. This helps you ensure that the robot state publisher is publishing the correct transformations.

You may have noticed that another window was launched - this is a GUI for the joint state publisher. The joint state publisher is another ROS 2 package which publishes the state for our non-fixed joints. You can manipulate this publisher through the small GUI and the new pose of the joints will be reflected in RVIz. Sliding the bars for any of the two wheels will rotate these joints. You can see this in action by viewing RVIZ as you sweep the sliders in the Joint State Publisher GUI.



drivewhl_l_joint -1.14
drivewhl_r_joint 0.87

Global Options	
Fixed Frame	base_link
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
✓ Status: Ok	
Visual Enabled	✓
Collision Enabled	
Update Interval	0
Alpha	0.6
Description Sou…	Topic
Description Topic	/robot_description
TF Prefix	
Links	
TF	✓
✓ Status: Ok	
Show Names	✓
Show Axes	✓
Show Arrows	✓

Visual Enabled
Whether to display the visual representation of the robot.

Note: We won't be interacting much with this package for Nav2, but if you would like to know more about the joint state publisher, feel free to have a look at the official Joint State Publisher Documentation.

At this point, you may already decide to stop with this tutorial since we have already achieved our objective of creating a URDF for a simple differential drive robot. The robot state publisher is now publishing the transforms derived from the URDF. These transforms can now be used by other packages (such as Nav2) to get information regarding the shape and structure of your robot. However, to properly use this URDF in a simulation, we need physical properties so that the robot reacts to physical environments like a real robot would. The visualization fields are only for visualization, not collision, so your robot will drive straight through obstacles. We'll get into adding these properties in our URDF in the next section.

Adding Physical Properties

As an additional section to this guide, we will be modifying our current URDF to include some of our robot's kinematic properties. This information may be used by physics simulators such as Gazebo to model and simulate how our robot will act in the virtual environment.

Let us first define macros containing the inertial properties of the geometric primitives we used in our project. Place the snippet below after our constants section in the URDF:

```
  <!-- Define intertial property macros  -->
  <xacro:macro name="box_inertia" params="m w h d">
    <inertial>
      <origin xyz="0 0 0" rpy="${pi/2} 0 ${pi/2}"/>
      <mass value="${m}"/>
      <inertia ixx="${(m/12) * (h*h + d*d)}" ixy="0.0" ixz="0.0" iyy="${(m/12) * (w*w + d*d)}" iyz="0.0" izz="${(m/12) * (w*w + h*h)}"/>
    </inertial>
  </xacro:macro>

  <xacro:macro name="cylinder_inertia" params="m r h">
```

(continues on next page)

(continued from previous page)

```
    <inertial>
      <origin xyz="0 0 0" rpy="${pi/2} 0 0" />
      <mass value="${m}"/>
      <inertia ixx="${(m/12) * (3*r*r + h*h)}" ixy = "0" ixz = "0" iyy="${(m/12) *␣
↪(3*r*r + h*h)}" iyz = "0" izz="${(m/2) * (r*r)}"/>
    </inertial>
  </xacro:macro>

  <xacro:macro name="sphere_inertia" params="m r">
    <inertial>
      <mass value="${m}"/>
      <inertia ixx="${(2/5) * m * (r*r)}" ixy="0.0" ixz="0.0" iyy="${(2/5) * m *␣
↪(r*r)}" iyz="0.0" izz="${(2/5) * m * (r*r)}"/>
    </inertial>
  </xacro:macro>
```

Let us start by adding collision areas to our `base_link` using the `<collision>` tag. We will also be using the box_inertia macro we defined before to add some inertial properties to our `base_link`. Include the following code snippet within `<link name="base_link">` tag of base_link in our URDF.

```
    <collision>
      <geometry>
        <box size="${base_length} ${base_width} ${base_height}"/>
      </geometry>
    </collision>

    <xacro:box_inertia m="15" w="${base_width}" d="${base_length}" h="${base_height}"/
↪>
```

Next, let us do the same for our wheel macros. Include the following code snippet within the `<link name="${prefix}_link">` tag of our wheel macros in our URDF.

```
      <collision>
        <origin xyz="0 0 0" rpy="${pi/2} 0 0"/>
        <geometry>
          <cylinder radius="${wheel_radius}" length="${wheel_width}"/>
        </geometry>
      </collision>

      <xacro:cylinder_inertia m="0.5" r="${wheel_radius}" h="${wheel_width}"/>
```

Lastly, let us add the similar properties to our spherical caster wheels. Include the following in the `<link name="front_caster">` tag of our caster wheel in the URDF.

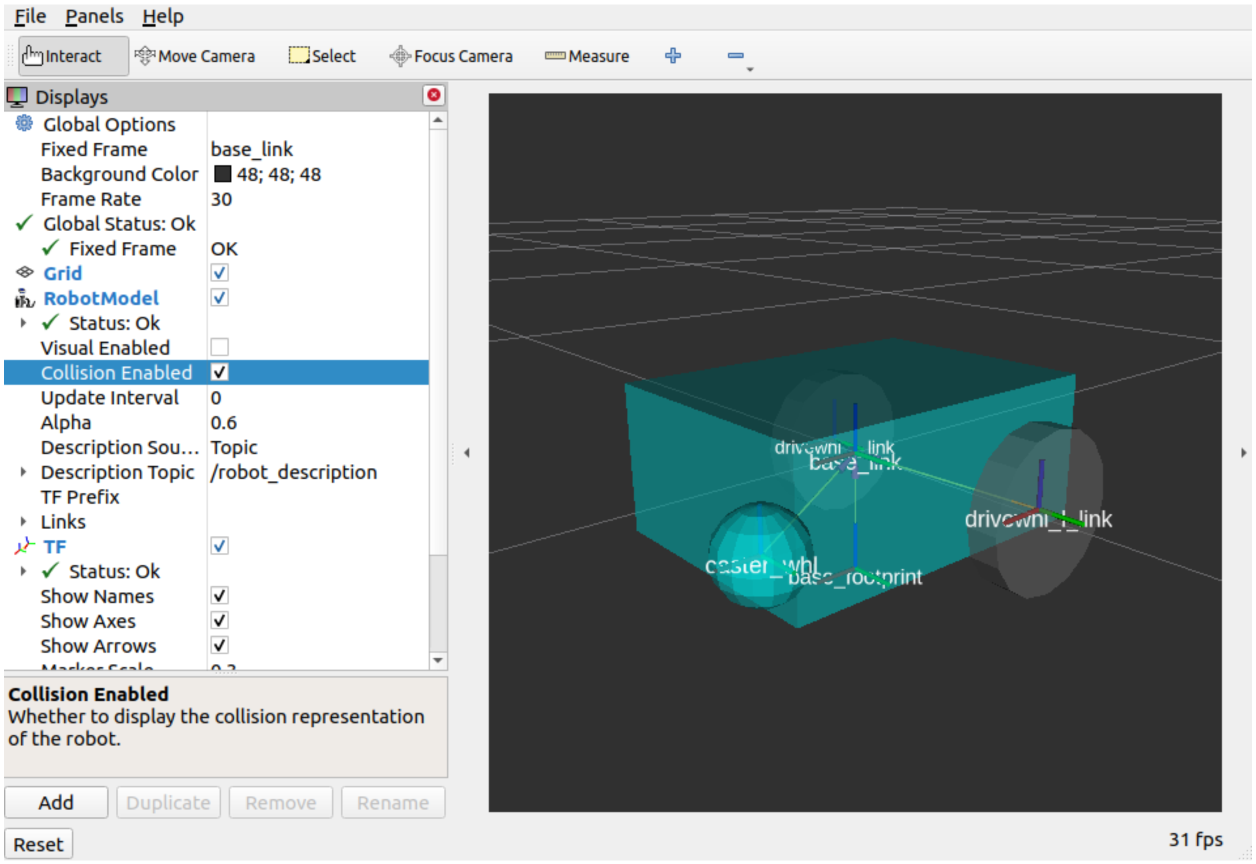
```
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <sphere radius="${(wheel_radius+wheel_zoff-(base_height/2))}"/>
      </geometry>
    </collision>

    <xacro:sphere_inertia m="0.5" r="${(wheel_radius+wheel_zoff-(base_height/2))}"/>
```

Note: We did not add any inertial or collision properties to our `base_footprint` link since this is a virtual and non-physical link.

Build your project and then launch RViz using the same commands in the previous section.

```
colcon build
. install/setup.bash
ros2 launch sam_bot_description display.launch.py
```

You can verify whether you have properly set up the collision areas by enabling `Collision Enabled` under `RobotModel` on the left pane (it may be easier to see if you also turn off `Visual Enabled`). For this tutorial we defined a collision area which is similar to our visual properties. Note that this may not always be the case since you may opt for simpler collision areas based on how your robot looks.



Global Options	
Fixed Frame	base_link
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
✓ Status: Ok	
Visual Enabled	☐
Collision Enabled	✓
Update Interval	0
Alpha	0.6
Description Sou…	Topic
Description Topic	/robot_description
TF Prefix	
Links	
TF	✓
✓ Status: Ok	
Show Names	✓
Show Axes	✓
Show Arrows	✓

Collision Enabled
Whether to display the collision representation of the robot.

For now, we will have to stop here since we will need to set up a lot more components to actually start simulating our robot in Gazebo. We will be coming back to this project during the course of these setup guides, and we will eventually see our robot move in a virtual environment once we get to the simulation sections. The major components that are missing from this work are the simulation plugins required to mimic your robot controllers. We will introduce those and add them to this URDF in the appropriate section.

Conclusion

And that's it. In this tutorial, you have successfully created a URDF for a simple differential drive robot. You have also set up a ROS 2 project that launches a robot publisher node, which then uses your URDF to publish the robot's transforms. We have also used RViz to visualize our robot to verify whether our URDF is correct. Lastly, we have added in some physical properties to our URDF in order to prepare it for simulation.

Feel free to use this tutorial as a template for your own robot. Remember that your main goal is to publish the correct transforms from your base_link up to your sensor_frames. Once these have been setup, then you may proceed to our other setup guides.

4.4.3 Setting Up Odometry

In this guide, we will be looking at how to integrate our robot's odometry system with Nav2. First we will provide a brief introduction on odometry, plus the necessary messages and transforms that need to be published for Nav2 to function correctly. Next, we will show how to setup odometry with two different cases. In the first case, we will show how to setup an odometry system for a robot with already available wheel encoders. In the second case, we will build a demo that simulates a functioning odometry system on `sam_bot` (the robot that we built in the previous section) using Gazebo. Afterwards, we will discuss how various sources of odometry can be fused to provide a smoothed odometry using the `robot_localization` package. Lastly, we will also show how to publish the `odom` => `base_link` transform using `robot_localization`.

See also:

The complete source code in this tutorial can be found in navigation2_tutorials repository under the `sam_bot_description` package. Note that the repository contains the full code after accomplishing all the tutorials in this guide.

Odometry Introduction

The odometry system provides a locally accurate estimate of a robot's pose and velocity based on its motion. The odometry information can be obtained from various sources such as IMU, LIDAR, RADAR, VIO, and wheel encoders. One thing to note is that IMUs drift over time while wheel encoders drift over distance traveled, thus they are often used together to counter each other's negative characteristics.

The `odom` frame and the transformation associated with it use a robot's odometry system to publish localization information that is continuous but becomes less accurate over time or distance (depending on the sensor modalities and drift). In spite of this, the information can still be used by the robot to navigate its immediate vicinity (e.g collision avoidance). To obtain consistently accurate odometry information over time, the `map` frame provides globally accurate information that is used to correct the `odom` frame.

As discussed in the previous guides and in REP 105, the `odom` frame is connected to the rest of the system and Nav2 through the `odom` => `base_link` transform. This transform is published by a tf2 broadcaster or by frameworks such as `robot_localization`, which also provide additional functionalities. We will be talking more about `robot_localization` in a following section.

In addition to the required `odom` => `base_link` transform, Nav2 also requires the publishing of `nav_msgs/Odometry` message because this message provides the velocity information of the robot. In detail, the `nav_msgs/Odometry` message contains the following information:

```
# This represents estimates of position and velocity in free space.
# The pose in this message should be specified in the coordinate frame given by␣
↪header.frame_id
# The twist in this message should be specified in the coordinate frame given by the␣
↪child_frame_id
```

(continues on next page)

(continued from previous page)

```
# Includes the frame id of the pose parent.
std_msgs/Header header

# Frame id the pose is pointing at. The twist is in this coordinate frame.
string child_frame_id

# Estimated pose that is typically relative to a fixed world frame.
geometry_msgs/PoseWithCovariance pose

# Estimated linear and angular velocity relative to child_frame_id.
geometry_msgs/TwistWithCovariance twist
```

This message tells us the estimates for the pose and velocity of the robot. The `header` message provides the timestamped data in a given coordinate frame. The `pose` message provides the position and orientation of the robot relative to the frame specified in `header.frame_id`. The `twist` message gives the linear and angular velocity relative to the frame defined in `child_frame_id`.

Setting Up Odometry on your Robot

Setting up the odometry system for Nav2 for your physical robot depends a lot on which odometry sensors are available with your robot. Due to the large number of configurations your robot may have, specific setup instructions will not be within the scope of this tutorial. Instead, we will provide some basic examples and useful resources to help you configure your robot for Nav2.

To start, we will use an example of a robot with wheel encoders as its odometry source. Note that wheel encoders are not required for Nav2 but it is common in most setups. The goal in setting up the odometry is to compute the odometry information and publish the `nav_msgs/Odometry` message and `odom` => `base_link` transform over ROS 2. To calculate this information, you will need to setup some code that will translate wheel encoder information into odometry information, similar to the snippet below:

```
linear = (right_wheel_est_vel + left_wheel_est_vel) / 2
angular = (right_wheel_est_vel - left_wheel_est_vel) / wheel_separation;
```

The `right_wheel_est_vel` and `left_wheel_est_vel` are the estimated velocities of the right and left wheels respectively, and the `wheel separation` is the distance between the wheels. The values of `right_wheel_est_vel` and `left_wheel_est_vel` can be obtained by simply getting the changes in the positions of the wheel joints over time. This information can then be used to publish the Nav2 requirements. A basic example on how to do this can be found in the Navigation documentation on odometry located here

An alternative to manually publishing this information that we recommend is through the `ros2_control` framework. The `ros2_control` framework contains various packages for real-time control of robots in ROS 2. For wheel encoders, `ros2_control` has a `diff_drive_controller` (differential drive controller) under the `ros2_controller` package. The `diff_drive_controller` takes in the `geometry_msgs/Twist` messages published on `cmd_vel` topic, computes odometry information, and publishes `nav_msgs/Odometry` messages on `odom` topic. Other packages that deal with different kind of sensors are also available in `ros2_control`.

See also:

For more information, see the ros2_control documentation and the Github repository of diff_drive_controller.

For other types of sensors such as IMU, VIO, etc, their respective ROS drivers should have documentation on how publish the odometry information. Keep in mind that Nav2 requires the `nav_msgs/Odometry` message and `odom` => `base_link` transforms to be published and this should be your goal when setting up your odometry system.

Simulating an Odometry System using Gazebo

In this section, we will be using Gazebo to simulate the odometry system of sam_bot, the robot that we built in the previous section of this tutorial series. You may go through that guide first or grab the complete source here.

Note: If you are working on your own physical robot and have already set up your odometry sensors, you may opt to skip this section and head onto the next one where we fuse IMU and odometry messages to provide a smooth odom => base_link transformation.

As an overview for this section, we will first setup Gazebo and the necessary packages required to make it work with ROS 2. Next, we will be adding Gazebo plugins, which simulate an IMU sensor and a differential drive odometry system, in order to publish sensor_msgs/Imu and nav_msgs/Odometry messages respectively. Lastly, we will spawn sam_bot in a Gazebo environment and verify the published sensor_msgs/Imu and nav_msgs/Odometry messages over ROS 2.

Setup and Prerequisites

Gazebo is a 3D simulator that allows us to observe how our virtual robot will function in a simulated environment. To start using Gazebo with ROS 2, follow the installation instructions in the Gazebo Installation Documentation.

We also need to install the gazebo_ros_pkgs package to simulate odometry and control the robot with ROS 2 in Gazebo:

```
sudo apt install ros-<ros2-distro>-gazebo-ros-pkgs
```

You can test if you have successfully set up your ROS 2 and Gazebo environments by following the instructions given here.

Note that we described sam_bot using URDF. However, Gazebo uses Simulation Description Format (SDF) to describe a robot in its simulated environment. Fortunately, Gazebo automatically translates compatible URDF files into SDF. The main requirement for the URDF to be compatible with Gazebo is to have an <inertia> element within each <link> element. This requirement is already satisfied in the URDF file of sam_bot, so it can already be used in Gazebo.

See also:

For more information on how to use URDF in Gazebo, see Tutorial: Using a URDF in Gazebo.

Adding Gazebo Plugins to a URDF

We will now add the IMU sensor and the differential drive plugins of Gazebo to our URDF. For an overview of the different plugins available in Gazebo, have a look at Tutorial: Using Gazebo plugins with ROS.

For our robot, we will be using the GazeboRosImuSensor which is a SensorPlugin. A SensorPlugin must be attached to a link, thus we will create an imu_link to which the IMU sensor will be attached. This link will be referenced under the <gazebo> element. Next, we will set /demo/imu as the topic to which the IMU will be publishing its information, and we will comply with REP145 by setting initalOrientationAsReference to false. We will also add some noise to the sensor configuration using Gazebo's sensor noise model.

Now, we will set up our IMU sensor plugin according to the description above by adding the following lines before the </robot> line in our URDF:

```
<link name="imu_link">
  <visual>
    <geometry>
      <box size="0.1 0.1 0.1"/>
    </geometry>
  </visual>

  <collision>
    <geometry>
      <box size="0.1 0.1 0.1"/>
    </geometry>
  </collision>

  <xacro:box_inertia m="0.1" w="0.1" d="0.1" h="0.1"/>
</link>

<joint name="imu_joint" type="fixed">
  <parent link="base_link"/>
  <child link="imu_link"/>
  <origin xyz="0 0 0.01"/>
</joint>

 <gazebo reference="imu_link">
  <sensor name="imu_sensor" type="imu">
   <plugin filename="libgazebo_ros_imu_sensor.so" name="imu_plugin">
      <ros>
        <namespace>/demo</namespace>
        <remapping>~/out:=imu</remapping>
      </ros>
      <initial_orientation_as_reference>false</initial_orientation_as_reference>
    </plugin>
    <always_on>true</always_on>
    <update_rate>100</update_rate>
    <visualize>true</visualize>
    <imu>
      <angular_velocity>
        <x>
          <noise type="gaussian">
            <mean>0.0</mean>
            <stddev>2e-4</stddev>
            <bias_mean>0.0000075</bias_mean>
            <bias_stddev>0.0000008</bias_stddev>
          </noise>
        </x>
        <y>
          <noise type="gaussian">
            <mean>0.0</mean>
            <stddev>2e-4</stddev>
            <bias_mean>0.0000075</bias_mean>
            <bias_stddev>0.0000008</bias_stddev>
          </noise>
        </y>
        <z>
          <noise type="gaussian">
            <mean>0.0</mean>
            <stddev>2e-4</stddev>
            <bias_mean>0.0000075</bias_mean>
```

(continues on next page)

(continued from previous page)

```
            <bias_stddev>0.0000008</bias_stddev>
          </noise>
        </z>
      </angular_velocity>
      <linear_acceleration>
        <x>
          <noise type="gaussian">
            <mean>0.0</mean>
            <stddev>1.7e-2</stddev>
            <bias_mean>0.1</bias_mean>
            <bias_stddev>0.001</bias_stddev>
          </noise>
        </x>
        <y>
          <noise type="gaussian">
            <mean>0.0</mean>
            <stddev>1.7e-2</stddev>
            <bias_mean>0.1</bias_mean>
            <bias_stddev>0.001</bias_stddev>
          </noise>
        </y>
        <z>
          <noise type="gaussian">
            <mean>0.0</mean>
            <stddev>1.7e-2</stddev>
            <bias_mean>0.1</bias_mean>
            <bias_stddev>0.001</bias_stddev>
          </noise>
        </z>
      </linear_acceleration>
    </imu>
  </sensor>
</gazebo>
```

Now, let us add the differential drive ModelPlugin. We will configure the plugin such that `nav_msgs/Odometry` messages are published on the `/demo/odom` topic. The joints of the left and right wheels will be set to the wheel joints of `sam_bot`. The wheel separation and wheel diameter are set according to the values of the defined values of `wheel_ygap` and `wheel_radius` respectively.

To include this plugin in our URDF, add the following lines after the `</gazebo>` tag of the IMU plugin:

```
<gazebo>
  <plugin name='diff_drive' filename='libgazebo_ros_diff_drive.so'>
    <ros>
      <namespace>/demo</namespace>
    </ros>

    <!-- wheels -->
    <left_joint>drivewhl_l_joint</left_joint>
    <right_joint>drivewhl_r_joint</right_joint>

    <!-- kinematics -->
    <wheel_separation>0.4</wheel_separation>
    <wheel_diameter>0.2</wheel_diameter>

    <!-- limits -->
    <max_wheel_torque>20</max_wheel_torque>
```

(continues on next page)

(continued from previous page)

```
    <max_wheel_acceleration>1.0</max_wheel_acceleration>

    <!-- output -->
    <publish_odom>true</publish_odom>
    <publish_odom_tf>false</publish_odom_tf>
    <publish_wheel_tf>true</publish_wheel_tf>

    <odometry_frame>odom</odometry_frame>
    <robot_base_frame>base_link</robot_base_frame>
  </plugin>
</gazebo>
```

Launch and Build Files

We will now edit our launch file, launch/display.launch.py, to spawn `sam_bot` in Gazebo. Since we will be simulating our robot, we can remove the GUI for the joint state publisher by deleting the following lines inside the `generate_launch_description()`:

```
joint_state_publisher_gui_node = launch_ros.actions.Node(
  package='joint_state_publisher_gui',
  executable='joint_state_publisher_gui',
  name='joint_state_publisher_gui',
  condition=launch.conditions.IfCondition(LaunchConfiguration('gui'))
)
```

Remove the following *gui* param:

```
DeclareLaunchArgument(name='gui', default_value='True',
                      description='Flag to enable joint_state_publisher_gui')
```

Remove the condition from the *joint_state_publisher_node*:

```
joint_state_publisher_node = launch_ros.actions.Node(
  package='joint_state_publisher',
  executable='joint_state_publisher',
  name='joint_state_publisher',
  condition=launch.conditions.UnlessCondition(LaunchConfiguration('gui')) # Remove␣
↪this line
)
```

Next, open package.xml and delete the line:

```
<exec_depend>joint_state_publisher_gui</exec_depend>
```

To launch Gazebo, add the following before the `joint_state_publisher_node,` line

```
launch.actions.ExecuteProcess(cmd=['gazebo', '--verbose', '-s', 'libgazebo_ros_init.so
↪', '-s', 'libgazebo_ros_factory.so'], output='screen'),
```

We will now add a node that spawns `sam_bot` in Gazebo. Open launch/display.launch.py again and paste the following lines before the `return launch.LaunchDescription([` line.

```
spawn_entity = launch_ros.actions.Node(
  package='gazebo_ros',
```

(continues on next page)

(continued from previous page)

```
  executable='spawn_entity.py',
  arguments=['-entity', 'sam_bot', '-topic', 'robot_description'],
  output='screen'
)
```

Then add the line `spawn_entity,` before the `rviz_node` line, as shown below.

```
      robot_state_publisher_node,
      spawn_entity,
      rviz_node
])
```

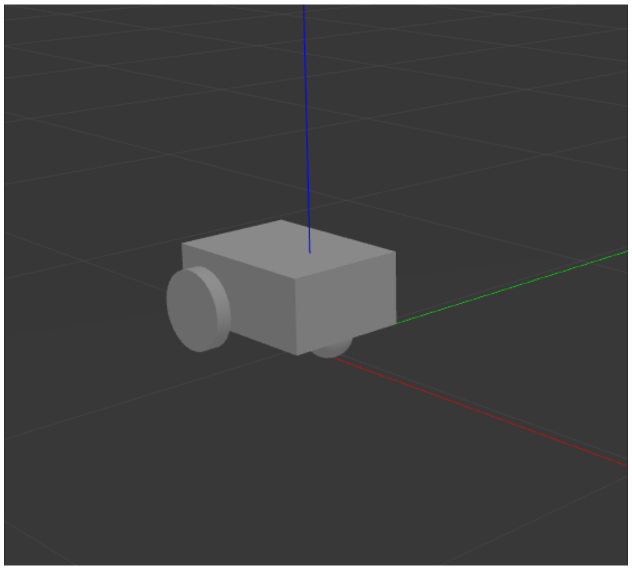
Build, Run and Verification

Let us run our package to check if the `/demo/imu` and `/demo/odom` topics are active in the system.

Navigate to the root of the project and execute the following lines:

```
colcon build
. install/setup.bash
ros2 launch sam_bot_description display.launch.py
```

Gazebo should launch and you should see a 3D model of `sam_bot`:



To see the active topics in the system, open a new terminal and execute:

```
ros2 topic list
```

You should see `/demo/imu` and `/demo/odom` in the list of topics.

To see more information about the topics, execute:

```
ros2 topic info /demo/imu
ros2 topic info /demo/odom
```

You should see an output similar to below:

```
Type: sensor_msgs/msg/Imu
Publisher count: 1
Subscription count: 0
```

```
Type: nav_msgs/msg/Odometry
Publisher count: 1
Subscription count: 0
```

Observe that the `/demo/imu` topic publishes `sensor_msgs/Imu` type messages while the `/demo/odom` topic publishes `nav_msgs/Odometry` type messages. The information being published on these topics come from the gazebo simulation of the IMU sensor and the differential drive respectively. Also note that both topics currently have no subscribers. In the next section, we will create a `robot_localization` node that will subscribe to these two topics. It will then use the messages published on both topics to provide a fused, locally accurate and smooth odometry information for Nav2.

Robot Localization Demo

The `robot_localization` package is used to provide a fused and locally accurate smooth odometry information from the data provided by `N` odometry sensor inputs. These information can be provided to the package through `nav_msgs/Odometry`, `sensor_msgs/Imu`, `geometry_msgs/PoseWithCovarianceStamped`, and `geometry_msgs/TwistWithCovarianceStamped` messages.

A usual robot setup consists of at least the wheel encoders and IMU as its odometry sensor sources. When multiple sources are provided to `robot_localization`, it is able to fuse the odometry information given by the sensors through the use of state estimation nodes. These nodes make use of either an Extended Kalman filter (`ekf_node`) or an Unscented Kalman Filter (`ukf_node`) to implement this fusion. In addition, the package also implements a `navsat_transform_node` which transforms geographic coordinates into the robot's world frame when working with GPS.

Fused sensor data is published by the `robot_localization` package through the `odometry/filtered` and the `accel/filtered` topics, if enabled in its configuration. In addition, it can also publish the `odom` => `base_link` transform on the `/tf` topic.

See also:

More details on `robot_localization` can be found in the official Robot Localization Documentation.

If your robot is only able to provide one odometry source, the use of `robot_localization` would have minimal effects aside from smoothing. In this case, an alternative approach is to publish transforms through a tf2 broadcaster in your single source of odometry node. Nevertheless, you can still opt to use `robot_localization` to publish the transforms and some smoothing properties may still be observed in the output.

See also:

For more information on how to write a tf2 broadcaster, you can check Writing a tf2 broadcaster (C++) (Python).

For the rest of this section, we will show how to use `robot_localization` to fuse the sensors of `sam_bot`. It will use the `sensor_msgs/Imu` messages published on `/demo/Imu` and the `nav_msgs/Odometry` message published on `/demo/odom` and then it will publish data on `odometry/filtered`, `accel/filtered`, and `/tf` topics.

Configuring Robot Localization

Let us now configure the `robot_localization` package to use an Extended Kalman Filter (`ekf_node`) to fuse odometry information and publish the `odom` => `base_link` transform.

First, install the `robot_localization` package using your machines package manager or by executing the following command:

```
sudo apt install ros-<ros2-distro>-robot-localization
```

Next, we specify the parameters of the `ekf_node` using a YAML file. Create a directory named `config` at the root of your project and create a file named `ekf.yaml`. Copy the following lines of code into your `ekf.yaml` file.

```
### ekf config file ###
ekf_filter_node:
    ros__parameters:
# The frequency, in Hz, at which the filter will output a position estimate. Note␣
↪that the filter will not begin
# computation until it receives at least one message from one of theinputs. It will␣
↪then run continuously at the
# frequency specified here, regardless of whether it receives more measurements.␣
↪Defaults to 30 if unspecified.
        frequency: 30.0

# ekf_localization_node and ukf_localization_node both use a 3D omnidirectional␣
↪motion model. If this parameter is
# set to true, no 3D information will be used in your state estimate. Use this if you␣
↪are operating in a planar
# environment and want to ignore the effect of small variations in the ground plane␣
↪that might otherwise be detected
# by, for example, an IMU. Defaults to false if unspecified.
        two_d_mode: false

# Whether to publish the acceleration state. Defaults to false if unspecified.
        publish_acceleration: true

# Whether to broadcast the transformation over the /tf topic. Defaultsto true if␣
↪unspecified.
        publish_tf: true

# 1. Set the map_frame, odom_frame, and base_link frames to the appropriate frame␣
↪names for your system.
#     1a. If your system does not have a map_frame, just remove it, and make sure
↪"world_frame" is set to the value of odom_frame.
# 2. If you are fusing continuous position data such as wheel encoder odometry,␣
↪visual odometry, or IMU data, set "world_frame"
#    to your odom_frame value. This is the default behavior for robot_localization's␣
↪state estimation nodes.
# 3. If you are fusing global absolute position data that is subject to discrete␣
↪jumps (e.g., GPS or position updates from landmark
#    observations) then:
#     3a. Set your "world_frame" to your map_frame value
#     3b. MAKE SURE something else is generating the odom->base_link transform. Note␣
↪that this can even be another state estimation node
#         from robot_localization! However, that instance should *not* fuse the␣
↪global data.
        map_frame: map              # Defaults to "map" if unspecified
        odom_frame: odom            # Defaults to "odom" if unspecified
```

(continues on next page)

(continued from previous page)

```
        base_link_frame: base_link  # Defaults to "base_link" ifunspecified
        world_frame: odom           # Defaults to the value ofodom_frame if␣
↪unspecified

        odom0: demo/odom
        odom0_config: [true,  true,  true,
                       false, false, false,
                       false, false, false,
                       false, false, true,
                       false, false, false]

        imu0: demo/imu
        imu0_config: [false, false, false,
                      true,  true,  true,
                      false, false, false,
                      false, false, false,
                      false, false, false]
```

In this configuration, we defined the parameter values of `frequency`, `two_d_mode`, `publish_acceleration`, `publish_tf`, `map_frame`, `odom_frame`, `base_link_frame`, and `world_frame`. For more information on the other parameters you can modify, see Parameters of state estimation nodes, and a sample `efk.yaml` can be found here.

To add a sensor input to the `ekf_filter_node`, add the next number in the sequence to its base name (odom, imu, pose, twist). In our case, we have one `nav_msgs/Odometry` and one `sensor_msgs/Imu` as inputs to the filter, thus we use `odom0` and `imu0`. We set the value of `odom0` to `demo/odom`, which is the topic that publishes the `nav_msgs/Odometry`. Similarly, we set the value of `imu0` to the topic that publishes `sensor_msgs/Imu`, which is `demo/imu`.

You can specify which values from a sensor are to be used by the filter using the `_config` parameter. The order of the values of this parameter is x, y, z, roll, pitch, yaw, vx, vy, vz, vroll, vpitch, vyaw, ax, ay, az. In our example, we set everything in `odom0_config` to `false` except the 1st, 2nd, 3rd, and 12th entries, which means the filter will only use the x, y, z, and the vyaw values of `odom0`.

In the `imu0_config` matrix, you'll notice that only roll, pitch, and yaw are used. Typical mobile robot-grade IMUs will also provide angular velocities and linear accelerations. For `robot_localization` to work properly, you should not fuse in multiple fields that are derivative of each other. Since angular velocity is fused internally to the IMU to provide the roll, pitch and yaw estimates, we should not fuse in the angular velocities used to derive that information. We also do not fuse in angular velocity due to the noisy characteristics it has when not using exceptionally high quality (and expensive) IMUs.

See also:

For more advise on configuration of input data to `robot_localization`, see Preparing Your Data for Use with robot_localization, and Configuring robot_localization.

Launch and Build Files

Now, let us add the ekf_node into the launch file. Open launch/display.launch.py and paste the following lines before the return launch.LaunchDescription([line.

```
robot_localization_node = launch_ros.actions.Node(
       package='robot_localization',
       executable='ekf_node',
       name='ekf_filter_node',
       output='screen',
       parameters=[os.path.join(pkg_share, 'config/ekf.yaml'), {'use_sim_time':␣
↪LaunchConfiguration('use_sim_time')}]
)
```

Next, add the following launch arguments within the return launch.LaunchDescription([block.

```
launch.actions.DeclareLaunchArgument(name='use_sim_time', default_value='True',
                                            description='Flag to enable use_sim_time
↪'),
```

Lastly, add robot_localization_node, above the rviz_node line to launch the robot localization node.

```
      robot_state_publisher_node,
      spawn_entity,
      robot_localization_node,
      rviz_node
])
```

Next, we need to add the robot_localization dependency to our package definition. Open package.xml and add the following line below the last <exec_depend> tag.

```
<exec_depend>robot_localization</exec_depend>
```

Lastly, open CMakeLists.txt and append the config directory inside the install(DIRECTORY...), as shown in the snippet below.

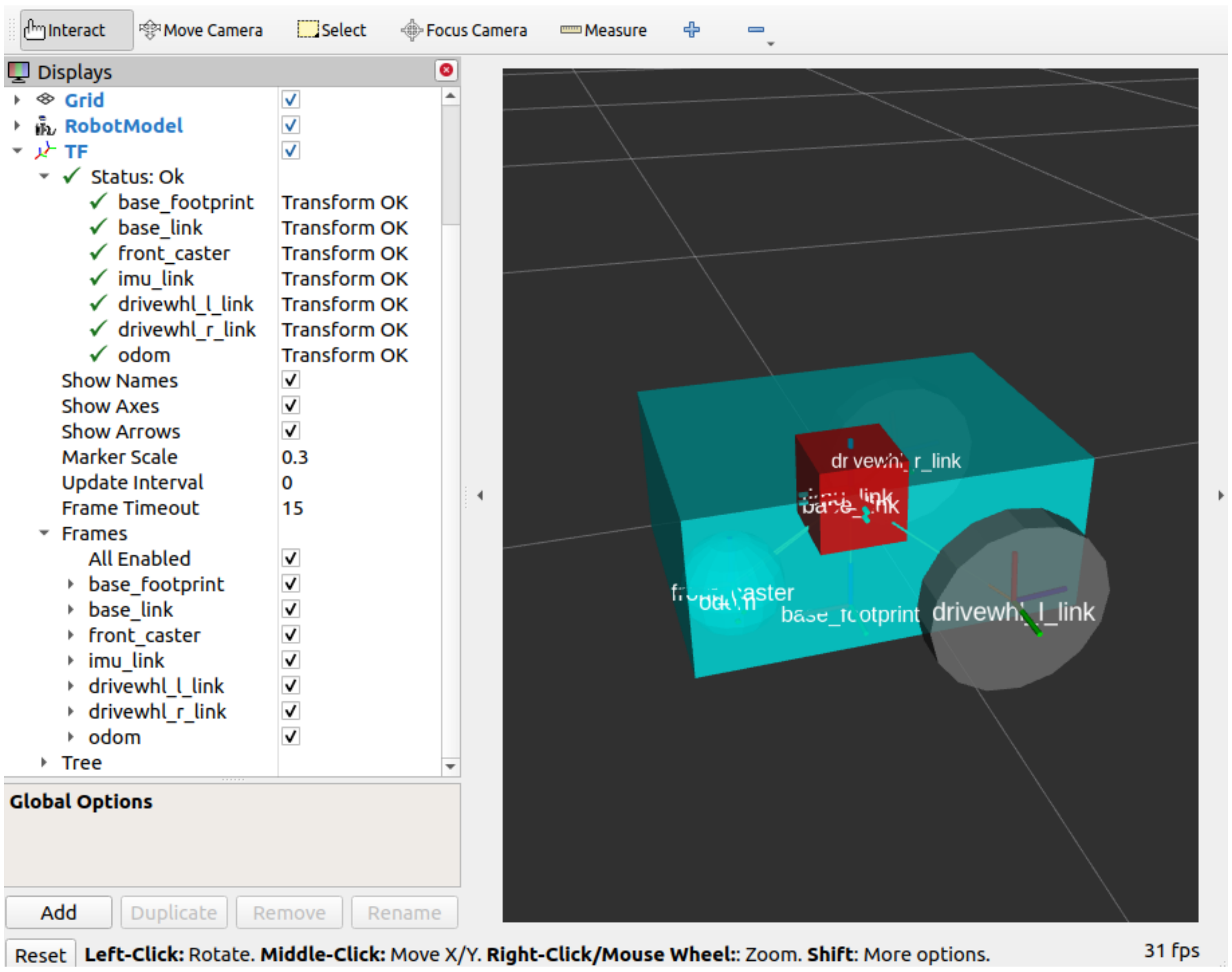
```
install(
  DIRECTORY src launch rviz config
  DESTINATION share/${PROJECT_NAME}
)
```

Build, Run and Verification

Let us now build and run our package. Navigate to the root of the project and execute the following lines:

```
colcon build
. install/setup.bash
ros2 launch sam_bot_description display.launch.py
```

Gazebo and RVIZ should launch. In the RVIZ window, you should see the model and TF frames of sam_bot:



Displays
Grid ✓
RobotModel ✓
TF ✓
Status: Ok
base_footprint Transform OK
base_link Transform OK
front_caster Transform OK
imu_link Transform OK
drivewhl_l_link Transform OK
drivewhl_r_link Transform OK
odom Transform OK
Show Names ✓
Show Axes ✓
Show Arrows ✓
Marker Scale 0.3
Update Interval 0
Frame Timeout 15
Frames
All Enabled ✓
base_footprint ✓
base_link ✓
front_caster ✓
imu_link ✓
drivewhl_l_link ✓
drivewhl_r_link ✓
odom ✓
Tree

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift:** More options.

Next, let us verify that the `odometry/filtered`, `accel/filtered`, and `/tf` topics are active in the system. Open a new terminal and execute:

```
ros2 topic list
```

You should see `odometry/filtered`, `accel/filtered`, and `/tf` in the list of the topics.

You can also check the subscriber count of these topics again by executing:

```
ros2 topic info /demo/imu
ros2 topic info /demo/odom
```

You should see that `/demo/imu` and `/demo/odom` now both have 1 subscriber each.

To verify that the `ekf_filter_node` are the subscribers of these topics, execute:

```
ros2 node info /ekf_filter_node
```

You should see an output as shown below.

```
/ekf_filter_node
Subscribers:
  /demo/imu: sensor_msgs/msg/Imu
  /demo/odom: nav_msgs/msg/Odometry
```

(continues on next page)

(continued from previous page)

```
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /set_pose: geometry_msgs/msg/PoseWithCovarianceStamped
Publishers:
  /accel/filtered: geometry_msgs/msg/AccelWithCovarianceStamped
  /diagnostics: diagnostic_msgs/msg/DiagnosticArray
  /odometry/filtered: nav_msgs/msg/Odometry
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /rosout: rcl_interfaces/msg/Log
  /tf: tf2_msgs/msg/TFMessage
Service Servers:
   ...
```

From the output above, we can see that the `ekf_filter_node` is subscribed to `/demo/imu` and `/demo/odom`. We can also see that the `ekf_filter_node` publishes on the `odometry/filtered`, `accel/filtered`, and `/tf` topics.

You may also verify that `robot_localization` is publishing the `odom` => `base_link` transform by using the tf2_echo utility. Run the folllowing command in a separate command line terminal:

```
ros2 run tf2_ros tf2_echo odom base_link
```

You should see a continuous output similar to what is shown below.

```
At time 8.842000000
- Translation: [0.003, -0.000, 0.127]
- Rotation: in Quaternion [-0.000, 0.092, 0.003, 0.996]
At time 9.842000000
- Translation: [0.002, -0.000, 0.127]
- Rotation: in Quaternion [-0.000, 0.092, 0.003, 0.996]
```

Conclusion

In this guide, we have discussed the messages and transforms that are expected by Nav2 from the odometry system. We have seen how to set up an odometry system and how to verify the published messages. We also have discussed how multiple odometry sensors can be used to provide a filtered and smoothed odometry using `robot_localization`. We have also checked if the `odom` => `base_link` transform is being published correctly by `robot_localization`.

4.4.4 Setting Up Sensors

In this guide, we will discuss the importance of the sensors in navigating a robot safely and how to set up the sensors with Nav2. In the first half of this tutorial, we will take a brief look at commonly used sensors and common sensor messages in Nav2. Next, we will add a basic sensor setup on our previously built simulated robot, `sam_bot`. Lastly, we will then verify the simulated sensor messages of `sam_bot` by visualizing them in RViz.

Once sensors have been set up on a robot, their readings can be used in mapping, localization, and perception tasks. In the second half of this guide, we will first discuss how mapping and localization use the sensor data. Then, we will also take a look at one of Nav2's packages, `nav2_costmap_2d`, which generates costmaps that will eventually be used in Nav2 path planning. We will set up basic configuration parameters for this package so it properly takes in sensor information from `sam_bot`. Lastly, we visualize a generated costmaps in RViz to verify its received data.

Sensor Introduction

Mobile robots are equipped with a multitude of sensors that allow them to see and perceive their environment. These sensors obtain information which can be used to build and maintain the map of the environment, to localize the robot on the map, and to see the obstacles in the environment. These tasks are essential to be able to safely and efficiently navigate a robot through a dynamic environment.

Examples of commonly used sensors are lidar, radar, RGB camera, depth camera, IMU, and GPS. To standardize the message formats of these sensors and allow for easier interoperation between vendors, ROS provides the `sensor_msgs` package that defines the common sensor interfaces. This also allows users to use any sensor vendor as long as it follows the standard format in `sensor_msgs`. In the next subsection, we introduce some of commonly used messages in navigation, namely the `sensor_msgs/LaserScan`, `sensor_msgs/PointCloud2`, `sensor_msgs/Range`, and `sensor_msgs/Image`.

Aside from the `sensor_msgs` package, there are also the `radar_msgs` and `vision_msgs` standard interfaces you should be aware of. The `radar_msgs` defines the messages for radar-specific sensors while the `vision_msgs` package defines the messages used in computer vision such as object detection, segmentation, and other machine learning models. Messages supported by this package are `vision_msgs/Classification2D`, `vision_msgs/Classification3D`, `vision_msgs/Detection2D`, and `vision_msgs/Detection3D`, to name a few.

See also:

For more information, see the API documentation of sensor_msgs, radar_msgs, and vision_msgs.

Your physical robot's sensors probably have ROS drivers written for them (e.g. a ROS node that connects to the sensors, populates data into messages, and publishes them for your robot to use) that follow the standard interface in the `sensor_msgs` package. The `sensor_msgs` package makes it easy for you to use many different sensors from different manufacturers. General software packages like Nav2 can then can read these standardized messages and perform tasks independent of the sensor hardware. On simulated robots such as `sam_bot`, Gazebo has sensor plugins which also publish their information following the `sensor_msgs` package.

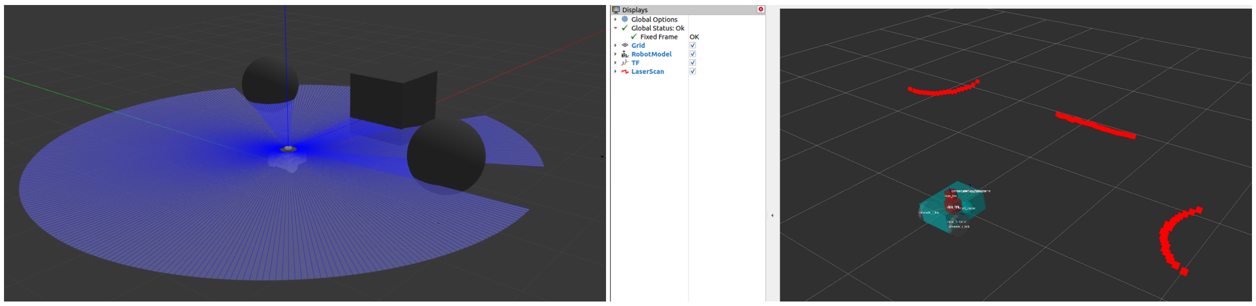
Common Sensor Messages

In this subsection, we discuss some of the common types of `sensor_msgs` you might encounter when setting up Nav2. We will provide a brief description for each sensor, an image of it being simulated in Gazebo and the corresponding visualization of the sensor readings in RViz.

Note: There are other types of `sensor_msgs` aside from the ones listed below. The complete list of messages and their definitions can be found in the sensor_msgs documentation.

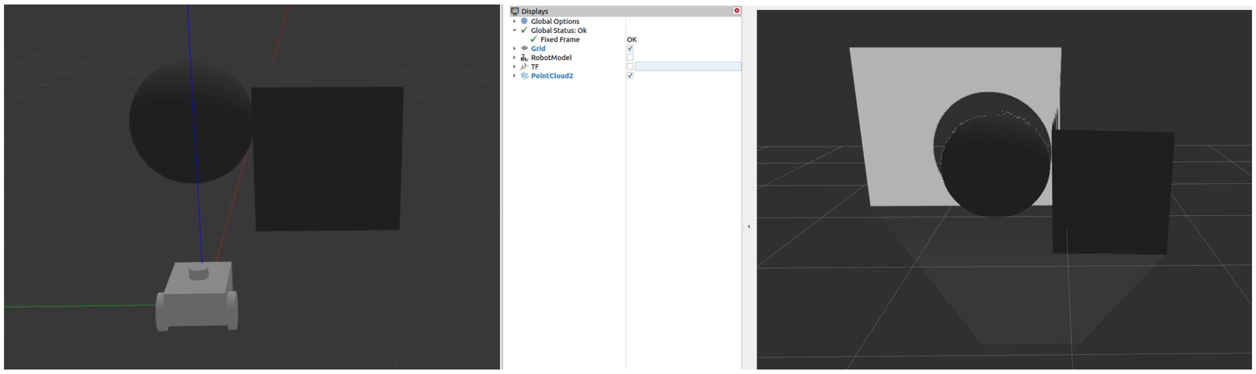
sensor_msgs/LaserScan

This message represents a single scan from a planar laser range-finder. This message is used in `slam_toolbox` and `nav2_amcl` for localization and mapping, or in `nav2_costmap_2d` for perception.



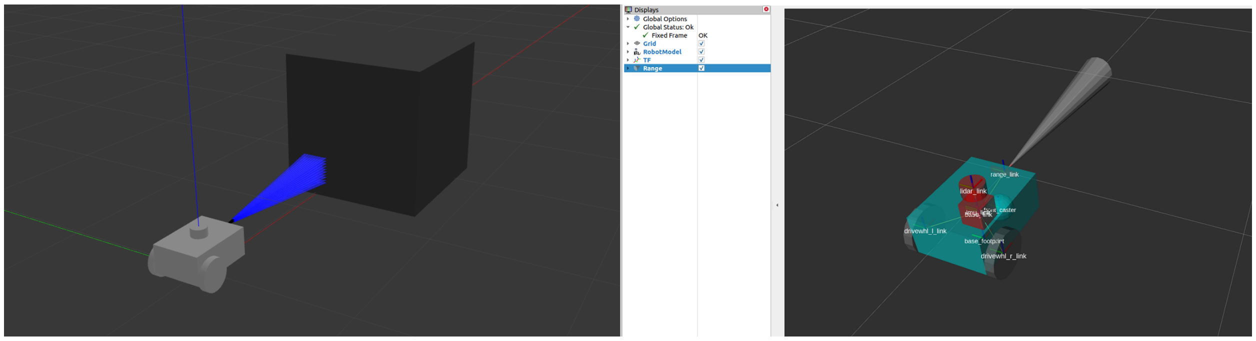
sensor_msgs/PointCloud2

This message holds a collection of 3D points, plus optional additional information about each point. This can be from a 3D lidar, a 2D lidar, a depth camera or more.



sensor_msgs/Range

This is a single range reading from an active ranger that emits energy and reports one range reading that is valid along an arc at the distance measured. A sonar, IR sensor, or 1D range finder are examples of sensors that use this message.



sensor_msgs/Image

This represents the sensor readings from RGB or depth camera, corresponding to RGB or range values.

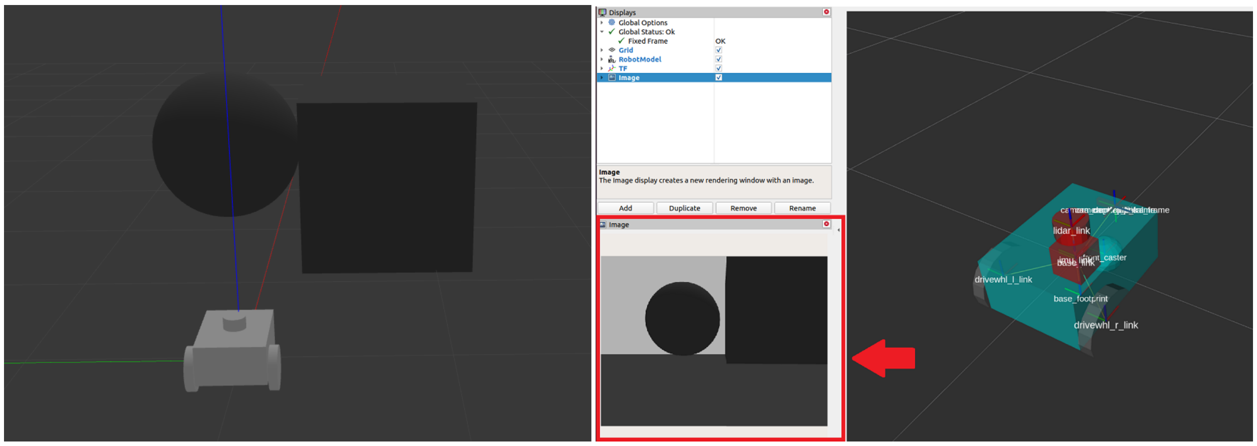


Image
The Image display creates a new rendering window with an image.

Simulating Sensors using Gazebo

To give you a better grasp of how to set up sensors on a simulated robot, we will build up on our previous tutorials and attach sensors to our simulated robot `sam_bot`. Similar to the previous tutorial where we used Gazebo plugins to add odometry sensors to `sam_bot`, we will be using the Gazebo plugins to simulate a lidar sensor and a depth camera on `sam_bot`. If you are working with a real robot, most of these steps are still required for setting up your URDF frames and it will not hurt to also add in the gazebo plugins for later use.

To be able to follow the rest of this section, make sure that you have properly installed Gazebo. You can follow the instructions at the Setup and Prerequisites of the previous tutorial to setup Gazebo.

Adding Gazebo Plugins to a URDF

Let us first add a lidar sensor to `sam_bot`. Open the URDF file, src/description/sam_bot_description.urdf and paste the following lines before the `</robot>` tag.

```xml
<link name="lidar_link">
  <inertial>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <mass value="0.125"/>
    <inertia ixx="0.001"  ixy="0"  ixz="0" iyy="0.001" iyz="0" izz="0.001" />
  </inertial>

  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
       <cylinder radius="0.0508" length="0.055"/>
    </geometry>
  </collision>

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
       <cylinder radius="0.0508" length="0.055"/>
    </geometry>
```

(continues on next page)

(continued from previous page)

```
  </visual>
</link>

<joint name="lidar_joint" type="fixed">
  <parent link="base_link"/>
  <child link="lidar_link"/>
  <origin xyz="0 0 0.12" rpy="0 0 0"/>
</joint>

<gazebo reference="lidar_link">
  <sensor name="lidar" type="ray">
    <always_on>true</always_on>
    <visualize>true</visualize>
    <update_rate>5</update_rate>
    <ray>
      <scan>
        <horizontal>
          <samples>360</samples>
          <resolution>1.000000</resolution>
          <min_angle>0.000000</min_angle>
          <max_angle>6.280000</max_angle>
        </horizontal>
      </scan>
      <range>
        <min>0.120000</min>
        <max>3.5</max>
        <resolution>0.015000</resolution>
      </range>
      <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
        <stddev>0.01</stddev>
      </noise>
    </ray>
    <plugin name="scan" filename="libgazebo_ros_ray_sensor.so">
      <ros>
        <remapping>~/out:=scan</remapping>
      </ros>
      <output_type>sensor_msgs/LaserScan</output_type>
      <frame_name>lidar_link</frame_name>
    </plugin>
  </sensor>
</gazebo>
```

In the code snippet above, we create a `lidar_link` which will be referenced by the `gazebo_ros_ray_sensor` plugin as the location to attach our sensor. We also set values to the simulated lidar's scan and range properties. Lastly, we set the `/scan` as the topic to which it will publish the `sensor_msgs/LaserScan` messages.

Next, let us add a depth camera to `sam_bot`. Paste the following lines after the `</gazebo>` tag of the lidar sensor.

```
<link name="camera_link">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="0.015 0.130 0.022"/>
    </geometry>
  </visual>
```

(continues on next page)

(continued from previous page)

```

  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="0.015 0.130 0.022"/>
    </geometry>
  </collision>

  <inertial>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <mass value="0.035"/>
    <inertia ixx="0.001"  ixy="0"  ixz="0" iyy="0.001" iyz="0" izz="0.001" />
  </inertial>
</link>

<joint name="camera_joint" type="fixed">
  <parent link="base_link"/>
  <child link="camera_link"/>
  <origin xyz="0.215 0 0.05" rpy="0 0 0"/>
</joint>

<link name="camera_depth_frame"/>

<joint name="camera_depth_joint" type="fixed">
  <origin xyz="0 0 0" rpy="${-pi/2} 0 ${-pi/2}"/>
  <parent link="camera_link"/>
  <child link="camera_depth_frame"/>
</joint>

<gazebo reference="camera_link">
  <sensor name="depth_camera" type="depth">
    <visualize>true</visualize>
    <update_rate>30.0</update_rate>
    <camera name="camera">
      <horizontal_fov>1.047198</horizontal_fov>
      <image>
        <width>640</width>
        <height>480</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.05</near>
        <far>3</far>
      </clip>
    </camera>
    <plugin name="depth_camera_controller" filename="libgazebo_ros_camera.so">
      <baseline>0.2</baseline>
      <alwaysOn>true</alwaysOn>
      <updateRate>0.0</updateRate>
      <frame_name>camera_depth_frame</frame_name>
      <pointCloudCutoff>0.5</pointCloudCutoff>
      <pointCloudCutoffMax>3.0</pointCloudCutoffMax>
      <distortionK1>0</distortionK1>
      <distortionK2>0</distortionK2>
      <distortionK3>0</distortionK3>
      <distortionT1>0</distortionT1>
      <distortionT2>0</distortionT2>
```

(continues on next page)

(continued from previous page)

```
      <CxPrime>0</CxPrime>
      <Cx>0</Cx>
      <Cy>0</Cy>
      <focalLength>0</focalLength>
      <hackBaseline>0</hackBaseline>
    </plugin>
  </sensor>
</gazebo>
```

Similar to the lidar sensor, we create `camera_link` which will be referenced by the `gazebo_ros_camera` plugin as the sensor attachment location. We also create a `camera_depth_frame` that is attached to the `camera_link` and will be set as the `<frame_name>` of the depth camera plugin. We also configure the plugin such that it will publish `sensor_msgs/Image` and `sensor_msgs/PointCloud2` messages to `/depth_camera/image_raw` and `/depth_camera/points` topics respectively. Lastly, we also set up other basic configuration properties for our depth camera.

Launch and Build Files

To verify that the sensors are set up properly and that they can see objects in our environemnt, let us launch `sam_bot` in a Gazebo world with objects. Let us create a Gazebo world with a single cube and a single sphere that are within the range of `sam_bot`'s sensors so we can verify if it can see the objects correctly.

To create the world, create a directory named `world` at the root of your project and create a file named `my_world.sdf` inside the `world` folder . Then copy the contents of world/my_world.sdf and paste them inside `my_world.sdf`.

Now, let us edit our launch file, launch/display.launch.py, to launch Gazebo with the world we just created. First, add the path of `my_world.sdf` by adding the following lines inside the `generate_launch_description()`:

```
world_path=os.path.join(pkg_share, 'world/my_world.sdf'),
```

Lastly, add the world path in the `launch.actions.ExecuteProcess(cmd=['gazebo',...` line, as shown below.

```
launch.actions.ExecuteProcess(cmd=['gazebo', '--verbose', '-s', 'libgazebo_ros_init.so
↪', '-s', 'libgazebo_ros_factory.so', world_path], output='screen'),
```

We also have to add the `world` directory to our `CMakeLists.txt` file. Open CmakeLists.txt and append the `world` directory inside the install(DIRECTORY...), as shown in the snippet below.

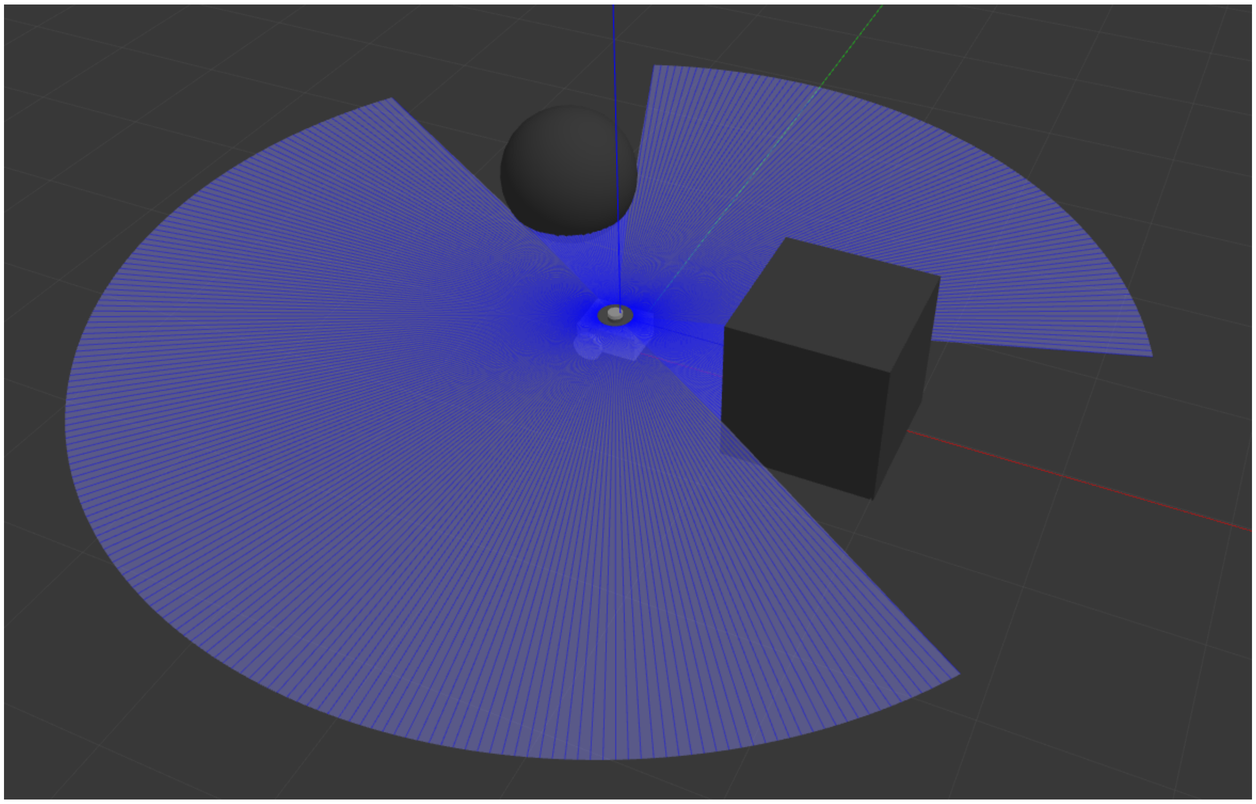
```
install(
  DIRECTORY src launch rviz config world
  DESTINATION share/${PROJECT_NAME}
)
```

Build, Run and Verification

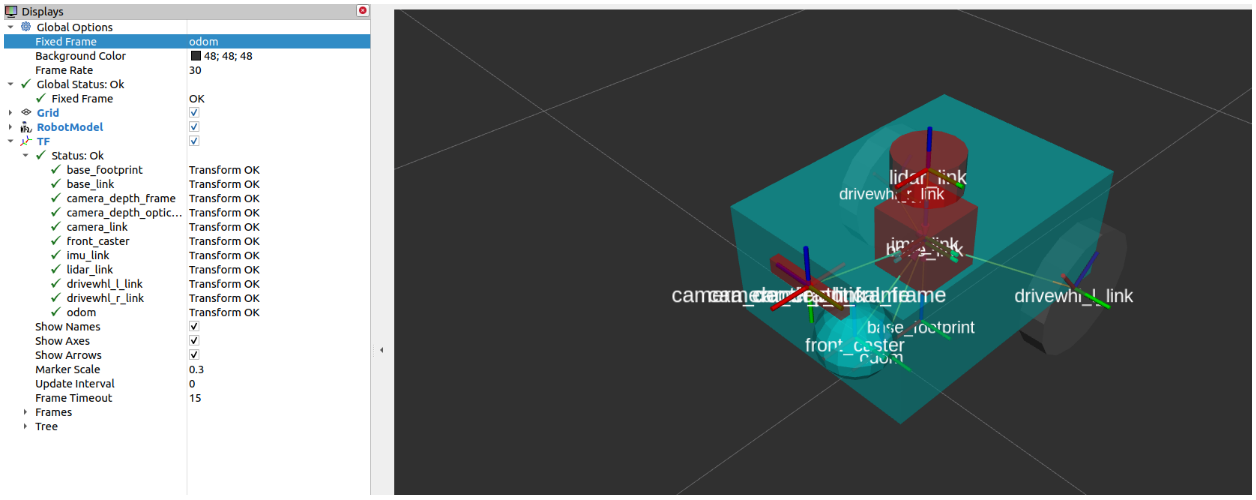
We can now build and run our project. Navigate to the root of the project and execute the following lines:

```
colcon build
. install/setup.bash
ros2 launch sam_bot_description display.launch.py
```

RViz and the Gazebo will then be launched with `sam_bot` present in both. In the Gazebo window, the world that we created should be launched and `sam_bot` should be spawned in that world. You should now be able to observe `sam_bot` with the 360 lidar sensor and the depth camera, as shown in the image below.

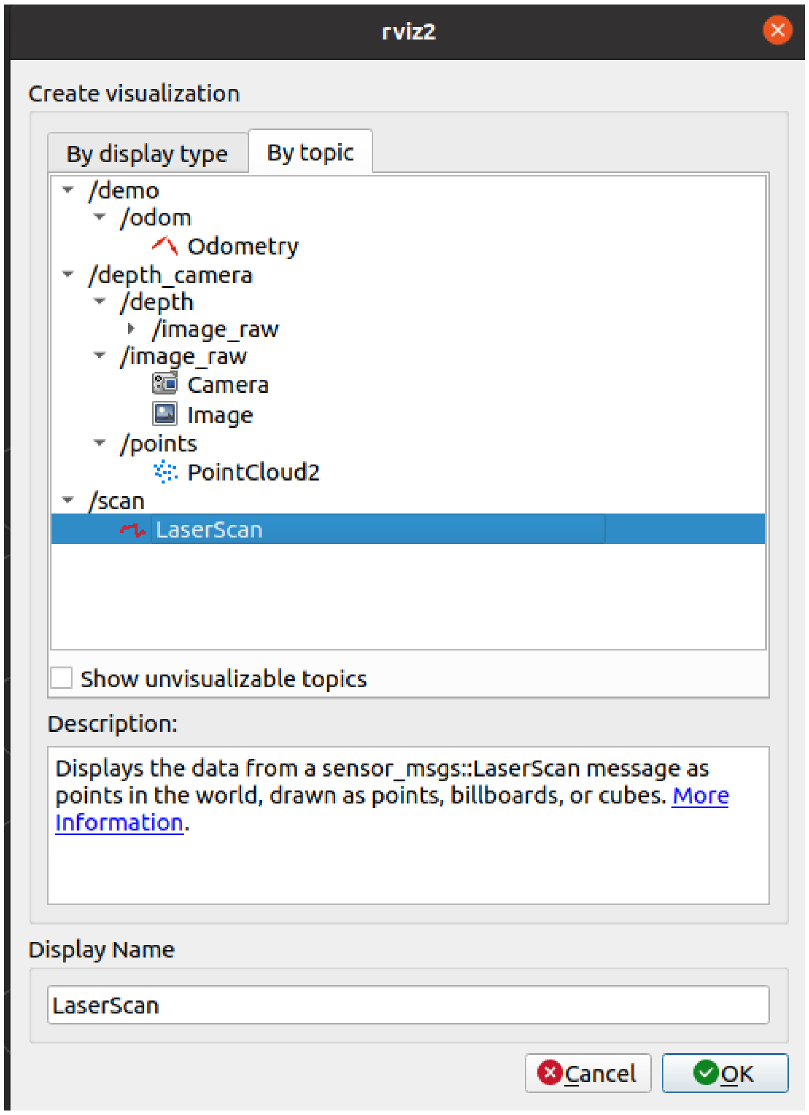


In the RViz window, we can verify if we have properly modeled our sensors and if the transforms of our newly added sensors are correct:



Displays	
Global Options	
Fixed Frame	odom
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
TF	✓
✓ Status: Ok	
✓ base_footprint	Transform OK
✓ base_link	Transform OK
✓ camera_depth_frame	Transform OK
✓ camera_depth_optic...	Transform OK
✓ camera_link	Transform OK
✓ front_caster	Transform OK
✓ imu_link	Transform OK
✓ lidar_link	Transform OK
✓ drivewhl_l_link	Transform OK
✓ drivewhl_r_link	Transform OK
✓ odom	Transform OK
Show Names	✓
Show Axes	✓
Show Arrows	✓
Marker Scale	0.3
Update Interval	0
Frame Timeout	15
Frames	
Tree	

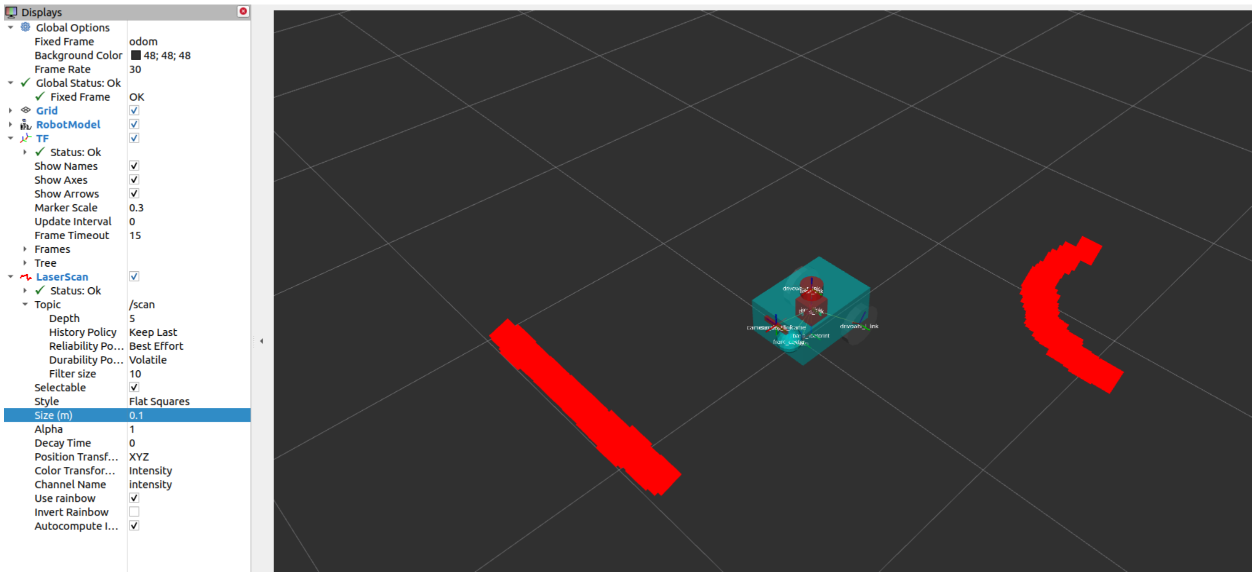
Lastly, we can also visualize the sensor readings in RViz. To visualize the `sensor_msgs/LaserScan` message published on `/scan` topic, click the add button at the bottom part of the RViz window. Then go to the `By topic` tab and select the `LaserScan` option under `/scan`, as shown below.



Displays the data from a sensor_msgs::LaserScan message as points in the world, drawn as points, billboards, or cubes. More Information.

LaserScan

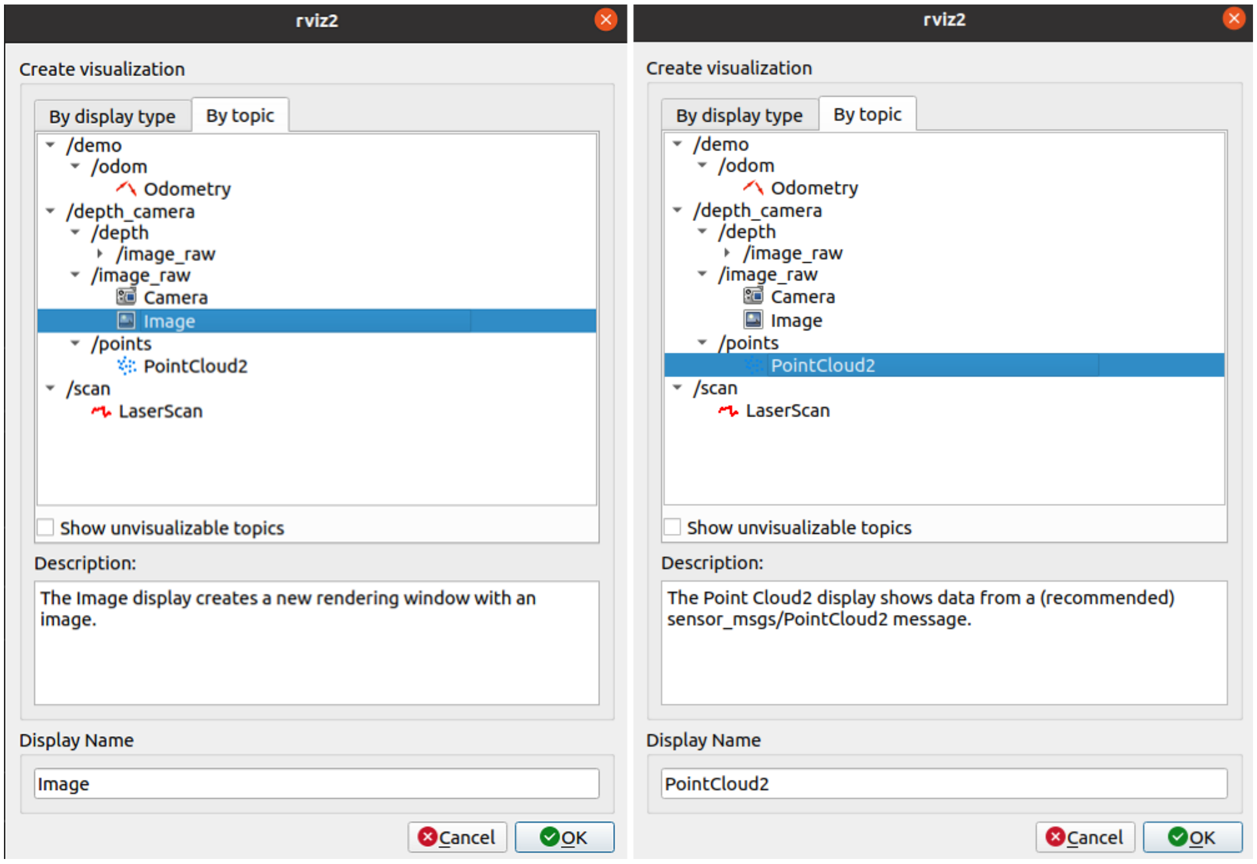
Next, set the `Reliability Policy` in RViz to `Best Effort` and set the `size` to 0.1 to see the points clearer. You should see the visualized `LaserScan` detection as shown below. This corresponds to the detected cube and sphere that we added to the Gazebo world.



Displays

Global Options	
Fixed Frame	odom
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
TF	✓
✓ Status: Ok	
Show Names	✓
Show Axes	✓
Show Arrows	✓
Marker Scale	0.3
Update Interval	0
Frame Timeout	15
Frames	
Tree	
LaserScan	✓
✓ Status: Ok	
Topic	/scan
Depth	5
History Policy	Keep Last
Reliability Po…	Best Effort
Durability Po…	Volatile
Filter size	10
Selectable	✓
Style	Flat Squares
Size (m)	0.1
Alpha	1
Decay Time	0
Position Transf…	XYZ
Color Transfor…	Intensity
Channel Name	intensity
Use rainbow	✓
Invert Rainbow	☐
Autocompute I…	✓

To visualize `sensor_msgs/Image` and `sensor_msgs/PointCloud2`, do the same for topics `/depth_camera/image_raw` and `/depth_camera/points` respectively:



rviz2

Create visualization

By display type | By topic

```
/demo
  /odom
    Odometry
/depth_camera
  /depth
    /image_raw
  /image_raw
    Camera
    Image
  /points
    PointCloud2
/scan
  LaserScan
```

☐ Show unvisualizable topics

Description:

The Image display creates a new rendering window with an image.

Display Name

Image

Cancel | OK

rviz2

Create visualization

By display type | By topic

```
/demo
  /odom
    Odometry
/depth_camera
  /depth
    /image_raw
  /image_raw
    Camera
    Image
  /points
    PointCloud2
/scan
  LaserScan
```

☐ Show unvisualizable topics

Description:

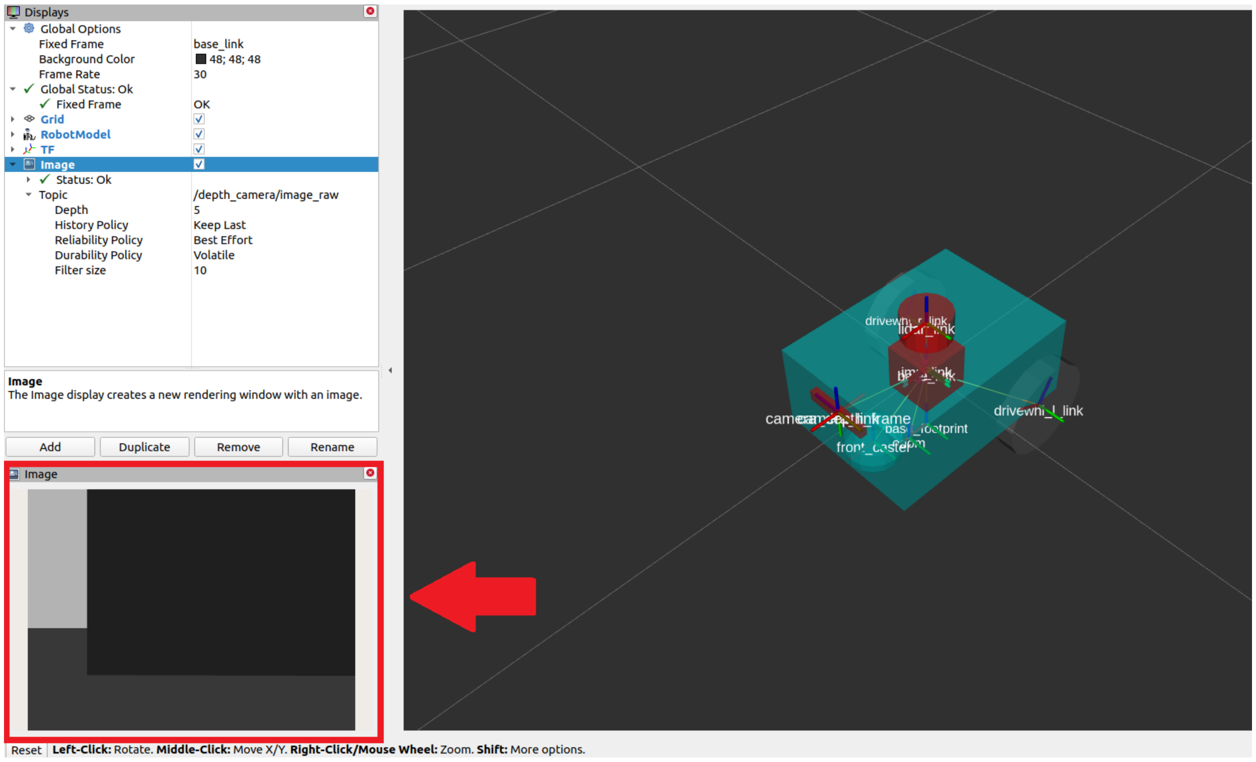
The Point Cloud2 display shows data from a (recommended) sensor_msgs/PointCloud2 message.

Display Name

PointCloud2

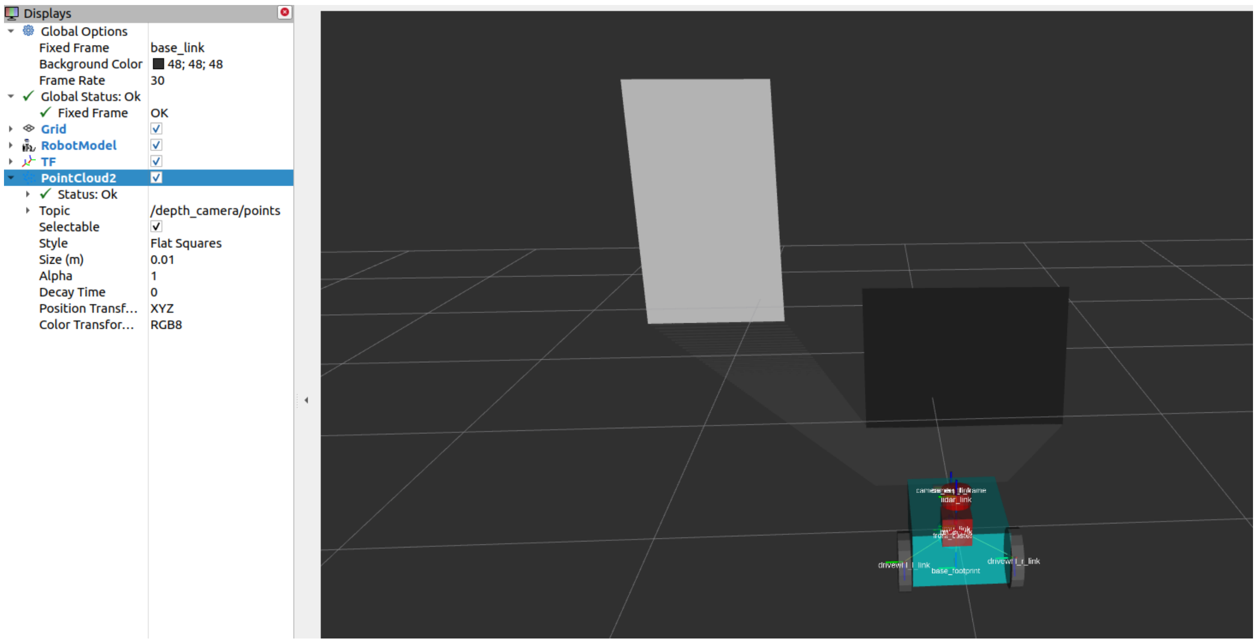
Cancel | OK

After adding the `/depth_camera/image_raw` topic in RViz, set the `Reliability Policy` in RViz to `Best Effort`. Then you should see the cube in the image window at the lower-left side of the RViz window, as shown below.



```
Displays
  Global Options
    Fixed Frame              base_link
    Background Color         ■ 48; 48; 48
    Frame Rate               30
  ✓ Global Status: Ok
    ✓ Fixed Frame            OK
  Grid                       ✓
  RobotModel                 ✓
  TF                         ✓
  Image                      ✓
    ✓ Status: Ok
    Topic                    /depth_camera/image_raw
      Depth                  5
      History Policy         Keep Last
      Reliability Policy     Best Effort
      Durability Policy      Volatile
      Filter size            10

Image
The Image display creates a new rendering window with an image.
```

You should also see the `sensor_msgs/PointCloud2`, as shown below.



```
Displays
  Global Options
    Fixed Frame              base_link
    Background Color         ■ 48; 48; 48
    Frame Rate               30
  ✓ Global Status: Ok
    ✓ Fixed Frame            OK
  Grid                       ✓
  RobotModel                 ✓
  TF                         ✓
  PointCloud2                ✓
    ✓ Status: Ok
    Topic                    /depth_camera/points
    Selectable               ✓
    Style                    Flat Squares
    Size (m)                 0.01
    Alpha                    1
    Decay Time               0
    Position Transf...       XYZ
    Color Transfor...        RGB8
```

Mapping and Localization

Now that we have a robot with its sensors set up, we can use the obtained sensor information to build a map of the environment and to localize the robot on the map. The `slam_toolbox` package is a set of tools and capabilities for 2D Simultaneous Localization and Mapping (SLAM) in potentially massive maps with ROS2. It is also one of the officially supported SLAM libraries in Nav2, and we recommend to use this package in situations you need to use SLAM on your robot setup. Aside from the `slam_toolbox`, localization can also be implemented through the `nav2_amcl` package. This package implements Adaptive Monte Carlo Localization (AMCL) which estimates the position and orientation of the robot in a map. Other techniques may also be available, please check Nav2 documentation for more information.

Both the `slam_toolbox` and `nav2_amcl` use information from the laser scan sensor to be able to perceive the robot's environment. Hence, to verify that they can access the laser scan sensor readings, we must make sure that they are subscribed to the correct topic that publishes the `sensor_msgs/LaserScan` message. This can be configured by setting their `scan_topic` parameters to the topic that publishes that message. It is a convention to publish the `sensor_msgs/LaserScan` messages to `/scan` topic. Thus, by default, the `scan_topic` parameter is set to `/scan`. Recall that when we added the lidar sensor to `sam_bot` in the previous section, we set the topic to which the lidar sensor will publish the `sensor_msgs/LaserScan` messages as `/scan`.

In-depth discussions on the complete configuration parameters will not be a scope of our tutorials since they can be pretty complex. Instead, we recommend you to have a look at their official documentation in the links below.

See also:

For the complete list of configuration parameters of `slam_toolbox`, see the Github repository of slam_toolbox.

For the complete list of configuration parameters and example configuration of `nav2_amcl`, see the AMCL Configuration Guide.

You can also refer to the (SLAM) Navigating While Mapping guide for the tutorial on how to use Nav2 with SLAM. You can verify that `slam_toolbox` and `nav2_amcl` have been correctly setup by visualizing the map and the robot's pose in RViz, similar to what was shown in the previous section.

Costmap 2D

The costmap 2D package makes use of the sensor information to provide a representation of the robot's environment in the form of an occupancy grid. The cells in the occupancy grid store cost values between 0-254 which denote a cost to travel through these zones. A cost of 0 means the cell is free while a cost of 254 means that the cell is lethally occupied. Values in between these extremes are used by navigation algorithms to steer your robot away from obstacles as a potential field. Costmaps in Nav2 are implemented through the `nav2_costmap_2d` package.

The costmap implementation consists of multiple layers, each of which has a certain function that contributes to a cell's overall cost. The package consists of the following layers, but are plugin-based to allow customization and new layers to be used as well: static layer, inflation layer, range layer, obstacle layer, and voxel layer. The static layer represents the map section of the costmap, obtained from the messages published to the `/map` topic like those produced by SLAM. The obstacle layer includes the objects detected by sensors that publish either or both the `LaserScan` and `PointCloud2` messages. The voxel layer is similar to the obstacle layer such that it can use either or both the `LaserScan` and `PointCloud2` sensor information but handles 3D data instead. The range layer allows for the inclusion of information provided by sonar and infrared sensors. Lastly, the inflation layer represents the added cost values around lethal obstacles such that our robot avoids navigating into obstacles due to the robot's geometry. In the next subsection of this tutorial, we will have some discussion about the basic configuration of the different layers in `nav2_costmap_2d`.

The layers are integrated into the costmap through a plugin interface and then inflated using a user-specified inflation radius, if the inflation layer is enabled. For a deeper discussion on costmap concepts, you can have a look at the ROS1 costmap_2D documentation. Note that the `nav2_costmap_2d` package is mostly a straightforward ROS2 port of the ROS1 navigation stack version with minor changes required for ROS2 support and some new layer plugins.

Configuring nav2_costmap_2d

In this subsection, we will show an example configuration of `nav2_costmap_2d` such that it uses the information provided by the lidar sensor of `sam_bot`. We will show an example configuration that uses static layer, obstacle layer, voxel layer, and inflation layer. We set both the obstacle and voxel layer to use the `LaserScan` messages published to the `/scan` topic by the lidar sensor. We also set some of the basic parameters to define how the detected obstacles are reflected in the costmap. Note that this configuration is to be included in the configuration file of Nav2.

```yaml
global_costmap:
  global_costmap:
    ros__parameters:
      update_frequency: 1.0
      publish_frequency: 1.0
      global_frame: map
      robot_base_frame: base_link
      use_sim_time: True
      robot_radius: 0.22
      resolution: 0.05
      track_unknown_space: false
      rolling_window: false
      plugins: ["static_layer", "obstacle_layer", "inflation_layer"]
      static_layer:
        plugin: "nav2_costmap_2d::StaticLayer"
        map_subscribe_transient_local: True
      obstacle_layer:
        plugin: "nav2_costmap_2d::ObstacleLayer"
        enabled: True
        observation_sources: scan
        scan:
          topic: /scan
          max_obstacle_height: 2.0
          clearing: True
          marking: True
          data_type: "LaserScan"
          raytrace_max_range: 3.0
          raytrace_min_range: 0.0
          obstacle_max_range: 2.5
          obstacle_min_range: 0.0
      inflation_layer:
        plugin: "nav2_costmap_2d::InflationLayer"
        cost_scaling_factor: 3.0
        inflation_radius: 0.55
      always_send_full_costmap: True

local_costmap:
  local_costmap:
    ros__parameters:
      update_frequency: 5.0
      publish_frequency: 2.0
      global_frame: odom
      robot_base_frame: base_link
      use_sim_time: True
      rolling_window: true
```

(continues on next page)

(continued from previous page)

```
46       width: 3
47       height: 3
48       resolution: 0.05
49       robot_radius: 0.22
50       plugins: ["voxel_layer", "inflation_layer"]
51       voxel_layer:
52         plugin: "nav2_costmap_2d::VoxelLayer"
53         enabled: True
54         publish_voxel_map: True
55         origin_z: 0.0
56         z_resolution: 0.05
57         z_voxels: 16
58         max_obstacle_height: 2.0
59         mark_threshold: 0
60         observation_sources: scan
61         scan:
62           topic: /scan
63           max_obstacle_height: 2.0
64           clearing: True
65           marking: True
66           data_type: "LaserScan"
67       inflation_layer:
68         plugin: "nav2_costmap_2d::InflationLayer"
69         cost_scaling_factor: 3.0
70         inflation_radius: 0.55
71       always_send_full_costmap: True
```

In the configuration above, notice that we set the parameters for two different costmaps: `global_costmap` and `local_costmap`. We set up two costmaps since the `global_costmap` is mainly used for long-term planning over the whole map while `local_costmap` is for short-term planning and collision avoidance.

The layers that we use for our configuration are defined in the `plugins` parameter, as shown in line 13 for the `global_costmap` and line 50 for the `local_costmap`. These values are set as a list of mapped layer names that also serve as namespaces for the layer parameters we set up starting at lines 14 and line 51. Note that each layer/namespace in this list must have a `plugin` parameter (as indicated in lines 15, 18, 32, 52, and 68) defining the type of plugin to be loaded for that specific layer.

For the static layer (lines 14-16), we set the `map_subscribe_transient_local` parameter to `True`. This sets the QoS settings for the map topic. Another important parameter for the static layer is the `map_topic` which defines the map topic to subscribe to. This defaults to `/map` topic when not defined.

For the obstacle layer (lines 17-30), we define its sensor source under the `observation_sources` parameter (line 20) as `scan` whose parameters are set up in lines 22-30. We set its `topic` parameter as the topic that publishes the defined sensor source and we set the `data_type` according to the sensor source it will use. In our configuration, the obstacle layer will use the `LaserScan` published by the lidar sensor to `/scan`.

Note that the obstacle layer and voxel layer can use either or both `LaserScan` and `PointCloud2` as their `data_type` but it is set to `LaserScan` by default. The code snippet below shows an example of using both the `LaserScan` and `PointCloud2` as the sensor sources. This may be particularly useful when setting up your own physical robot.

```
obstacle_layer:
  plugin: "nav2_costmap_2d::ObstacleLayer"
  enabled: True
  observation_sources: scan pointcloud
  scan:
    topic: /scan
```

(continues on next page)

(continued from previous page)

```
    data_type: "LaserScan"
  pointcloud:
    topic: /depth_camera/points
    data_type: "PointCloud2"
```

For the other parameters of the obstacle layer, the `max_obstacle_height` parameter sets the maximum height of the sensor reading to return to the occupancy grid. The minimum height of the sensor reading can also be set using the `min_obstacle_height` parameter, which defaults to 0 since we did not set it in the configation. The `clearing` parameter is used to set whether the obstacle is to be removed from the costmap or not. The clearing operation is done by raytracing through the grid. The maximum and minimum range to raytrace clear objects from the costmap is set using the `raytrace_max_range` and `raytrace_min_range` respectively. The `marking` parameter is used to set whether the inserted obstacle is marked into the costmap or not. We also set the maximum and minimum range to mark obstacles in the costmap through the `obstacle_max_range` and `obstacle_min_range` respectively.

For the inflation layer (lines 31-34 and 67-70), we set the exponential decay factor across the inflation radius using the `cost_scaling_factor` parameter. The value of the radius to inflate around lethal obstacles is defined using the `inflation_radius`.

For the voxel layer (lines 51-66), we set the `publish_voxel_map` parameter to `True` to enable the publishing of the 3D voxel grid. The resolution of the voxels in height is defined using the `z_resolution` parameter, while the number of voxels in each column is defined using the `z_voxels` parameter. The `mark_threshold` parameter sets the minimum number of voxels in a column to mark as occupied in the occupancy grid. We set the `observation_sources` parameter of the voxel layer to `scan`, and we set the scan parameters (in lines 61-66) similar to the parameters that we have discussed for the obstacle layer. As defined in its `topic` and `data_type` parameters, the voxel layer will use the `LaserScan` published on the `/scan` topic by the lidar scanner.

Note that the we are not using a range layer for our configuration but it may be useful for your own robot setup. For the range layer, its basic parameters are the `topics`, `input_sensor_type`, and `clear_on_max_reading` parameters. The range topics to subscribe to are defined in the `topics` parameter. The `input_sensor_type` is set to either `ALL`, `VARIABLE`, or `FIXED`. The `clear_on_max_reading` is a boolean parameter that sets whether to clear the sensor readings on max range. Have a look at the configuration guide in the link below in case you need to set it up.

See also:

For more information on `nav2_costmap_2d` and the complete list of layer plugin parameters, see the Costmap 2D Configuration Guide.

Build, Run and Verification

We will first launch `display.launch.py` which launches the robot state publisher that provides the `base_link` => `sensors` transformations in our URDF. It also launches Gazebo that acts as our physics simulator and also provides the `odom` => `base_link` from the differential drive plugin, which we added to `sam_bot` in the previous guide, Simulating an Odometry System Using Gazebo. It also launches RViz which we can use to visualize the robot and sensor information.

Then we will launch `slam_toolbox` to publish to `/map` topic and provide the `map` => `odom` transform. Recall that the `map` => `odom` transform is one of the primary requirements of the Nav2 system. The messages published on the `/map` topic will then be used by the static layer of the `global_costmap`.

After we have properly setup our robot description, odometry sensors, and necessary transforms, we will finally launch the Nav2 system itself. For now, we will only be exploring the costmap generation system of Nav2. After launching Nav2, we will visualize the costmaps in RViz to confirm our output.

Launching Description Nodes, RViz and Gazebo

Let us now launch our Robot Description Nodes, RViz and Gazebo through the launch file `display.launch.py`. Open a new terminal and execute the lines below.

```
colcon build
. install/setup.bash
ros2 launch sam_bot_description display.launch.py
```

RViz and the Gazebo should now be launched with `sam_bot` present in both. Recall that the `base_link` => `sensors` transform is now being published by `robot_state_publisher` and the `odom` => `base_link` transform by our Gazebo plugins. Both transforms should now be dislpayed show without errors in RViz.

Launching slam_toolbox

To be able to launch `slam_toolbox`, make sure that you have installed the `slam_toolbox` package by executing the following command:

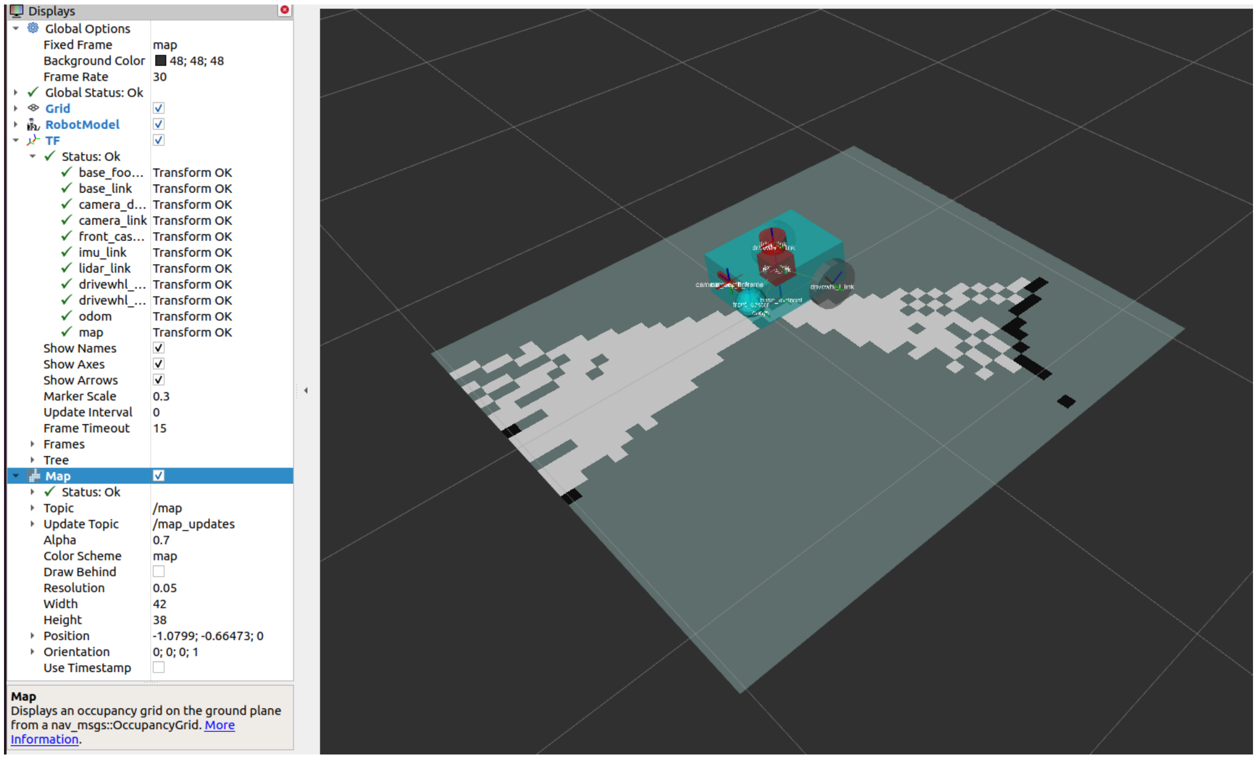
```
sudo apt install ros-<ros2-distro>-slam-toolbox
```

We will launch the `async_slam_toolbox_node` of `slam_toolbox` using the package's built-in launch files. Open a new terminal and then execute the following lines:

```
ros2 launch slam_toolbox online_async_launch.py
```

The `slam_toolbox` should now be publishing to the `/map` topic and providing the `map` => `odom` transform.

We can verify in RViz that the `/map` topic is being published. In the RViz window, click the add button at the bottom-left part then go to `By topic` tab then select the `Map` under the `/map` topic. You should be able to visualize the message received in the `/map` as shown in the image below.



Displays

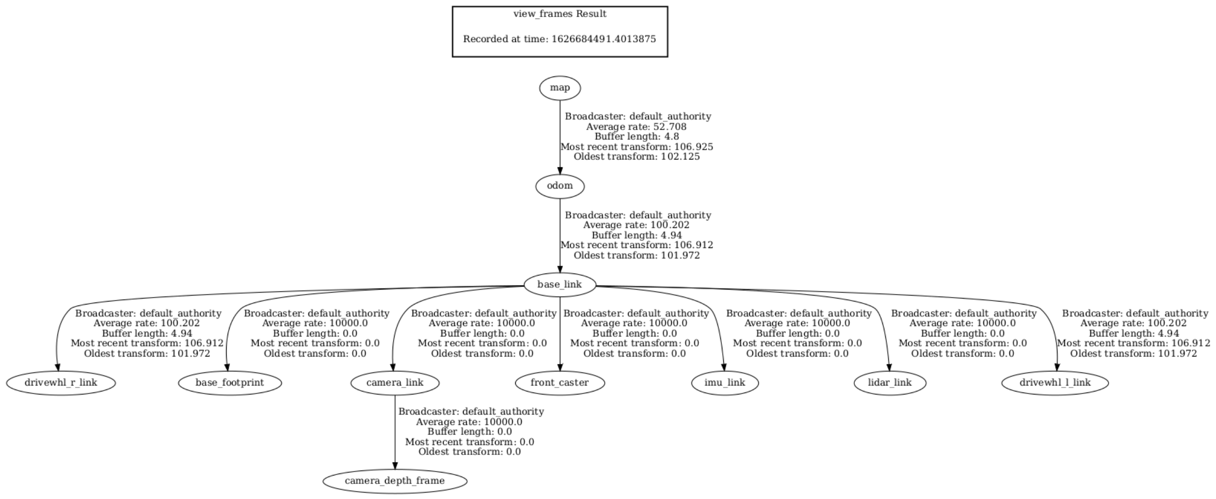
Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
Grid	✓
RobotModel	✓
TF	✓
✓ Status: Ok	
✓ base_foo...	Transform OK
✓ base_link	Transform OK
✓ camera_d...	Transform OK
✓ camera_link	Transform OK
✓ front_cas...	Transform OK
✓ imu_link	Transform OK
✓ lidar_link	Transform OK
✓ drivewhl_...	Transform OK
✓ drivewhl_...	Transform OK
✓ odom	Transform OK
✓ map	Transform OK
Show Names	✓
Show Axes	✓
Show Arrows	✓
Marker Scale	0.3
Update Interval	0
Frame Timeout	15
Frames	
Tree	
Map	✓
✓ Status: Ok	
Topic	/map
Update Topic	/map_updates
Alpha	0.7
Color Scheme	map
Draw Behind	☐
Resolution	0.05
Width	42
Height	38
Position	-1.0799; -0.66473; 0
Orientation	0; 0; 0; 1
Use Timestamp	☐

Map
Displays an occupancy grid on the ground plane from a nav_msgs::OccupancyGrid. More Information.

We can also check that the transforms are correct by executing the following lines in a new terminal:

```
ros2 run tf2_tools view_frames.py
```

Note: For Galactic and newer, it should be `view_frames` and not `view_frames.py` The line above will create a `frames.pdf` file that shows the current transform tree. Your tranform tree should be similar to the one shown below:


view_frames Result
Recorded at time: 1626684491.4013875
map
Broadcaster: default_authority
Average rate: 52.708
Buffer length: 4.8
Most recent transform: 106.925
Oldest transform: 102.125
odom
Broadcaster: default_authority
Average rate: 100.202
Buffer length: 4.94
Most recent transform: 106.912
Oldest transform: 101.972
base_link
Broadcaster: default_authority
Average rate: 100.202
Buffer length: 4.94
Most recent transform: 106.912
Oldest transform: 101.972
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
Broadcaster: default_authority
Average rate: 100.202
Buffer length: 4.94
Most recent transform: 106.912
Oldest transform: 101.972
drivewhl_r_link
base_footprint
camera_link
front_caster
imu_link
lidar_link
drivewhl_l_link
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
camera_depth_frame

Launching Nav2

First, Make sure you have installed the Nav2 packages by executing the following:

```
sudo apt install ros-<ros2-distro>-navigation2
sudo apt install ros-<ros2-distro>-nav2-bringup
```

We will now launch Nav2 using the `nav2_bringup`'s built-in launch file, `navigation_launch.py` . Open a new terminal and execute the following:

```
ros2 launch nav2_bringup navigation_launch.py
```

Note that the parameters of the `nav2_costmap_2d` that we discussed in the previous subsection are included in the default parameters of `navigation_launch.py`. Aside from the `nav2_costmap_2d` parameters, it also contains parameters for the other nodes that are included in Nav2 implementation.

After we have properly set up and launched Nav2, the `/global_costmap` and `/local_costmap` topics should now be active.

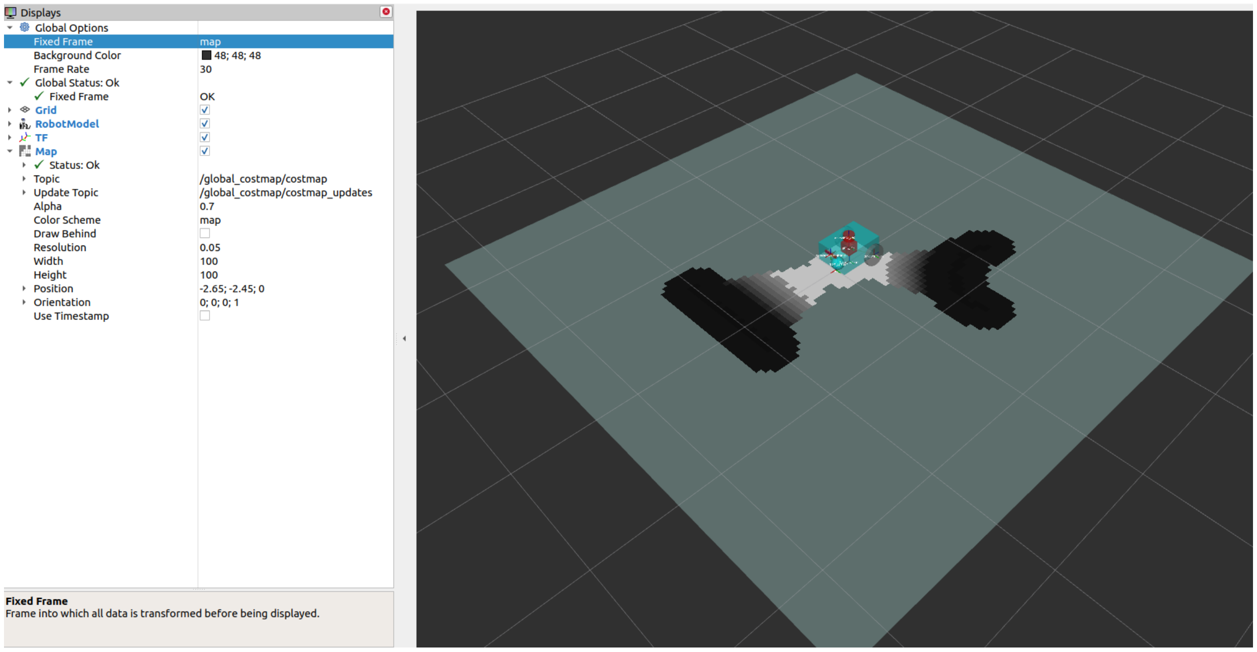
Note: To make the costmaps show up, run the 3 commands in this order

1. Launching Description Nodes, RViz and Gazebo - in logs wait for "Connected to gazebo master"
2. Launching slam_toolbox - in logs wait for "Registering sensor"
3. Launching Nav2 - in logs wait for "Creating bond timer"

Visualizing Costmaps in RViz

The `global_costmap`, `local_costmap` and the voxel representation of the detected obstacles can be visualized in RViz.

To visualize the `global_costmap` in RViz, click the add button at the bottom-left part of the RViz window. Go to `By topic` tab then select the `Map` under the `/global_costmap/costmap` topic. The `global_costmap` should show in the RViz window, as shown below. The `global_costmap` shows areas which should be avoided (black) by our robot when it navigates our simulated world in Gazebo.

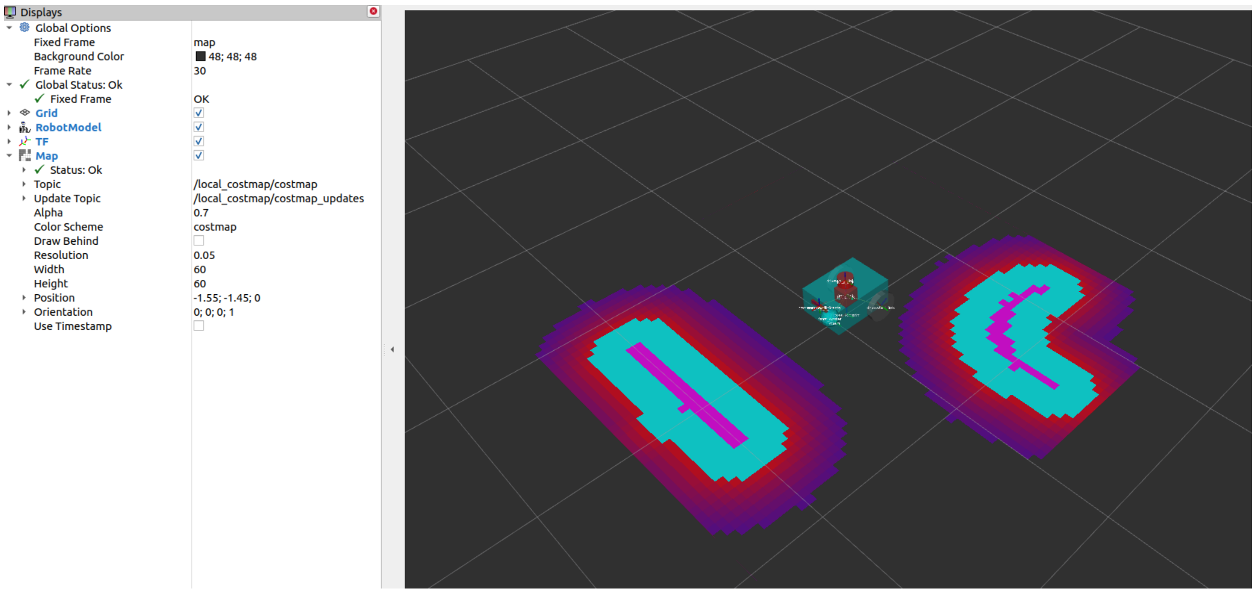


Displays

Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
TF	✓
Map	✓
✓ Status: Ok	
Topic	/global_costmap/costmap
Update Topic	/global_costmap/costmap_updates
Alpha	0.7
Color Scheme	map
Draw Behind	☐
Resolution	0.05
Width	100
Height	100
Position	-2.65; -2.45; 0
Orientation	0; 0; 0; 1
Use Timestamp	☐

Fixed Frame
Frame into which all data is transformed before being displayed.

To visualize the `local_costmap` in RViz, select the `Map` under the `/local_costmap/costmap` topic. Set the `color scheme` in RViz to `costmap` to make it appear similar to the image below.



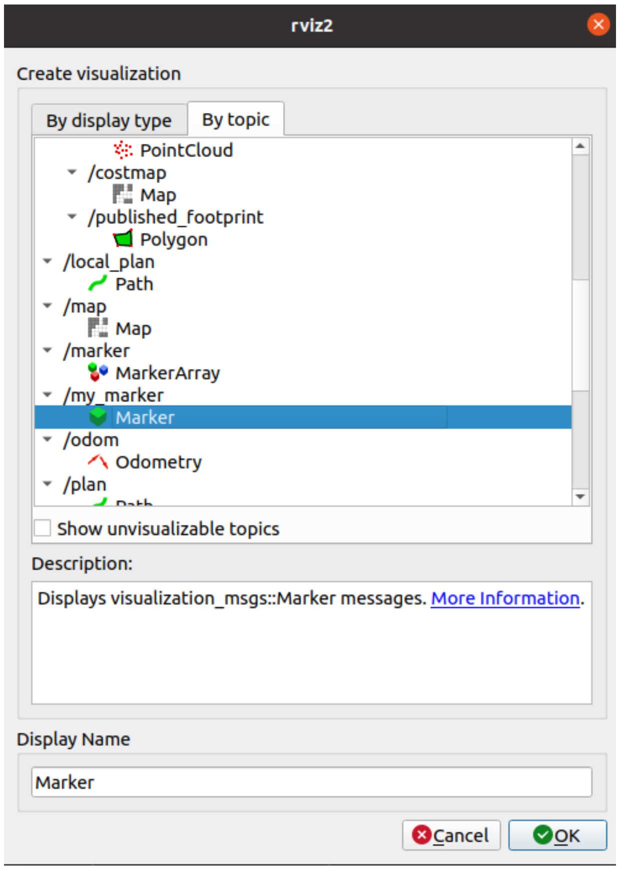
Displays

Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
TF	✓
Map	✓
✓ Status: Ok	
Topic	/local_costmap/costmap
Update Topic	/local_costmap/costmap_updates
Alpha	0.7
Color Scheme	costmap
Draw Behind	☐
Resolution	0.05
Width	60
Height	60
Position	-1.55; -1.45; 0
Orientation	0; 0; 0; 1
Use Timestamp	☐

To visualize the voxel representation of the detected object, open a new terminal and execute the following lines:

```
ros2 run nav2_costmap_2d nav2_costmap_2d_markers voxel_grid:=/local_costmap/voxel_
↪grid visualization_marker:=/my_marker
```

The line above sets the topic where the the markers will be published to `/my_marker`. To see the markers in RViz, select `Marker` under the `/my_marker` topic, as shown below.


rviz2

Create visualization

By display type | By topic

- PointCloud
- /costmap
  - Map
- /published_footprint
  - Polygon
- /local_plan
  - Path
- /map
  - Map
- /marker
  - MarkerArray
- /my_marker
  - Marker
- /odom
  - Odometry
- /plan
  - Path

☐ Show unvisualizable topics

Description:

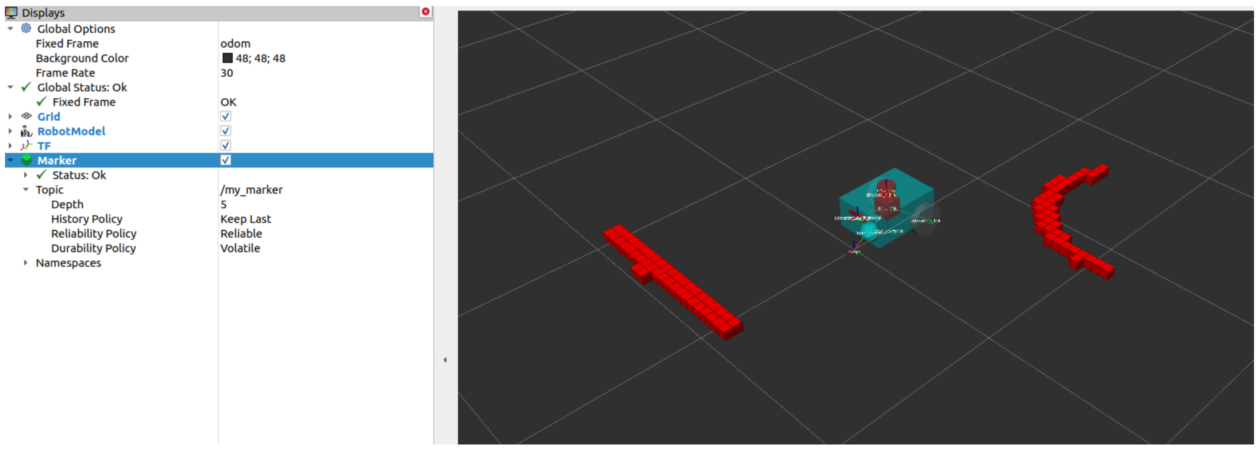
Displays visualization_msgs::Marker messages. More Information.

Display Name

Marker

Cancel | OK

Then set the `fixed frame` in RViz to `odom` and you should now see the voxels in RViz, which represent the cube and the sphere that we have in the Gazebo world:


Displays

Global Options	
Fixed Frame	odom
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
TF	✓
Marker	✓
✓ Status: Ok	
Topic	/my_marker
Depth	5
History Policy	Keep Last
Reliability Policy	Reliable
Durability Policy	Volatile
Namespaces	

Conclusion

In this section of our robot setup guide, we have discussed the importance of sensor information for different tasks associated with Nav2. More specifically, tasks such as mapping (SLAM), localization (AMCL), and perception (costmap) tasks.

We also had a discussion on the common types of sensor messages in Nav2 which standardize the message formats for different sensor vendors. We also discussed how to add sensors to a simulated robot using Gazebo and how to verify that the sensors are working correctly through RViz.

Lastly, we set up a basic configuration for the `nav2_costmap_2d` package using different layers to produce a global and local costmap. We then verify our work by visualizing these costmaps in RViz.

4.4.5 Setting Up the Robot's Footprint

In this guide, we will discuss how to configure the footprint of your robot for the navigation algorithms used by Nav2. We will also show a sample footprint configuration on `sam_bot`, the simulated robot that we have been building in this series of setup guides. Lastly, we will also show the visualization of `sam_bot`'s footprint in RViz to ensure that we have set it up correctly.

Footprint Introduction

The footprint outlines the robot's 2D shape when projected to the ground and is primarily used by Nav2 to avoid collisions during planning. The algorithms involved in this task makes sure that the robot does not collide with the obstacles in the costmap while it computes the robot's paths or plans.

The footprint is set up using the `footprint` or `robot_radius` parameter of the global and local costmaps which we tackled in the previous tutorials (*Setting Up Sensors Guide*). The value defined in the `footprint` parameter is an ordered vector of 2-D points defining the robot's footprint with the `base_link` frame as the origin. The first and last points in the vector are joined into the last line segment to close the footprint's shape. As an alternative, you may also use the `robot_radius` parameter wherein circular footprint is automatically generated and centered at `base_link`. In cases both the `footprint` and `robot_radius` parameters have been defined in the configuration, the `footprint` is used.

See also:

A section in the previous guide, *Configuring nav2_costmap_2d*, explains how to configure basic costmap parameters. Please refer to that guide for more details on costmap configuration.

For the global costmap footprint, the decision to choose between the `robot_radius` (circular) or `footprint` (polygon) parameter depends on the robot, its environment, and the path planning algorithm you will use. Even if you are working with a non-circular robot, there may be situations where a circular footprint is acceptable. For example, path planning algorithms like NavFn assume that the robot is circular since it only checks for collision per grid cell, so it will not be necessary to outline the robot's exact shape for its footprint. On the other hand, algorithms such as Smac Planner's Hybrid-A* perform collision checking on the robot's polygon-shaped footprint if possible and necessary. Hence, it might be useful to use a polygon-shaped footprint. Another example is having a small RC car sized robot roaming a warehouse. This robot is so small it won't need to make confined maneuvers – thusly approximating it with the largest cross-sectional radius is a good time-saving optimization.

For the local costmap footprint, it is typical for non-circular robots to be set up with `footprint` (polygon). Some situations where this is not recommended is when you do not have enough computing resources to implement collision avoidance algorithms on a polygon-shaped footprint. Another possible reason to use `robot_radius` (circular) for the local costmap is when the robot is very small relative to its environment such that precise collision avoidance is not necessary. However, generally the local trajectory planner should use the actual footprint polygon of the robot.

Configuring the Robot's Footprint

In this section, we will configure the footprint of sam_bot such that footprint (polygon) is used for the local costmap and robot_radius (circular) is used for the global costmap. We will utilize the default configuration file of Nav2 with a modified footprint parameter for the global and local costmaps.

Note: The complete source code for sam_bot can be found in navigation2_tutorials repository.

Under the config directory, create a new file named nav2_params.yaml. Next, copy the contents of config/nav2_params.yaml and paste them into the newly created file. The contents of config/nav2_params.yaml are copied from the default configuration file of Nav2 but with changes in the footprint and robot_radius parameters to match the shape of sam_bot.

See also:

The default configuration file for Nav2 can be found in the official Navigation2 repository.

Below is the code snippet from nav2_params.yaml defining the local costmap footprint. In this configuration file, the footprint parameter of the local costmap has already been set with a rectangular-shaped footprint. This box is centered at the base_link frame of sam_bot.

```
resolution: 0.05
footprint: "[ [0.21, 0.195], [0.21, -0.195], [-0.21, -0.195], [-0.21, 0.195] ]"
plugins: ["voxel_layer", "inflation_layer"]
```

For the global costmap, we have already set the robot_radius parameter to create a circular footprint that matches sam_bot's size and centered at base_link. The parameter that was modified is shown in the code snippet below.

```
use_sim_time: True
robot_radius: 0.3
resolution: 0.05
```

Build, Run and Verification

We will now confirm that we have properly set up sam_bot's footprint.

First, we launch launch/display.launch.py to launch the robot state publisher, spawn sam_bot in Gazebo, and visualize sam_bot and its footprint in Rviz. The robot state publisher publishes the base_link => sensors transforms defined in sam_bot's URDF, while Gazebo's differential drive plugin publishes the odom => base_link transform. Open a new terminal and execute the lines below.

```
colcon build
. install/setup.bash
ros2 launch sam_bot_description display.launch.py
```

After launching display.launch.py, RViz and Gazebo should open. RViz should show sam_bot, the frames of sam_bot's parts, and the odom frame without errors. Gazebo should show sam_bot with a sphere and a cube detectable by sambot's lidar sensor.

Next, we will publish the map => odom transform using the static_transform_publisher. We publish the map => odom transform as static in this guide as a simple way to publish the transform and visualize the footprint. Open a new terminal and execute the lines below.

```
ros2 run tf2_ros static_transform_publisher 0 0 0 0 0 0 map odom
```

The `map` => `odom` transform should now be being published and the `map` frame should be added in RViz without errors.

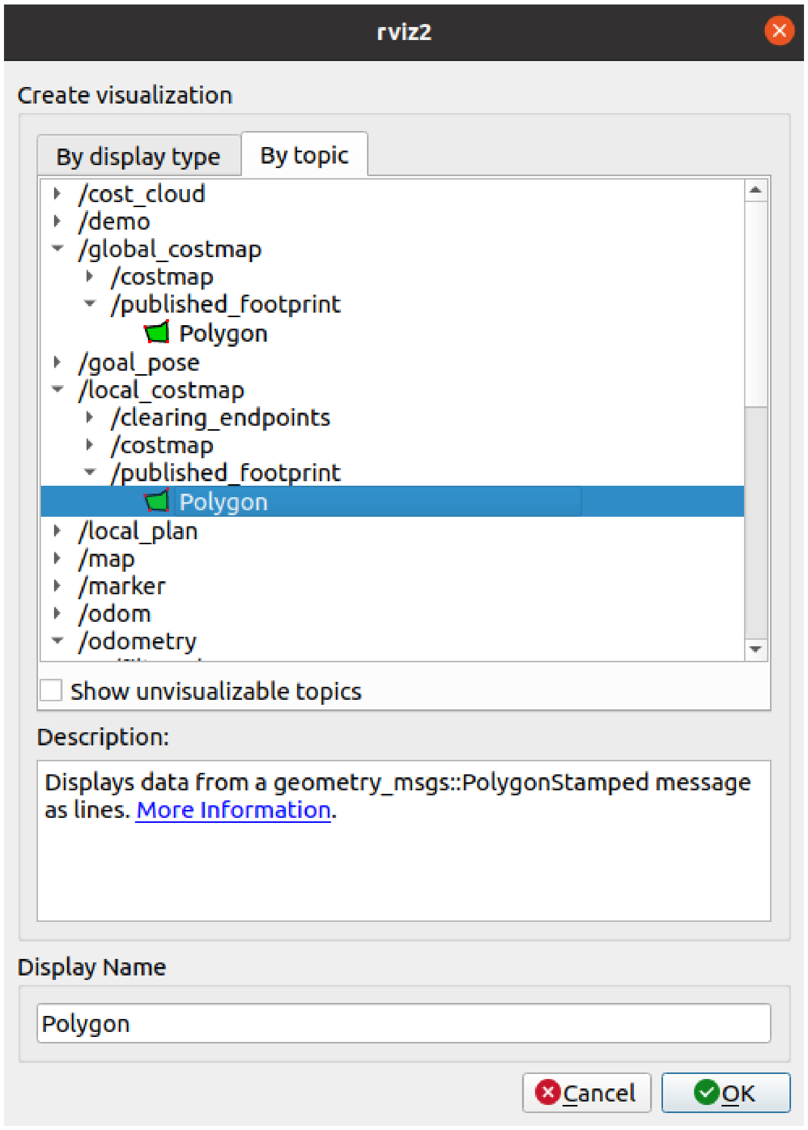
Lastly, we will launch Nav2 using the `nav2_params.yaml` configuration file we just made and `navigation_launch.py`, the built-in launch file of `nav2_bringup`. Open a new terminal and execute the following:

```
ros2 launch nav2_bringup navigation_launch.py params_file:=<full/path/to/config/nav2_
↪params.yaml>
```

We should now be able to visualize the footprints in RViz, which will be discussed in the next section.

Visualizing Footprint in RViz

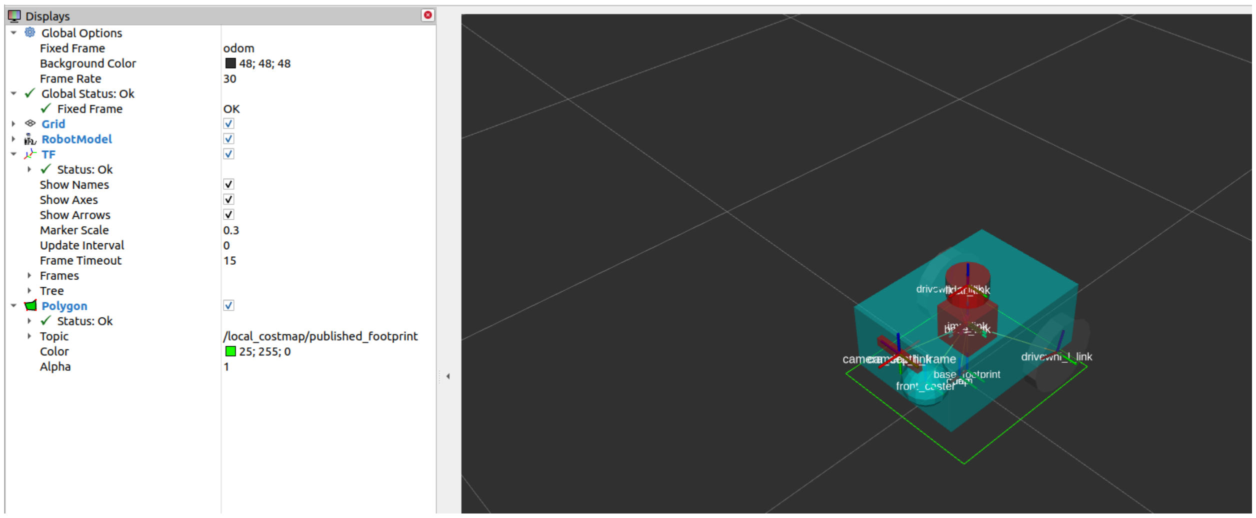
To visualize the footprint of the local costmap, click the add button at the bottom-left part of the RViz window. Under the `By topic` tab, select the `Polygon` under the `/local_costmap/published_footprint` topic, as shown below.



Displays data from a geometry_msgs::PolygonStamped message as lines. More Information.

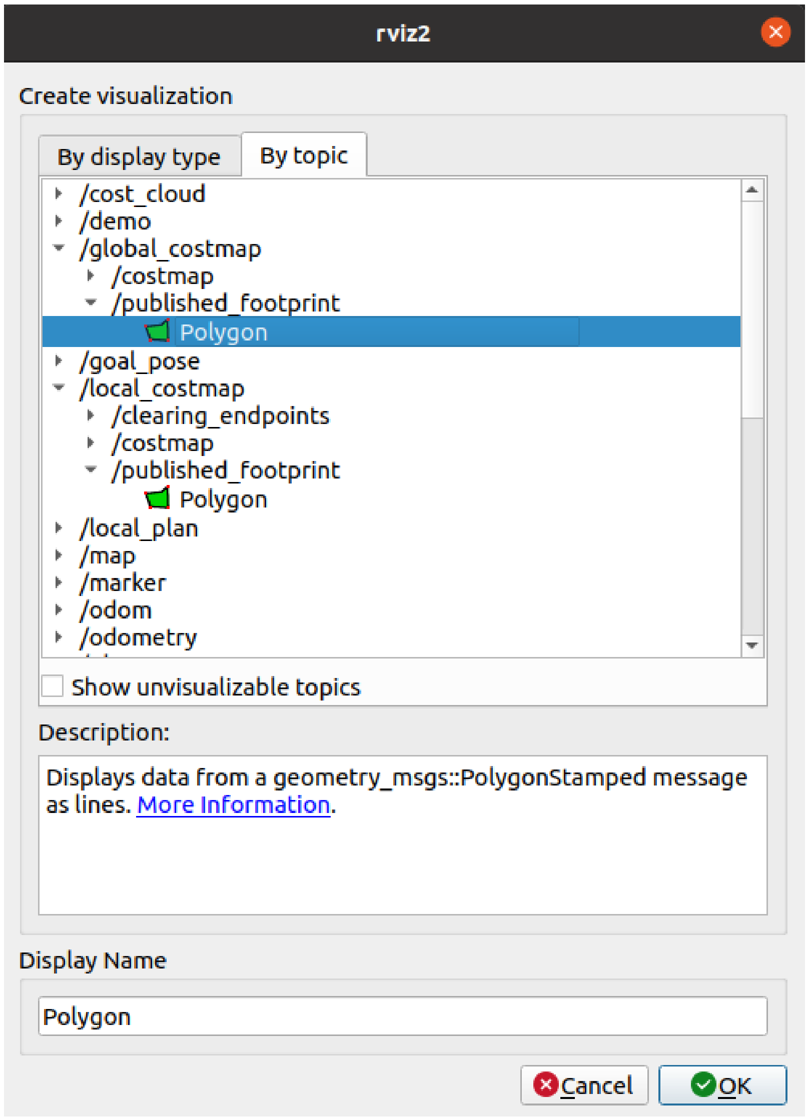
Polygon

Set the fixed frame in RViz to `odom` and you should see the rectangular-shaped footprint of `sam_bot`:



Displays	
Global Options	
Fixed Frame	odom
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
TF	✓
✓ Status: Ok	
Show Names	✓
Show Axes	✓
Show Arrows	✓
Marker Scale	0.3
Update Interval	0
Frame Timeout	15
Frames	
Tree	
Polygon	✓
✓ Status: Ok	
Topic	/local_costmap/published_footprint
Color	🟩 25; 255; 0
Alpha	1

On the other hand, for the global costmap, click the add button at the bottom-left part of the RViz window. Go to `By topic` tab then select the `Polygon` under the `/global_costmap/published_footprint` topic, as shown below.



rviz2

Create visualization

By display type | By topic

- /cost_cloud
- /demo
- /global_costmap
  - /costmap
  - /published_footprint
    - Polygon
- /goal_pose
- /local_costmap
  - /clearing_endpoints
  - /costmap
  - /published_footprint
    - Polygon
- /local_plan
- /map
- /marker
- /odom
- /odometry

☐ Show unvisualizable topics

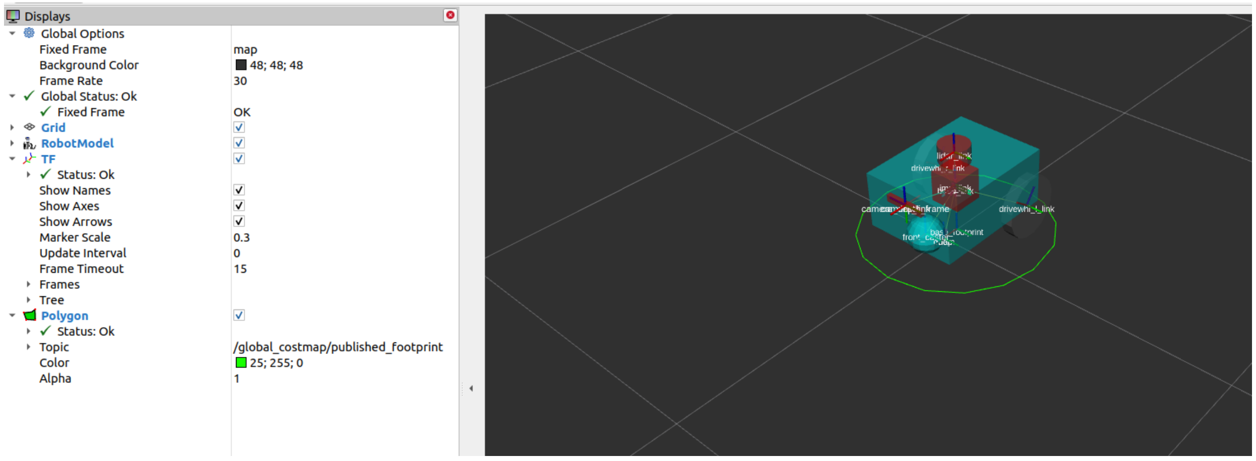
Description:

Displays data from a geometry_msgs::PolygonStamped message as lines. More Information.

Display Name

Polygon

Set the fixed frame in RViz to `map` and you should see the circular footprint of `sam_bot`:



Displays

Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
TF	✓
✓ Status: Ok	
Show Names	✓
Show Axes	✓
Show Arrows	✓
Marker Scale	0.3
Update Interval	0
Frame Timeout	15
Frames	
Tree	
Polygon	✓
✓ Status: Ok	
Topic	/global_costmap/published_footprint
Color	🟩 25; 255; 0
Alpha	1

Conclusion

In this guide, we have shown how to configure a circular and polygon-shaped footprint for your robot. This footprint is important since it plays a major role in Nav2's pathfinding algorithms function.

As a demo, we have configured the costmap footprint parameters of `sam_bot`. We set the local costmap to use a polygon-shaped footprint following `sam_bot`'s shape while we set the the global costmap to use a circular footprint. Lastly, we visualized and confirmed the footprints of the local and global costmaps in RViz.

4.4.6 Setting Up Navigation Plugins

In this part of the guide, we discuss how your robot can utilize different planner and controller algorithms to complete navigation tasks. We will discuss some of the available algorithm plugins you can use based on your robot type and environment.

Planner and Controller Servers

Navigation algorithms are implemented in Nav2 through the use of plugins running on ROS action servers - the planner, controller and behavior servers (among others). For this section, we discuss the planner and controller servers, which are the heart of the navigation stack. These servers may implement one or more algorithm plugins each with its own configuration tailored for a specific action or robot state. This guide will highlight the different algorithms based on the type of robot and environment the robot is deployed in. This guide will not cover behaviors, smoothers, etc as those are dependent on application and not hardware/environment to offer generalized advice.

The planner server is responsible for implementing algorithms that compute the robot's path. For example, one plugin can be configured to compute a simple shortest path between two relatively near locations while another plugin computes for a path to locations that cover the entire robot environment.

The controller server generates the appropriate control efforts needed for a robot to complete a task in its local environment. These tasks include but are not limited to: following a path generated by the planner server, avoiding dynamic obstacles along this path, and even charging at a docking station.

As mentioned before, the planner and controller servers host a map of one or multiple plugins wherein a certain plugin will be used for a certain environment, scenario, or task. For instance, the controller server can have a plugin for following a path when in long corridors to stay in the middle of the aisle, and then another plugin for avoiding dynamic obstacles in a crowded place. Selecting which planning algorithm to execute based on the robot's task can be done through the behavior tree of your navigation system or application server.

See also:

For a more in-depth discussion on Navigation Servers, we suggest to have a look at the Navigation Servers section under the Navigation Concepts category.

Selecting the Algorithm Plugins

In this section, we discuss some of the available algorithm plugins for the planner and controller servers. We also discuss the purpose of each algorithm, and for which type of robot they are recommended to be used. Lastly, we show some sample yaml configuration that specifies the plugin to be used for each server.

Note: The algorithm plugins you can use are not limited to the ones mentioned in this section. You may create custom plugins as well and new plugins are added to Nav2 regularly. For tutorials on how to write your own plugins, please see Writing a New Planner Plugin and Writing a New Controller Plugin. The list of all available Nav2 plugins and their descriptions can be found in Navigation Plugins Section.

Planner Server

The algorithm plugins for the planner server find the robot's path using a representation of its environment captured by its different sensors. Some of these algorithms operate by searching through the environment's grid space while others expand the robot's possible states while accounting for path feasibility.

As mentioned, the planner server may utilize plugins that work on the grid space such as the `NavFn Planner`, `Smac Planner 2D`, and `Theta Star Planner`. The NavFn planner is a navigation function planner that uses either Dijkstra or A*. Next, the Smac 2D planner implements a 2D A* algorithm using 4 or 8 connected neighborhoods with a smoother and multi-resolution query. Lastly, the Theta Star planner is an implementation of Theta* using either line of sight to create non-discretely oriented path segments.

One issue you may encounter when using algorithms that work on the grid space is that there is no guarantee that a drivable path can be generated for any type of robot. For example, it is not guaranteed that the `NavFn Planner` can plan a feasible path for a non-circular robot in a tight space since it uses the circular footprint of a robot (by approximating the robot's largest cross-sectional radius) and checks for collisions per costmap grid cell. In addition, these algorithms are also not suitable for ackermann and legged robots since they have turning constraints. That being said, these plugins are best used on robots that can drive in any direction or rotate safely in place, such as **circular differential and circular omnidirectional** robots.

Another planner plugin is the Smac Hybrid-A* planner that supports **arbitrary shaped ackermann and legged** robots. It is a highly optimized and fully reconfigurable Hybrid-A* implementation supporting Dubin and Reeds-Shepp motion models. This algorithm expands the robot's candidate paths while considering the robot's minimum turning radius constraint and the robot's full footprint for collision avoidance. Thus, this plugin is suitable for arbitrary shaped robots that require full footprint collision checking. It may also be used for high-speed robots that must be navigated carefully to not flip over, skid, or dump load at high speeds.

There is also the `Smac Lattice planner` plugin which is based on a State Lattice planner. This plugin functions by expanding the robot state space while ensuring the path complies with the robot's kinematic constraints. The algorithm provides minimum control sets which allows it to support **differential, omnidirectional, and ackermann** vehicles of any shape and size with minimal reconfiguration.

Summary

Plugin Name	Supported Robot Types
NavFn Planner	Circular Differential, Circular Omnidirectional
Smac Planner 2D	
Theta Star Planner	
Smac Hybrid-A* Planner	Non-circular or Circular Ackermann, Non-circular or Circular Legged
Smac Lattice Planner	Non-circular Differential, Non-circular Omnidirectional

Example Configuration

```yaml
planner_server:
  ros__parameters:
    planner_plugins: ['GridBased']
    GridBased:
      plugin: 'nav2_navfn_planner/NavfnPlanner'
```

An example configuration of the planner server is shown above. The `planner_plugins` parameter accepts a list of mapped planner plugin names. For each plugin namespace defined in `planner_plugins` (`GridBased` in our example), we specify the type of plugin to be loaded in the `plugin` parameter. Additional configurations must then be specified in this namespace based on the algorithm to be used. Please see the Configuration Guide for more details.

Controller Server

The default controller plugin is the DWB controller. It implements a modified Dynamic Window Approach (DWA) algorithm with configurable plugins to compute the control commands for the robot. This controller makes use of a `Trajectory Generator plugin` that generates the set of possible trajectories. These are then evaluated by one or more `Critic plugins`, each of which may give a different score based on how they are configured. The sum of the scores from these `Critic plugins` determine the overall score of a trajectory. The best scoring trajectory then determines the output command velocity.

The `DWB controller` can be used on **circular or non-circular differential, and circular or non-circular omnidirectional** robots. It may also be configured for **ackerman and legged** robots if it is given a `Trajectory Generation plugin` that produces a set of possible trajectories that considers the robot's minimum curvature constraint.

Another example of a controller server plugin is the TEB controller which is an MPC time optimal controller. It implements the Timed Elastic Band (TEB) approach which optimizes the robot's trajectory based on its execution time, distance from obstacles, and feasibility with respect to the robot's kinematic constraints. This controller can be used on **differential, omnidirectional, ackermann, and legged** robots.

The last example for this section is the Regulated Pure Pursuit controller (RPP) . This controller implements a variant of the pure pursuit algorithm with added regulation heuristic functions to manage collision and velocity constraints. This variation is implemented to target the needs of service or industrial robots and is suitable for use with **differential, ackermann, and legged** robots.

Summary

Plugin Name	Supported Robot Types	Task
DWB controller	Differential, Omnidirectional	Dynamic obstacle avoidance
TEB Controller	Differential, Omnidirectional, Ackermann, Legged	
RPP controller	Differential, Ackermann, Legged	Exact path following

All of these algorithms work for both circular and non-circular robots.

Example Configuration

```
controller_server:
  ros__parameters:
    controller_plugins: ["FollowPath"]
    FollowPath:
       plugin: "dwb_core::DWBLocalPlanner"
```

Shown above is a sample basic configuration of the controller server. The list of mapped names for controller plugins are defined in the `controller_plugins` parameter. Similar to the planner server, each namespace defined in the `controller_plugins` (`FollowPath` in our example) must define the type of plugin it will use in the `plugin` parameter. Additional configurations must also be made for the selected algorithm in the namespace. Please see the Configuration Guide for more details.

Note: The planner and controller servers, along with the other servers of Nav2, are launched in ROS 2 through lifecycle nodes. Lifecycle nodes allow for easier bringup and teardown of the servers. Lifecycle node management will be discussed in the next tutorial.

Conclusion

In this tutorial, we have discussed the roles and configuration of Nav2's planner and controller servers. To summarize, these servers host a map of one or more algorithm plugins which are selected depending on your robot's structure and surroundings. We also described a few of the available plugins for the planner and controller servers to help you identify which plugins are suitable for your robot. Lastly, we also provided some simple configuration examples to show how these plugins are instantiated on the servers. You can refer to the configuration guide of the algorithm plugin you will select for more details on its configuration parameters.

4.5 General Tutorials

Navigation2 Tutorials

4.5.1 Navigating with a Physical Turtlebot 3

- *Overview*
- *Requirements*
- *Tutorial Steps*

Overview

This tutorial shows how to control and navigate Turtlebot 3 using the ROS 2 Nav2 on a physical Turtlebot 3 robot. Before completing this tutorials, completing *Getting Started* is highly recommended especially if you are new to ROS and Nav2.

This tutorial may take about 1 hour to complete. It depends on your experience with ROS, robots, and what computer system you have.

Requirements

You must install Nav2, Turtlebot3. If you don't have them installed, please follow *Getting Started*.

Tutorial Steps

0- Setup Your Enviroment Variables

Run the following commands first whenever you open a new terminal during this tutorial.

- `source /opt/ros/<ros2-distro>/setup.bash`
- `export TURTLEBOT3_MODEL=waffle`

1- Launch Turtlebot 3

You will need to launch your robot's interface,

```
ros2 launch turtlebot3_bringup robot.launch.py use_sim_time:=False
```

2- Launch Nav2

You need to have a map of the environment where you want to Navigate Turtlebot 3, or create one live with SLAM.

In case you are interested, there is a use case tutorial which shows how to use Nav2 with SLAM. *(SLAM) Navigating While Mapping*.

Required files:

- `your-map.map`
- `your-map.yaml`

`<your_map>.yaml` is the configuration file for the map we want to provide Nav2. In this case, it has the map resolution value, threshold values for obstacles and free spaces, and a map file location. You need to make sure these values are correct. More information about the map.yaml can be found here.

Launch Nav2. If you set autostart:=False, you need to click on the start button in RViz to initialize the nodes. Make sure *use_sim time* is set to **False**, because we want to use the system time instead of the time simulation time from Gazebo.

```
ros2 launch nav2_bringup bringup_launch.py use_sim_time:=False
autostart:=False map:=/path/to/your-map.yaml
```

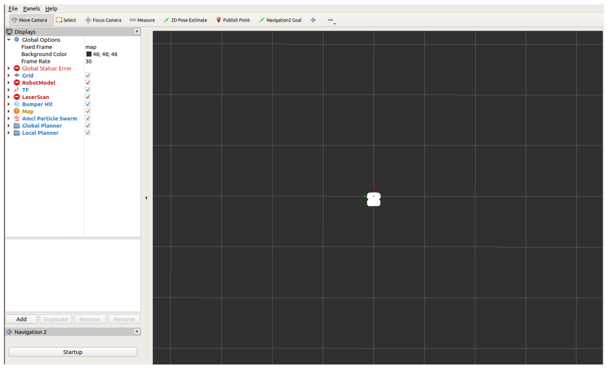
Note: Don't forget to change **/path/to/your-map.yaml** to the actual path to the your-map.yaml file.

3- Launch RVIZ

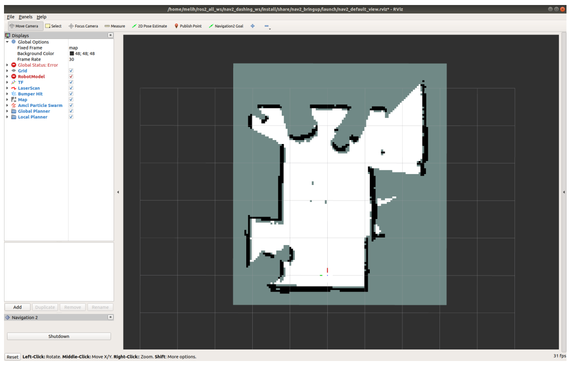
Launch RVIZ with a pre-defined configuration file.

```
ros2 run rviz2 rviz2 -d $(ros2 pkg prefix nav2_bringup)/share/
nav2_bringup/rviz/nav2_default_view.rviz
```

Now, you should see a shadow of Turtlebot 3 robot model in the center of the plot in Rviz. Click on the Start button (Bottom Left) if you set the auto_start parameter to false. Then, the map should appear in RViz.



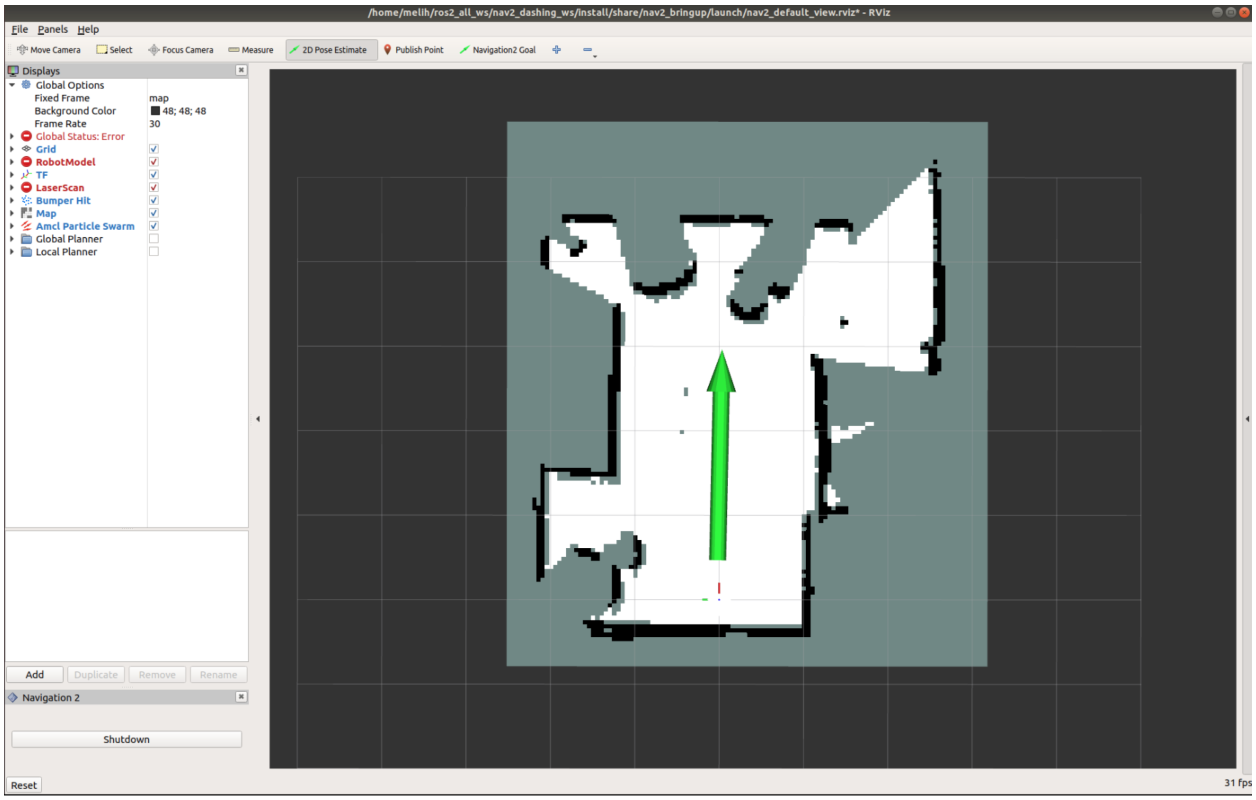
Fixed Frame map
Background Color 48; 48; 48
Frame Rate 30



4- Initialize the Location of Turtlebot 3

First, find where the robot is on the map. Check where your robot is in the room.

Set the pose of the robot in RViz. Click on the 2D Pose Estimate button and point the location of the robot on the map. The direction of the green arrow is the orientation of Turtlebot.

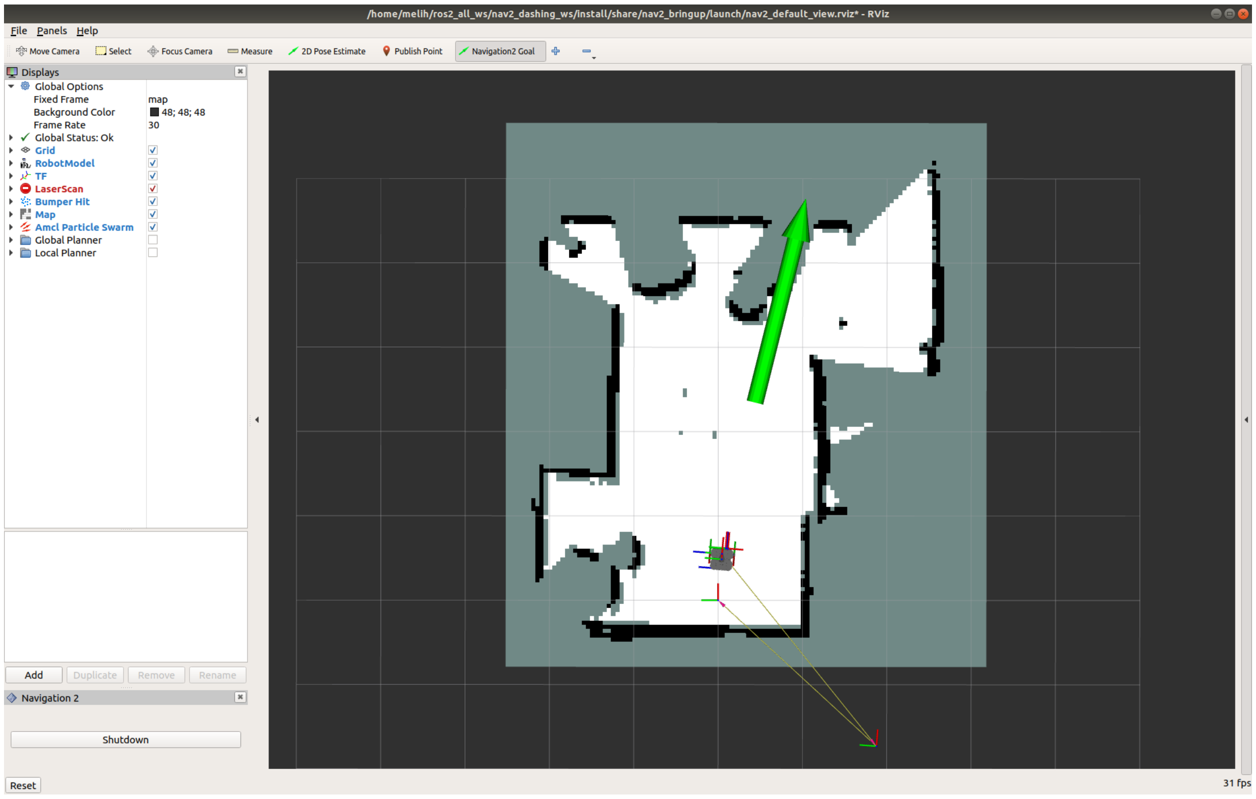


Now, the 3D model of Turtlebot should move to that location. A small error in the estimated location is tolerable.

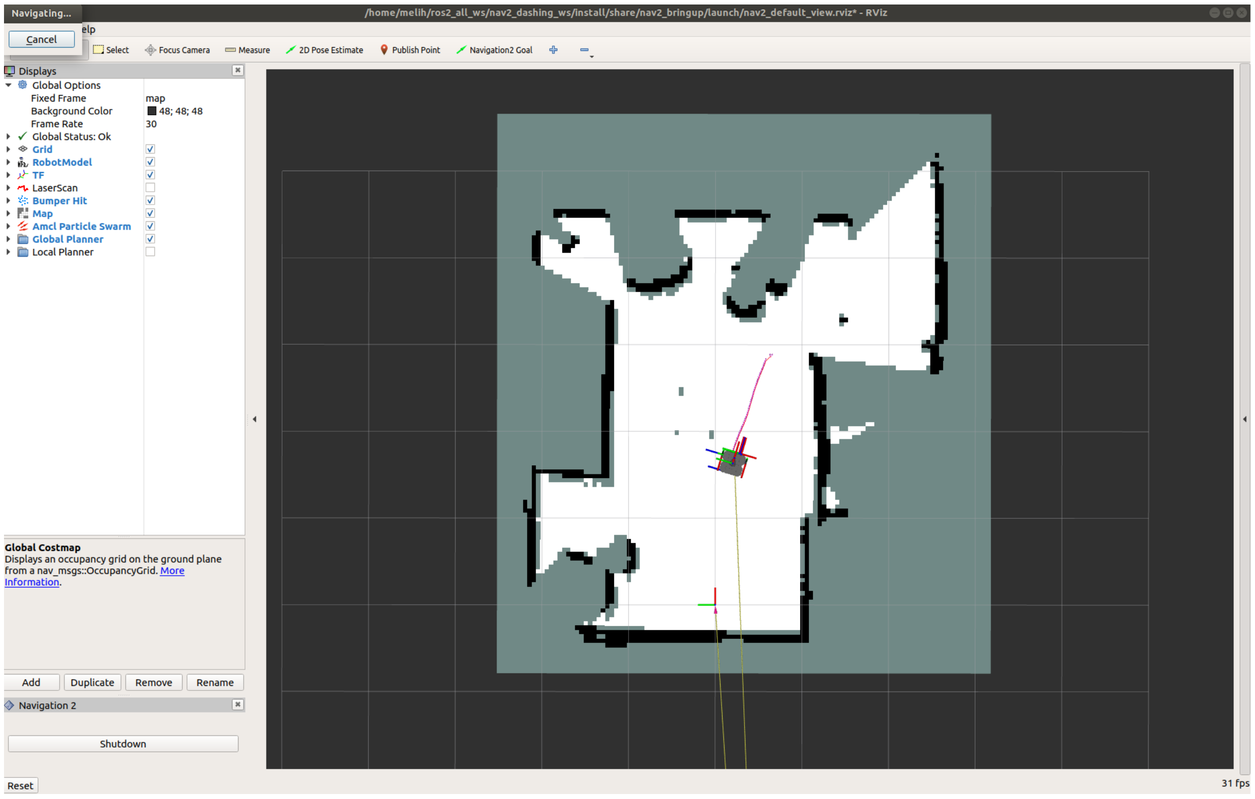
5- Send a Goal Pose

Pick a target location for Turtlebot on the map. You can send Turtlebot 3 a goal position and a goal orientation by using the **Nav2 Goal** or the **GoalTool** buttons.

Note: Nav2 Goal button uses a ROS 2 Action to send the goal and the GoalTool publishes the goal to a topic.



Once you define the target pose, Nav2 will find a global path and start navigating the robot on the map.



Global Options
Fixed Frame map
Background Color 48; 48; 48
Frame Rate 30
Global Status: Ok

Global Costmap
Displays an occupancy grid on the ground plane from a nav_msgs::OccupancyGrid. More Information.

Now, you can see that Turtlebot 3 moves towards the goal position in the room. See the video below.

4.5.2 (SLAM) Navigating While Mapping

- *Overview*
- *Requirements*
- *Tutorial Steps*

Overview

This document explains how to use Nav2 with SLAM. The following steps show ROS 2 users how to generate occupancy grid maps and use Nav2 to move their robot around. This tutorial applies to both simulated and physical robots, but will be completed here on physical robot.

Before completing this tutorial, completing the *Getting Started* is highly recommended especially if you are new to ROS and Navigation2.

In this tutorial we'll be using SLAM Toolbox. More information can be found in the ROSCon talk for SLAM Toolbox

Requirements

You must install Navigation2, Turtlebot3, and SLAM Toolbox. If you don't have them installed, please follow *Getting Started*.

SLAM Toolbox can be installed via:

```
sudo apt install ros-<ros2-distro>-slam-toolbox
```

or from built from source in your workspace with:

```
git clone -b <ros2-distro>-devel git@github.com:stevemacenski/
slam_toolbox.git
```

Tutorial Steps

0- Launch Robot Interfaces

For this tutorial, we will use the turtlebot3. If you have another robot, replace with your robot specific interfaces. Typically, this includes the robot state publisher of the URDF, simulated or physical robot interfaces, controllers, safety nodes, and the like.

Run the following commands first whenever you open a new terminal during this tutorial.

- `source /opt/ros/<ros2-distro>/setup.bash`
- `export TURTLEBOT3_MODEL=waffle`

Launch your robot's interface and robot state publisher,

```
ros2 launch turtlebot3_bringup robot.launch.py
```

1- Launch Navigation2

Launch Navigation without nav2_amcl and nav2_map_server. It is assumed that the SLAM node(s) will publish to /map topic and provide the map->odom transform.

```
ros2 launch nav2_bringup navigation_launch.py
```

2- Launch SLAM

Bring up your choice of SLAM implementation. Make sure it provides the map->odom transform and /map topic. Run Rviz and add the topics you want to visualize such as /map, /tf, /laserscan etc. For this tutorial, we will use SLAM Toolbox.

```
ros2 launch slam_toolbox online_async_launch.py
```

3- Working with SLAM

Move your robot by requesting a goal through RViz or the ROS 2 CLI, ie:

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

You should see the map update live! To save this map to file:

```
ros2 run nav2_map_server map_saver_cli -f ~/map
```

4- Getting Started Simplification

If you're only interested in running SLAM in the turtlebot3 getting started sandbox world, we also provide a simple way to enable SLAM as a launch configuration. Rather than individually launching the interfaces, navigation, and SLAM, you can continue to use the `tb3_simulation_launch.py` with `slam` config set to true. We provide the instructions above with the assumption that you'd like to run SLAM on your own robot which would have separated simulation / robot interfaces and navigation launch files that are combined in `tb3_simulation_launch.py` for the purposes of easy testing.

```
ros2 launch nav2_bringup tb3_simulation_launch.py slam:=True
```

4.5.3 (STVL) Using an External Costmap Plugin

- *Overview*
- *Costmap2D and STVL*
- *Tutorial Steps*

Overview

This tutorial shows how to load and use an external plugin. This example uses the Spatio Temporal Voxel Layer (STVL) costmap pluginlib plugin as an example. STVL is a demonstrative pluginlib plugin and the same process can be followed for other costmap plugins as well as plugin planners, controllers, and behaviors.

Before completing this tutorial, please look at the previous two tutorials on navigation in simulation and physical hardware, if available. This tutorial assumes knowledge of navigation and basic understanding of costmaps.

Note: For Ubuntu 20.04 users before December 2021, there's a known issue with OpenVDB and its binaries with `libjmalloc`. If you see an error such as `Could not load library LoadLibrary error: /usr/lib/x86_64-linux-gnu/libjemalloc.so.2: cannot allocate memory in static TLS block`, it can be resolved with `export LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libjemalloc.so.2` until new binaries are released of OpenVDB.

Costmap2D and STVL

Costmap 2D is the data object we use to buffer sensor information into a global view that the robot will use to create plans and control efforts. Within Costmap2D, there are pluginlib plugin interfaces available to create custom behaviors loadable at runtime. Examples of included pluginlib plugins for Costmap2D are the Obstacle Layer, Voxel Layer, Static Layer, and Inflation Layer.

However, these are simply example plugins offered by the base implementation. Another available pluginlib plugin for Costmap2D in Navigation2 is STVL.

STVL is another 3D perception plugin similar to the Voxel Layer. A more detailed overview of how it works can be found in this repo, however it buffers 3D data from depth cameras, sonars, lidars, and more into a sparse volumetic world model and removes voxels over time proportional with a sensor model and time-based expiration. This can be especially useful for robots in highly dynamic envrionments and decreases the resource utilization for 3D sensor processing by up to 2x. STVL also treats 3D lidars and radars as first class citizens for support. The ROSCon talk for STVL can be found in this video.

Tutorial Steps

0- Setup

Follow the same process as in *Getting Started* for installing and setting up a robot for hardware testing or simulation, as applicable. Ensure ROS 2, Navigation2, and Gazebo are installed.

1- Install STVL

STVL can be installed in ROS 2 via the ROS Build Farm:

- `sudo apt install ros-<ros2-distro>-spatio-temporal-voxel-layer`

It can also be built from source by cloning the repository into your Navigation2 workspace:

- `git clone -b <ros2-distro>-devel git@github.com:stevemacenski/spatio_temporal_voxel_layer`

1- Modify Navigation2 Parameter

STVL is an optional plugin, like all plugins, in Costmap2D. Costmap Plugins in Navigation2 are loaded in the `plugin_names` and `plugin_types` variables inside of their respective costmaps. For example, the following will load the static and obstacle layer plugins into the name `static_layer` and `obstacle_layer`, respectively:

```yaml
global_costmap:
  global_costmap:
    ros__parameters:
      use_sim_time: True
      plugins: ["static_layer", "obstacle_layer"]
```

Note: For Galactic or later, `plugin_names` and `plugin_types` have been replaced with a single `plugins` string vector for plugin names. The types are now defined in the `plugin_name` namespace in the `plugin:` field (e.g. `plugin: MyPlugin::Plugin`). Inline comments in the code blocks will help guide you through this.

To load the STVL plugin, a new plugin name and type must be added. For example, if the application required an STVL layer and no obstacle layer, our file would be:

```
global_costmap:
  global_costmap:
    ros__parameters:
      use_sim_time: True
      plugins: ["static_layer", "stvl_layer"]
```

Similar to the Voxel Layer, after registering the plugin, we can add the configuration of the STVL layer under the namespace `stvl_layer`. An example fully-described parameterization of an STVL configuration is:

```
stvl_layer:
  plugin: "spatio_temporal_voxel_layer/SpatioTemporalVoxelLayer"
  enabled: true
  voxel_decay: 15.
  decay_model: 0
  voxel_size: 0.05
  track_unknown_space: true
  unknown_threshold: 15
  mark_threshold: 0
  update_footprint_enabled: true
  combination_method: 1
  origin_z: 0.0
  publish_voxel_map: true
  transform_tolerance: 0.2
  mapping_mode: false
  map_save_duration: 60.0
  observation_sources: pointcloud
  pointcloud:
    data_type: PointCloud2
    topic: /intel_realsense_r200_depth/points
    marking: true
    clearing: true
    obstacle_range: 3.0
    min_obstacle_height: 0.0
    max_obstacle_height: 2.0
    expected_update_rate: 0.0
    observation_persistence: 0.0
    inf_is_valid: false
    filter: "voxel"
    voxel_min_points: 0
    clear_after_reading: true
    max_z: 7.0
    min_z: 0.1
    vertical_fov_angle: 0.8745
    horizontal_fov_angle: 1.048
    decay_acceleration: 15.0
    model_type: 0
```

Please copy-paste the text above, with the `plugin_names` and `plugin_types` registration, into your `nav2_params.yaml` to enable STVL in your application. Make sure to change both the local and global costmaps.

Note: Pluginlib plugins for other Navigation2 servers such as planning, behavior, and control can be set in this same way.

2- Launch Navigation2

Follow the same process as in *Getting Started* to launch a simulated robot in gazebo with Navigation2. Navigation2 is now using STVL as its 3D sensing costmap layer.

3- RVIZ

With RViz open and `publish_voxel_map: true`, you can visualize the underlying data structure's 3D grid using the `{local, global}_costmap/voxel_grid` topics. Note: It is recommended in RViz to set the `PointCloud2` Size to your voxel size and the style to `Boxes` with a neutral color for best visualization.

4.5.4 Groot - Interacting with Behavior Trees

Connect to a remote server to visualize the status of a BehaviorTree in real-time.

- *Overview*
- *Visualize Behavior Trees*
- *Edit Behavior Trees*
- *Adding A Custom Node*

Overview

Groot is the companion application of the BehaviorTree.CPP library used to create, edit, and visualize behavior trees. Behavior Trees are deeply integrated into Nav2, used as the main method of orchestrating task server logic across a complex navigation and autonomy stack. Behavior Trees, in short BTs, consist of many nodes completing different tasks and control the flow of logic, similar to a Hierarchical or Finite State Machine, but organized in a tree structure. These nodes are of types: *Action*, *Condition*, *Control*, or *Decorator*, and are described in more detail in *Navigation Concepts* and BehaviorTree.CPP.

Writing a New Behavior Tree Plugin offers a well written example of creating a simple `Action` node if creating new BT nodes is of interest. This tutorial will focus solely on launching Groot, visualizing a Behavior Tree, and modifying that tree for a given customization, assuming a library of BT nodes. Luckily, Nav2 provides a robust number of BT nodes for your use out of the box, enumerated in *Navigation Plugins*.

A BT configuration file in BehaviorTree.CPP is an XML file. This is used to dynamically load the BT node plugins at run-time from the appropriate libraries mapped to their names. The XML format is defined in detail here. Therefore, Groot needs to have a list of nodes it has access to and important metadata about them like their type and ports (or parameters). We refer to this as the "palette" of nodes later in the tutorial.

In the video above you can see Groot side-by-side with RVIz and a test platform 100% equipped with ROS-enabled hardware from SIEMENS. Groot not only displays the current Behavior Tree while the robot is operating. Note: Before ROS 2 Humble, live Groot behavior tree monitoring during execution was supported in Nav2. This was removed due to buggy support in BT.CPP / Groot for changing behavior trees on the fly, see *Galactic to Humble* for more details.

Visualize Behavior Trees

To display a Behavior Tree like that in Figure **??**, we will first start the Groot executable. Out of the box, Groot can only display Behavior Trees and nodes that are from the defaults in BT.CPP, since it does not know anything about Nav2 or your other projects. Therefore, we must point Groot to our palette, or index, of Nav2 / custom behavior tree nodes:

1. Open Groot in editor mode. Now, Groot should look like in Figure **??**.
2. Select the *Load palette from file* option either via the context menu or the import icon in the top middle of the menu bar.
3. Open the file */path/to/navigation2/nav2_behavior_tree/nav2_tree_nodes.xml* to import all the custom behavior tree nodes used for navigation. This is the palette of Nav2 custom behavior tree nodes. Now, Groot should look like in Figure **??**.
4. Select *Load tree* option near the top left corner
5. Browse the tree you want to visualize, then select *OK*. The Nav2 BTs exist in */path/to/navigation2/nav2_bt_navigator/behavior_trees/*

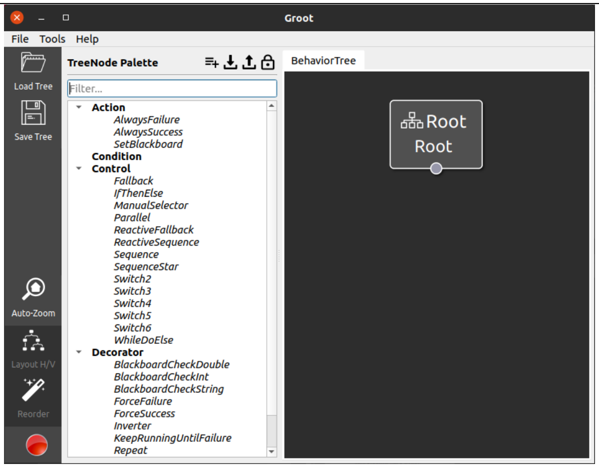


Figure 4.1: Default Editor View

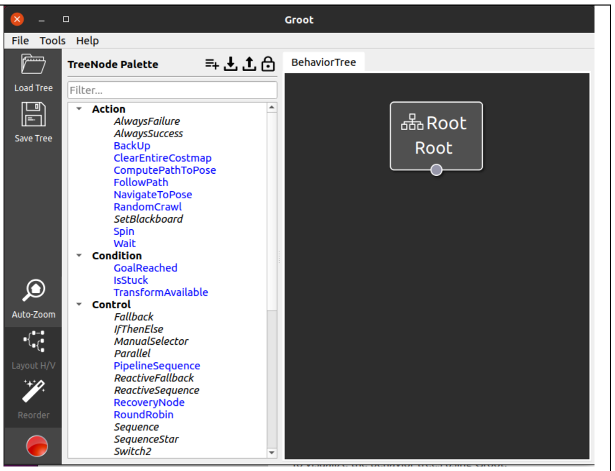


Figure 4.2: Editor with Custom Nodes loaded in blue

If you select the default tree *navigate_w_replanning_and_recovery.xml*, then a Groot editor should look like Figure **??**.

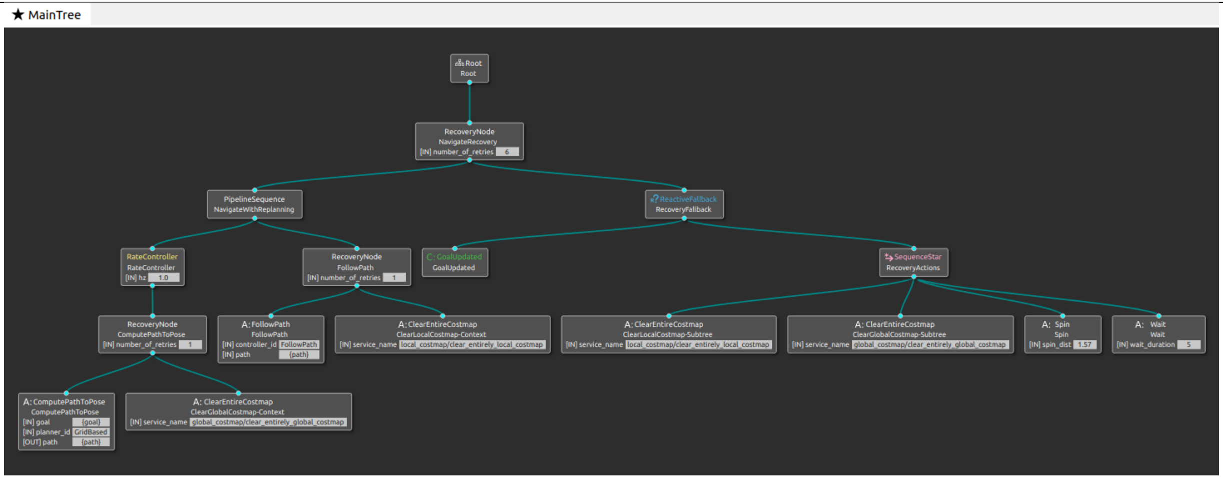


Figure 4.3: Full Nav2 Default BehaviorTree

Note: If a tree cannot be visualized because some nodes are missing in the palette, you might need to add it to your palette. While we try to keep Nav2's BT nodes and palettes in sync, if you notice one is missing, please file a ticket or pull request and we should have that updated quickly.

Edit Behavior Trees

Now that you have a Nav2 BT open in Groot in editor mode, you should be able to trivially modify it using the GUI. Starting from a screen like that shown in Figure **??**, you can pull in new nodes from the side panel to add them to the workspace. You may then connect the nodes using a "drag and drop" motion between the node's input and output ports to assemble the new nodes into the tree.

If you select a given node, you can change metadata about it such as its name or values of parameterizable ports. When you're done modifying, simply save the new configuration file and use that on your robot the next time!

Adding A Custom Node

Each node in the behavior tree holds a specialized function. Sometimes, its useful to create new nodes and add them to your palette during the design process - perhaps before the implementations themselves exist. This helps designers abstract away the implementation specifics of the nodes from the higher level logic of the tree itself and how they'd like to interact with a given node (e.g. type, ports, etc). Within Groot, you may create new custom nodes to add to your tree and export these new nodes back to your palette. Implementing the node itself needs to be done separately from Groot, which is described in *Writing a New Behavior Tree Plugin*.

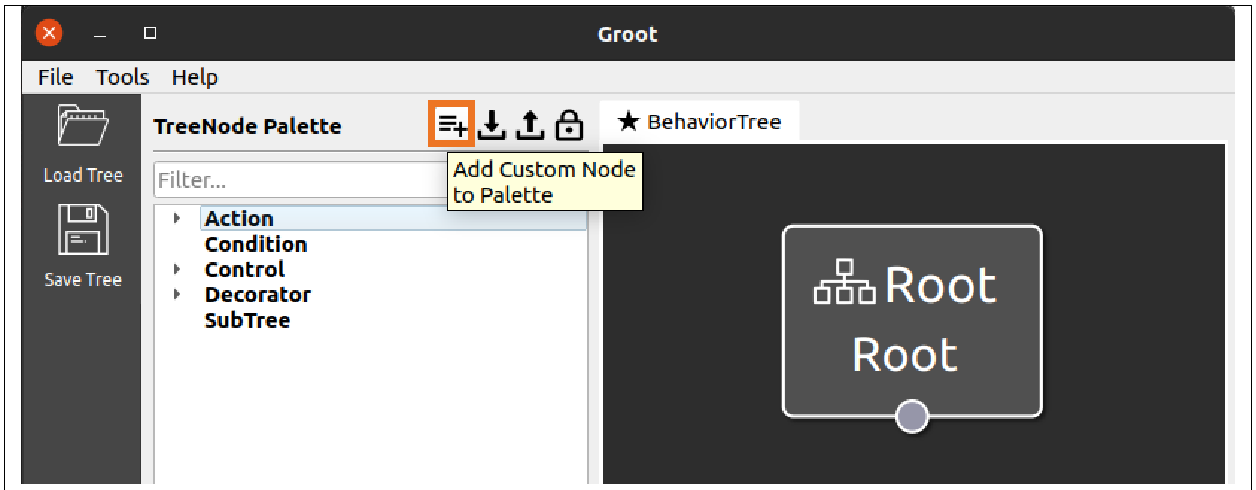
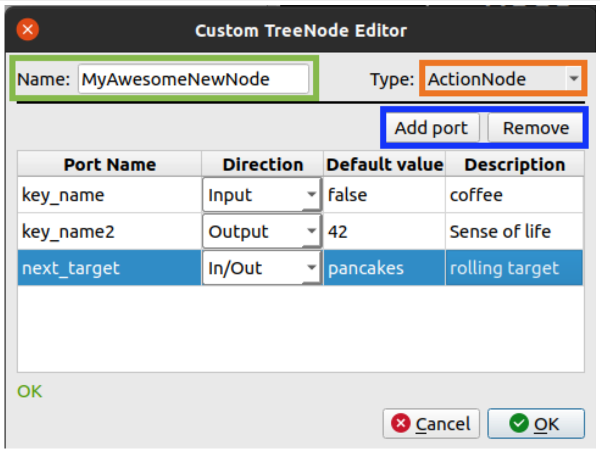


Figure 4.4: Create a new Custom Node

Creating a new custom node can be started by clicking the orange marked icon in Figure **??**, while Groot is in Editor mode. This should load a new window, similar to Figure **??**. In this new window, it asks you to fill in the metadata about this new node, in order to create it. It will ask you for standard information such as name (green box), type of node (orange box), and any optional ports for parameterization or access to blackboard variables (blue box).

After completing, select *OK* in Figure **??**, the new custom node should appear in blue in the *TreeNode Palette* as in Figure **??**.



Custom TreeNode Editor

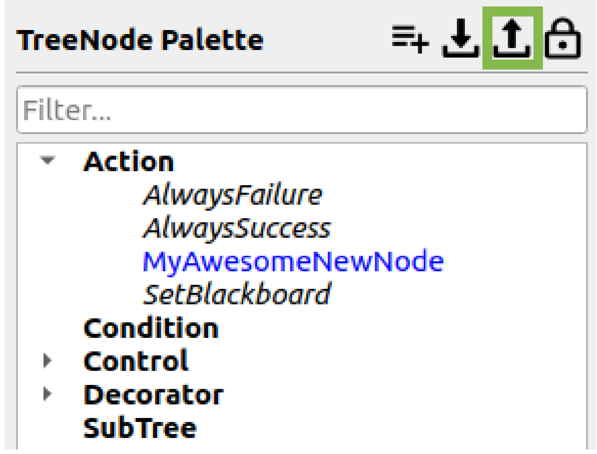
Name: MyAwesomeNewNode

Type: ActionNode

Port Name	Direction	Default value	Description
key_name	Input	false	coffee
key_name2	Output	42	Sense of life
next_target	In/Out	pancakes	rolling target

OK

Figure 4.5: UI to describing new Nodes



TreeNode Palette

- Action
  - AlwaysFailure
  - AlwaysSuccess
  - MyAwesomeNewNode
  - SetBlackboard
- Condition
- Control
- Decorator
- SubTree

Figure 4.6: Exporting the new Custom Node

Before starting to create a new BT based on the new custom nodes, it is recommend to export the newly created nodes to save in case of Groot crashing. This can be performed with the icon highlighted in green from Figure **??**. The resulting XML output from the node created in Figure **??** can be seen below. You can see more examples in Nav2's BT Node Palette XML.

```
<root>
    <TreeNodesModel>
        <Action ID="MyAwesomeNewNode">
            <input_port name="key_name" default="false">coffee</input_port>
            <output_port name="key_name2" default="42">Sense of life</output_port>
            <inout_port name="next_target" default="pancakes">rolling target</inout_
↪port>
        </Action>
    </TreeNodesModel>
</root>
```

4.5.5 Camera Calibration

- *Overview*
- *Requirements*
- *Tutorial Steps*

Overview

This tutorial shows how to obtain calibration parameters for monocular camera.

Requirements

1- Install Camera Calibration Parser, Camera Info Manager and Launch Testing Ament Cmake using operating system's package manager:

```
sudo apt install ros-<ros2-distro>-camera-calibration-parsers
```

```
sudo apt install ros-<ros2-distro>-camera-info-manager
```

```
sudo apt install ros-<ros2-distro>-launch-testing-ament-cmake
```

2- Image Pipeline need to be built from source in your workspace with:

```
git clone – b <ros2-distro> git@github.com:ros-perception/
image_pipeline.git
```

Also, make sure you have the following:

- A large checkerboard with known dimensions. This tutorial uses a 7x9 checkerboard with 20mm squares. **Calibration uses the interior vertex points of the checkerboard, so an "8x10" board uses the interior vertex parameter "7x9" as in the example below.** The checkerboard with set dimensions can be downloaded from here.
- A well-lit area clear of obstructions and other check board patterns
- A monocular camera publishing images over ROS

Tutorial Steps

1- Start a terminal in your GUI

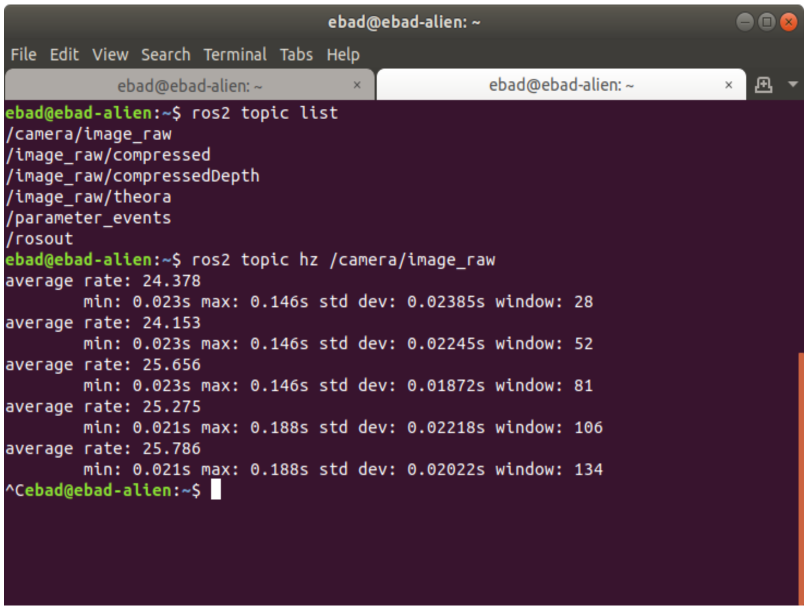
2- Launch the ROS driver for your specific camera.

3- Make sure camera is publishing images over ROS. This can be tested by running:

```
ros2 topic list
```

4- This will show you all the topics published make sure that there is an image_raw topic /camera/image_raw. To confirm that its a real topic and actually publishing check topic hz:

```
ros2 topic hz /camera/image_raw
```



```
ebad@ebad-alien:~$ ros2 topic list
/camera/image_raw
/image_raw/compressed
/image_raw/compressedDepth
/image_raw/theora
/parameter_events
/rosout
ebad@ebad-alien:~$ ros2 topic hz /camera/image_raw
average rate: 24.378
        min: 0.023s max: 0.146s std dev: 0.02385s window: 28
average rate: 24.153
        min: 0.023s max: 0.146s std dev: 0.02245s window: 52
average rate: 25.656
        min: 0.023s max: 0.146s std dev: 0.01872s window: 81
average rate: 25.275
        min: 0.021s max: 0.188s std dev: 0.02218s window: 106
average rate: 25.786
        min: 0.021s max: 0.188s std dev: 0.02022s window: 134
^Cebad@ebad-alien:~$
```

5- Start the camera calibration node

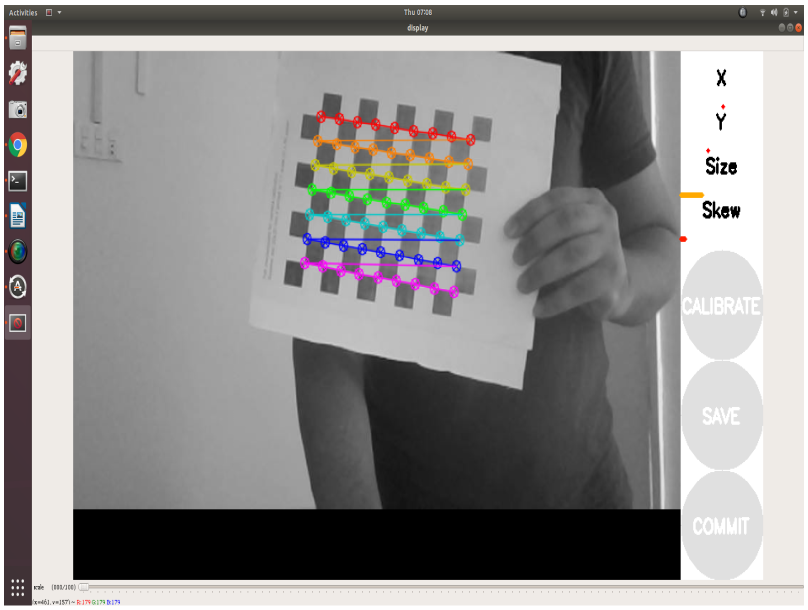
    ros2 run camera_calibration cameracalibrator --size 7x9 --square 0.02
    --ros-args -r image:=/my_camera/image_raw -p camera:=/my_camera

```
Camera Name:

-c, --camera_name
        name of the camera to appear in the calibration file


Chessboard Options:

You must specify one or more chessboards as pairs of --size and--square options.

  -p PATTERN, --pattern=PATTERN
                    calibration pattern to detect - 'chessboard','circles', 'acircles
↪','charuco'
  -s SIZE, --size=SIZE
                    chessboard size as NxM, counting interior corners (e.g. a␣
↪standard chessboard is 7x7)
  -q SQUARE, --square=SQUARE
                    chessboard square size in meters

ROS Communication Options:

 --approximate=APPROXIMATE
                    allow specified slop (in seconds) when pairing images from␣
↪unsynchronized stereo cameras
 --no-service-check
                    disable check for set_camera_info services at startup
```

(continues on next page)

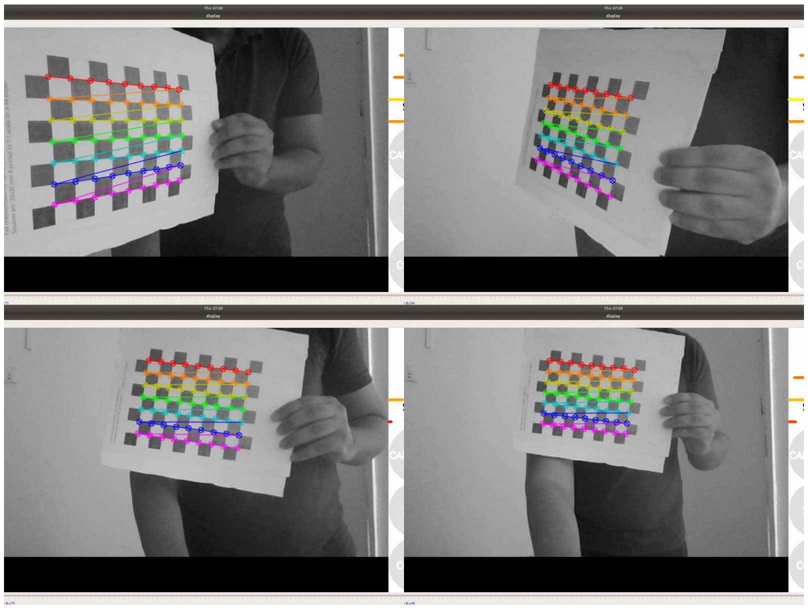
(continued from previous page)

```
Calibration Optimizer Options:

 --fix-principal-point
                     fix the principal point at the image center
 --fix-aspect-ratio
                     enforce focal lengths (fx, fy) are equal
 --zero-tangent-dist
                     set tangential distortion coefficients (p1, p2) to
                     zero
 -k NUM_COEFFS, --k-coefficients=NUM_COEFFS
                     number of radial distortion coefficients to use (up to
                     6, default 2)
 --disable_calib_cb_fast_check
                     uses the CALIB_CB_FAST_CHECK flag for findChessboardCorners

    This will open a calibration window which highlight the checkerboard.
```

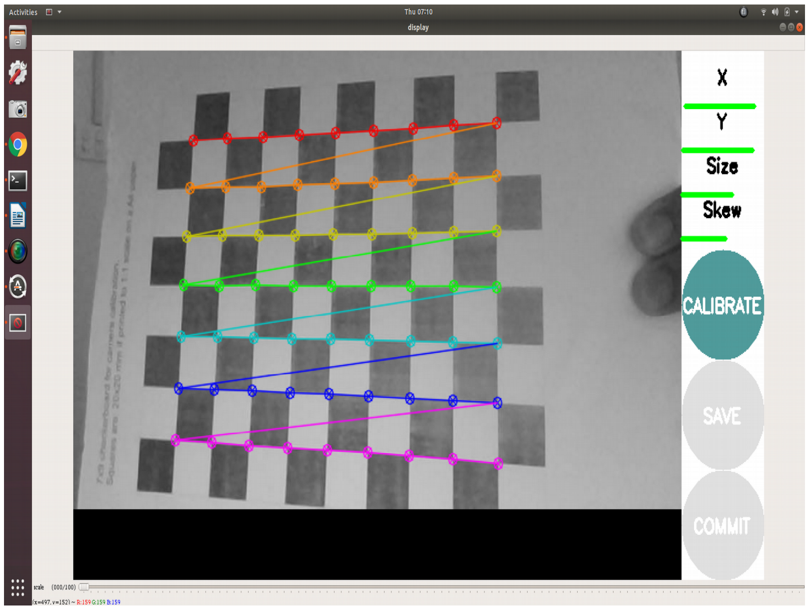


6- In order to get a good calibration you will need to move the checkerboard around in the camera frame such that:

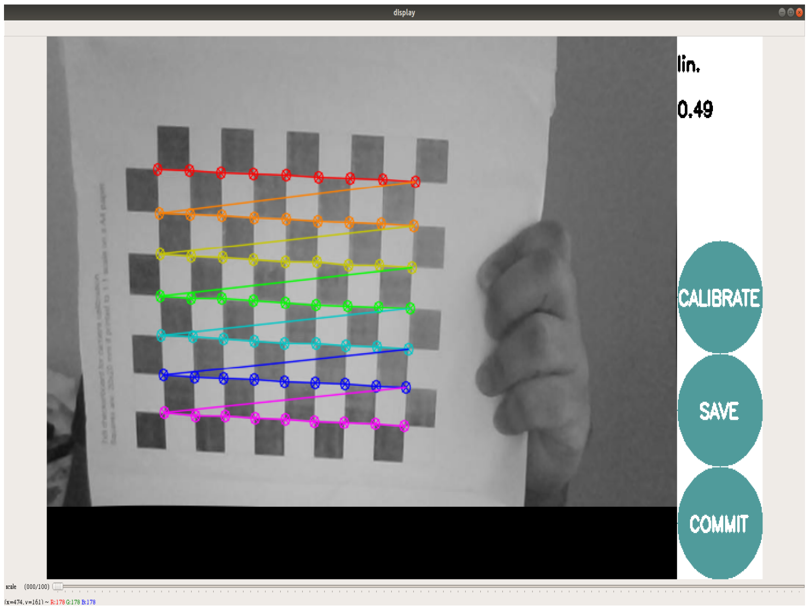
- **checkerboard on the camera’s left, right, top and bottom of field of view** ∘ X bar - left/right in field of view
    - ∘ Y bar - top/bottom in field of view
    - ∘ Size bar - toward/away and tilt from the camera
- checkerboard filling the whole field of view
- checkerboard tilted to the left, right, top and bottom (Skew)

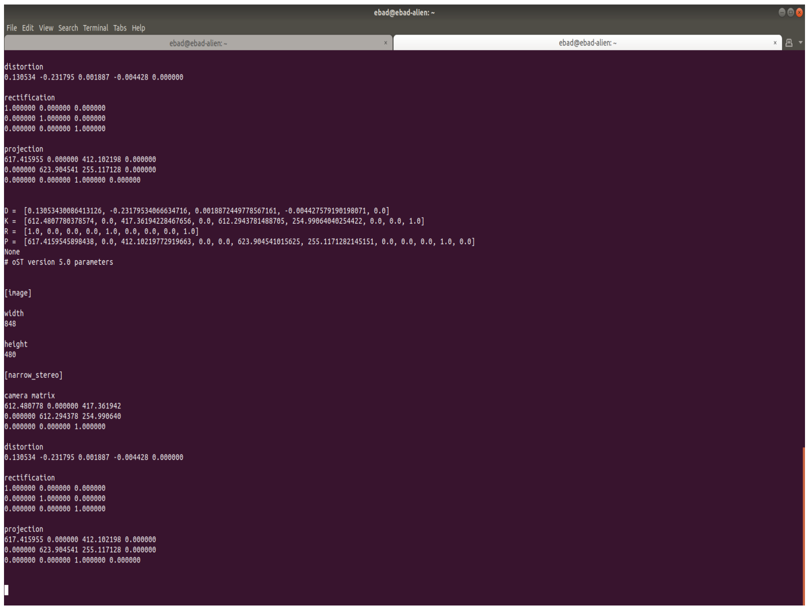


7- As the checkerboard is moved around the 4 bars on the calibration sidebar increases in length. When all then the 4 bars are green and enough data is available for calibration the CALIBRATE button will light up. Click it to see the results. It takes around the minute for calibration to take place.



8- After the calibration is completed the save and commit buttons light up. And you can also see the result in terminal.





```
distortion
0.130534 -0.231795 0.001887 -0.004428 0.000000

rectification
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 1.000000

projection
617.415955 0.000000 412.102198 0.000000
0.000000 623.904541 255.117128 0.000000
0.000000 0.000000 1.000000 0.000000


D =  [0.13053430086413126, -0.23179534066634716, 0.0018872449778567161, -0.004427579190198071, 0.0]
K =  [612.4807780378574, 0.0, 417.36194228467656, 0.0, 612.2943781488705, 254.99064040254422, 0.0, 0.0, 1.0]
R =  [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
P =  [617.4159545898438, 0.0, 412.10219772919663, 0.0, 0.0, 623.904541015625, 255.1171282145151, 0.0, 0.0, 0.0, 1.0, 0.0]
None
# oST version 5.0 parameters


[image]

width
848

height
480

[narrow_stereo]

camera matrix
612.480778 0.000000 417.361942
0.000000 612.294378 254.990640
0.000000 0.000000 1.000000

distortion
0.130534 -0.231795 0.001887 -0.004428 0.000000

rectification
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 1.000000

projection
617.415955 0.000000 412.102198 0.000000
0.000000 623.904541 255.117128 0.000000
0.000000 0.000000 1.000000 0.000000
```

9-Press the save button to see the result. Data is saved to “/tmp/calibrationdata.tar.gz”

10-To use the the calibration file unzip the calibration.tar.gz `tar -xvf calibration.tar.gz`

11-In the folder images used for calibration are available and also “**ost.yaml**” and “**ost.txt**”. You can use the yaml file which contains the calibration parameters as directed by the camera driver.

4.5.6 Get Backtrace in ROS 2 / Nav2

- *Overview*
- *Preliminaries*
- *From a Node*
- *From a Launch File*
- *From Nav2 Bringup*
- *Automatic backtrace on crash*

Overview

This document explains one set of methods for getting backtraces for ROS 2 and Nav2. There are many ways to accomplish this, but this is a good starting point for new C++ developers without GDB experience.

The following steps show ROS 2 users how to modify the Nav2 stack to get traces from specific servers when they encounter a problem. This tutorial applies to both simulated and physical robots.

This will cover how to get a backtrace from a specific node using `ros2 run`, from a launch file representing a single node using `ros2 launch`, and from a more complex orchestration of nodes. By the end of this tutorial, you should be able to get a backtrace when you notice a server crashing in ROS 2.

Preliminaries

GDB is the most popular debugger for C++ on Unix systems. It can be used to determine the reason for a crash and track threads. It may also be used to add breakpoints in your code to check values in memory a particular points in your software.

Using GDB is a critical skill for all software developers working on C/C++. Many IDEs will have some kind of debugger or profiler built in, but with ROS, there are few IDEs to choose. Therefore it’s important to understand how to use these raw tools you have available rather than relying on an IDE to provide them. Further, understanding these tools is a fundamental skill of C/C++ development and leaving it up to your IDE can be problematic if you change roles and no longer have access to it or are doing development on the fly through an ssh session to a remote asset.

Using GDB luckily is fairly simple after you have the basics under your belt. The first step is to add `-g` to your compiler flags for the ROS package you want to profile / debug. This flag builds debug symbols that GDB and valgrind can read to tell you specific lines of code in your project are failing and why. If you do not set this flag, you can still get backtraces but it will not provide line numbers for failures. Be sure to remove this flag after debugging, it will slow down performance at run-time.

Adding the following line to your `CMakeLists.txt` for your project should do the trick. If your project already has a `add_compile_options()`, you can simply add `-g` to it. Then simply rebuild your workspace with this package `colcon build --packages-select <package-name>`. It may take a little longer than usual to compile.

```
add_compile_options(-g)
```

Now you're ready to debug your code! If this was a non-ROS project, at this point you might do something like below. Here we're launching a GDB session and telling our program to immediately run. Once your program crashes, it will return a gdb session prompt denoted by `(gdb)`. At this prompt you can access the information you're interested in. However, since this is a ROS project with lots of node configurations and other things going on, this isn't a great option for beginners or those that don't like tons of commandline work and understanding the filesystem.

```
gdb ex run --args /path/to/exe/program
```

Below are sections to describe the 3 major situations you could run into with ROS 2-based systems. Read the section that best describes the problem you're attempting to solve.

From a Node

Just as in our non-ROS example, we need to setup a GDB session before launching our ROS 2 node. While we could set this up through the commandline with some knowledge of the ROS 2 file system, we can instead use the launch `--prefix` option the kind folks at Open Robotics provided for us.

`--prefix` will execute some bits of code before our `ros2` command allowing us to insert some information. If you attempted to do `gdb ex run --args ros2 run <pkg> <node>` as analog to our example in the preliminaries, you'd find that it couldn't find the `ros2` command. If you're even more clever, you'd find that trying to source your workspace would also fail for similar reasons.

Rather than having to revert to finding the install path of the executable and typing it all out, we can instead use `--prefix`. This allows us to use the same `ros2 run` syntax you're used to without having to worry about some of the GDB details.

```
ros2 run --prefix 'gdb -ex run --args' <pkg> <node> --all-other-launch arguments
```

Just as before, this prefix will launch a GDB session and run the node you requested with all the additional commandline arguments. You should now have your node running and should be chugging along with some debug printing.

Once your server crashes, you'll see a prompt like below. At this point you can now get a backtrace.

```
(gdb)
```

In this session, type `backtrace` and it will provide you with a backtrace. Copy this for your needs. For example:

```
(gdb) backtrace
#0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
#1  0x00007ffff79cc859 in __GI_abort () at abort.c:79
#2  0x00007ffff7c52951 in ?? () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#3  0x00007ffff7c5e47c in ?? () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#4  0x00007ffff7c5e4e7 in std::terminate() () from /usr/lib/x86_64-linux-gnu/
↪libstdc++.so.6
#5  0x00007ffff7c5e799 in __cxa_throw () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#6  0x00007ffff7c553eb in ?? () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#7  0x000055555555936c in std::vector<int, std::allocator<int> >::_M_range_check (
    this=0x5555555cfdb0, __n=100) at /usr/include/c++/9/bits/stl_vector.h:1070
#8  0x0000555555558e1d in std::vector<int, std::allocator<int> >::at␣
↪(this=0x5555555cfdb0,
    __n=100) at /usr/include/c++/9/bits/stl_vector.h:1091
#9  0x000055555555828b in GDBTester::VectorCrash (this=0x5555555cfb40)
    at /home/steve/Documents/nav2_ws/src/gdb_test_pkg/src/gdb_test_node.cpp:44
```

(continues on next page)

(continued from previous page)

```
#10 0x0000555555559cfc in main (argc=1, argv=0x7fffffffc108)
    at /home/steve/Documents/nav2_ws/src/gdb_test_pkg/src/main.cpp:25
```

In this example you should read this in the following way, starting at the bottom:

- In the main function, on line 25 we call a function VectorCrash.
- In VectorCrash, on line 44, we crashed in the Vector's `at()` method with input `100`.
- It crashed in `at()` on STL vector line 1091 after throwing an exception from a range check failure.

These traces take some time to get used to reading, but in general, start at the bottom and follow it up the stack until you see the line it crashed on. Then you can deduce why it crashed. When you are done with GDB, type `quit` and it will exit the session and kill any processes still up. It may ask you if you want to kill some threads at the end, say yes.

From a Launch File

Just as in our non-ROS example, we need to setup a GDB session before launching our ROS 2 launch file. While we could set this up through the commandline, we can instead make use of the same mechanics that we did in the `ros2 run` node example, now using a launch file.

In your launch file, find the node that you're interested in debugging. For this section, we assume that your launch file contains only a single node (and potentially other information as well). The `Node` function used in the `launch_ros` package will take in a field `prefix` taking a list of prefix arguments. We will insert the GDB snippet here with one change from our node example, use of `xterm`. `xterm` will pop up a new terminal window to show and interact with GDB. We do this because of issues handling `stdin` on launch files (e.g. if you hit control+C, are you talking to GDB or launch?). See this ticket for more information. See below for an example debugging SLAM Toolbox.

```
start_sync_slam_toolbox_node = Node(
    parameters=[
      get_package_share_directory("slam_toolbox") + '/config/mapper_params_online_
↪sync.yaml',
      {'use_sim_time': use_sim_time}
    ],
    package='slam_toolbox',
    executable='sync_slam_toolbox_node',
    name='slam_toolbox',
    prefix=['xterm -e gdb -ex run --args'],
    output='screen')
```

Just as before, this prefix will launch a GDB session, now in `xterm` and run the launch file you requested with all the additional launch arguments defined.

Once your server crashes, you'll see a prompt like below, now in the `xterm` session. At this point you can now get a backtrace.

```
(gdb)
```

In this session, type `backtrace` and it will provide you with a backtrace. Copy this for your needs. See the example trace in the section above for an example.

These traces take some time to get used to reading, but in general, start at the bottom and follow it up the stack until you see the line it crashed on. Then you can deduce why it crashed. When you are done with GDB, type `quit` and it will exit the session and kill any processes still up. It may ask you if you want to kill some threads at the end, say yes.

From Large Project

Working with launch files with multiple nodes is a little different so you can interact with your GDB session without being bogged down by other logging in the same terminal. For this reason, when working with larger launch files, its good to pull out the specific server you're interested in and launching it seperately. These instructions are targeting Nav2, but are applicable to any large project with many nodes of any type in a series of launch file(s).

As such, for this case, when you see a crash you'd like to investigate, its beneficial to separate this server from the others.

If your server of interest is being launched from a nested launch file (e.g. an included launch file) you may want to do the following:

- Comment out the launch file inclusion from the parent launch file
- Recompile the package of interest with `-g` flag for debug symbols
- Launch the parent launch file in a terminal
- Launch the server's launch file in another terminal following the instructions in *From a Launch File*.

Alternatively, if you server of interest is being launched in these files directly (e.g. you see a `Node`, `LifecycleNode`, or inside a `ComponentContainer`), you will need to seperate this from the others:

- Comment out the node's inclusion from the parent launch file
- Recompile the package of interest with `-g` flag for debug symbols
- Launch the parent launch file in a terminal
- Launch the server's node in another terminal following the instructions in *From a Node*.

Note: Note that in this case, you may need to remap or provide parameter files to this node if it was previously provided by the launch file. Using `--ros-args` you can give it the path to the new parameters file, remaps, or names. See this ROS 2 tutorial for the commandline arguments required.

We understand this can be a pain, so it might encourage you to rather have each node possible as a separately included launch file to make debugging easier. An example set of arguments might be `--ros-args -r __node:=<node_name> --params-file /absolute/path/to/params.yaml` (as a template).

Once your server crashes, you'll see a prompt like below in the specific server's terminal. At this point you can now get a backtrace.

```
(gdb)
```

In this session, type `backtrace` and it will provide you with a backtrace. Copy this for your needs. See the example trace in the section above for an example.

These traces take some time to get used to reading, but in general, start at the bottom and follow it up the stack until you see the line it crashed on. Then you can deduce why it crashed. When you are done with GDB, type `quit` and it will exit the session and kill any processes still up. It may ask you if you want to kill some threads at the end, say yes.

From Nav2 Bringup

To debug directly from the nav2 bringup launch files you may want to do the following:

- Add `prefix=['xterm -e gdb -ex run --args']` to the non-composed node in the appropriate launch file.
- Recompile the package of interest with `-g` flag for debug symbols.
- Launch normally with `ros2 launch nav2_bringup tb3_simulation_launch.py use_composition:=False`. A seperate xterm window will open with the proccess of intrest running in gdb.

Note: Turning off composition has serious performance impacts. If this is important to you please follow "From Large Project".

Once your server crashes, you'll see a prompt like below in the xterm window. At this point you can now get a backtrace.

```
(gdb)
```

In this session, type `backtrace` and it will provide you with a backtrace. Copy this for your needs. See the example trace in the section above for an example.

These traces take some time to get used to reading, but in general, start at the bottom and follow it up the stack until you see the line it crashed on. Then you can deduce why it crashed. When you are done with GDB, type `quit` and it will exit the session and kill any processes still up. It may ask you if you want to kill some threads at the end, say yes.

Automatic backtrace on crash

The backward-cpp library provides beautiful stack traces, and the backward_ros wrapper simplifies its integration.

Just add it as a dependency and *find_package* it in your CMakeLists and the backward libraries will be injected in all your executables and libraries.

4.5.7 Profiling in ROS 2 / Nav2

Overview

This document explains one method for profiling applications in ROS 2 / Nav2. The aim of profiling is to generate files that can be analyzed to see where compute time and resources are spent during the execution of a program. This can be useful to determine where the bottlenecks in your program exist and where things might be able to be improved.

The following steps show ROS 2 users how to modify the Nav2 stack to get profiling information about a particular server / algorithm when they encounter a situation they'd like to understand better. This tutorial applies to both simulated and physical robots.

Preliminaries

This tutorial makes use of two tools, callgrind from the `Valgrind` set of tools and `kcachegrind`. Valgrind is used to get the profiling information about the program and kcachegrind is the visualization engine used to interprete this information to do useful work.

Thus, we must install them.

```
sudo apt install valgrind kcachegrind
```

More information can be found in the Valgrind manual including additional valgrind arguments that can be used to specify more information.

Generally speaking, to use valgrind we need to compile with debugging information. This can be done by passing `-g` as a compiling option or compile `CMAKE_BUILD_TYPE` as `Debug` or `RelWithDebInfo`. Then, we run our program using valgrind to capture the run-time statistics for later analysis. These are stored in `callgrind.out.XXX` files, where the suffix is the PID of the process. kcachegrind is used to visualize and analyze the results of the program execution.

```
      # CMakeLists.txt
add_compile_options(-g)
```

```
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo
```

```
valgrind --tool=callgrind [your-program] [program options]

kcachegrind callgrind.out.XXX
```

Profile from a Node

As in our generic example, for a given node, we need to compile with debug flags to capture the information for profiling with Valgrind. This can be done easily from the commandline. Note that we use `--packages-select` to only compile with this flag for the packages we want to profile nodes within.

```
colcon build --packages-select <packages of interest> --cmake-args -DCMAKE_BUILD_
↪TYPE=RelWithDebInfo
```

Optionally, you may add the following line to the `CMakeLists.txt` of the package you're looking to profile. This may be preferable when you have a workspace with many packages but would like to only compile a subset with debug information using a single `colcon build` invokation.

It is important that this should be added to both the host server and plugin packages(s) if you would like the results of a plugin's run-time profile.

```
add_compile_options(-pg)
```

After either compiling method, this node should be run in its own terminal to isolate it from the rest of your system. Thus, this should not be composed in the same process as other nodes. To run a ROS 2 node with valgrind, this can be done from the commandline via:

```
ros2 run --prefix 'valgrind --tool=callgrind' <pkg> <node> --all-other-launch␣
↪arguments
```

An example of this might be used for profiling the controller server with a particular controller plugin loaded. Both `nav2_controller` and the plugin package of interest are compiled with debug flags. In the example below, we are running a ROS 2 node with remapped topics and a path to its parameter file:

```
ros2 run --prefix 'valgrind --tool=callgrind' nav2_controller controller_server --ros-
↪args -r __node:=controller_server -r cmd_vel:=cmd_vel_nav --params-file /path/to/
↪nav2_bringup/params/nav2_params.yaml
```

Once sufficient data has been collected, cleanly exit the process with Control+C.

Profile from a Launch File

Just as in the Node example, we must also compile with debug flags when profiling a node from launch. We can complete the same valgrind call as from the commandline as within a launch file using launch prefixes.

As our example before, this is how we'd launch the `controller_server` node from inside a launch file.

```
start_controller_server_node = Node(
    parameters=[
      get_package_share_directory("nav2_bringup") + '/params/nav2_params.yaml',
      {'use_sim_time': use_sim_time}
    ],
    package='nav2_controller',
    executable='controller_server',
    name='controller_server',
    prefix=['xterm -e valgrind --tool=callgrind'],
    output='screen')
```

Note that just like before, we should isolate this process from others. So this should not be run with any other nodes in this launch file nor use node composition when profiling a particular node.

Once sufficient data has been collected, cleanly exit the process with Control+C.

From Nav2 Bringup

Because Nav2 bringup has more than one node per launch file (and in the case `use_composition=true`, more than one per process), it is necessary to separate out a particular node that you're interested in profiling from the rest of the system. As previously described, once they're isolated in either a launch file or as a node to be launched on the commandline, they can easily be run to collect the callgrind information.

The steps within Nav2 are as follows:

- Remove server node from the `navigation_launch.py`, ensuring to remove from both composed and non-composed options within the file
- In a separate launch file or using `ros2 run` CLI, start up the node you'd like to profile using the instructions above

- Launch Nav2 as usual with the missing node
- Once your data has been collected, control+C and cleanly finish the profiled process and the rest of the navigation

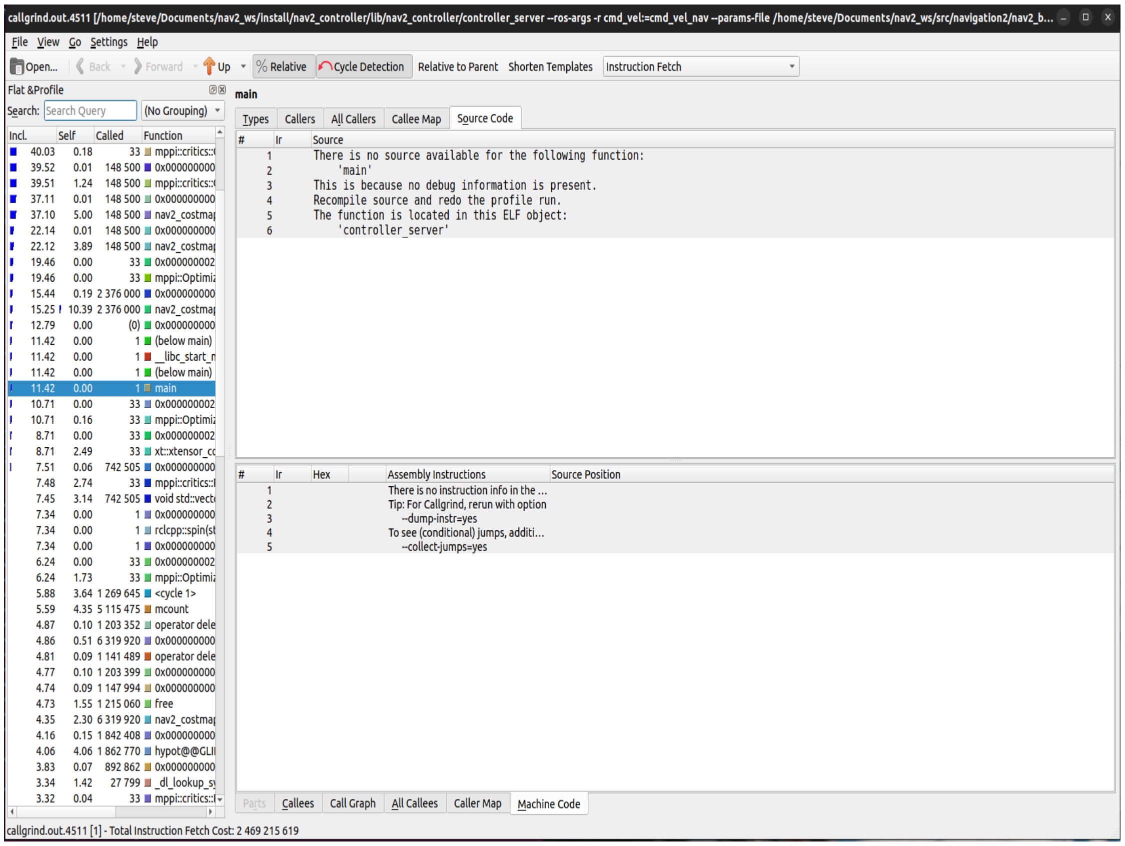
It is important that the profiler node is launched before Nav2 so that it can take the signals from the lifecycle manager to transition up.

Interpreting Results

Once you have your `callgrind` results, regardless of if you did it through a node, launch file, Nav2, or elsewhere, now we can analyze the results from the profiler to identify bottlenecks or potential areas of improvement. Using `kcachegrind`:

```
kcachegrind callgrind.out.XXX
```

This should open a window looking like below. The left side shows all of the calls and their relative percentages of compute time they and their children functions utilized.


callgrind.out.4511 [/home/steve/Documents/nav2_ws/install/nav2_controller/lib/nav2_controller/controller_server --ros-args -r cmd_vel:=cmd_vel_nav --params-file /home/steve/Documents/nav2_ws/src/navigation2/nav2_b...
File View Go Settings Help
Open... Back Forward Up % Relative Cycle Detection Relative to Parent Shorten Templates Instruction Fetch
Flat &Profile
Search: Search Query (No Grouping)
main
Types Callers All Callers Callee Map Source Code
Parts Callees Call Graph All Callees Caller Map Machine Code

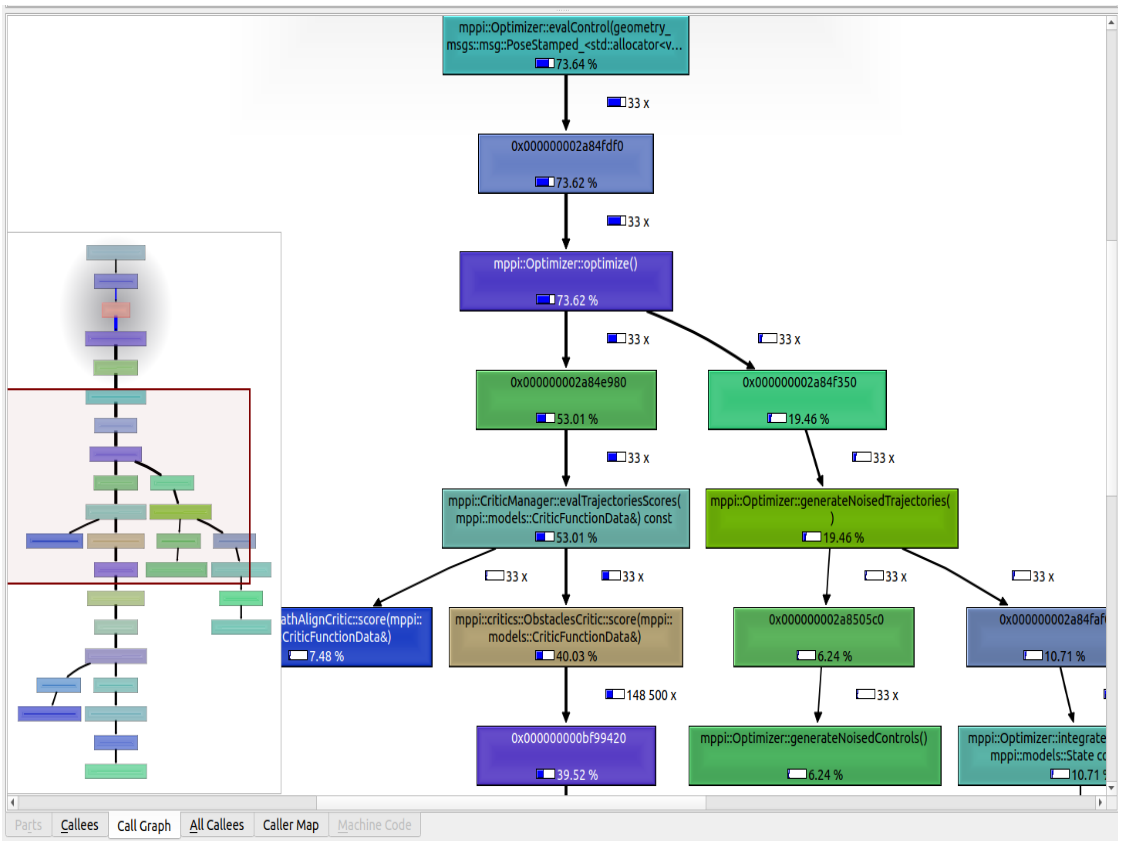
Incl.	Self	Called	Function
40.03	0.18	33	mppi::critics::
39.52	0.01	148 500	0x000000000
39.51	1.24	148 500	mppi::critics::
37.11	0.01	148 500	0x000000000
37.10	5.00	148 500	nav2_costma
22.14	0.01	148 500	0x000000000
22.12	3.89	148 500	nav2_costma
19.46	0.00	33	0x000000002
19.46	0.00	33	mppi::Optimi
15.44	0.19	2 376 000	0x000000000
15.25	10.39	2 376 000	nav2_costma
12.79	0.00	(0)	0x000000000
11.42	0.00	1	(below main)
11.42	0.00	1	__libc_start_
11.42	0.00	1	(below main)
11.42	0.00	1	main
10.71	0.00	33	0x000000002
10.71	0.16	33	mppi::Optimi
8.71	0.00	33	0x000000002
8.71	2.49	33	xt::xtensor_c
7.51	0.06	742 505	0x000000000
7.48	2.74	33	mppi::critics::
7.45	3.14	742 505	void std::vect
7.34	0.00	1	0x000000000
7.34	0.00	1	rclcpp::spin(st
7.34	0.00	1	0x000000000
6.24	0.00	33	0x000000002
6.24	1.73	33	mppi::Optimi
5.88	3.64	1 269 645	<cycle 1>
5.59	4.35	5 115 475	mcount
4.87	0.10	1 203 352	operator dele
4.86	0.51	6 319 920	0x000000000
4.81	0.09	1 141 489	operator dele
4.77	0.10	1 203 399	0x000000000
4.74	0.09	1 147 994	0x000000000
4.73	1.55	1 215 060	free
4.35	2.30	6 319 920	nav2_costma
4.16	0.15	1 842 408	0x000000000
4.06	4.06	1 862 770	hypot@@GLI
3.83	0.07	892 862	0x000000000
3.34	1.42	27 799	_dl_lookup_s
3.32	0.04	33	mppi::critics::

```
#   Ir   Source
1        There is no source available for the following function:
             'main'
3        This is because no debug information is present.
4        Recompile source and redo the profile run.
5        The function is located in this ELF object:
             'controller_server'
```

```
#   Ir   Hex   Assembly Instructions                Source Position
1                There is no instruction info in the ...
2                Tip: For Callgrind, rerun with option
3                   --dump-instr=yes
4                To see (conditional) jumps, additi...
5                   --collect-jumps=yes
```

callgrind.out.4511 [1] - Total Instruction Fetch Cost: 2 469 215 619

If you select the top level entry on the left sidebar, then select "Call Graph" at the bottom of the right workspace, it should show you a call graph of where the compute time was spent as a graph of method calls. This can be exceptionally helpful to find the methods where the most time is spent.


mppi::Optimizer::evalControl(geometry_msgs::msg::PoseStamped_<std::allocator<v...
73.64 %
33 x
0x000000002a84fdf0
73.62 %
33 x
mppi::Optimizer::optimize()
73.62 %
33 x
33 x
0x000000002a84e980
53.01 %
0x000000002a84f350
19.46 %
33 x
33 x
mppi::CriticManager::evalTrajectoriesScores(mppi::models::CriticFunctionData&) const
53.01 %
mppi::Optimizer::generateNoisedTrajectories( )
19.46 %
33 x
33 x
33 x
33 x
athAlignCritic::score(mppi::CriticFunctionData&)
7.48 %
mppi::critics::ObstaclesCritic::score(mppi::models::CriticFunctionData&)
40.03 %
0x000000002a8505c0
6.24 %
0x000000002a84faf
10.71 %
148 500 x
33 x
0x000000000bf99420
39.52 %
mppi::Optimizer::generateNoisedControls()
6.24 %
mppi::Optimizer::integrate
mppi::models::State c
10.71
Parts
Callees
Call Graph
All Callees
Caller Map
Machine Code

4.5.8 Dynamic Object Following

- *Overview*
- *Tutorial Steps*

Overview

This tutorial shows how to use Nav2 for a different task other than going from point A to point B. In this case, we will use Nav2 to follow a moving object at a distance indefinitely.

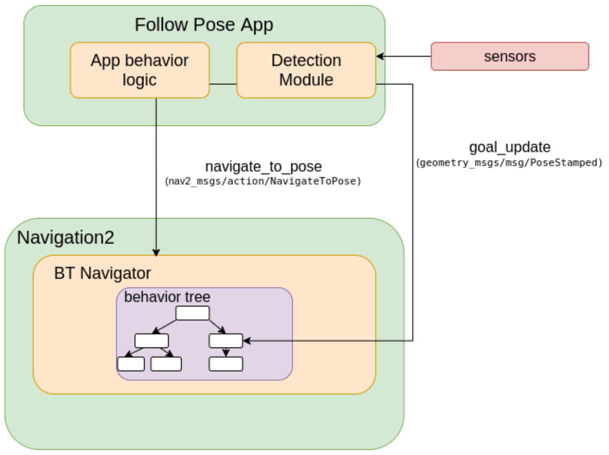
This task is useful in cases such as following a person or another robot. Below are some sample videos of applications that could be created using this capability. The "Carry My Luggage" RoboCup @ Home test, in which the CATIE Robotics team performs the test successfully and this real (future) world application:

The requirements for this task are as follows:

- Changes are limited to the behavior tree used to navigate. This behavior tree can be selected in the `NavigateToPose` action when required, or it can be the default behavior tree. It is made up of run-time configurable plugins.
- The configuration of the planner and the controller will not be modified.
- The action will indefinitely run until it is canceled by who initiated it.

The detection of the dynamic object (like a person) to follow is outside the scope of this tutorial. As shown in the following diagram, your application should provide a detector for the object(s) of interest, send the initial pose to the

NavigateToPose action, and update it on a topic for the duration of the task. Many different types of detectors exist that you can leverage for this application:



Tutorial Steps

0- Create the Behavior Tree

Let's start from this simple behavior tree. This behavior tree replans a new path at 1 hz and passes that path to the controller to follow:

```
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <PipelineSequence name="NavigateWithReplanning">
      <RateController hz="1.0">
        <ComputePathToPose goal="{goal}" path="{path}" planner_id="GridBased"/>
      </RateController>
      <FollowPath path="{path}" controller_id="FollowPath"/>
    </PipelineSequence>
  </BehaviorTree>
</root>
```

First, let's make this behavior run until there's a failure. For this purpose, we will use the KeepRunningUntilFailure control node.

```
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <PipelineSequence name="NavigateWithReplanning">
      <RateController hz="1.0">
        <ComputePathToPose goal="{goal}" path="{path}" planner_id="GridBased"/>
      </RateController>
      <KeepRunningUntilFailure>
        <FollowPath path="{path}" controller_id="FollowPath"/>
      </KeepRunningUntilFailure>
    </PipelineSequence>
  </BehaviorTree>
</root>
```

We will then use the decorator GoalUpdater to accept updates of the dynamic object pose we're trying to follow. This node takes as input the current goal and subscribes to the topic /goal_update. It sets the new goal as updated_goal if a new goal on that topic is received.

```
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <PipelineSequence name="NavigateWithReplanning">
      <RateController hz="1.0">
        <GoalUpdater input_goal="{goal}" output_goal="{updated_goal}">
          <ComputePathToPose goal="{updated_goal}" path="{path}" planner_id="GridBased
↪"/>
        </GoalUpdater>
      </RateController>
      <KeepRunningUntilFailure>
        <FollowPath path="{path}" controller_id="FollowPath"/>
      </KeepRunningUntilFailure>
    </PipelineSequence>
  </BehaviorTree>
</root>
```

To stay at a certain distance from the target, we will use the action node `TruncatePath`. This node modifies a path making it shorter so we don't try to navigate into the object of interest. We can set up the desired distance to the goal using the input port `distance`.

```
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <PipelineSequence name="NavigateWithReplanning">
      <RateController hz="1.0">
        <Sequence>
          <GoalUpdater input_goal="{goal}" output_goal="{updated_goal}">
            <ComputePathToPose goal="{updated_goal}" path="{path}" planner_id=
↪"GridBased"/>
          </GoalUpdater>
         <TruncatePath distance="1.0" input_path="{path}" output_path="{truncated_
↪path}"/>
        </Sequence>
      </RateController>
      <KeepRunningUntilFailure>
        <FollowPath path="{truncated_path}" controller_id="FollowPath"/>
      </KeepRunningUntilFailure>
    </PipelineSequence>
  </BehaviorTree>
</root>
```

Now, you may save this behavior tree and use it in our navigation task.

For reference, this exact behavior tree is made available to you batteries included in the `nav2_bt_navigator` package.

1- Setup Rviz clicked point

We are going to use RViz instead of a full application so you can test at home without finding a detector to get started. We will use the "clicked point" button on the toolbar to substitute object detections to provide goal updates to Nav2. This button allows you to publish coordinates in the topic `/clicked_point`. This point needs to be sent to the behavior tree, using the program `clicked_point_to_pose`, from this repo. Clone this repo in your workspace, build, and type in a terminal.

```
ros2 run nav2_test_utils clicked_point_to_pose
```

Optionally, you can remap this topic in your rviz configuration file to `goal_updates`.

2- Run Dynamic Object Following in Nav2 Simulation

Start Nav2 in one terminal:

```
ros2 launch nav2_bringup tb3_simulation_launch.py default_bt_xml_filename:=/
path/to/bt.xml
```

Open RViz and, after initialize the robot position, command the robot to navigate to any position. Use the button clicked point to simulate a new detection of the object of interest, as shown in the video in the head of this tutorial.

When you have a detector detecting your obstacle at a higher rate (1 hz, 10 hz, 100 hz) you will see a far more reactive robot following your detected object of interest!

4.5.9 Navigating with Keepout Zones

- *Overview*
- *Requirements*
- *Tutorial Steps*

Overview

This tutorial shows how to simply utilize keep-out/safety zones where robots can't enter and preferred lanes for robots moving in industrial environments and warehouses. All this functionality is being covered by `KeepoutFilter` costmap filter plugin which will be enabled and used in this document.

Requirements

It is assumed that ROS 2, Gazebo and TurtleBot3 packages are installed or built locally. Please make sure that Nav2 project is also built locally as it was made in *Build and Install*.

Tutorial Steps

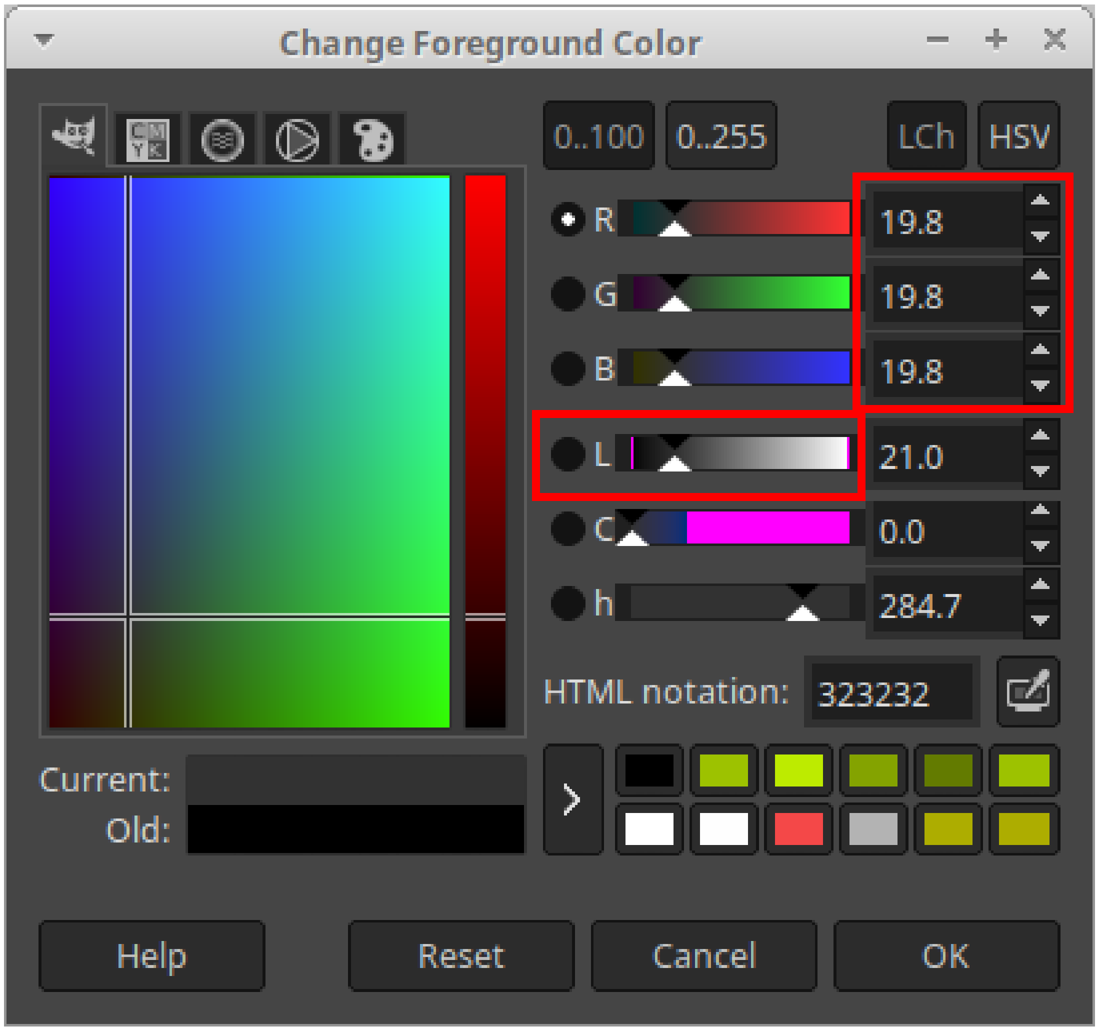
1. Prepare filter mask

As was written in *Navigation Concepts*, any Costmap Filter (including Keepout Filter) are reading the data marked in a filter mask file. Filter mask - is the usual Nav2 2D-map distributed through PGM, PNG or BMP raster file with its metadata containing in a YAML file. The following steps help to understand how to make a new filter mask:

Create a new image with a PGM/PNG/BMP format: copy turtlebot3_world.pgm main map which will be used in a world simulation from a `Nav2` repository to a new `keepout_mask.pgm` file.

Open `keepout_mask.pgm` in your favourite raster graphics editor (as an example could be taken GIMP editor). The lightness of each pixel on the mask means an encoded information for the specific costmap filter you are going to use. Color lightness of each pixel belongs to the `[0..255]` range (or `[0..100]` in percent scale), where `0` means black color and `255` - white. Another term "darkness" will be understood as the exact opposite of lightness. In other words `color_darkness = 100% - color_lightness`.

In the GIMP lightness is expressed through color components value (e.g. `R` in percent scale) and might be set by moving `L` slider in color changing tool:



R 19.8
G 19.8
B 19.8
L 21.0
C 0.0
h 284.7
HTML notation: 323232

The incoming mask file is being read by the Map Server and converted into `OccupancyGrid` values from `[0..100]` range (where `0` means free cell, `100` - occupied, anything in between - less or more occupied cells on map) or be equal to `-1` for unknown value. In Nav2 stack each map has `mode` attribute which could be `trinary`, `scale` or `raw`. Depending on `mode` selected, the color lightness of PGM/PNG/BMP is being converted to `OccupancyGrid` by one of the following principles:

- `trinary` (default mode): Darkness >= `occupied_thresh` means that map occupied (`100`). Darkness <= `free_thresh` - map free (`0`). Anything in between - unknown status on map (`-1`).
- `scale`: Alpha < `1.0` - unknown. Darkness >= `occupied_thresh` means that map occupied (`100`). Darkness <= `free_thresh` - map free (`0`). Anything in between - linearly interpolate to nearest integer from `[0..100]` range.
- `raw`: Lightness = `0` (dark color) means that map is free (`0`). Lightness = `100` (in absolute value) - map is occupied (`100`). Anything in between - `OccupancyGrid` value = lightness. Lightness >= `101` - unknown (`-1`).

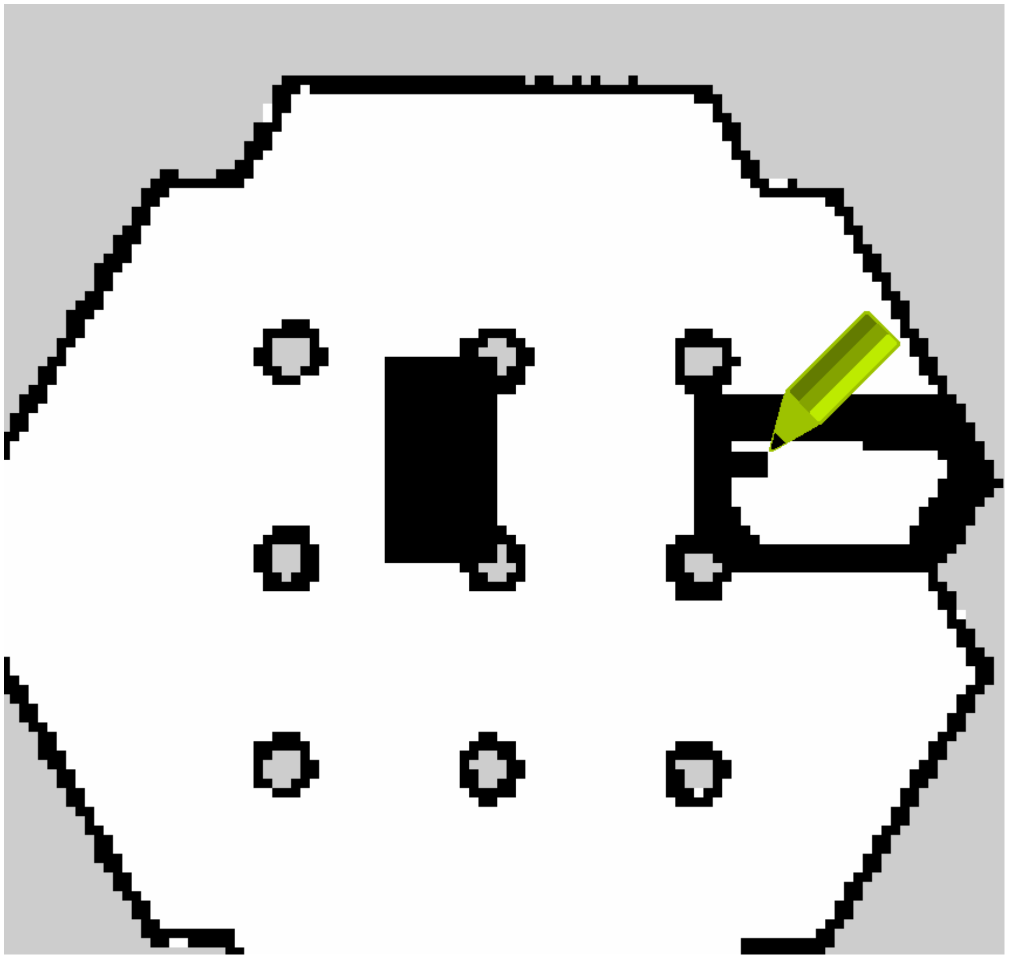
where `free_thresh` and `occupied_thresh` thresholds are expressed in percentage of maximum lightness/darkness level (`255`). Map mode and thresholds are placed in YAML metadata file (see below) as `mode`, `free_thresh` and `occupied_thresh` fields.

Note: There is another parameter in a YAML metadata file called `negate`. By default it is set to `false`. When it is set to `true`, blacker pixels will be considered as free, whiter pixels - as occupied. In this case we should count color lightness instead of darkness for `trinary` and `scale` modes. `negate` has no effect on `raw` mode.

For Keepout Filter `OccupancyGrid` value is proportional to the passibility of area corresponting to this cell: higher values means more impassable areas. Cells with occupied values covers keep-out zones where robot will never enter or pass through. `KeepoutFilter` can also act as a "weighted areas layer" by setting the `OccupancyGrid` to something between `[1-99]` non-occupied values. Robot is allowed to move in these areas, however its presence there would be "undesirable" there (the higher the value, the sooner planners will try to get the robot out of this area).

Keepout Filter also covers preferred lanes case, where robots should moving only on pre-defined lanes and permitted areas e.g. in warehouses. To use this feaure you need to prepare the mask image where the lanes and permitted areas will be marked with free values while all other areas will be occupied. TIP for drawing the mask in a `trinary` or `scale` mode: typically, amount of pixels belonging to lanes are much less than pixels covering other areas. In this case initially all lanes data might be drawn with a black pencil over white background and then (just before saving a PGM) "color inversion" tool in a image raster editor might be used.

For simplicity, in the example fill the areas with black color (in `trinary` mode this means occupied map) that you are going to mark as a keep-out zones:



After all keepout areas will be filled save the `keepout_mask.pgm` image.

Like all other maps, filter mask should have its own YAML metadata file. Copy turtlebot3_world.yaml to `keepout_mask.yaml`. Open `keepout_mask.yaml` and correct `image` field to a newly made PGM mask:

```
image: turtlebot3_world.pgm
```

(continues on next page)

(continued from previous page)

```
->
image: keepout_mask.pgm
```

Since filter mask image was created as a copy of main map, other fields of YAML-file do not need to be changed. Save `keepout_mask.yaml` and new filter mask is ready to use.

Note: World map itself and filter mask could have different sizes, origin and resolution which might be useful e.g. for cases when filter mask is covering smaller areas on maps or when one filter mask is used repeatedly many times (like annotating a keepout zone for same shape rooms in the hotel). For this case, you need to correct `resolution` and `origin` fields in YAML as well so that the filter mask is correctly laid on top of the original map.

Note: Another important note is that since Costmap2D does not support orientation, the last third "yaw" component of the `origin` vector should be equal to zero. For example: `origin:  [1.25, -5.18, 0.0]`.

2. Configure Costmap Filter Info Publisher Server

Each costmap filter reads incoming meta-information (such as filter type or data conversion coefficients) in a messages of `nav2_msgs/CostmapFilterInfo` type. These messages are being published by Costmap Filter Info Publisher Server. The server is running as a lifecycle node. According to the design document, `nav2_msgs/CostmapFilterInfo` messages are going in a pair with `OccupancyGrid` filter mask topic. Therefore, along with Costmap Filter Info Publisher Server there should be enabled a new instance of Map Server configured to publish filter mask.

In order to enable Keepout Filter in your configuration, both servers should be enabled as a lifecycle nodes in Python launch-file. For example, this might look as follows:

```python
import os

from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node
from nav2_common.launch import RewrittenYaml


def generate_launch_description():
    # Get the launch directory
    costmap_filters_demo_dir = get_package_share_directory('nav2_costmap_filters_demo
↪')

    # Create our own temporary YAML files that include substitutions
    lifecycle_nodes = ['filter_mask_server', 'costmap_filter_info_server']

    # Parameters
    namespace = LaunchConfiguration('namespace')
    use_sim_time = LaunchConfiguration('use_sim_time')
    autostart = LaunchConfiguration('autostart')
    params_file = LaunchConfiguration('params_file')
    mask_yaml_file = LaunchConfiguration('mask')
```

(continues on next page)

(continued from previous page)

```
    # Declare the launch arguments
    declare_namespace_cmd = DeclareLaunchArgument(
        'namespace',
        default_value='',
        description='Top-level namespace')

    declare_use_sim_time_cmd = DeclareLaunchArgument(
        'use_sim_time',
        default_value='true',
        description='Use simulation (Gazebo) clock if true')

    declare_autostart_cmd = DeclareLaunchArgument(
        'autostart', default_value='true',
        description='Automatically startup the nav2 stack')

    declare_params_file_cmd = DeclareLaunchArgument(
            'params_file',
            default_value=os.path.join(costmap_filters_demo_dir, 'params', 'keepout_
↪params.yaml'),
            description='Full path to the ROS 2 parameters file to use')

    declare_mask_yaml_file_cmd = DeclareLaunchArgument(
            'mask',
            default_value=os.path.join(costmap_filters_demo_dir, 'maps', 'keepout_
↪mask.yaml'),
            description='Full path to filter mask yaml file to load')

    # Make re-written yaml
    param_substitutions = {
        'use_sim_time': use_sim_time,
        'yaml_filename': mask_yaml_file}

    configured_params = RewrittenYaml(
        source_file=params_file,
        root_key=namespace,
        param_rewrites=param_substitutions,
        convert_types=True)

    # Nodes launching commands
    start_lifecycle_manager_cmd = Node(
            package='nav2_lifecycle_manager',
            executable='lifecycle_manager',
            name='lifecycle_manager_costmap_filters',
            namespace=namespace,
            output='screen',
            emulate_tty=True,  # https://github.com/ros2/launch/issues/188
            parameters=[{'use_sim_time': use_sim_time},
                        {'autostart': autostart},
                        {'node_names': lifecycle_nodes}])

    start_map_server_cmd = Node(
            package='nav2_map_server',
            executable='map_server',
            name='filter_mask_server',
            namespace=namespace,
            output='screen',
```

(continues on next page)

(continued from previous page)

```
            emulate_tty=True,  # https://github.com/ros2/launch/issues/188
            parameters=[configured_params])

    start_costmap_filter_info_server_cmd = Node(
            package='nav2_map_server',
            executable='costmap_filter_info_server',
            name='costmap_filter_info_server',
            namespace=namespace,
            output='screen',
            emulate_tty=True,  # https://github.com/ros2/launch/issues/188
            parameters=[configured_params])

    ld = LaunchDescription()

    ld.add_action(declare_namespace_cmd)
    ld.add_action(declare_use_sim_time_cmd)
    ld.add_action(declare_autostart_cmd)
    ld.add_action(declare_params_file_cmd)
    ld.add_action(declare_mask_yaml_file_cmd)

    ld.add_action(start_lifecycle_manager_cmd)
    ld.add_action(start_map_server_cmd)
    ld.add_action(start_costmap_filter_info_server_cmd)

    return ld
```

where the `params_file` variable should be set to a YAML-file having ROS parameters for Costmap Filter Info Publisher Server and Map Server nodes. These parameters and their meaning are listed at *Map Server / Saver* page. Please, refer to it for more information. The example of `params_file` could be found below:

```
costmap_filter_info_server:
  ros__parameters:
    use_sim_time: true
    type: 0
    filter_info_topic: "/costmap_filter_info"
    mask_topic: "/keepout_filter_mask"
    base: 0.0
    multiplier: 1.0
filter_mask_server:
  ros__parameters:
    use_sim_time: true
    frame_id: "map"
    topic_name: "/keepout_filter_mask"
    yaml_filename: "keepout_mask.yaml"
```

Note, that:

- For Keepout Filter the `type` of costmap filter should be set to `0`.
- Filter mask topic name should be the equal for `mask_topic` parameter of Costmap Filter Info Publisher Server and `topic_name` parameter of Map Server.
- According to the Costmap Filters design, `OccupancyGrid` values are being linearly transformed into feature map in a filter space. For a Keepout Filter these values are directly passed as a filter space values without a linear conversion. Even though `base` and `multiplier` coefficients are not used in Keepout Filter, they should be set to `0.0` and `1.0` accordingly in order to explicitly show that we have one-to-one conversion from `OccupancyGrid` values -> to a filter value space.

Ready-to-go standalone Python launch-script, YAML-file with ROS parameters and filter mask example for Keepout Filter could be found in a nav2_costmap_filters_demo directory of `navigation2_tutorials` repository. To simply run Filter Info Publisher Server and Map Server tuned on Turtlebot3 standard simulation written at *Getting Started*, build the demo and launch `costmap_filter_info.launch.py` as follows:

```
$ mkdir -p ~/tutorials_ws/src
$ cd ~/tutorials_ws/src
$ git clone https://github.com/ros-planning/navigation2_tutorials.git
$ cd ~/tutorials_ws
$ colcon build --symlink-install --packages-select nav2_costmap_filters_demo
$ source ~/tutorials_ws/install/setup.bash
$ ros2 launch nav2_costmap_filters_demo costmap_filter_info.launch.py params_
↪file:=src/navigation2_tutorials/nav2_costmap_filters_demo/params/keepout_params.
↪yaml mask:=src/navigation2_tutorials/nav2_costmap_filters_demo/maps/keepout_mask.
↪yaml
```

3. Enable Keepout Filter

Costmap Filters are Costamp2D plugins. You can enable the `KeepoutFilter` plugin in Costmap2D by adding `keepout_filter` to the `plugins` parameter in `nav2_params.yaml`. You can place it in the `global_costmap` for planning with keepouts and `local_costmap` to make sure the robot won't attempt to drive through a keepout zone. The KeepoutFilter plugin should have the following parameters defined:

- `plugin`: type of plugin. In our case `nav2_costmap_2d::KeepoutFilter`.
- `filter_info_topic`: filter info topic name. This need to be equal to `filter_info_topic` parameter of Costmap Filter Info Publisher Server from the chapter above.

Full list of parameters supported by `KeepoutFilter` are listed at *Keepout Filter Parameters* page.

It is important to note that enabling `KeepoutFilter` for `global_costmap` only will cause the path planner to build plans bypassing keepout zones. Enabling `KeepoutFilter` for `local_costmap` only will cause the robot to not enter keepout zones, but the path may still go through them. So, the best practice is to enable `KeepoutFilter` for global and local costmaps simultaneously by adding it both in `global_costmap` and `local_costmap` in `nav2_params.yaml`. However it does not always have to be true. In some cases keepout zones don't have to be the same for global and local costmaps, e.g. if the robot doesn't allowed to intentionally go inside keepout zones, but if its there, the robot can drive in and out really quick if it clips an edge or corner. For this case, there is not need to use extra resources of the local costmap copy.

To enable `KeepoutFilter` with same mask for both global and local costmaps, use the following configuration:

```
global_costmap:
  global_costmap:
    ros__parameters:
      ...
      plugins: ["static_layer", "obstacle_layer", "inflation_layer"]
      filters: ["keepout_filter"]
      ...
      keepout_filter:
        plugin: "nav2_costmap_2d::KeepoutFilter"
        enabled: True
        filter_info_topic: "/costmap_filter_info"
...
local_costmap:
  local_costmap:
    ros__parameters:
      ...
```

(continues on next page)

(continued from previous page)

```
      plugins: ["voxel_layer", "inflation_layer"]
      filters: ["keepout_filter"]
      ...
      keepout_filter:
        plugin: "nav2_costmap_2d::KeepoutFilter"
        enabled: True
        filter_info_topic: "/costmap_filter_info"
```

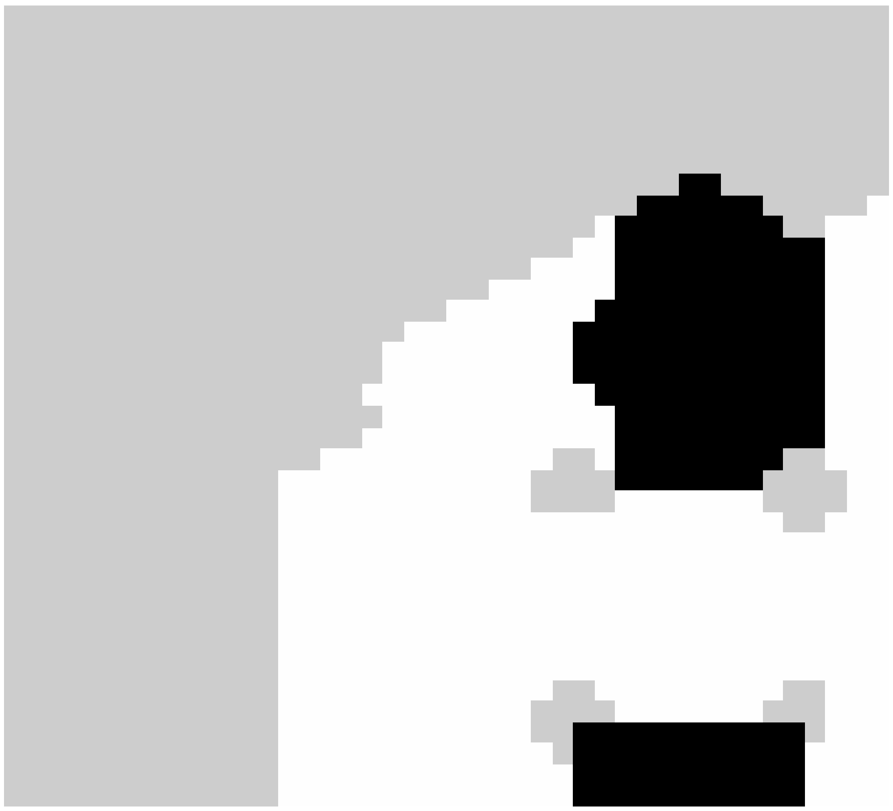
Note: All costmap filters should be enabled through a `filters` parameter – though it is technically possible to include in the layered costmap itself. This is separated from the layer plugins to prevent interference in the layers, particularly the inflation layer.

4. Run Nav2 stack

After Costmap Filter Info Publisher Server and Map Server were launched and Keepout Filter was enabled for global/local costmaps, run Nav2 stack as written in *Getting Started*:

```
ros2 launch nav2_bringup tb3_simulation_launch.py
```

And check that filter is working properly as in the pictures below (first picture shows keepout filter enabled for the global costmap, second - differently-sized `keepout_mask.pgm` filter mask):



4.5.10 Navigating with Speed Limits

- *Overview*
- *Requirements*
- *Tutorial Steps*

Overview

This tutorial shows how to simply utilize Speed Filter which is designed to limit the maximum speed of robots in speed restriction areas marked on a map. This functionality is being covered by `SpeedFilter` costmap filter plugin which will be enabled and used in this document.

Requirements

It is assumed that ROS 2, Gazebo and TurtleBot3 packages are installed or built locally. Please make sure that the Nav2 project is also built locally as it was made in *Build and Install*.

Tutorial Steps

1. Prepare filter mask

As was written in *Navigation Concepts*, any Costmap Filter (including Speed Filter) is reading the data marked in a filter mask file. All information about filter masks, their types, detailed structure and how to make a new one is written in a *Navigating with Keepout Zones* tutorial at `1. Prepare filter masks` chapter. The principal of drawing the filter mask for Speed Filter is the same as for Keepout Filter (to annotate a map with the requested zones), except that `OccupancyGrid` mask values have another meaning: these values are encoded speed limits for the areas corresponding to the cell on map.

Let's look, how it is being decoded. As we know, `OccupancyGrid` values are belonging to the `[0..100]` range. For Speed Filter `0` value means no speed limit in the area corresponding zero-cell on mask. Values from `[1..100]` range are being linearly converted into a speed limit value by the following formula:

```
speed_limit = filter_mask_data * multiplier + base;
```

where:

- `filter_mask_data` - is an `OccupancyGrid` value of the corresponding cell on mask where maximum speed should be restricted.
- `base` and `multiplier` are coefficients taken from `nav2_msgs/CostmapFilterInfo` messages published by Costmap Filter Info Server (see in next chapter below).

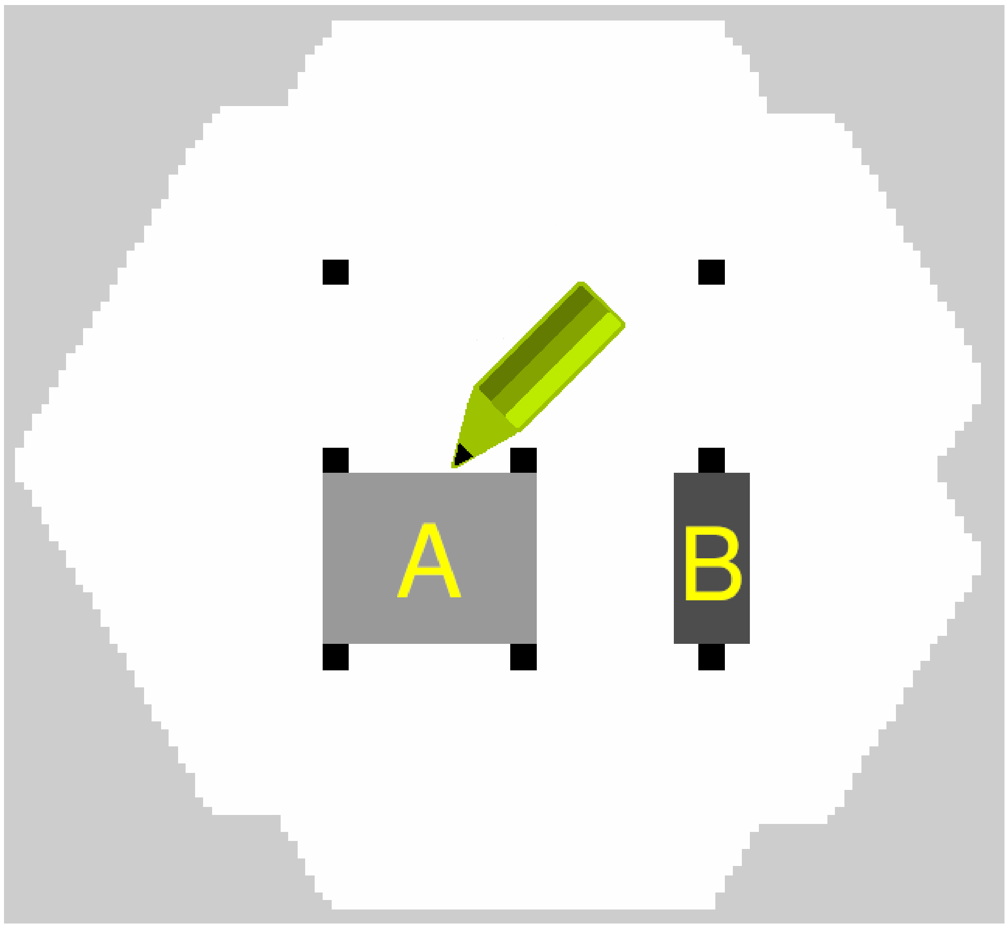
The decoded `speed_limit` value may have one of two meanings:

- Speed limit expressed in a percent from maximum robot speed.
- Speed limit expressed in absolute values (e.g. in `m/s`).

The meaning used by Speed Filter is being read from `nav2_msgs/CostmapFilterInfo` messages. In this tutorial we will use the first type of speed restriction expressed in a percent from maximum robot speed.

Note: For speed restriction expressed in a percent, `speed_limit` will be used exactly as a percent belonging to `[0..100]` range, not `[0.0..1.0]` range.

Create a new image with a PGM/PNG/BMP format: copy `turtlebot3_world.pgm` main map which will be used in a world simulation from a Nav2 repository to a new `speed_mask.pgm` file. Open `speed_mask.pgm` in your favourite raster graphics editor and fill speed restricted areas with grey colors. In our example darker colors will indicate areas with higher speed restriction:



Area "A" is filled with `40%` gray color, area "B" - with `70%` gray, that means that speed restriction will take `100% - 40% = 60%` in area "A" and `100% - 70% = 30%` in area "B" from maximum speed value allowed for this robot. We will use `scale` map mode with no thresholds. In this mode darker colors will have higher `OccupancyGrid` values. E.g. for area "B" with `70%` of gray `OccupancyGrid` data will be equal to `70`. So in order to hit the target, we need to choose `base = 100.0` and `multiplier = -1.0`. This will reverse the scale `OccupancyGrid` values to a desired one. No thresholds (`free_thresh occupied_thresh`) were chosen for the convenience in the `yaml` file: to have 1:1 full range conversion of lightness value from filter mask -> to speed restriction percent.

Note: It is typical but not a mandatory selection of `base` and `multiplier`. For example, you can choose map mode to be `raw`. In this case color lightness is being directly converted into `OccupancyGrid` values. For masks saved in a `raw` mode, `base` and `multiplier` will be equal to `0.0` and `1.0` accordingly.

Another important thing is that it is not necessary to use the whole `[0..100]` percent scale. `base` and `multiplier` coefficients could be chosen so that the speed restriction values would belong to somewhere in the middle of percent range. E.g. `base = 40.0`, `multiplier = 0.1` will give speed restrictions from `[40.0%..50.0%]` range with a step of `0.1%`. This might be useful for fine tuning.

After all speed restriction areas will be filled, save the `speed_mask.pgm` image.

Like all other maps, the filter mask should have its own YAML metadata file. Copy turtlebot3_world.yaml to speed_mask.yaml. Open speed_mask.yaml and update the fields as shown below (as mentioned before for the scale mode to use whole color lightness range there should be no thresholds: free_thresh = 0.0 and occupied_thresh = 1.0):

```
image: turtlebot3_world.pgm
->
image: speed_mask.pgm

mode: trinary
->
mode: scale

occupied_thresh: 0.65
free_thresh: 0.196
->
occupied_thresh: 1.0
free_thresh: 0.0
```

Since Costmap2D does not support orientation, the last third "yaw" component of the origin vector should be equal to zero (for example: origin: [1.25, -5.18, 0.0]). Save speed_mask.yaml and the new filter mask is ready to use.

Note: World map itself and filter mask could have different sizes, origin and resolution which might be useful (e.g. for cases when filter mask is covering smaller areas on maps or when one filter mask is used repeatedly many times, like annotating a speed restricted area for same shape rooms in the hotel). For this case, you need to correct resolution and origin fields in YAML as well so that the filter mask is correctly laid on top of the original map. This example shows using the main map as a base, but that is not required.

2. Configure Costmap Filter Info Publisher Server

Each costmap filter reads incoming meta-information (such as filter type or data conversion coefficients) in messages of nav2_msgs/CostmapFilterInfo type. These messages are being published by Costmap Filter Info Publisher Server. The server is running as a lifecycle node. According to the design document, nav2_msgs/CostmapFilterInfo messages are going in a pair with OccupancyGrid filter mask topic. Therefore, along with Costmap Filter Info Publisher Server there should be enabled a new instance of Map Server configured to publish filter masks.

In order to enable Speed Filter in your configuration, both servers should be enabled as lifecycle nodes in Python launch-file. For example, this might look as follows:

```
import os

from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node
from nav2_common.launch import RewrittenYaml


def generate_launch_description():
    # Get the launch directory
    costmap_filters_demo_dir = get_package_share_directory('nav2_costmap_filters_demo
↪')
```

(continues on next page)

(continued from previous page)

```
    # Create our own temporary YAML files that include substitutions
    lifecycle_nodes = ['filter_mask_server', 'costmap_filter_info_server']

    # Parameters
    namespace = LaunchConfiguration('namespace')
    use_sim_time = LaunchConfiguration('use_sim_time')
    autostart = LaunchConfiguration('autostart')
    params_file = LaunchConfiguration('params_file')
    mask_yaml_file = LaunchConfiguration('mask')

    # Declare the launch arguments
    declare_namespace_cmd = DeclareLaunchArgument(
        'namespace',
        default_value='',
        description='Top-level namespace')

    declare_use_sim_time_cmd = DeclareLaunchArgument(
        'use_sim_time',
        default_value='true',
        description='Use simulation (Gazebo) clock if true')

    declare_autostart_cmd = DeclareLaunchArgument(
        'autostart', default_value='true',
        description='Automatically startup the nav2 stack')

    declare_params_file_cmd = DeclareLaunchArgument(
            'params_file',
            default_value=os.path.join(costmap_filters_demo_dir, 'params', 'speed_
↪params.yaml'),
            description='Full path to the ROS 2 parameters file to use')

    declare_mask_yaml_file_cmd = DeclareLaunchArgument(
            'mask',
            default_value=os.path.join(costmap_filters_demo_dir, 'maps', 'speed_mask.
↪yaml'),
            description='Full path to filter mask yaml file to load')

    # Make re-written yaml
    param_substitutions = {
        'use_sim_time': use_sim_time,
        'yaml_filename': mask_yaml_file}

    configured_params = RewrittenYaml(
        source_file=params_file,
        root_key=namespace,
        param_rewrites=param_substitutions,
        convert_types=True)

    # Nodes launching commands
    start_lifecycle_manager_cmd = Node(
            package='nav2_lifecycle_manager',
            executable='lifecycle_manager',
            name='lifecycle_manager_costmap_filters',
            namespace=namespace,
            output='screen',
            emulate_tty=True,  # https://github.com/ros2/launch/issues/188
```

(continues on next page)

(continued from previous page)

```
            parameters=[{'use_sim_time': use_sim_time},
                        {'autostart': autostart},
                        {'node_names': lifecycle_nodes}])

    start_map_server_cmd = Node(
            package='nav2_map_server',
            executable='map_server',
            name='filter_mask_server',
            namespace=namespace,
            output='screen',
            emulate_tty=True,  # https://github.com/ros2/launch/issues/188
            parameters=[configured_params])

    start_costmap_filter_info_server_cmd = Node(
            package='nav2_map_server',
            executable='costmap_filter_info_server',
            name='costmap_filter_info_server',
            namespace=namespace,
            output='screen',
            emulate_tty=True,  # https://github.com/ros2/launch/issues/188
            parameters=[configured_params])

    ld = LaunchDescription()

    ld.add_action(declare_namespace_cmd)
    ld.add_action(declare_use_sim_time_cmd)
    ld.add_action(declare_autostart_cmd)
    ld.add_action(declare_params_file_cmd)
    ld.add_action(declare_mask_yaml_file_cmd)

    ld.add_action(start_lifecycle_manager_cmd)
    ld.add_action(start_map_server_cmd)
    ld.add_action(start_costmap_filter_info_server_cmd)

    return ld
```

where the `params_file` variable should be set to a YAML-file having ROS parameters for Costmap Filter Info Publisher Server and Map Server nodes. These parameters and their meaning are listed at *Map Server / Saver* page. Please, refer to it for more information. The example of `params_file` could be found below:

```
costmap_filter_info_server:
  ros__parameters:
    use_sim_time: true
    type: 1
    filter_info_topic: "/costmap_filter_info"
    mask_topic: "/speed_filter_mask"
    base: 100.0
    multiplier: -1.0
filter_mask_server:
  ros__parameters:
    use_sim_time: true
    frame_id: "map"
    topic_name: "/speed_filter_mask"
    yaml_filename: "speed_mask.yaml"
```

Note, that:

- For Speed Filter setting speed restrictions in a percent from maximum speed, the `type` of costmap filter should be set to `1`. All possible costmap filter types could be found at *Map Server / Saver* page.
- Filter mask topic name should be the equal for `mask_topic` parameter of Costmap Filter Info Publisher Server and `topic_name` parameter of Map Server.
- As was described in a previous chapter, `base` and `multiplier` should be set to `100.0` and `-1.0` accordingly for the purposes of this tutorial example.

Ready-to-go standalone Python launch-script, YAML-file with ROS parameters and filter mask example for Speed Filter could be found in a nav2_costmap_filters_demo directory of `navigation2_tutorials` repository. To simply run Filter Info Publisher Server and Map Server tuned on Turtlebot3 standard simulation written at *Getting Started*, build the demo and launch `costmap_filter_info.launch.py` as follows:

```
$ mkdir -p ~/tutorials_ws/src
$ cd ~/tutorials_ws/src
$ git clone https://github.com/ros-planning/navigation2_tutorials.git
$ cd ~/tutorials_ws
$ colcon build --symlink-install --packages-select nav2_costmap_filters_demo
$ source ~/tutorials_ws/install/setup.bash
$ ros2 launch nav2_costmap_filters_demo costmap_filter_info.launch.py params_
↪file:=src/navigation2_tutorials/nav2_costmap_filters_demo/params/speed_params.yaml␣
↪mask:=src/navigation2_tutorials/nav2_costmap_filters_demo/maps/speed_mask.yaml
```

3. Enable Speed Filter

Costmap Filters are Costmap2D plugins. You can enable the `SpeedFilter` plugin in Costmap2D by adding `speed_filter` to the `plugins` parameter in `nav2_params.yaml`. The Speed Filter plugin should have the following parameters defined:

- `plugin`: type of plugin. In our case `nav2_costmap_2d::SpeedFilter`.
- `filter_info_topic`: filter info topic name. This needs to be equal to `filter_info_topic` parameter of Costmap Filter Info Publisher Server from the chapter above.
- `speed_limit_topic`: name of topic to publish speed limit to.

Full list of parameters supported by `SpeedFilter` are listed at the *Speed Filter Parameters* page.

You can place the plugin either in the `global_costmap` section in `nav2_params.yaml` to have speed restriction mask applied to global costmap or in the `local_costmap` to apply speed mask to the local costmap. However, `SpeedFilter` plugin should never be enabled simultaneously for global and local costmaps. Otherwise, it can lead to unwanted multiple "speed restriction" - "no restriction" message chains on speed restriction boundaries, that will cause jerking of the robot or another unpredictable behaviour.

In this tutorial, we will enable Speed Filter for the global costmap. For this use the following configuration:

```
global_costmap:
  global_costmap:
    ros__parameters:
      ...
      plugins: ["static_layer", "obstacle_layer", "inflation_layer"]
      filters: ["speed_filter"]
      ...
      speed_filter:
        plugin: "nav2_costmap_2d::SpeedFilter"
        enabled: True
        filter_info_topic: "/costmap_filter_info"
        speed_limit_topic: "/speed_limit"
```

As stated in the design, Speed Filter publishes speed restricting messages targeted for a Controller Server so that it could restrict maximum speed of the robot when it needed. Controller Server has a `speed_limit_topic` ROS parameter for that, which should be set to the same as in `speed_filter` plugin value. This topic in the map server could also be used to any number of other speed-restricted applications beyond the speed limiting zones, such as dynamically adjusting maximum speed by payload mass.

Set `speed_limit_topic` parameter of a Controller Server to the same value as it set for `speed_filter` plugin:

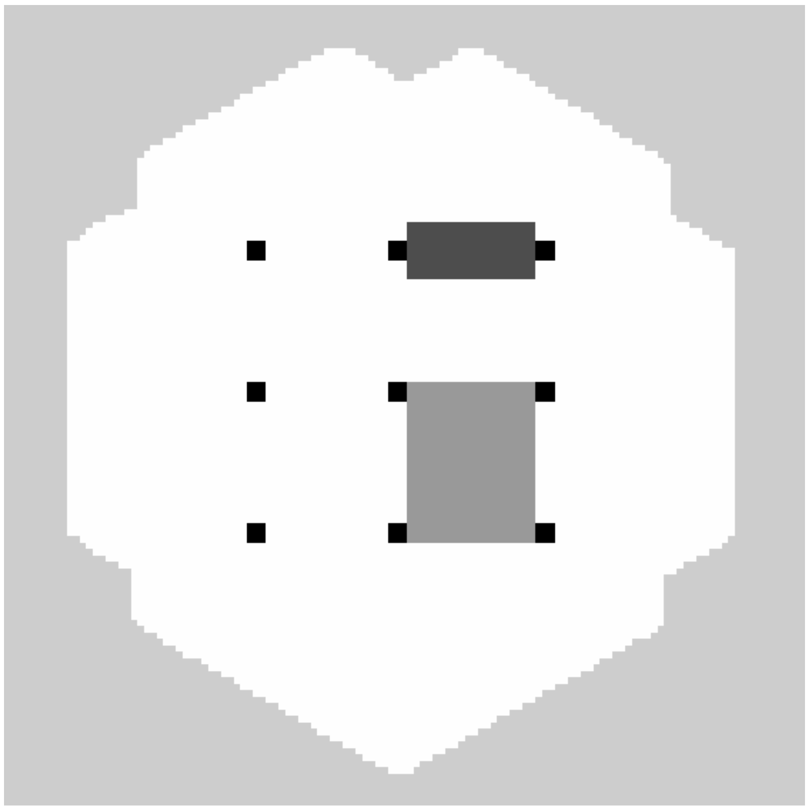
```
controller_server:
  ros__parameters:
    ...
    speed_limit_topic: "/speed_limit"
```

4. Run Nav2 stack

After Costmap Filter Info Publisher Server and Map Server were launched and Speed Filter was enabled for global/local costmap, run Nav2 stack as written in *Getting Started*:

```
ros2 launch nav2_bringup tb3_simulation_launch.py
```

For better visualization of speed filter mask, in RViz in the left `Displays` pane unfold `Map` and change `Topic` from `/map` -> to `/speed_filter_mask`. Set the goal behind the speed restriction areas and check that the filter is working properly: robot should slow down when going through a speed restricting areas. Below is how it might look (first picture shows speed filter enabled for the global costmap, second - `speed_mask.pgm` filter mask):



4.5.11 Using Rotation Shim Controller

Overview

This tutorial will discuss how to set up your robot to use the `RotationShimController` to help create intuitive, rotate-in-place, behavior for your robot while starting out to track a path. The goal of this tutorial is to explain to the reader the value of the controller, how to configure it, how to configure the primary controller with it, and finally an example of it in use.

Before starting this tutorial, completing the *Getting Started* is highly recommended especially if you are new to ROS and Nav2. The requirements are having the latest install of Nav2 / ROS 2 containing this package.

What is the Rotation Shim Controller?

This was developed due to quirks in TEB and DWB, but applicable to any other controller plugin type that you'd like to have rotation in place behavior with. `TEB`'s behavior tends to whip the robot around with small turns, or when the path is starting at a very different heading than current, in a somewhat surprising way due to the elastic band approach. `DWB` can be tuned to have any type of behavior, but typically to tune it to be an excellent path follower also makes it less optimally capable of smooth transitions to new paths at far away headings – there are always trade offs. Giving both TEB and DWB a better starting point to start tracking a path makes tuning the controllers significantly easier and creates more intuitive results for on-lookers.

Note that it is not required to use this with **any** plugin. Many users are perfectly successful without using this controller, but if a robot may rotate in place before beginning its path tracking task (or others), it can be advantageous to do so.

The `nav2_rotation_shim_controller` will check the rough heading difference with respect to the robot and a newly received path. If within a threshold, it will pass the request onto the `primary_controller` to execute the task. If it is outside of the threshold, this controller will rotate the robot in place towards that path heading. Once it is within the tolerance, it will then pass off control-execution from this rotation shim controller onto the primary controller plugin. At this point, the robot's main plugin will take control for a smooth hand off into the task.

The `RotationShimController` is most suitable for:

- Robots that can rotate in place, such as differential and omnidirectional robots.
- Preference to rotate in place when starting to track a new path that is at a significantly different heading than the robot's current heading – or when tuning your controller for its task makes tight rotations difficult.
- Using planners that are non-kinematically feasible, such as NavFn, Theta*, or Smac 2D (Feasible planners such as Smac Hybrid-A* and State Lattice will start search from the robot's actual starting heading, requiring no rotation since their paths are guaranteed drivable by physical constraints).

Note: Regulated Pure Pursuit has this built in so it is not necessary to pair with RPP. However, it is applicable to all others. See *Navigation Plugins* for a full list of current controller plugins.

Configuring Rotation Shim Controller

This controller is a *shim* because it is placed between the primary controller plugin and the controller server. It takes commands and pre-processes them to rotate to the heading and then passes off execution-control to the primary plugin once that condition is met - acting as a simple pass through.

As such, its configuration looks very similar to that of any other plugin. In the code block below, you can see that we've added the `RotationShimController` as the plugin for path tracking in the controller server. You can see that we've also configured it below with its internal parameters, `angular_dist_threshold` through `max_angular_accel`.

```
controller_server:
  ros__parameters:
    use_sim_time: True
    controller_frequency: 20.0
    min_x_velocity_threshold: 0.001
    min_y_velocity_threshold: 0.5
    min_theta_velocity_threshold: 0.001
    progress_checker_plugins: ["progress_checker"] # progress_checker_plugin:
↪"progress_checker" For Humble and older
    goal_checker_plugins: ["goal_checker"]
    controller_plugins: ["FollowPath"]
    progress_checker:
      plugin: "nav2_controller::SimpleProgressChecker"
      required_movement_radius: 0.5
      movement_time_allowance: 10.0
    goal_checker:
      plugin: "nav2_controller::SimpleGoalChecker"
      xy_goal_tolerance: 0.25
      yaw_goal_tolerance: 0.25
      stateful: True
    FollowPath:
      plugin: "nav2_rotation_shim_controller::RotationShimController"
      angular_dist_threshold: 0.785
      forward_sampling_distance: 0.5
      rotate_to_heading_angular_vel: 1.8
      max_angular_accel: 3.2
      simulate_ahead_time: 1.0
```

The Rotation Shim Controller is very simple and only has a couple of parameters to dictate the conditions it should be enacted.

- `angular_dist_threshold`: The angular distance (in radians) apart from the robot's current heading and the approximated path heading to trigger the rotation behavior. Once the robot is within this threshold, control is handed over to the primary controller plugin.
- `forward_sampling_distance`: The distance (in meters) away from the robot to select a point on the path to approximate the path's starting heading at. This is analogous to a "lookahead" point.
- `rotate_to_heading_angular_vel`: The angular velocity (in rad/s) to have the robot rotate to heading by, when the behavior is enacted.
- `max_angular_accel`: The angular acceleration (in rad/s/s) to have the robot rotate to heading by, when the behavior is enacted.
- `simulate_ahead_time`: The Time (s) to forward project the rotation command to check for collision

Configuring Primary Controller

There is one more remaining parameter of the `RotationShimController` not mentioned above, the `primary_controller`. This is the type of controller that your application would like to use as the primary modus operandi. It will share the same name and yaml namespace as the shim plugin. You can observe this below with the primary controller set the `DWB` (with the progress and goal checkers removed for brevity).

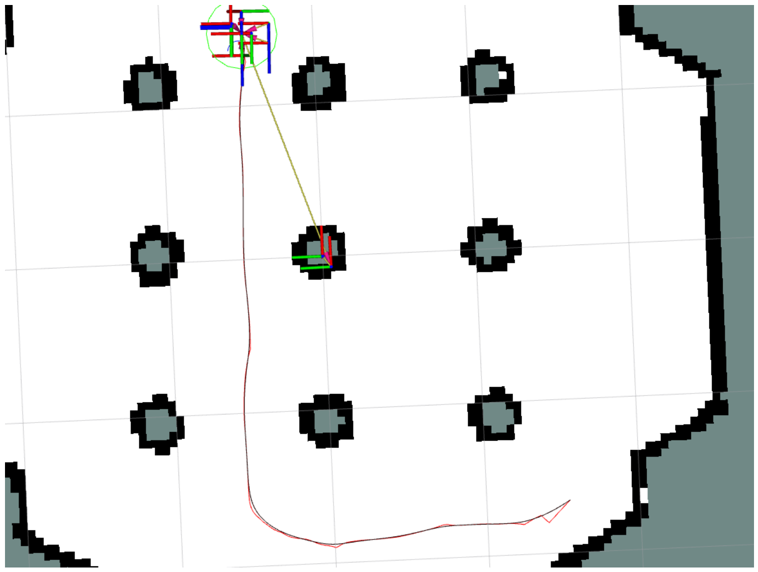
```yaml
controller_server:
  ros__parameters:
    use_sim_time: True
    controller_frequency: 20.0
    min_x_velocity_threshold: 0.001
    min_y_velocity_threshold: 0.5
    min_theta_velocity_threshold: 0.001
    controller_plugins: ["FollowPath"]
    FollowPath:
      plugin: "nav2_rotation_shim_controller::RotationShimController"
      primary_controller: "dwb_core::DWBLocalPlanner"
      angular_dist_threshold: 0.785
      forward_sampling_distance: 0.5
      rotate_to_heading_angular_vel: 1.8
      max_angular_accel: 3.2
      simulate_ahead_time: 1.0

      # DWB parameters
      ...
      ...
      ...
```

An important note is that **within the same yaml namespace**, you may also include any `primary_controller` specific parameters required for a robot. Thusly, after `max_angular_accel`, you can include any of `DWB`'s parameters for your platform.

Demo Execution

4.5.12 Adding a Smoother to a BT

- *Overview*
- *Requirements*
- *Tutorial Steps*



Overview

This tutorial shows how to add a smoother to your behavior tree to smooth output paths from a path planner. Before completing this tutorials, completing *Getting Started* is highly recommended especially if you are new to ROS and Nav2.

Requirements

You must install Nav2, Turtlebot3. If you don't have them installed, please follow *Getting Started*. You must also have a working behavior tree, such as those provided by the Nav2 BT Navigator package, for editing. You should also have a copy of the `nav2_params.yaml` file for your system to edit as well.

Tutorial Steps

0- Familiarization with the Smoother BT Node

The *SmoothPath* BT node is a behavior tree node that interacts with the smoother task server similar to that which you may expect to find for the planner or controller servers. It contains the action client to call the server and specifies its arguments and return types as BT ports. It too calls the server via an action interface that may be seperately interacted with via other servers and client library languages.

Please see the BT node's configuration page to familiarize yourself with all aspects, but the core ports to note are the `unsmoothed_path` input port and the `smoothed_path` output port. The first takes in a raw path from a planning algorithm and the latter will set the value of the smoothed output path post-smoothing. Other ports are available that fully implements the Smoother Server's action API.

1- Specifying a Smoother Plugin

In order to use a smoother in your BT node, you must first configure the smoother server itself to contain the smoother plugins of interest. These plugins implement the specific algorithms that you would like to use.

For each smoother plugin you would like to use, a name must be given to it (e.g. `simple_smoother`, `curvature_smoother`). This name is its `smoother_id` for other servers to interact with this algorithm from a request to the Smoother Server's action interface.

Under each name, the parameters for that particular algorithm must be specified along with the `plugin` name for pluginlib to load a given algorithm's library. An example configuration of 2 smoother plugins is shown below that could be used in the `nav2_params.yaml` for your robot.

```
smoother_server:
  ros__parameters:
    costmap_topic: global_costmap/costmap_raw
    footprint_topic: global_costmap/published_footprint
    robot_base_frame: base_link
    transform_timeout: 0.1
    smoother_plugins: ["simple_smoother", "curvature_smoother"]
    simple_smoother:
      plugin: "nav2_smoother::SimpleSmoother"
      tolerance: 1.0e-10
      do_refinement: True
    curvature_smoother:
      plugin: "nav2_ceres_costaware_smoother/CeresCostawareSmoother"
```

2- Modifying your BT XML

Now that you have selected and configured the smoother server for your given plugin(s), it is time to use those smoother(s) in your behavior tree for navigation behavior. While there are many places / ways to use this in a BT, what is shown below is probably the most likely situation you would want to use the smoother in (to smooth a path returned by the path planner and then using that smoothed path for path tracking).

Note: If you use only a single type of smoothing algorithm, there is no need to specify the `smoother_id` in the BT XML entry. Since there is only a single option, that will be used for any request that does not specifically request a smoother plugin. However, if you leverage multiple smoother plugins, you **must** populate the `smoother_id` XML port.

A given behavior tree will have a line:

```
<ComputePathToPose goal="{goal}" path="{path}" planner_id="GridBased" error_code_id="
↪{compute_path_error_code}"/>
```

This line calls the planner server and return a path to the `path` blackboard variable in the behavior tree. We are going to replace that line with the following to compute the path, smooth the path, and finally replace the `path` blackboard variable with the new smoothed path that the system will now interact with:

```
<Sequence name="ComputeAndSmoothPath">
  <ComputePathToPose goal="{goal}" path="{path}" planner_id="GridBased" error_code_id=
↪"{compute_path_error_code}"/>
  <SmoothPath unsmoothed_path="{path}" smoothed_path="{path}" error_code_id="
↪{smoother_error_code}"/>
</Sequence>
```

If you wish to have recoveries for the smoother error codes, such as triggering the system recoveries branch of a behavior tree:

```
<Sequence name= "TryToResolveSmootherErrorCodes">
  <WouldASmootherRecoveryHelp error_code="{smoother_error_code}">
  <!-- recovery to resolve smoother error code goes here -->
<Sequence/>
```

And its as simple as that! You can now compile or use this behavior tree in your system and see that the plans are now smoothed and the controllers are now tracking this smoothed path. The image at the top of the tutorial shows the unsmoothed path from NavFn (red) and the smoothed path (black). Note the smoother approach to goal, turns, and transitions in the straight-line segments.

If you wish to see the difference, but not track the smoothed path, you may wish to remove the `smoothed_path="{path}"` portion to compute the smoothed path, but not replace the original path with it. Instead, the topic `/smoothed_path` contains this information published by the smoother server for visualization or use by other parts of the system. You may also remap the smoothed path to another blackboard variable to interact with it in other parts of the behavior tree (e.g. `smoothed_path="{smoothed_path}"`).

4.5.13 Using Collision Monitor

- *Overview*
- *Requirements*
- *Preparing Nav2 stack*
- *Configuring Collision Monitor*
- *Demo Execution*

Overview

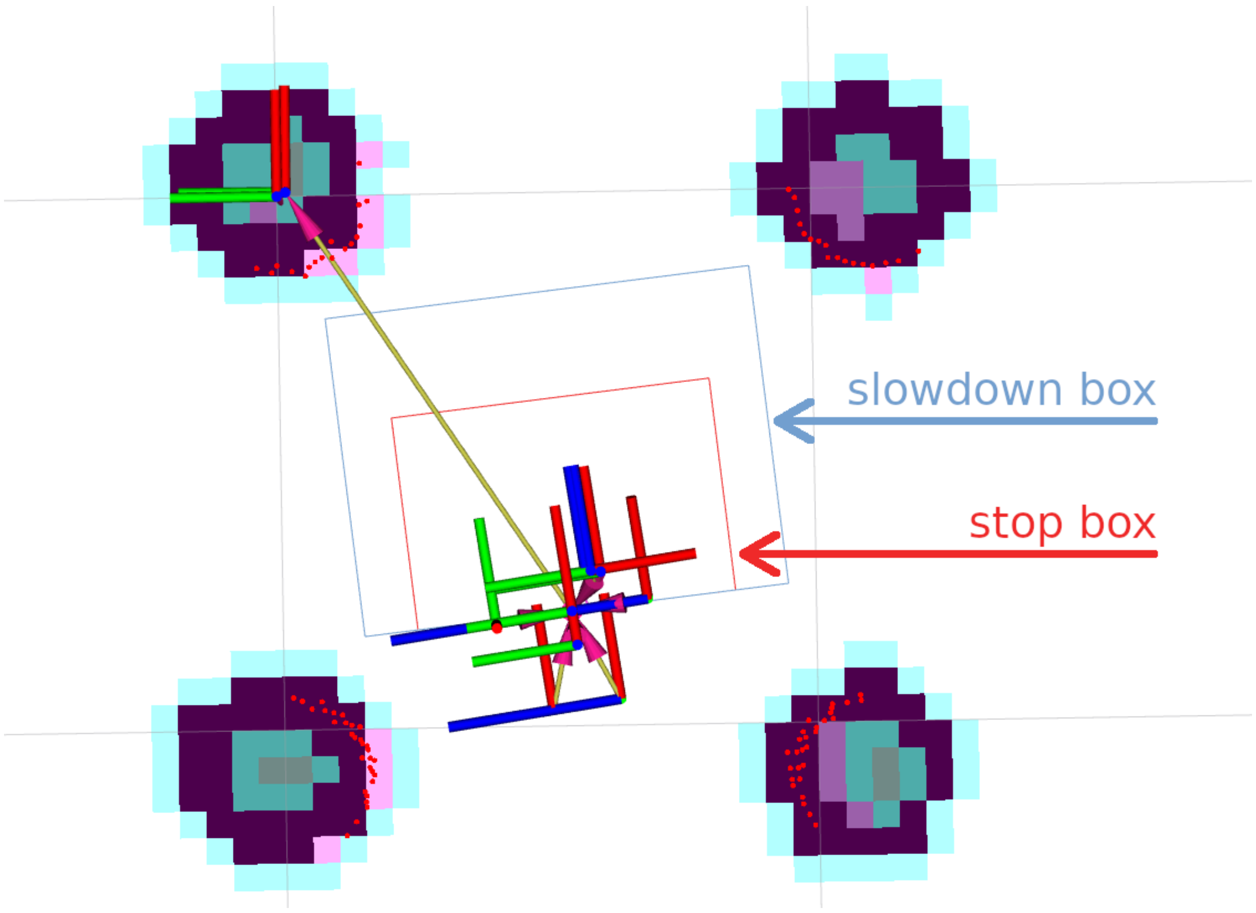
This tutorial shows how to use a Collision Monitor with Nav2 stack. Based on this tutorial, you can setup it for your environment and needs.

Requirements

It is assumed ROS2 and Nav2 dependent packages are installed or built locally. Please make sure that Nav2 project is also built locally as it was made in *Build and Install*.

Configuring Collision Monitor

The Collision Monitor node has its own `collision_monitor_node.launch.py` launch-file and preset parameters in the `collision_monitor_params.yaml` file for demonstration, though its trivial to add this to Nav2's main launch file if being used in practice. For the demonstration, two shapes will be created - an inner stop and a larger slowdown bounding boxes placed in the front of the robot:



If more than 3 points will appear inside a slowdown box, the robot will decrease its speed to `30%` from its value. For the cases when obstacles are dangerously close to the robot, inner stop zone will work. For this setup, the following lines should be added into `collision_monitor_params.yaml` parameters file. Stop box is named as `PolygonStop` and slowdown bounding box - as `PolygonSlow`:

```
polygons: ["PolygonStop", "PolygonSlow"]
PolygonStop:
  type: "polygon"
  points: [0.4, 0.3, 0.4, -0.3, 0.0, -0.3, 0.0, 0.3]
  action_type: "stop"
  min_points: 4  # max_points: 3 for Humble
  visualize: True
  polygon_pub_topic: "polygon_stop"
PolygonSlow:
  type: "polygon"
  points: [0.6, 0.4, 0.6, -0.4, 0.0, -0.4, 0.0, 0.4]
  action_type: "slowdown"
  min_points: 4  # max_points: 3 for Humble
  slowdown_ratio: 0.3
  visualize: True
  polygon_pub_topic: "polygon_slowdown"
```

Note: The circle shape could be used instead of polygon, e.g. for the case of omni-directional robots where the collision can occur from any direction. However, for the tutorial needs, let's focus our view on polygons. For the same reason, we leave out of scope the Approach model. Both of these cases could be easily enabled by referencing to the

Collision Monitor configuration guide.

Note: Both polygon shapes in the tutorial were set statically. However, there is an ability to dynamically adjust them over time using topic messages containing vertices points for polygons or footprints. For more information, please refer to the configuration guide.

For the working configuration, at least one data source should be added. In current demonstration, it is used laser scanner (though `PointCloud2` and Range/Sonar/IR sensors are also possible), which is described by the following lines for Collision Monitor node:

```
observation_sources: ["scan"]
scan:
  type: "scan"
  topic: "scan"
```

Set topic names, frame ID-s and timeouts to work correctly with a default Nav2 setup. The whole `nav2_collision_monitor/params/collision_monitor_params.yaml` file in this case will look as follows:

```
collision_monitor:
  ros__parameters:
    use_sim_time: True
    base_frame_id: "base_footprint"
    odom_frame_id: "odom"
    cmd_vel_in_topic: "cmd_vel_raw"
    cmd_vel_out_topic: "cmd_vel"
    transform_tolerance: 0.5
    source_timeout: 5.0
    stop_pub_timeout: 2.0
    polygons: ["PolygonStop", "PolygonSlow"]
    PolygonStop:
      type: "polygon"
      points: [0.4, 0.3, 0.4, -0.3, 0.0, -0.3, 0.0, 0.3]
      action_type: "stop"
      min_points: 4  # max_points: 3 for Humble
      visualize: True
      polygon_pub_topic: "polygon_stop"
    PolygonSlow:
      type: "polygon"
      points: [0.6, 0.4, 0.6, -0.4, 0.0, -0.4, 0.0, 0.4]
      action_type: "slowdown"
      min_points: 4  # max_points: 3 for Humble
      slowdown_ratio: 0.3
      visualize: True
      polygon_pub_topic: "polygon_slowdown"
    observation_sources: ["scan"]
    scan:
      type: "scan"
      topic: "scan"
```

Preparing Nav2 stack

The Collision Monitor is designed to operate below Nav2 as an independent safety node. This acts as a filter on the `cmd_vel` topic coming out of the Controller Server. If no such zone is triggered, then the Controller's `cmd_vel` is used. Else, it is scaled or set to stop as appropriate. For correct operation of the Collision Monitor with the Controller, it is required to add the `cmd_vel -> cmd_vel_raw` remapping to the `navigation_launch.py` bringup script as presented below:

```
Node(
    package='nav2_controller',
    executable='controller_server',
    output='screen',
    respawn=use_respawn,
    respawn_delay=2.0,
    parameters=[configured_params],
+   remappings=remappings + [('cmd_vel', 'cmd_vel_raw')]),
...
ComposableNode(
    package='nav2_controller',
    plugin='nav2_controller::ControllerServer',
    name='controller_server',
    parameters=[configured_params],
+   remappings=remappings + [('cmd_vel', 'cmd_vel_raw')]),
```

Please note, that the remapped `cmd_vel_raw` topic should match to the input velocity `cmd_vel_in_topic` parameter value of the Collision Monitor node, and the output velocity `cmd_vel_out_topic` parameter value should be actual `cmd_vel` to fit the replacement.

Demo Execution

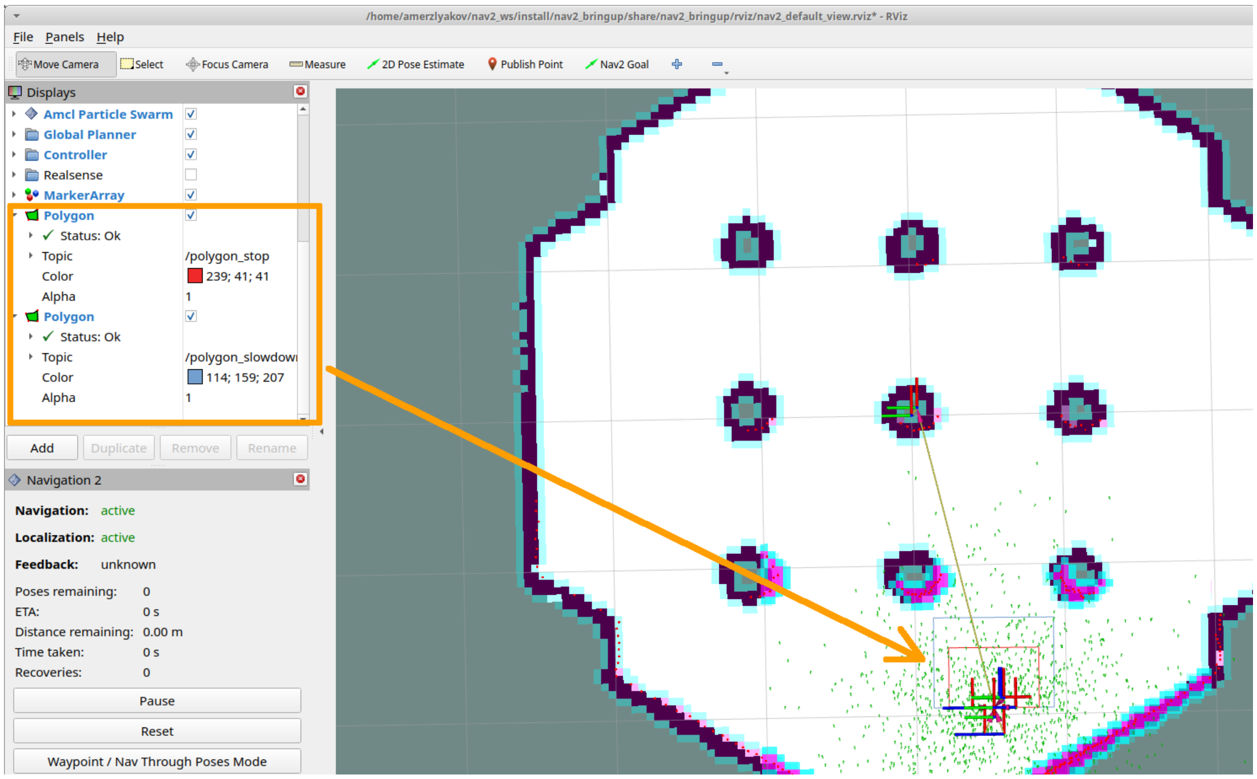
Once Collision Monitor node has been tuned and `cmd_vel` topics remapped, Collision Monitor node is ready to run. For that, run Nav2 stack as written in *Getting Started*:

```
ros2 launch nav2_bringup tb3_simulation_launch.py headless:=False
```

In parallel console, launch Collision Monitor node by using its launch-file:

```
ros2 launch nav2_collision_monitor collision_monitor_node.launch.py
```

Since both `PolygonStop` and `PolygonSlow` polygons will have their own publishers, they could be added to visualization as shown at the picture below:

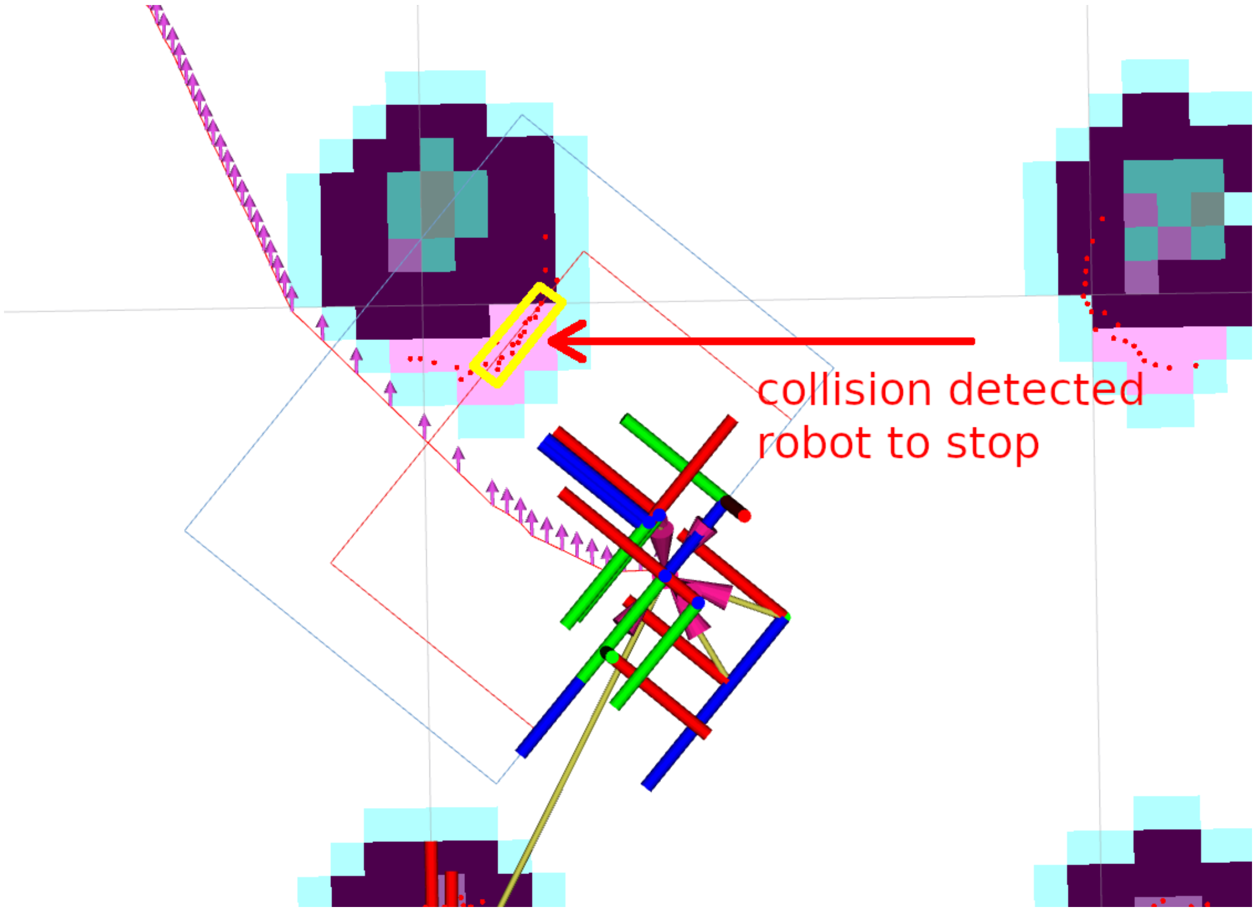


Polygon	✓
Status: Ok	
Topic	/polygon_stop
Color	🟥 239; 41; 41
Alpha	1
Polygon	✓
Status: Ok	
Topic	/polygon_slowdowi
Color	🟦 114; 159; 207
Alpha	1

Navigation: active
Localization: active
Feedback: unknown

Poses remaining: 0
ETA: 0 s
Distance remaining: 0.00 m
Time taken: 0 s
Recoveries: 0

Set the initial pose and then put Nav2 goal on map. The robot will start its movement, slowing down while running near the obstacles, and stopping in close proximity to them:



4.5.14 Adding a New Nav2 Task Server

A nav2 task server consists of server side logic to complete different types of requests, usually called by the autonomy system or through the Behavior Tree Navigator. In this guide, we will discuss the core components needed to add a new task server to Nav2 (ex. Controller, Behavior, Smoother, Planner Servers). Namely, how to set up your new Lifecycle-Component Node for launch and state management and the communication of semantically meaningful error codes (if necessary).

While this tutorial does not cover how to add the complementary Behavior Tree Node to interact with this new Task Server, that is covered at length in *Writing a New Behavior Tree Plugin* so this Task Server can be invoked in the BTs in BT Navigator.

If you've created a new Task Server that may have general reuse for the community, consider contacting the maintainers to add it to the Nav2 project! Nav2 gets better by contributions by users like you!

Lifecycle Nodes

The Lifecycle node is the first key component of a nav2 task server. Lifecycle nodes were introduced in ROS 2 to systematically manage the bringup and shutdown of the different nodes involved in the robot's operation. The use of Lifecycle nodes ensures that all nodes are successfully instantiated before they begin their execution and Nav2 shuts down all nodes if there is any unresponsive node.

Lifecycle nodes contain state machine transitions that enable deterministic behavior in ROS 2 servers. The Lifecycle node transitions in Nav2 are handled by the `Lifecycle Manager`. The Lifecycle Manager transitions the states of the Lifecycle nodes and provides greater control over the state of a system.

The primary states of a Lifecycle node are `Unconfigured`, `Inactive`, `Active`, and `Finalized`. A Lifecycle node starts in an `Unconfigured` state after being instantiated. The Lifecycle Manager transitions a node from `Unconfigured` to `Inactive` by implementing the `Configurating` transition. The `Configurating` transition sets up all configuration parameters and prepares any required setup such as memory allocation and the set up of the static publication and subscription topics. A node in the `Inactive` state is allowed to reconfigure its parameters and but cannot perform any processing. From the `Inactive` state, the Lifecyle Manager implements the `Activating` transition state to transition the node from `Inactive` to `Active`, which is the main state. A node in the `Active` state is allowed to perform any processing operation. In case a node crashes, the Lifecycle Manager shuts down the system to prevent any critical failures. On shutdown, the necessary cleanup operations are performed and the nodes are transitioned to the `Finalized` state via `Deactivating`, `CleaningUp`, and `ShuttingDown` transition states.

See also:

For more information on Lifecycle management, see the article on Managed Nodes.

You may wish to integrate your own nodes into the Nav2 framework or add new lifecycle nodes to your system. As an example, we will add a new notional lifecycle node `sensor_driver`, and have it be controlled via the Nav2 Lifecycle Manager to ensure sensor feeds are available before activating navigation. You can do so by adding a `sensor_driver` node in your launch file and adding it to the list of nodes to be activated by the `lifecycle_manager` before navigation, as shown in the example below.

```
lifecycle_nodes = ['sensor_driver',
                   'controller_server',
                   'smoother_server',
                   'planner_server',
                   'behavior_server',
                   'bt_navigator',
                   'waypoint_follower']

...

Node(
    package='nav2_sensor_driver',
    executable='sensor_driver',
    name='sensor_driver',
    output='screen',
    parameters=[configured_params],
    remappings=remappings),

Node(
    package='nav2_lifecycle_manager',
    executable='lifecycle_manager',
    name='lifecycle_manager_navigation',
    output='screen',
    parameters=[{'autostart': autostart},
                {'node_names': lifecycle_nodes}]),
```

In the snippet above, the nodes to be handled by the Lifecycle Manager are set using the `node_names` parameter. The `node_names` parameter takes in an ordered list of nodes to bringup through the Lifecycle transition. As shown in the snippet, the `node_names` parameter takes in `lifecycle_nodes` which contains the list of nodes to be added to the Lifecycle Manager. The Lifecycle Manager implements bringup transitions (`Configuring` and `Activating`) to the nodes one-by-one and in order, while the nodes are processed in reverse order for shutdown transitions. Hence, the `sensor_driver` is listed first before the other navigation servers so that the sensor data is available before the navigation servers are activated.

Two other parameters of the Lifecycle Manager are `autostart` and `bond_timeout`. Set `autostart` to `true` if you want to set the transition nodes to the `Active` state on startup. Otherwise, you will need to manually trigger Lifecycle Manager to transition up the system. The `bond_timeout` sets the waiting time to decide when to transition down all of the nodes if a node is not responding.

Note: More information on Lifecycle Manager parameters can be found in the Configuration Guide of Lifecycle Manager

Composition

Composition is the second key component nav2 task servers that was introduced to reduce the memory and CPU resources by putting multiple nodes in a single process. In Nav2, Composition can be used to compose all Nav2 nodes in a single process instead of launching them separately. This is useful for deployment on embedded systems where developers need to optimize resource usage.

See also:

More information on Composition can be found here.

In the following section, we give an example on how to add a new Nav2 server, which we notionally call the `route_server`, to our system.

We make use of the launch files to compose different servers into a single process. The process is established by the `ComposableNodeContainer` container that is populated with composition nodes via `ComposableNode`. This container can then be launched and used the same as any other Nav2 node.

1. Add a new `ComposableNode()` instance in your launch file pointing to the component container of your choice.

```
container = ComposableNodeContainer(
    name='my_container',
    namespace='',
    package='rclcpp_components',
    executable='component_container',
    composable_node_descriptions=[
        ComposableNode(
            package='nav2_route_server',
            plugin='nav2_route_server::RouteServer',
            name='nav2_route_server'),
    ],
    output='screen',
)
```

   See also:

   See example in composition demo's composition_demo.launch.py.

2. Add the package containing the server to your `package.xml` file.

```
<exec_depend>nav2_route_server</exec_depend>
```

Error codes

Your nav2 task server may also wish to return a 'error_code' in its action response (though not required). If there are semantically meaningful and actionable types of failures for your system, this is a systemic way to communicate those failures which may be automatically aggregated into the responses of the navigation system to your application.

It is important to note that error codes from 0-9999 are reserved for internal nav2 servers with each server offset by 100 while external servers start at 10000 and end at 65535. The table below shows the current servers along with the expected error code structure.

Server Name	Reserved	RANGE
…	NONE=0, UNKNOWN=1	2-99
Controller Server	NONE=0, UNKNOWN=100	101-199
Planner Server (compute_path_to_pose)	NONE=0, UNKNOWN=200	201-299
Planner Server (compute_path_through_poses)	NONE=0, UNKNOWN=300	301-399
…	…	
Smoother Server	NONE=0, UNKNOWN=500	501-599
Waypoint Follower Server	NONE=0, UNKNOWN=600	601-699
Behavior Server	NONE=0	701-799
…	…	
Last Nav2 Server	NONE=0, UNKNOWN=9900	9901-9999
First External Server	NONE=0, UNKNOWN=10000	10001-10099
…	…	

Error codes are attached to the response of the action message. An example can be seen below for the route server. Note that by convention we set the error code field within the message definition to `error_code`.

```
# Error codes
# Note: The expected priority order of the errors should match the message order
uint16 NONE=0 # 0 is reserved for NONE
uint16 UNKNOWN=10000 # first error code in the sequence is reserved for UNKNOWN

# User Error codes below
int16 INVILAD_START=10001
int16 NO_VALID_ROUTE=10002

#goal definition
route_msgs/PoseStamped goal
route_msgs/PoseStamped start
string route_id
---
#result definition
nav_msgs/Route route
builtin_interfaces/Duration route_time
uint16 error_code
---
```

As stated in the message, the priority order of the errors should match the message order, 0 is reserved for NONE and the first error code in the sequence is reserved for UNKNOWN. Since the the route server is a external server, the errors codes start at 10000 and go up to 10099.

In order to propigate your server's error code to the rest of the system it must be added to the nav2_params.yaml file.

The *error_code_id_names* inside of the BT Navigator define what error codes to look for on the blackboard by the server. The lowest error code of the sequence is then returned - whereas the code enums increase the higher up in the software stack - giving higher priority to lower-level failures.

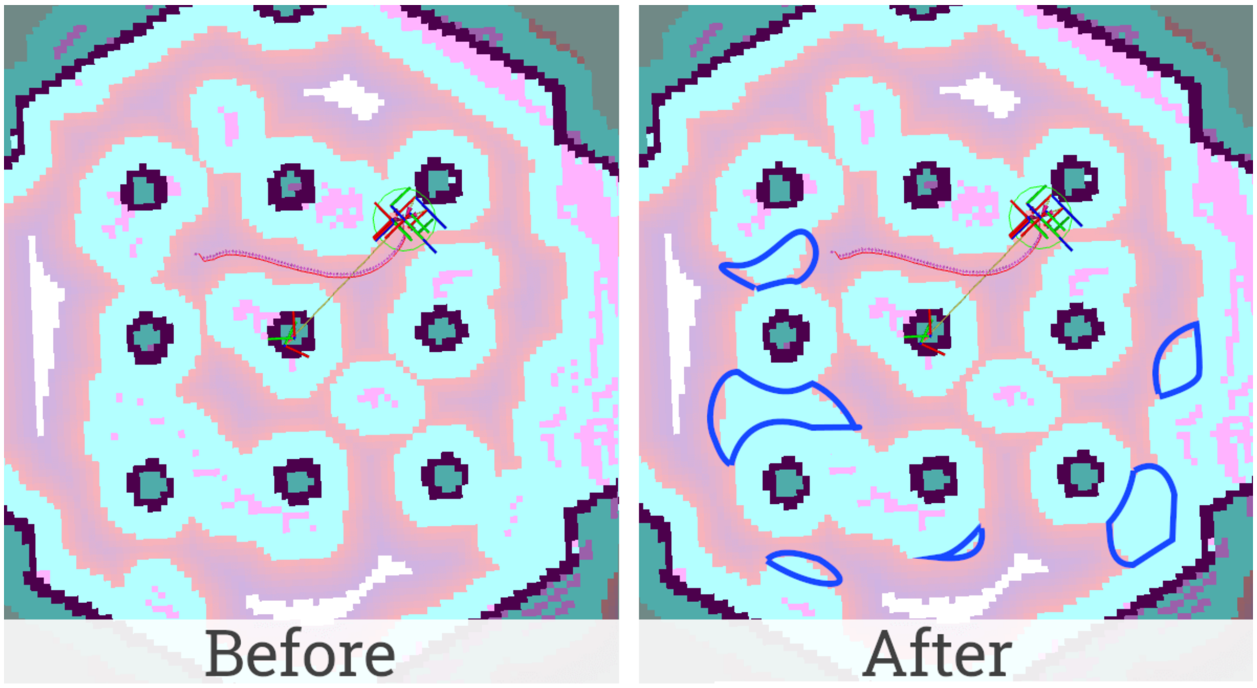
```
error_code_id_names:
    - compute_path_error_code_id
    - follow_path_error_code_id
    - route_error_code_id
```

Conclusion

In this section of the guide, we have discussed Lifecycle Nodes, Composition and Error Codes which are new and important concepts in ROS 2. We also showed how to implement Lifecycle Nodes, Composition and Error Codes to your newly created nodes/servers with Nav2. These three concepts are helpful to efficiently run your system and therefore are encouraged to be used throughout Nav2.

4.5.15 Filtering of Noise-Induced Obstacles

- *Overview*
- *Requirements*
- *Tutorial Steps*
- *How it works*



Overview

Noisy sensor measurements can cause to errors in `Voxel Layer` or `Obstacle Layer`. As a result, salt and pepper noise may appear on the costmap. This noise creates false obstacles that prevent the robot from finding the best path on the map. While the images above show both salt and pepper noise as well as error due to mislocalization, this layer will only remove sensor noise, not mislocalized artifacts misaligned with the static map. This tutorial shows how to configure filtering of false obstacles caused by noise. This functionality is provided by the `DenoiseLayer` costmap layer plugin which will be enabled and used in this document.

Requirements

It is assumed that ROS 2, Gazebo and TurtleBot3 packages are installed or built locally. Please make sure that Nav2 project is also built locally as it was made in *Build and Install*.

Tutorial Steps

1. Enable Denoise Layer

Denoise Layer is Costmap2D plugin. You can enable the `DenoiseLayer` plugin in Costmap2D by adding `denoise_layer` to the `plugins` parameter in `nav2_params.yaml`. You can place it in the `global_costmap` and (or) `local_costmap` to filter noise on a global or local map. The DenoiseLayer plugin should have the following parameter defined:

- `plugin`: type of plugin. In our case `nav2_costmap_2d::DenoiseLayer`.

Full list of parameters supported by `DenoiseLayer` are listed at *Denoise Layer Parameters* page.

It is important to note that `DenoiseLayer` typically should be placed before the inflation layer. This is required to prevent inflation from noise-induced obstacles. Moreover, `DenoiseLayer` processes only obstacle information in the costmap. Values `INSCRIBED_INFLATED_OBSTACLE`, `LETHAL_OBSTACLE` and optionally `NO_INFORMATION` will be interpreted as obstacle cell. Cells with any other values will be interpreted as `FREE_SPACE` when processed (won't be distorted in the cost map). If a cell with an obstacle is recognized as noise, it will be replaced by `FREE_SPACE` after processing.

To enable `DenoiseLayer` for both global and local costmaps, use the following configuration:

```
global_costmap:
  global_costmap:
    ros__parameters:
      ...
      plugins: ["static_layer", "obstacle_layer", "denoise_layer", "inflation_layer"]
      ...
      denoise_layer:
        plugin: "nav2_costmap_2d::DenoiseLayer"
        enabled: True
...
local_costmap:
  local_costmap:
    ros__parameters:
      ...
      plugins: ["voxel_layer", "denoise_layer", inflation_layer"]
      ...
      keepout_filter:
        plugin: "nav2_costmap_2d::DenoiseLayer"
        enabled: True
```

Note: The key to success in filtering noise is to understand its type and choose the right `DenoiseLayer` parameters. The default parameters are focused on fast removal of standalone obstacles. More formally, an obstacle is discarded if there are no obstacles among the adjacent eight cells. This should be sufficient in typical cases.

If some sensor generates intercorrelated noise-induced obstacles and small obstacles in the world are unlikely, small groups of obstacles can be removed. To configure the `DenoiseLayer` to such cases and understand how it works, refer to the section *How it works*.

Warning: Use this plugin to filter the global costmap with caution. It introduces potential performance issues. For example in case of typically-high-range lidars (20+ meters) update window can be massive making processing time unacceptably long. It is worth taking this into account as an application designer.

2. Run Nav2 stack

After Denoise Layer was enabled for global/local costmaps, run Nav2 stack as written in *Getting Started*:

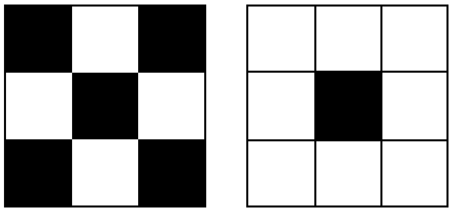
```
ros2 launch nav2_bringup tb3_simulation_launch.py headless:=False
```

And check that filter is working properly: with the default parameters, no standalone obstacles should remain on the cost map. This can be checked, for example, in RViz main window displaying local and global costmaps after removing unnecessary particles (illustrated at the top of this tutorial).

How it works

The plugin is based on two algorithms.

When parameter `minimal_group_size` = 2, the first algorithm turns on. It apply erosion function with kernel from image below (left if `group_connectivity_type` = 4 or right if `group_connectivity_type` = 8) to the costmap. White color of the kernel pixel means to use the value, black means to ignore it.



As a result of erosion function the neighbors image is created. Each possible position of the kernel on the costmap corresponds to one pixel of the neighbors image. The pixel value of this image is equal to the maximum of 4/8 costmap pixels corresponding to the white pixels of the mask. In other words, the pixel of the neighbors image is equal to the obstacle code if there is an obstacle nearby, the free space code in other case. After that, obstacles corresponding to free space code on neighbors image are removed.

This process is illustrated below. On the left side of the image is a costmap, on the right is a neighbors image. White pixels are free space, black pixels are obstacles, `group_connectivity_type` = 4. Obstacles marked at the end of the animation will be removed.

When parameter `minimal_group_size` > 2, the second algorithm is executed. This is a generalized solution that allows you to remove groups of adjacent obstacles if their total number is less than `minimal_group_size`. To select groups of adjacent obstacles, the algorithm performs their segmentation. The type of cell connectivity in one segment is determined by the parameter `group_connectivity_type`. Next, the size of each segment is calculated. Obstacles segments with size less than the `minimal_group_size` are replaced with empty cells. This algorithm is about 10 times slower than first, so use it with caution and only when necessary. Its execution time depends on the size of the processed map fragment (and not depend on the value of `minimal_group_size`).

This algorithm is illustrated in the animation below (`group_connectivity_type` = 8). Obstacles marked at the end of the animation will be removed (groups that size less 3).

4.6 Plugin Tutorials

Navigation2 Tutorials

4.6.1 Writing a New Costmap2D Plugin

- *Overview*
- *Requirements*
- *Tutorial Steps*

Overview

This tutorial shows how to create your own simple plugin for Costmap2D.

Before starting the tutorial, please check this video which contains information about Costmap2D layers design and plugins basic operational principals.

Requirements

It is assumed that ROS 2, Gazebo and TurtleBot3 packages are installed or built locally. Please make sure that Nav2 project is also built locally as it was made in *Build and Install*.

Tutorial Steps

1- Write a new Costmap2D plugin

For a demonstration, this example will create a costmap plugin that puts repeating cost gradients in the costmap. The annotated code for this tutorial can be found in navigation2_tutorials repository as the `nav2_gradient_costmap_plugin` ROS 2-package. Please refer to it when making your own layer plugin for Costmap2D.

The plugin class `nav2_gradient_costmap_plugin::GradientLayer` is inherited from basic class `nav2_costmap_2d::Layer`:

```
namespace nav2_gradient_costmap_plugin
{

class GradientLayer : public nav2_costmap_2d::Layer
```

The basic class provides the set of virtual methods API for working with costmap layers in a plugin. These methods are called at runtime by LayeredCostmap. The list of methods, their description, and necessity to have these methods in plugin's code is presented in the table below:

Virtual method	Method description	Requires override?
onInitialize()	Method is called at the end of plugin initialization. There is usually declarations of ROS parameters. This is where any required initialization should occur.	No
updateBounds()	Method is called to ask the plugin: which area of costmap layer it needs to update. The method has 3 input parameters: robot position and orientation, and 4 output parameters: pointers to window bounds. These bounds are used for performance reasons: to update the area inside the window where new info is available, avoiding updates of the whole costmap on every iteration.	Yes
updateCosts()	Method is called each time when costmap re-calculation is required. It updates the costmap layer only within its bounds window. The method has 4 input parameters: calculation window bounds, and 1 output parameter: reference to a resulting costmap master_grid. The Layer class provides the plugin with an internal costmap, costmap_, for updates. The master_grid should be updated with values within the window bounds using one of the following update methods: updateWithAddition(), updateWithMax(), updateWithOverwrite() or updateWithTrueOverwrite().	Yes
matchSize()	Method is called each time when map size was changed.	No
onFootprintChanged()	Method is called each time when footprint was changed.	No
reset()	It may have any code to be executed during costmap reset.	Yes

In our example these methods have the following functionality:

1. GradientLayer::onInitialize() contains declaration of a ROS parameter with its default value:

```
declareParameter("enabled", rclcpp::ParameterValue(true));
node_->get_parameter(name_ + "." + "enabled", enabled_);
```

and sets need_recalculation_ bounds recalculation indicator:

```
need_recalculation_ = false;
```

2. GradientLayer::updateBounds() re-calculates window bounds if need_recalculation_ is true and updates them regardless of need_recalculation_ value.

3. GradientLayer::updateCosts() - in this method the gradient is writing directly to the resulting costmap master_grid without merging with previous layers. This is equal to working with internal costmap_ and then calling updateWithTrueOverwrite() method. Here is the gradient making algorithm for master costmap:

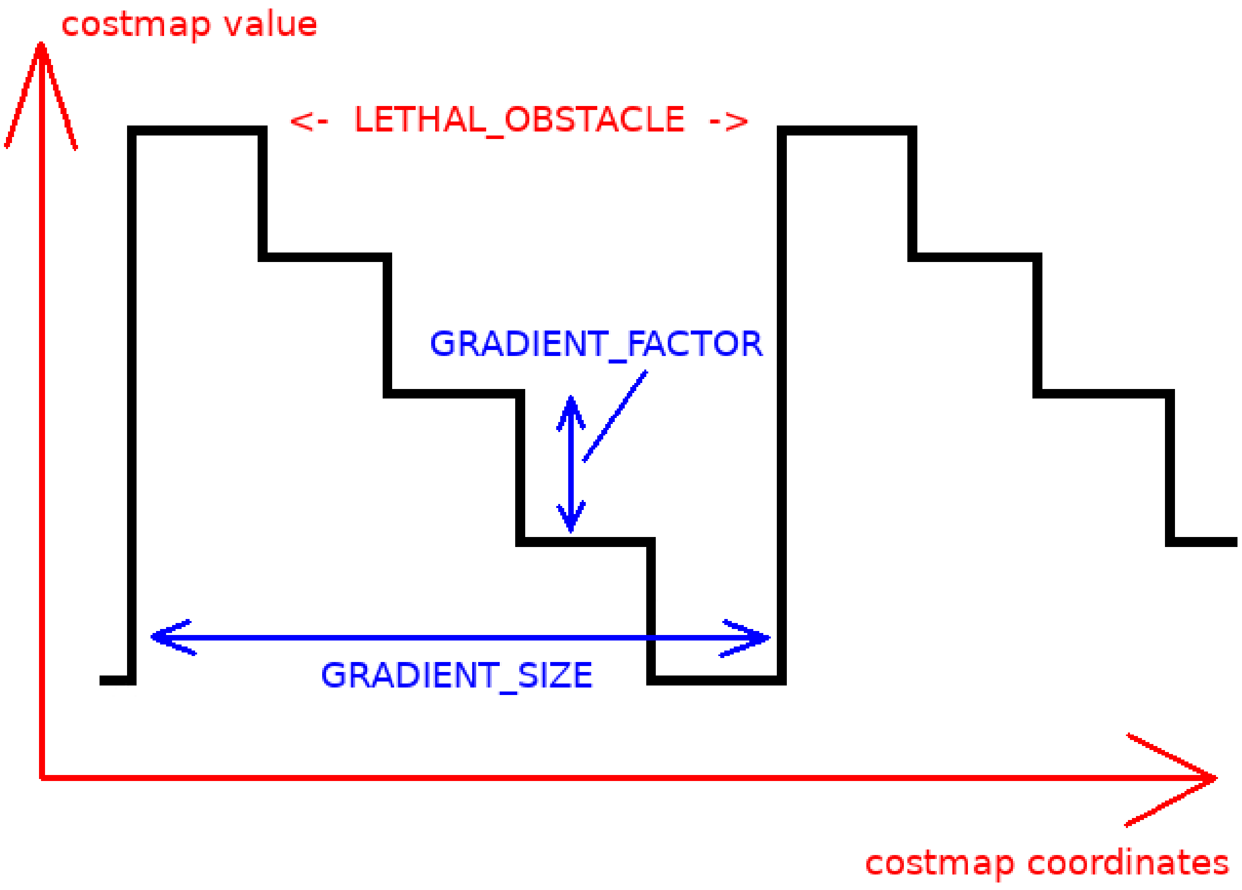
```
int gradient_index;
for (int j = min_j; j < max_j; j++) {
  // Reset gradient_index each time when reaching the end of re-calculated window
  // by OY axis.
  gradient_index = 0;
  for (int i = min_i; i < max_i; i++) {
```

(continues on next page)

(continued from previous page)

```
      int index = master_grid.getIndex(i, j);
      // setting the gradient cost
      unsigned char cost = (LETHAL_OBSTACLE - gradient_index*GRADIENT_FACTOR)%255;
      if (gradient_index <= GRADIENT_SIZE) {
        gradient_index++;
      } else {
        gradient_index = 0;
      }
      master_array[index] = cost;
    }
  }
```

where the GRADIENT_SIZE is the size of each gradient period in map cells, GRADIENT_FACTOR - decrement of costmap's value per each step:



These parameters are defined in plugin's header file.

4. GradientLayer::onFootprintChanged() just resets need_recalculation_ value.
5. GradientLayer::reset() method is dummy: it is not used in this example plugin. It remains there since pure virtual function reset() in parent Layer class required to be overridden.

2- Export and make GradientLayer plugin

The written plugin will be loaded at runtime as its basic parent class and then will be called by plugin handling modules (for costmap2d by `LayeredCostmap`). Pluginlib opens a given plugin in run-time and provides methods from exported classes to be callable. The mechanism of class exporting tells pluginlib which basic class should be used during these calls. This allows to extend an application by plugins without knowing application source code or recompiling it.

In our example the `nav2_gradient_costmap_plugin::GradientLayer` plugin's class should be dynamically loaded as a `nav2_costmap_2d::Layer` basic class. For this the plugin should be registered as follows:

1. Plugin's class should be registered with a basic type of loaded class. For this there is a special macro `PLUGINLIB_EXPORT_CLASS` should be added to any source-file composing the plugin library:

```
#include "pluginlib/class_list_macros.hpp"
PLUGINLIB_EXPORT_CLASS(nav2_gradient_costmap_plugin::GradientLayer, nav2_costmap_
↪2d::Layer)
```

This part is usually placed at the end of cpp-file where the plugin class was written (in our example `gradient_layer.cpp`). It is good practice to place these lines at the end of the file, but technically, you can also place at the top.

2. Plugin's inormation should be stored to plugin description file. This is done by using separate XML (in our example `gradient_plugins.xml`) in the plugin's package. This file contains information about:

- `path`: Path and name of library where plugin is placed.
- `name`: Plugin type referenced in `plugin_types` parameter (see next section for more details). It could be whatever you want.
- `type`: Plugin class with namespace taken from the source code.
- `basic_class_type`: Basic parent class from which plugin class was derived.
- `description`: Plugin description in a text form.

```
<library path="nav2_gradient_costmap_plugin_core">
  <class name="nav2_gradient_costmap_plugin/GradientLayer" type="nav2_gradient_
↪costmap_plugin::GradientLayer" base_class_type="nav2_costmap_2d::Layer">
    <description>This is an example plugin which puts repeating costs gradients to␣
↪costmap</description>
  </class>
</library>
```

The export of plugin is performed by including `pluginlib_export_plugin_description_file()` cmake-function into `CMakeLists.txt`. This function installs plugin description file into `share` directory and sets ament indexes for plugin description XML to be discoverable as a plugin of selected type:

```
pluginlib_export_plugin_description_file(nav2_costmap_2d gradient_layer.xml)
```

Plugin description file is also should be added to `package.xml`. `costmap_2d` is the package of the interface definition, for our case `Layer`, and requires a path to the xml file:

```
<export>
  <costmap_2d plugin="${prefix}/gradient_layer.xml" />
  ...
</export>
```

After everything is done put the plugin package into `src` directory of a certain ROS 2-workspace, build the plugin package (`colcon build --packages-select nav2_gradient_costmap_plugin --symlink-install`) and source `setup.bash` file when it necessary.

Now the plugin is ready to use.

3- Enable the plugin in Costmap2D

At the next step it is required to tell Costmap2D about new plugin. For that the plugin should be added to `plugin_names` and `plugin_types` lists in `nav2_params.yaml` optionally for `local_costmap/global_costmap` in order to be enabled in run-time for Controller/Planner Server. `plugin_names` list contains the names of plugin objects. These names could be anything you want. `plugin_types` contains types of listed in `plugin_names` objects. These types should correspond to `name` field of plugin class specified in plugin description XML-file.

Note: For Galactic or later, `plugin_names` and `plugin_types` have been replaced with a single `plugins` string vector for plugin names. The types are now defined in the `plugin_name` namespace in the `plugin:` field (e.g. `plugin: MyPlugin::Plugin`). Inline comments in the code blocks will help guide you through this.

For example:

```
--- a/nav2_bringup/bringup/params/nav2_params.yaml
+++ b/nav2_bringup/bringup/params/nav2_params.yaml
@@ -124,8 +124,8 @@ local_costmap:
       width: 3
       height: 3
       resolution: 0.05
-      plugins: ["obstacle_layer", "voxel_layer", "inflation_layer"]
+      plugins: ["obstacle_layer", "voxel_layer", "gradient_layer"]
       robot_radius: 0.22
       inflation_layer:
         cost_scaling_factor: 3.0
@@ -171,8 +171,8 @@ global_costmap:
       robot_base_frame: base_link
       global_frame: map
       use_sim_time: True
-      plugins: ["static_layer", "obstacle_layer", "voxel_layer", "inflation_layer"]
+      plugins: ["static_layer", "obstacle_layer", "voxel_layer", "gradient_layer"]
       robot_radius: 0.22
       resolution: 0.05
       obstacle_layer:
```

YAML-file may also contain the list of parameters (if any) for each plugin, identified by plugins object name.

NOTE: there could be many simultaneously loaded plugin objects of one type. For this, `plugin_names` list should contain different plugins names whether the `plugin_types` will remain the same types. For example:

```
plugins: ["obstacle_layer", "gradient_layer_1", "gradient_layer_2"]
```

In this case each plugin object will be handled by its own parameters tree in a YAML-file, like:

```
gradient_layer_1:
  plugin: nav2_gradient_costmap_plugin/GradientLayer
  enabled: True
  ...
```

(continues on next page)

(continued from previous page)

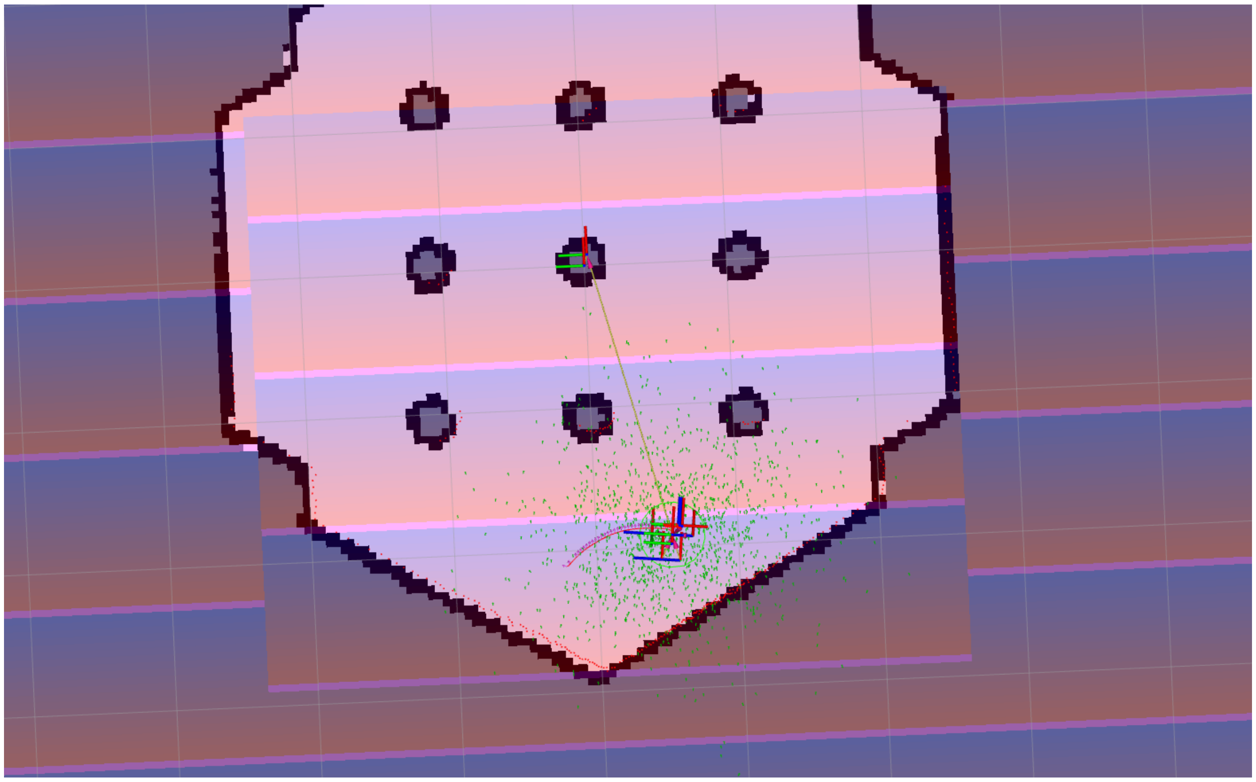
```
gradient_layer_2:
  plugin: nav2_gradient_costmap_plugin/GradientLayer
  enabled: False
  ...
```

4- Run GradientLayer plugin

Run Turtlebot3 simulation with enabled Nav2. Detailed instructions how to make it are written at *Getting Started*. Below is shortcut command for that:

```
$ ros2 launch nav2_bringup tb3_simulation_launch.py
```

Then goto RViz and click on the "2D Pose Estimate" button at the top and point the location on map as it was described in *Getting Started*. Robot will be localized on map and the result should be as presented at picture below. There, the gradient costmap can be seen. There are also 2 noticeable things: dynamically updated by `GradientLayer::updateCosts()` costmap within its bounds and global path curved by gradient:



4.6.2 Writing a New Planner Plugin

- *Overview*
- *Requirements*
- *Tutorial Steps*

Overview

This tutorial shows how to create your own planner plugin.

Requirements

- ROS 2 (binary or build-from-source)
- Nav2 (Including dependencies)
- Gazebo
- Turtlebot3

Tutorial Steps

1- Creating a new Planner Plugin

We will create a simple straight-line planner. The annotated code in this tutorial can be found in navigation_tutorials repository as the `nav2_straightline_planner`. This package can be considered as a reference for writing planner plugin.

Our example plugin inherits from the base class `nav2_core::GlobalPlanner`. The base class provides 5 pure virtual methods to implement a planner plugin. The plugin will be used by the planner server to compute trajectories. Let's learn more about the methods needed to write a planner plugin.

Virtual method	**Method description**	**Requires override?**
configure()	Method is called at when planner server enters on_configure state. Ideally this methods should perform declarations of ROS parameters and initialization of planner's member variables. This method takes 4 input params: shared pointer to parent node, planner name, tf buffer pointer and shared pointer to costmap.	Yes
activate()	Method is called when planner server enters on_activate state. Ideally this method should implement operations which are neccessary before planner goes to an active state.	Yes
deactivate()	Method is called when planner server enters on_deactivate state. Ideally this method should implement operations which are neccessary before planner goes to an inactive state.	Yes
cleanup()	Method is called when planner server goes to on_cleanup state. Ideally this method should clean up resoures which are created for the planner.	Yes
createPlan()	Method is called when planner server demands a global plan for specified start and goal pose. This method returns *nav_msgs::msg::Path* carrying global plan. This method takes 2 input parmas: start pose and goal pose.	Yes

For this tutorial, we will be using methods `StraightLine::configure()` and `StraightLine::createPlan()` to create straight-line planner.

In planners, `configure()` method must set member variables from ROS parameters and any initialization required,

```
node_ = parent;
tf_ = tf;
name_ = name;
costmap_ = costmap_ros->getCostmap();
global_frame_ = costmap_ros->getGlobalFrameID();

// Parameter initialization
nav2_util::declare_parameter_if_not_declared(node_, name_ + ".interpolation_resolution
↪", rclcpp::ParameterValue(0.1));
node_->get_parameter(name_ + ".interpolation_resolution", interpolation_resolution_);
```

Here, `name_ + ".interpolation_resolution"` is fetching the ROS parameters `interpolation_resolution` which is specific to our planner. Nav2 allows the loading of multiple plugins, and to keep things organized, each plugin is mapped to some ID/name. Now if we want to retrieve the parameters for that specific plugin, we use `<mapped_name_of_plugin>.<name_of_parameter>` as done in the above snippet. For example, our example planner is mapped to the name "GridBased" and to retrieve the `interpolation_resolution` parameter which is specific to "GridBased", we used `Gridbased.interpolation_resolution`. In other words, `GridBased` is used as a namespace for plugin-specific parameters. We will see more on this when we discuss the parameters file (or params file).

In `createPlan()` method, we need to create a path from the given start to goal poses. The `StraightLine::createPlan()` is called using start pose and goal pose to solve the global path planning problem. Upon succeeding, it converts the path to the `nav_msgs::msg::Path` and returns to the planner server. Below annotation shows the implementation of this method.

```
nav_msgs::msg::Path global_path;

// Checking if the goal and start state is in the global frame
if (start.header.frame_id != global_frame_) {
  RCLCPP_ERROR(
    node_->get_logger(), "Planner will only except start position from %s frame",
    global_frame_.c_str());
  return global_path;
}

if (goal.header.frame_id != global_frame_) {
  RCLCPP_INFO(
    node_->get_logger(), "Planner will only except goal position from %s frame",
    global_frame_.c_str());
  return global_path;
}

global_path.poses.clear();
global_path.header.stamp = node_->now();
global_path.header.frame_id = global_frame_;
// calculating the number of loops for current value of interpolation_resolution_
int total_number_of_loop = std::hypot(
  goal.pose.position.x - start.pose.position.x,
  goal.pose.position.y - start.pose.position.y) /
  interpolation_resolution_;
double x_increment = (goal.pose.position.x - start.pose.position.x) / total_number_of_
↪loop;
double y_increment = (goal.pose.position.y - start.pose.position.y) / total_number_of_
↪loop;
```

(continues on next page)

(continued from previous page)

```

for (int i = 0; i < total_number_of_loop; ++i) {
  geometry_msgs::msg::PoseStamped pose;
  pose.pose.position.x = start.pose.position.x + x_increment * i;
  pose.pose.position.y = start.pose.position.y + y_increment * i;
  pose.pose.position.z = 0.0;
  pose.pose.orientation.x = 0.0;
  pose.pose.orientation.y = 0.0;
  pose.pose.orientation.z = 0.0;
  pose.pose.orientation.w = 1.0;
  pose.header.stamp = node_->now();
  pose.header.frame_id = global_frame_;
  global_path.poses.push_back(pose);
}

global_path.poses.push_back(goal);

return global_path;
```

The remaining methods are not used but it's mandatory to override them. As per the rules, we did override all but left them blank.

2- Exporting the planner plugin

Now that we have created our custom planner, we need to export our planner plugin so that it will be visible to the planner server. Plugins are loaded at runtime and if they are not visible, then our planner server won't be able to load it. In ROS 2, exporting and loading plugins is handled by `pluginlib`.

Coming back to our tutorial, class `nav2_straightline_planner::StraightLine` is loaded dynamically as `nav2_core::GlobalPlanner` which is our base class.

1. To export the planner, we need to provide two lines

```
#include "pluginlib/class_list_macros.hpp"
PLUGINLIB_EXPORT_CLASS(nav2_straightline_planner::StraightLine, nav2_
↪core::GlobalPlanner)
```

Note that it requires pluginlib to export out plugin's class. Pluginlib would provide as macro `PLUGINLIB_EXPORT_CLASS` which does all the work of exporting.

It is good practice to place these lines at the end of the file, but technically, you can also write at the top.

2. Next step would be to create plugin's description file in the root directory of the package. For example, `global_planner_plugin.xml` file in our tutorial package. This file contains the following information

- `library path`: Plugin's library name and its location.
- `class name`: Name of the class.
- `class type`: Type of class.
- `base class`: Name of the base class.
- `description`: Description of the plugin.

```
<library path="nav2_straightline_planner_plugin">
  <class name="nav2_straightline_planner/StraightLine" type="nav2_straightline_
↪planner::StraightLine" base_class_type="nav2_core::GlobalPlanner">
```

(continues on next page)

(continued from previous page)

```
    <description>This is an example plugin which produces straight path.</description>
  </class>
</library>
```

3. Next step would be to export plugin using `CMakeLists.txt` by using cmake function `pluginlib_export_plugin_description_file()`. This function installs plugin description file to `share` directory and sets ament indexes to make it discoverable.

```
pluginlib_export_plugin_description_file(nav2_core global_planner_plugin.xml)
```

4. Plugin description file should also be added to `package.xml`

```
<export>
  <build_type>ament_cmake</build_type>
  <nav2_core plugin="${prefix}/global_planner_plugin.xml" />
</export>
```

5. Compile and it should be registered. Next, we'll use this plugin.

3- Pass the plugin name through params file

To enable the plugin, we need to modify the `nav2_params.yaml` file as below to replace following params

Note: For Galactic or later, `plugin_names` and `plugin_types` have been replaced with a single `plugins` string vector for plugin names. The types are now defined in the `plugin_name` namespace in the `plugin:` field (e.g. `plugin: MyPlugin::Plugin`). Inline comments in the code blocks will help guide you through this.

```
planner_server:
  ros__parameters:
    plugins: ["GridBased"]
    use_sim_time: True
    GridBased:
      plugin: "nav2_navfn_planner/NavfnPlanner" # For Foxy and later
      tolerance: 2.0
      use_astar: false
      allow_unknown: true
```

with

```
planner_server:
  ros__parameters:
    plugins: ["GridBased"]
    use_sim_time: True
    GridBased:
      plugin: "nav2_straightline_planner/StraightLine"
      interpolation_resolution: 0.1
```

In the above snippet, you can observe the mapping of our `nav2_straightline_planner/StraightLine` planner to its id `GridBased`. To pass plugin-specific parameters, we have used `<plugin_id>.<plugin_specific_parameter>`.

4- Run StraightLine plugin

Run Turtlebot3 simulation with enabled navigation2. Detailed instruction how to make it are written at *Getting Started*. Below is shortcut command for that:

```
$ ros2 launch nav2_bringup tb3_simulation_launch.py params_file:=/path/to/your_params_
↪file.yaml
```

Then goto RViz and click on the "2D Pose Estimate" button at the top and point to the location on map as it was described in *Getting Started*. Robot will localize on the map and then click on "Navigation2 goal" and click on the pose where you want your planner to consider a goal pose. After that planner will plan the path and robot will start moving towards the goal.

4.6.3 Writing a New Controller Plugin

- *Overview*
- *Requirements*
- *Tutorial Steps*

Overview

This tutorial shows how to create your own controller plugin.

In this tutorial, we will be implementing the pure pursuit path tracking algorithm based on this paper. It is recommended you go through it.

Note: This tutorial is based on a previously existing simplified version of the Regulated Pure Pursuit controller now in the Nav2 stack. You can find the source code matching this tutorial here.

Requirements

- ROS 2 (binary or build-from-source)
- Nav2 (Including dependencies)
- Gazebo
- Turtlebot3

Tutorial Steps

1- Create a new Controller Plugin

We will be implementing the pure pursuit controller. The annotated code in this tutorial can be found in navigation_tutorials repository as the `nav2_pure_pursuit_controller`. This package can be considered as a reference for writing your own controller plugin.

Our example plugin class `nav2_pure_pursuit_controller::PurePursuitController` inherits from the base class `nav2_core::Controller`. The base class provides a set of virtual methods to implement a controller plugin. These methods are called at runtime by the controller server to compute velocity commands. The list of methods, their descriptions, and necessity are presented in the table below:

Virtual method	Method description	Requires override?
configure()	Method is called when controller server enters on_configure state. Ideally this method should perform declarations of ROS parameters and initialization of controller's member variables. This method takes 4 input params: weak pointer to parent node, controller name, tf buffer pointer and shared pointer to costmap.	Yes
activate()	Method is called when controller server enters on_activate state. Ideally this method should implement operations which are neccessary before controller goes to an active state.	Yes
deactivate()	Method is called when controller server enters on_deactivate state. Ideally this method should implement operations which are neccessary before controller goes to an inactive state.	Yes
cleanup()	Method is called when controller server goes to on_cleanup state. Ideally this method should clean up resources which are created for the controller.	Yes
setPlan()	Method is called when the global plan is updated. Ideally this method should perform operations that transform the global plan and store it.	Yes
computeVelocityCommands()	Method is called when a new velocity command is demanded by the controller server in-order for the robot to follow the global path. This method returns a *geometry_msgs::msg::TwistStamped* which represents the velocity command for the robot to drive. This method passes 2 parameters: reference to the current robot pose and its current velocity.	Yes
setSpeedLimit()	Method is called when it is required to limit the maximum linear speed of the robot. Speed limit could be expressed in absolute value (m/s) or in percentage from maximum robot speed. Note that typically, maximum rotational speed is being limited proportionally to the change of maximum linear speed, in order to keep current robot behavior untouched.	Yes

In this tutorial, we will use the methods `PurePursuitController::configure`, `PurePursuitController::setPlan` and `PurePursuitController::computeVelocityCommands`.

In controllers, `configure()` method must set member variables from ROS parameters and perform any initialization required.

```
void PurePursuitController::configure(
  const rclcpp_lifecycle::LifecycleNode::WeakPtr & parent,
  std::string name, std::shared_ptr<tf2_ros::Buffer> tf,
  std::shared_ptr<nav2_costmap_2d::Costmap2DROS> costmap_ros)
{
  node_ = parent;
  auto node = node_.lock();

  costmap_ros_ = costmap_ros;
  tf_ = tf;
  plugin_name_ = name;
  logger_ = node->get_logger();
  clock_ = node->get_clock();

  declare_parameter_if_not_declared(
    node, plugin_name_ + ".desired_linear_vel", rclcpp::ParameterValue(
      0.2));
  declare_parameter_if_not_declared(
    node, plugin_name_ + ".lookahead_dist",
    rclcpp::ParameterValue(0.4));
  declare_parameter_if_not_declared(
    node, plugin_name_ + ".max_angular_vel", rclcpp::ParameterValue(
      1.0));
  declare_parameter_if_not_declared(
```

(continues on next page)

(continued from previous page)

```
    node, plugin_name_ + ".transform_tolerance", rclcpp::ParameterValue(
      0.1));

  node->get_parameter(plugin_name_ + ".desired_linear_vel", desired_linear_vel_);
  node->get_parameter(plugin_name_ + ".lookahead_dist", lookahead_dist_);
  node->get_parameter(plugin_name_ + ".max_angular_vel", max_angular_vel_);
  double transform_tolerance;
  node->get_parameter(plugin_name_ + ".transform_tolerance", transform_tolerance);
  transform_tolerance_ = rclcpp::Duration::from_seconds(transform_tolerance);
}
```

Here, `plugin_name_ + ".desired_linear_vel"` is fetching the ROS parameter `desired_linear_vel` which is specific to our controller. Nav2 allows loading of multiple plugins, and to keep things organized, each plugin is mapped to some ID/name. Now, if we want to retrieve the parameters for that specific plugin, we use `<mapped_name_of_plugin>.<name_of_parameter>` as done in the above snippet. For example, our example controller is mapped to the name `FollowPath` and to retrieve the `desired_linear_vel` parameter, which is specific to "FollowPath", we used `FollowPath.desired_linear_vel`. In other words, `FollowPath` is used as a namespace for plugin-specific parameters. We will see more on this when we discuss the parameters file (or params file).

The passed-in arguments are stored in member variables so that they can be used at a later stage if needed.

In `setPlan()` method, we receive the updated global path for the robot to follow. In our example, we transform the received global path into the frame of the robot and then store this transformed global path for later use.

```
void PurePursuitController::setPlan(const nav_msgs::msg::Path & path)
{
  // Transform global path into the robot's frame
  global_plan_ = transformGlobalPlan(path);
}
```

The computation for the desired velocity happens in the `computeVelocityCommands()` method. It is used to calculate the desired velocity command given the current velocity and pose. The third argument - is a pointer to the `nav2_core::GoalChecker`, that checks whether a goal has been reached. In our example, this won't be used. In the case of pure pursuit, the algorithm computes velocity commands such that the robot tries to follow the global path as closely as possible. This algorithm assumes a constant linear velocity and computes the angular velocity based on the curvature of the global path.

```
geometry_msgs::msg::TwistStamped PurePursuitController::computeVelocityCommands(
  const geometry_msgs::msg::PoseStamped & pose,
  const geometry_msgs::msg::Twist & velocity,
  nav2_core::GoalChecker * /*goal_checker*/)
{
  // Find the first pose which is at a distance greater than the specified lookahead␣
↪distance
  auto goal_pose = std::find_if(
    global_plan_.poses.begin(), global_plan_.poses.end(),
    [&](const auto & global_plan_pose) {
      return hypot(
        global_plan_pose.pose.position.x,
        global_plan_pose.pose.position.y) >= lookahead_dist_;
    })->pose;

  double linear_vel, angular_vel;

  // If the goal pose is in front of the robot then compute the velocity using the␣
↪pure pursuit algorithm
```

(continues on next page)

(continued from previous page)

```
  // else rotate with the max angular velocity until the goal pose is in front of the␣
↪robot
  if (goal_pose.position.x > 0) {

    auto curvature = 2.0 * goal_pose.position.y /
      (goal_pose.position.x * goal_pose.position.x + goal_pose.position.y * goal_pose.
↪position.y);
    linear_vel = desired_linear_vel_;
    angular_vel = desired_linear_vel_ * curvature;
  } else {
    linear_vel = 0.0;
    angular_vel = max_angular_vel_;
  }

  // Create and publish a TwistStamped message with the desired velocity
  geometry_msgs::msg::TwistStamped cmd_vel;
  cmd_vel.header.frame_id = pose.header.frame_id;
  cmd_vel.header.stamp = clock_->now();
  cmd_vel.twist.linear.x = linear_vel;
  cmd_vel.twist.angular.z = max(
    -1.0 * abs(max_angular_vel_), min(
      angular_vel, abs(
        max_angular_vel_)));

  return cmd_vel;
}
```

The remaining methods are not used, but it's mandatory to override them. As per the rules, we did override all but left them empty.

2- Exporting the controller plugin

Now that we have created our custom controller, we need to export our controller plugin so that it will be visible to the controller server. Plugins are loaded at runtime, and if they are not visible, then our controller server won't be able to load them. In ROS 2, exporting and loading plugins is handled by `pluginlib`.

Coming back to our tutorial, class `nav2_pure_pursuit_controller::PurePursuitController` is loaded dynamically as `nav2_core::Controller` which is our base class.

1. To export the controller, we need to provide two lines

```
#include "pluginlib/class_list_macros.hpp"
PLUGINLIB_EXPORT_CLASS(nav2_pure_pursuit_controller::PurePursuitController, nav2_
↪core::Controller)
```

Note that it requires pluginlib to export out the plugin's class. Pluginlib would provide as macro `PLUGINLIB_EXPORT_CLASS`, which does all the work of exporting.

It is good practice to place these lines at the end of the file, but technically, you can also write at the top.

2. The next step would be to create the plugin's description file in the root directory of the package. For example, `pure_pursuit_controller_plugin.xml` file in our tutorial package. This file contains the following information

- `library path`: Plugin's library name and its location.
- `class name`: Name of the class.

- `class type`: Type of class.
- `base class`: Name of the base class.
- `description`: Description of the plugin.

```
<library path="nav2_pure_pursuit_controller">
  <class type="nav2_pure_pursuit_controller::PurePursuitController" base_class_type=
↪"nav2_core::Controller">
    <description>
      This is pure pursuit controller
    </description>
  </class>
</library>
```

3. Next step would be to export plugin using `CMakeLists.txt` by using CMake function `pluginlib_export_plugin_description_file()`. This function installs the plugin description file to `share` directory and sets ament indexes to make it discoverable.

```
pluginlib_export_plugin_description_file(nav2_core pure_pursuit_controller_plugin.xml)
```

4. The plugin description file should also be added to `package.xml`

```
<export>
  <build_type>ament_cmake</build_type>
  <nav2_core plugin="${prefix}/pure_pursuit_controller_plugin.xml" />
</export>
```

5. Compile, and it should be registered. Next, we'll use this plugin.

3- Pass the plugin name through the params file

To enable the plugin, we need to modify the `nav2_params.yaml` file as below

```
controller_server:
  ros__parameters:
    controller_plugins: ["FollowPath"]

    FollowPath:
      plugin: "nav2_pure_pursuit_controller::PurePursuitController"
      debug_trajectory_details: True
      desired_linear_vel: 0.2
      lookahead_dist: 0.4
      max_angular_vel: 1.0
      transform_tolerance: 1.0
```

In the above snippet, you can observe the mapping of our `nav2_pure_pursuit_controller/PurePursuitController` controller to its id `FollowPath`. To pass plugin-specific parameters we have used `<plugin_id>.<plugin_specific_parameter>`.

4- Run Pure Pursuit Controller plugin

Run Turtlebot3 simulation with enabled Nav2. Detailed instructions on how to make it run are written at *Getting Started*. Below is a shortcut command for that:

```
$ ros2 launch nav2_bringup tb3_simulation_launch.py params_file:=/path/to/your_params_
↪file.yaml
```

Then goto RViz and click on the "2D Pose Estimate" button at the top and point the location on the map as it was described in *Getting Started*. The robot will localize on the map and then click on the "Nav2 goal" and click on the pose where you want your robot to navigate to. After that controller will make the robot follow the global path.

4.6.4 Writing a New Behavior Tree Plugin

- *Overview*
- *Requirements*
- *Tutorial Steps*

Overview

This tutorial shows how to create your own behavior tree (BT) plugin. The BT plugins are used as nodes in the behavior tree XML processed by the BT Navigator for navigation logic.

Requirements

- ROS 2 (binary or build-from-source)
- Nav2 (Including dependencies)
- Gazebo
- Turtlebot3

Tutorial Steps

1- Creating a new BT Plugin

We will create a simple BT plugin node to perform an action on another server. For this example, we're going to analyze the simplest behavior tree action node in the `nav2_behavior_tree` package, the `wait` node. Beyond this example of an action BT node, you can also create custom decorator, condition, and control nodes. Each node type has a unique role in the behavior tree to perform actions like planning, control the flow of the BT, check the status of a condition, or modify the output of other BT nodes.

The code in this tutorial can be found in nav2_behavior_tree package as the `wait_action` node. This action node can be considered as a reference for writing other action node plugins.

Our example plugin inherits from the base class `nav2_behavior_tree::BtActionNode`. The base class is a wrapper on the BehaviorTree.CPP `BT::ActionNodeBase` that simplifies BT action nodes that utilize ROS 2 action clients. An `BTActionNode` is both a BT action and uses ROS 2 action network interfaces for calling a remote server to do some work.

When working with other types of BT nodes (e.g. decorator, control, condition) use the corresponding BT node, `BT::DecoratorNode`, `BT::ControlNode`, or `BT::ConditionNode`. For BT action nodes that do *not* utilize ROS 2 action interfaces, use the `BT::ActionNodeBase` base class itself.

The `BTActionNode` class provides 5 virtual methods to use, in addition to the information provided in the constructor. Let's learn more about the methods needed to write a BT action plugin.

method	Method description	Required?
Constructor	Constructor to indicate the corresponding XML tag name to that matches the plugin, the name of the action server to call using the plugin, and any BehaviorTree.CPP special configurations required.	Yes
providedPorts()	A function to define the input and output ports a BT node may have. These are analogous to parameters that are defined in the BT XML by hardcoded values or by the value of output ports of other nodes.	Yes
on_tick()	Method is called when this BT node is ticked by the behavior tree while executing. This should be used to get dynamic updates like new blackboard values, input ports, or parameters. May also reset state for the action.	No
on_wait_for_result()	Method is called when the behavior tree node is waiting for a result from the ROS 2 action server it called. This could be used to check for updates to preempt the current task, check for a timeout, or anything to compute while waiting for the action to complete.	No
on_success()	Method is called when the ROS 2 action server returns a successful result. Returns the value the BT node will report back to the tree.	No
on_aborted()	Method is called when the ROS 2 action server returns an aborted result. Returns the value the BT node will report back to the tree.	No
on_cancelled()	Method is called when the ROS 2 action server returns a cancelled result. Returns the value the BT node will report back to the tree.	No

For this tutorial, we will only be using the `on_tick()` method.

In the constructor, we need to get any non-variable parameters that apply to the behavior tree node. In this example, we need to get the value of the duration to sleep from the input port of the behavior tree XML.

```
WaitAction::WaitAction(
  const std::string & xml_tag_name,
  const std::string & action_name,
  const BT::NodeConfiguration & conf)
: BtActionNode<nav2_msgs::action::Wait>(xml_tag_name, action_name, conf)
{
  int duration;
  getInput("wait_duration", duration);
  if (duration <= 0) {
    RCLCPP_WARN(
      node_->get_logger(), "Wait duration is negative or zero "
      "(%i). Setting to positive.", duration);
    duration *= -1;
  }

  goal_.time.sec = duration;
}
```

Here, we give the input of the `xml_tag_name`, which tells the BT node plugin the string in the XML that corresponds to this node. This will be seen later when we register this BT node as a plugin. It also takes in the string name of the action server that it will call to execute some behavior. Finally, a set of configurations that we can safely ignore for the purposes of most node plugins.

We then call the `BTActionNode` constructor. As can be seen, it's templated by the ROS 2 action type, so we give it the `nav2_msgs::action::Wait` action message type and forward our other inputs. The `BTActionNode` has the `tick()` method, which is called directly by the behavior tree when this node is called from the tree. `on_tick()` is then called before sending the action client goal.

In the body of the constructor, we get the input port `getInput` of the parameter `wait_duration`, which can be configured independently for every instance of the `wait` node in the tree. It is set in the `duration` parameter and inserted into the `goal_`. The `goal_` class variable is the goal that the ROS 2 action client will send to the action server. So in this example, we set the duration to the time we want to wait by so that the action server knows the specifics of our request.

The `providedPorts()` method gives us the opportunity to define input or output ports. Ports can be thought of as parameters that the behavior tree node has access to from the behavior tree itself. For our example, there is only a single input port, the `wait_duration` which can be set in the BT XML for each instance of the `wait` recovery. We set the type, `int`, the default `1`, the name `wait_duration`, and a description of the port `Wait time`.

```
static BT::PortsList providedPorts()
{
  return providedBasicPorts(
    {
      BT::InputPort<int>("wait_duration", 1, "Wait time")
    });
}
```

The `on_tick()` method is called when the behavior tree ticks a specific node. For the wait BT node, we simply want to notify a counter on the blackboard that an action plugin that corresponds to a recovery was ticked. This is useful to keep metrics about the number of recoveries executed during a specific navigation run. You could also log or update the `goal_` waiting duration if that is a variable input.

```
void WaitAction::on_tick()
{
  increment_recovery_count();
}
```

The remaining methods are not used and are not mandatory to override them. Only some BT node plugins will require overriding `on_wait_for_result()` to check for preemption or check a timeout. The success, aborted, and cancelled methods will default to `SUCCESS`, `FAILURE`, `SUCCESS` respectively, if not overridden.

2- Exporting the planner plugin

Now that we have created our custom BT node, we need to export our plugin so that it would be visible to the behavior tree when it loads a custom BT XML. Plugins are loaded at runtime, and if they are not visible, then our BT Navigator server won't be able to load them or use them. In BehaviorTree.CPP, exporting and loading plugins is handled by the `BT_REGISTER_NODES` macro.

```
BT_REGISTER_NODES(factory)
{
  BT::NodeBuilder builder =
    [](const std::string & name, const BT::NodeConfiguration & config)
    {
      return std::make_unique<nav2_behavior_tree::WaitAction>(name, "wait", config);
    };

  factory.registerBuilder<nav2_behavior_tree::WaitAction>("Wait", builder);
}
```

In this macro, we must create a `NodeBuilder` so that our custom action node can have a non-default constructor signature (for the action and xml names). This lambda will return a unique pointer to the behavior tree node we have created. Fill in the constructor with the relevant information, giving it the `name` and `config` given in the function arguments. Then define the ROS 2 action server's name that this BT node will call, in this case, it's the `Wait` action.

We finally give the builder to a factory to register. `Wait` given to the factory is the name in the behavior tree XML file that corresponds to this BT node plugin. An example can be seen below, where the `Wait` BT XML node specifies a non-variable input port `wait_duration` of 5 seconds.

```
<Wait wait_duration="5"/>
```

3- Add plugin library name to config

In order for the BT Navigator node to discover the plugin we've just registered, we need to list the plugin library name under the bt_navigator node in the configuration YAML file. Configuration should look similar to the one shown below. Take note of nav2_wait_action_bt_node listed under plugin_lib_names.

```
bt_navigator:
  ros__parameters:
    use_sim_time: True
    global_frame: map
    robot_base_frame: base_link
    odom_topic: /odom
    default_bt_xml_filename: "navigate_w_replanning_and_recovery.xml"
    plugin_lib_names:
    - nav2_back_up_action_bt_node # other plugin
    - nav2_wait_action_bt_node    # our new plugin
```

4- Run Your Custom plugin

Now you can use a behavior tree with your custom BT node. For example, the `navigate_w_replanning_and_recovery.xml` file is shown below.

Select this BT XML file in your specific navigation request in `NavigateToPose` or as the default behavior tree in the BT Navigator's configuration yaml file.

```
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <RecoveryNode number_of_retries="6" name="NavigateRecovery">
      <PipelineSequence name="NavigateWithReplanning">
        <RateController hz="1.0">
          <RecoveryNode number_of_retries="1" name="ComputePathToPose">
            <ComputePathToPose goal="{goal}" path="{path}" planner_id="GridBased"/>
            <ClearEntireCostmap name="ClearGlobalCostmap-Context" service_name=
↪"global_costmap/clear_entirely_global_costmap"/>
          </RecoveryNode>
        </RateController>
        <RecoveryNode number_of_retries="1" name="FollowPath">
          <FollowPath path="{path}" controller_id="FollowPath"/>
          <ClearEntireCostmap name="ClearLocalCostmap-Context" service_name="local_
↪costmap/clear_entirely_local_costmap"/>
        </RecoveryNode>
      </PipelineSequence>
      <ReactiveFallback name="RecoveryFallback">
```

(continues on next page)

(continued from previous page)

```
        <GoalUpdated/>
        <SequenceStar name="RecoveryActions">
          <ClearEntireCostmap name="ClearLocalCostmap-Subtree" service_name="local_
↪costmap/clear_entirely_local_costmap"/>
          <ClearEntireCostmap name="ClearGlobalCostmap-Subtree" service_name="global_
↪costmap/clear_entirely_global_costmap"/>
          <Spin spin_dist="1.57"/>
          <Wait wait_duration="5"/>
        </SequenceStar>
      </ReactiveFallback>
    </RecoveryNode>
  </BehaviorTree>
</root>
```

4.6.5 Writing a New Behavior Plugin

- *Overview*
- *Requirements*
- *Tutorial Steps*

Overview

This tutorial shows how to create you own Behavior Plugin. The Behavior Plugins live in the behavior server. Unlike the planner and controller servers, each behavior will host its own unique action server. The planners and controllers have the same API as they accomplish the same task. However, recoveries can be used to do a wide variety of tasks, so each behavior can have its own unique action message definition and server. This allows for massive flexibility in the behavior server enabling any behavior action imaginable that doesn't need to have other reuse.

Requirements

- ROS 2 (binary or build-from-source)
- Nav2 (Including dependencies)
- Gazebo
- Turtlebot3

Tutorial Steps

1- Creating a new Behavior Plugin

We will create a simple send sms behavior. It will use Twilio to send a message via SMS to a remote operations center. The code in this tutorial can be found in navigation_tutorials repository as `nav2_sms_behavior`. This package can be a considered as a reference for writing Behavior Plugin.

Our example plugin implements the plugin class of `nav2_core::Behavior`. However, we have a nice wrapper for actions in `nav2_behaviors`, so we use the `nav2_behaviors::TimedBehavior` base class for this application instead. This wrapper class derives from the `nav2_core` class so it can be used as a plugin, but handles the vast majority of ROS 2 action server boiler plate required.

The base class from `nav2_core` provides 4 pure virtual methods to implement a Behavior Plugin. The plugin will be used by the behavior server to host the plugins, but each plugin will provide their own unique action server interface. Let's learn more about the methods needed to write a Behavior Plugin **if you did not use the ``nav2_behaviors`` wrapper**.

Virtual method	Method description	Requires override?
configure()	Method is called at when server enters on_configure state. Ideally this method should perform declarations of ROS parameters and initialization of behavior's member variables. This method takes 4 input parameters: shared pointer to parent node, behavior name, tf buffer pointer and shared pointer to a collision checker.	Yes
activate()	Method is called when behavior server enters on_activate state. Ideally this method should implement operations which are neccessary before the behavior goes to an active state.	Yes
deactivate()	Method is called when behavior server enters on_deactivate state. Ideally this method should implement operations which are neccessary before behavior goes to an inactive state.	Yes
cleanup()	Method is called when behavior server goes to on_cleanup state. Ideally this method should clean up resources which are created for the behavior.	Yes

For the `nav2_behaviors` wrapper, which provides the ROS 2 action interface and boilerplate, we have 4 virtual methods to implement. This tutorial uses this wrapper so these are the main elements we will address.

Virtual method	Method description	Requires override?
onRun()	Method is called immediately when a new behavior action request is received. Gives the action goal to process and should start behavior initialization / process.	Yes
onCycleUpdate()	Method is called at the behavior update rate and should complete any necessary updates. An example for spinning is computing the command velocity for the current cycle, publishing it and checking for completion.	Yes
onConfigure()	Method is called when behavior server enters on_configure state. Ideally this method should implement operations which are neccessary before behavior goes to a configured state (get parameters, etc).	No
onCleanup()	Method is called when behavior server goes to on_cleanup state. Ideally this method should clean up resources which are created for the behavior.	No
onActionCompletion()	Method is called when the action has completed. Ideally, this method should populate the action result.	No

For this tutorial, we will be using methods `onRun()`, `onCycleUpdate()`, and `onConfigure()` to create the SMS behavior. `onConfigure()` will be skipped for brevity, but only declares parameters.

In recoveries, `onRun()` method must set any initial state and kick off the behavior. For the case of our call for help behavior, we can trivially compute all of our needs in this method.

```
Status SendSms::onRun(const std::shared_ptr<const Action::Goal> command)
{
  std::string response;
  bool message_success = _twilio->send_message(
```

(continues on next page)

(continued from previous page)

```
    _to_number,
    _from_number,
    command->message,
    response,
    "",
    false);

  if (!message_success) {
    RCLCPP_INFO(node_->get_logger(), "SMS send failed.");
    return ResultStatus{Status::FAILED};
  }

  RCLCPP_INFO(node_->get_logger(), "SMS sent successfully!");
  return ResultStatus{Status::SUCCEEDED};
}
```

We receive an action goal, `command`, which we want to process. `command` contains a field `message` that contains the message we want to communicate to our mothership. This is the "call for help" message that we want to send via SMS to our brothers in arms in the operations center.

We use the service Twilio to complete this task. Please create an account and get all the relavent information needed for creating the service (e.g. `account_sid`, `auth_token`, and a phone number). You can set these values as parameters in your configuration files corresponding to the `onConfigure()` parameter declarations.

We use the `_twilio` object to send our message with your account information from the configuration file. We send the message and log to screen whether or not the message was sent successfully or not. We return a `FAILED` or `SUCCEEDED` depending on this value to be returned to the action client.

`onCycleUpdate()` is trivially simple as a result of our short-running behavior. If the behavior was instead longer running like spinning, navigating to a safe area, or getting out of a bad spot and waiting for help, then this function would be checking for timeouts or computing control values. For our example, we simply return success because we already completed our mission in `onRun()`.

```
Status SendSms::onCycleUpdate()
{
  return Status::SUCCEEDED;
}
```

The remaining methods are not used and are not mandatory to override them.

2- Exporting the Behavior Plugin

Now that we have created our custom behavior, we need to export our Behavior Plugin so that it would be visible to the behavior server. Plugins are loaded at runtime and if they are not visible, then our behavior server won't be able to load it. In ROS 2, exporting and loading plugins is handled by `pluginlib`.

Coming to our tutorial, class `nav2_sms_bahavior::SendSms` is loaded dynamically as `nav2_core::Behavior` which is our base class.

1. To export the behavior, we need to provide two lines

```
#include "pluginlib/class_list_macros.hpp"
PLUGINLIB_EXPORT_CLASS(nav2_sms_bahavior::SendSms, nav2_core::Behavior)
```

Note that it requires pluginlib to export out plugin's class. Pluginlib would provide as macro `PLUGINLIB_EXPORT_CLASS` which does all the work of exporting.

It is good practice to place these lines at the end of the file but technically, you can also write at the top.

2. Next step would be to create plugin's description file in the root directory of the package. For example, `behavior_plugin.xml` file in our tutorial package. This file contains following information

- `library path`: Plugin's library name and it's location.
- `class name`: Name of the class.
- `class type`: Type of class.
- `base class`: Name of the base class.
- `description`: Description of the plugin.

```
<library path="nav2_sms_behavior_plugin">
  <class name="nav2_sms_behavior/SendSms" type="nav2_sms_behavior::SendSms" base_
↪class_type="nav2_core::Behavior">
    <description>This is an example plugin which produces an SMS text message␣
↪recovery.</description>
  </class>
</library>
```

3. Next step would be to export plugin using `CMakeLists.txt` by using cmake function `pluginlib_export_plugin_description_file()`. This function installs plugin description file to `share` directory and sets ament indexes to make it discoverable.

```
pluginlib_export_plugin_description_file(nav2_core behavior_plugin.xml)
```

4. Plugin description file should also be added to `package.xml`

```
<export>
  <build_type>ament_cmake</build_type>
  <nav2_core plugin="${prefix}/behavior_plugin.xml" />
</export>
```

5. Compile and it should be registered. Next, we'll use this plugin.

3- Pass the plugin name through params file

To enable the plugin, we need to modify the `nav2_params.yaml` file as below to replace following params

```
behavior_server:  # Humble and later
recoveries_server:  # Galactic and earlier
  ros__parameters:
    costmap_topic: local_costmap/costmap_raw
    footprint_topic: local_costmap/published_footprint
    cycle_frequency: 10.0
    behavior_plugins: ["spin", "backup", "wait"]  # Humble and later
    recovery_plugins: ["spin", "backup", "wait"]  # Galactic and earlier
    spin:
      plugin: "nav2_behaviors/Spin"
    backup:
      plugin: "nav2_behaviors/BackUp"
    wait:
      plugin: "nav2_behaviors/Wait"
    global_frame: odom
    robot_base_frame: base_link
```

(continues on next page)

(continued from previous page)

```
    transform_timeout: 0.1
    use_sim_time: true
    simulate_ahead_time: 2.0
    max_rotational_vel: 1.0
    min_rotational_vel: 0.4
    rotational_acc_lim: 3.2
```

with

```
behavior_server:  # Humble and newer
recoveries_server:  # Galactic and earlier
  ros__parameters:
    local_costmap_topic: local_costmap/costmap_raw
    local_footprint_topic: local_costmap/published_footprint
    global_costmap_topic: global_costmap/costmap_raw
    global_footprint_topic: global_costmap/published_footprint
    cycle_frequency: 10.0
    behavior_plugins: ["spin", "backup", "wait","send_sms"]  # Humble and newer
    recovery_plugins: ["spin", "backup", "wait","send_sms"]  # Galactic and earlier
    spin:
      plugin: "nav2_behaviors/Spin"
    backup:
      plugin: "nav2_behaviors/BackUp"
    wait:
      plugin: "nav2_behaviors/Wait"
    send_sms:
      plugin: "nav2_sms_behavior/SendSms"
    account_sid: ... # your sid
    auth_token: ... # your token
    from_number: ... # your number
    to_number: ... # the operations center number
    global_frame: odom
    robot_base_frame: base_link
    transform_timeout: 0.1
    use_sim_time: true
    simulate_ahead_time: 2.0
    max_rotational_vel: 1.0
    min_rotational_vel: 0.4
    rotational_acc_lim: 3.2
```

In the above snippet, you can observe that we add the SMS behavior under the `send_sms` ROS 2 action server name. We also tell the behavior server that the `send_sms` is of type `SendSms` and give it our parameters for your Twilio account.

4- Run Behavior Plugin

Run Turtlebot3 simulation with enabled Nav2. Detailed instruction how to make it are written at *Getting Started*. Below is shortcut command for that:

```
$ ros2 launch nav2_bringup tb3_simulation_launch.py params_file:=/path/to/your_params_
↪file.yaml
```

In a new terminal run:

```
$ ros2 action send_goal "send_sms" nav2_sms_behavior/action/SendSms "{message : Hello!
↪! Navigation2 World }"
```

4.6.6 Writing a New Navigator Plugin

- *Overview*
- *Requirements*
- *Tutorial Steps*

Overview

This tutorial shows how to create your own behavior-tree navigator plugin based on the `nav2_core::BehaviorTreeNavigator` base class.

In this tutorial, we will be reviewing the `Navigate to Pose` behavior-tree navigator plugin, which is the foundational navigator of Nav2 and complimentary behavior to ROS 1 Navigation. This completes point-to-point navigation. This tutorial will be reviewing the code and structure as of ROS 2 Iron. While small variations may be made over time, this should be sufficient to get started writing your own navigator if you choose as we do not expect major API changes on this system.

It may be beneficial to write your own Navigator if you have a custom action message definition you'd like to use with Navigation rather than the provided `NavigateToPose` or `NavigateThroughPoses` interfaces (e.g. doing complete coverage or containing additional constraint information). The role of the Navigators are to extract information from requests to pass to the behavior tree / blackboard, populate feedback and responses, and maintain the state of the behavior tree if relevant. The behavior tree XML will define the actual navigation logic used.

Requirements

- ROS 2 (binary or build-from-source)
- Nav2 (Including dependencies)
- Gazebo
- Turtlebot3

Tutorial Steps

1- Create a new Navigator Plugin

We will be implementing pure point-to-point navigation behavior. The code in this tutorial can be found in Nav2's BT Navigator package as the `NavigateToPoseNavigator`. This package can be considered as a reference for writing your own plugin.

Our example plugin class `nav2_bt_navigator::NavigateToPoseNavigator` inherits from the base class `nav2_core::BehaviorTreeNavigator`. The base class provides a set of virtual methods to implement a navigator plugin. These methods are called at runtime by the BT Navigator server or as a response to ROS 2 actions to process a navigation request.

Note that this class has itself a base class of `NavigatorBase`. This class is to provide a non-templated base-class for use in loading the plugins into vectors for storage and calls for basic state transition in the lifecycle node. Its members (e.g. `on_XYZ`) are implemented for you in `BehaviorTreeNavigator` and marked as `final` so they are not possible to be overrided by the user. The API that you will be implementing for your navigator are the virtual methods within `BehaviorTreeNavigator`, not `NavigatorBase`. These `on_XYZ` APIs are implemented in necessary functions in `BehaviorTreeNavigator` to handle boilerplate logic regarding the behavior tree and action server to minimize code duplication across the navigator implementations (e.g. `on_configure` will create the action server, register callbacks, populate the blackboard with some necessary basic information, and then call a user-defined `configure` function for any additional user-specific needs).

The list of methods, their descriptions, and necessity are presented in the table below:

Virtual method	Method description	Requires override?
getDefaultBTFilepath()	Method is called on initialization to retrieve the default BT filepath to use for navigation. This may be done via parameters, hardcoded logic, sentinal files, etc.	Yes
configure()	Method is called when BT navigator server enters on_configure state. This method should implement operations which are neccessary before navigator goes to an active state, such as getting parameters, setting up the blackboard, etc.	No
activate()	Method is called when BT navigator server enters on_activate state. This method should implement operations which are neccessary before navigator goes to an active state, such as create clients and subscriptions.	No
deactivate()	Method is called when BT navigator server enters on_deactivate state. This method should implement operations which are neccessary before navigator goes to an inactive state.	No
cleanup()	Method is called when BT navigator server goes to on_cleanup state. This method should clean up resources which are created for the navigator.	No
goalReceived()	Method is called when a new goal is received by the action server to process. It may accept or deny this goal with its return signature. If accepted, it may need to load the appropriate parameters from the request (e.g. which BT to use), add request parameters to the blackboard for your applications use, or reset internal state.	Yes
onLoop()	Method is called periodically while the behavior tree is looping to check statuses or more commonly to publish action feedback statuses to the client.	Yes
onPreempt()	Method is called when a new goal is requesting preemption over the existing goal currently being processed. If the new goal is viable, it should make all appropriate updates to the BT and blackboard such that this new request may immediately start being processed without hard cancelation of the initial task (e.g. preemption).	Yes
goalCompleted()	Method is called when a goal is completed to populate the action result object or do any additional checks required at the end of a task.	Yes
getName()	Method is called to get the name of this navigator type	Yes

In the Navigate to Pose Navigator, `configure()` method must determine the blackboard parameter names where the goal and paths are being stored, as these are key values for processing feedback in `onLoop` and for the different behavior tree nodes to communicate this information between themselves. Additionally and uniquely to this navigator type, we also create a client to itself and a subscription to the `goal_pose` topic such that requests from the default configurations of Rviz2 using the *Goal Pose* tool will be processed.

```
bool NavigateToPoseNavigator::configure(
  rclcpp_lifecycle::LifecycleNode::WeakPtr parent_node,
  std::shared_ptr<nav2_util::OdomSmoother> odom_smoother)
{
```

(continues on next page)

(continued from previous page)

```
  start_time_ = rclcpp::Time(0);
  auto node = parent_node.lock();

  if (!node->has_parameter("goal_blackboard_id")) {
    node->declare_parameter("goal_blackboard_id", std::string("goal"));
  }

  goal_blackboard_id_ = node->get_parameter("goal_blackboard_id").as_string();

  if (!node->has_parameter("path_blackboard_id")) {
    node->declare_parameter("path_blackboard_id", std::string("path"));
  }

  path_blackboard_id_ = node->get_parameter("path_blackboard_id").as_string();

  // Odometry smoother object for getting current speed
  odom_smoother_ = odom_smoother;

  self_client_ = rclcpp_action::create_client<ActionT>(node, getName());

  goal_sub_ = node->create_subscription<geometry_msgs::msg::PoseStamped>(
    "goal_pose",
    rclcpp::SystemDefaultsQoS(),
    std::bind(&NavigateToPoseNavigator::onGoalPoseReceived, this, std::placeholders::_
↪1));
  return true;
}
```

The values of the blackboard IDs are stored alongside the odometry smoother the BT Navigator provides for populating meaningful feedback later. Complimentary to this, the `cleanup` method will reset these resources. The activate and deactivate methods are not used in this particular navigator.

```
bool NavigateToPoseNavigator::cleanup()
{
  goal_sub_.reset();
  self_client_.reset();
  return true;
}
```

In the `getDefaultBTFilepath()`, we use a parameter `default_nav_to_pose_bt_xml` to get the default behavior tree XML file to use if none is provided by the navigation request and to initialize the BT Navigator with a behavior tree hot-loaded. If one is not provided in the parameter files, then we grab a known and reasonable default XML file in the `nav2_bt_navigator` package:

```
std::string NavigateToPoseNavigator::getDefaultBTFilepath(
  rclcpp_lifecycle::LifecycleNode::WeakPtr parent_node)
{
  std::string default_bt_xml_filename;
  auto node = parent_node.lock();

  if (!node->has_parameter("default_nav_to_pose_bt_xml")) {
    std::string pkg_share_dir =
      ament_index_cpp::get_package_share_directory("nav2_bt_navigator");
    node->declare_parameter<std::string>(
      "default_nav_to_pose_bt_xml",
      pkg_share_dir +
```

(continues on next page)

(continued from previous page)

```
      "/behavior_trees/navigate_to_pose_w_replanning_and_recovery.xml");
  }

  node->get_parameter("default_nav_to_pose_bt_xml", default_bt_xml_filename);

  return default_bt_xml_filename;
}
```

When a goal is received, we need to determine if this goal is valid and should be processed. The `goalReceived` method provides you the `goal` and a return value if it is being processed or not. This information is sent back to the action server to notify the client. In this case, we want to make sure that the goal's behavior tree is valid or else we cannot proceed. If it is valid, then we can initialize the goal pose onto the blackboard and reset some state in order to cleanly process this new request.

```
bool NavigateToPoseNavigator::goalReceived(ActionT::Goal::ConstSharedPtr goal)
{
  auto bt_xml_filename = goal->behavior_tree;

  if (!bt_action_server_->loadBehaviorTree(bt_xml_filename)) {
    RCLCPP_ERROR(
      logger_, "BT file not found: %s. Navigation canceled.",
      bt_xml_filename.c_str());
    return false;
  }

  initializeGoalPose(goal);

  return true;
}
```

Once this goal is completed, we need to populate the Action's result, if required and meaningful. In this navigator's case, it contains no result information when the navigation request was completed successfully, so this method is empty. For other navigator types, you may populate the `result` object provided.

```
void NavigateToPoseNavigator::goalCompleted(
  typename ActionT::Result::SharedPtr /*result*/,
  const nav2_behavior_tree::BtStatus /*final_bt_status*/)
{
}
```

If however a goal is preempted (e.g. a new action request comes in while an existing request is being processed), the `onPreempt()` method is called to determine if the new request is genuine and appropriate to preempt the currently processing goal. For example, it might not be wise to accept a preemption request if that request is fundamentally different in nature from an existing behavior tree task or when your existing task is of a higher priority.

```
void NavigateToPoseNavigator::onPreempt(ActionT::Goal::ConstSharedPtr goal)
{
  RCLCPP_INFO(logger_, "Received goal preemption request");

  if (goal->behavior_tree == bt_action_server_->getCurrentBTFilename() ||
    (goal->behavior_tree.empty() &&
    bt_action_server_->getCurrentBTFilename() == bt_action_server_->
↪getDefaultBTFilename()))
  {
    // if pending goal requests the same BT as the current goal, accept the pending␣
↪goal
```

(continues on next page)

(continued from previous page)

```
    // if pending goal has an empty behavior_tree field, it requests the default BT␣
↪file
    // accept the pending goal if the current goal is running the default BT file
    initializeGoalPose(bt_action_server_->acceptPendingGoal());
  } else {
    RCLCPP_WARN(
      logger_,
      "Preemption request was rejected since the requested BT XML file is not the␣
↪same "
      "as the one that the current goal is executing. Preemption with a new BT is␣
↪invalid "
      "since it would require cancellation of the previous goal instead of true␣
↪preemption."
      "\nCancel the current goal and send a new action request if you want to use a "
      "different BT XML file. For now, continuing to track the last goal until␣
↪completion.");
    bt_action_server_->terminatePendingGoal();
  }
}
```

Note that here you can also see the `initializeGoalPose` method called. This method will set the goal parameters for this navigator onto the blackboard and reset important state information to cleanly re-use a behavior tree without old state information, as shown below:

```
void
NavigateToPoseNavigator::initializeGoalPose(ActionT::Goal::ConstSharedPtr goal)
{
  RCLCPP_INFO(
    logger_, "Begin navigating from current location to (%.2f, %.2f)",
    goal->pose.pose.position.x, goal->pose.pose.position.y);

  // Reset state for new action feedback
  start_time_ = clock_->now();
  auto blackboard = bt_action_server_->getBlackboard();
  blackboard->set<int>("number_recoveries", 0);  // NOLINT

  // Update the goal pose on the blackboard
  blackboard->set<geometry_msgs::msg::PoseStamped>(goal_blackboard_id_, goal->pose);
}
```

The recovery counter and start time are both important feedback terms for a client to understand the state of the current task (e.g. if its failing, having problems, or taking exceptionally long). The setting of the goal on the blackboard is taken by the `ComputePathToPose` BT Action node to plan a new route to the goal (and then who's path is communicated to the `FollowPath` BT node via the blackboard ID previously set).

The final function implemented is `onLoop`, which is simplified below for tutorial purposes. While anything can be done in this method, which is called as the BT is looping through the tree, it is common to use this as an opportunity to populate any necessary feedback about the state of the navigation request, robot, or metadata that a client might be interested in.

```
void NavigateToPoseNavigator::onLoop()
{
  auto feedback_msg = std::make_shared<ActionT::Feedback>();

  geometry_msgs::msg::PoseStamped current_pose = ...;
  auto blackboard = bt_action_server_->getBlackboard();
```

(continues on next page)

(continued from previous page)

```
  nav_msgs::msg::Path current_path;
  blackboard->get<nav_msgs::msg::Path>(path_blackboard_id_, current_path);

  ...

  feedback_msg->distance_remaining = distance_remaining;
  feedback_msg->estimated_time_remaining = estimated_time_remaining;

  int recovery_count = 0;
  blackboard->get<int>("number_recoveries", recovery_count);
  feedback_msg->number_of_recoveries = recovery_count;
  feedback_msg->current_pose = current_pose;
  feedback_msg->navigation_time = clock_->now() - start_time_;

  bt_action_server_->publishFeedback(feedback_msg);
}
```

2- Exporting the navigator plugin

Now that we have created our custom navigator, we need to export our plugin so that it would be visible to the BT Navigator server. Plugins are loaded at runtime, and if they are not visible, then our server won't be able to load it. In ROS 2, exporting and loading plugins is handled by `pluginlib`.

Coming to our tutorial, class `nav2_bt_navigator::NavigateToPoseNavigator` is loaded dynamically as `nav2_core::NavigatorBase` which is our base class due to the subtleties previously described.

1. To export the controller, we need to provide two lines

```
#include "pluginlib/class_list_macros.hpp"
PLUGINLIB_EXPORT_CLASS(nav2_bt_navigator::NavigateToPoseNavigator, nav2_
↪core::NavigatorBase)
```

Note that it requires pluginlib to export out the plugin's class. Pluginlib would provide as macro `PLUGINLIB_EXPORT_CLASS`, which does all the work of exporting.

It is good practice to place these lines at the end of the file, but technically, you can also write at the top.

2. The next step would be to create the plugin's description file in the root directory of the package. For example, `navigator_plugin.xml` file in our tutorial package. This file contains the following information

- `library path`: Plugin's library name and it's location.
- `class name`: Name of the class.
- `class type`: Type of class.
- `base class`: Name of the base class.
- `description`: Description of the plugin.

```
<library path="nav2_bt_navigator">
  <class type="nav2_bt_navigator::NavigateToPoseNavigator" base_class_type="nav2_
↪core::NavigatorBase">
    <description>
      This is pure point-to-point navigation
    </description>
  </class>
</library>
```

3. Next step would be to export plugin using `CMakeLists.txt` by using CMake function `pluginlib_export_plugin_description_file()`. This function installs the plugin description file to `share` directory and sets ament indexes to make it discoverable.

```
pluginlib_export_plugin_description_file(nav2_core navigator_plugin.xml)
```

4. The plugin description file should also be added to `package.xml`

```
<export>
  <build_type>ament_cmake</build_type>
  <nav2_core plugin="${prefix}/navigator_plugin.xml" />
</export>
```

5. Compile, and it should be registered. Next, we'll use this plugin.

3- Pass the plugin name through the params file

To enable the plugin, we need to modify the `nav2_params.yaml` file as below

```
bt_navigator:
  ros__parameters:
    use_sim_time: true
    global_frame: map
    robot_base_frame: base_link
    transform_tolerance: 0.1
    default_nav_to_pose_bt_xml: replace/with/path/to/bt.xml # or $(find-pkg-share my_
↪package)/behavior_tree/my_nav_to_pose_bt.xml
    default_nav_through_poses_bt_xml: replace/with/path/to/bt.xml # or $(find-pkg-
↪share my_package)/behavior_tree/my_nav_through_poses_bt.xml
    goal_blackboard_id: goal
    goals_blackboard_id: goals
    path_blackboard_id: path
    navigators: ['navigate_to_pose', 'navigate_through_poses']
    navigate_to_pose:
      plugin: "nav2_bt_navigator/NavigateToPoseNavigator"
    navigate_through_poses:
      plugin: "nav2_bt_navigator/NavigateThroughPosesNavigator"
```

In the above snippet, you can observe the mapping of our `nav2_bt_navigator/NavigateToPoseNavigator` plugin to its id `navigate_to_pose`. To pass plugin-specific parameters we have used `<plugin_id>.<plugin_specific_parameter>`.

4- Run plugin

Run Turtlebot3 simulation with enabled Nav2. Detailed instructions on how to make it run are written at *Getting Started*. Below is a shortcut command for that:

```
$ ros2 launch nav2_bringup tb3_simulation_launch.py params_file:=/path/to/your_params_
↪file.yaml
```

Then goto RViz and click on the "2D Pose Estimate" button at the top and point the location on the map as it was described in *Getting Started*. The robot will localize on the map and then click on the "Nav2 goal" and click on the pose where you want your robot to navigate to. After that navigator will take over with the behavior tree XML file behavior definition provided to it.

4.7 Configuration Guide

This guide provides a process through which the user can adjust the tunable parameters to obtain the best navigation performance.

4.7.1 Behavior-Tree Navigator

Source code on Github.

The BT Navigator (Behavior Tree Navigator) module implements the NavigateToPose task interface. It is a Behavior Tree-based implementation of navigation that is intended to allow for flexibility in the navigation task and provide a way to easily specify complex robot behaviors, including recovery.

Consider checking out the *Groot - Interacting with Behavior Trees* tutorial for using Groot to visualize and modify behavior trees.

Parameters

navigators

Type	Default
vector<string>	{'navigate_to_pose', 'navigate_through_poses'}

Description New to Iron: Plugins for navigator types implementing the `nav2_core::BehaviorTreeNavigator` interface. They implement custom action servers with custom interface definitions and use that data to populate and process behavior tree navigation requests. Plugin classes are defined under the same namespace, see examples below. Defaults correspond to the `NavigateToPoseNavigator` and `NavigateThroughPosesNavigator` navigators.

default_nav_to_pose_bt_xml

Type	Default
string	N/A

Description Path to the default behavior tree XML description for `NavigateToPose`, see *Behavior Tree XML Nodes* for details on this file. Used to be `default_bt_xml_filename` pre-Galactic. You can use substitution to specify file path like `$(find-pkg-share my_package)/behavior_tree/my_nav_to_pose_bt.xml`.

default_nav_through_poses_bt_xml

Type	Default
string	N/A

Description Path to the default behavior tree XML description for `NavigateThroughPoses`, see *Behavior Tree XML Nodes* for details on this file. New to Galactic after `NavigateThroughPoses` was added. You can use substitution to specify file path like `$(find-pkg-share my_package)/behavior_tree/my_nav_through_poses_bt.xml`.

plugin_lib_names

Type	Default
vector<string>	["nav2_compute_path_to_pose_action_bt_node", "nav2_follow_path_action_bt_node", "nav2_back_up_action_bt_node", "nav2_spin_action_bt_node", "nav2_wait_action_bt_node", "nav2_clear_costmap_service_bt_node", "nav2_is_stuck_condition_bt_node", "nav2_goal_reached_condition_bt_node", "nav2_initial_pose_received_condition_bt_node", "nav2_goal_updated_condition_bt_node", "nav2_reinitialize_global_localization_service_bt_node", "nav2_rate_controller_bt_node", "nav2_distance_controller_bt_node", "nav2_speed_controller_bt_node", "nav2_recovery_node_bt_node", "nav2_pipeline_sequence_bt_node", "nav2_round_robin_node_bt_node", "nav2_transform_available_condition_bt_node", "nav2_time_expired_condition_bt_node", "nav2_distance_traveled_condition_bt_node", "nav2_single_trigger_bt_node"]

Description List of behavior tree node shared libraries.

bt_loop_duration

Type	Default
int	10

Description Duration (in milliseconds) for each iteration of BT execution.

default_server_timeout

Type	Default
int	20

Description Default timeout value (in milliseconds) while a BT action node is waiting for acknowledgement from an action server. This value will be overwritten for a BT node if the input port "server_timeout" is provided.

transform_tolerance

Type	Default	Unit
double	0.1	seconds

Description TF transform tolerance.

global_frame

Type	Default
string	map

Description Reference frame.

robot_base_frame

Type	Default
string	base_link

Description Path to behavior tree XML description.

odom_topic

Type	Default
string	odom

Description Topic on which odometry is published

goal_blackboard_id

Type	Default
string	"goal"

Description Blackboard variable to use to supply the goal to the behavior tree for `NavigateToPose`. Should match ports of BT XML file.

path_blackboard_id

Type	Default
string	"path"

Description Blackboard variable to get the path from the behavior tree for `NavigateThroughPoses` feedback. Should match port names of BT XML file.

goals_blackboard_id

Type	Default
string	"goals"

Description Blackboard variable to use to supply the goals to the behavior tree for `NavigateThroughPoses`. Should match ports of BT XML file.

use_sim_time

Type	Default
bool	false

Description Use time provided by simulation.

error_code_names

Type	Default
vector<string>	["compute_path_error_code", "follow_path_error_code"]

Description List of of error codes to compare.

Example

```
bt_navigator:
  ros__parameters:
    use_sim_time: true
    global_frame: map
    robot_base_frame: base_link
    transform_tolerance: 0.1
    default_nav_to_pose_bt_xml: replace/with/path/to/bt.xml # or $(find-pkg-share my_
↪package)/behavior_tree/my_nav_to_pose_bt.xml
    default_nav_through_poses_bt_xml: replace/with/path/to/bt.xml # or $(find-pkg-
↪share my_package)/behavior_tree/my_nav_through_poses_bt.xml
    goal_blackboard_id: goal
    goals_blackboard_id: goals
    path_blackboard_id: path
    navigators: ['navigate_to_pose', 'navigate_through_poses']
```

(continues on next page)

(continued from previous page)

```
    navigate_to_pose:
      plugin: "nav2_bt_navigator/NavigateToPoseNavigator"
    navigate_through_poses:
      plugin: "nav2_bt_navigator/NavigateThroughPosesNavigator"
    plugin_lib_names:
      - nav2_compute_path_to_pose_action_bt_node
      - nav2_follow_path_action_bt_node
      - nav2_back_up_action_bt_node
      - nav2_spin_action_bt_node
      - nav2_wait_action_bt_node
      - nav2_clear_costmap_service_bt_node
      - nav2_is_stuck_condition_bt_node
      - nav2_goal_reached_condition_bt_node
      - nav2_initial_pose_received_condition_bt_node
      - nav2_goal_updated_condition_bt_node
      - nav2_reinitialize_global_localization_service_bt_node
      - nav2_rate_controller_bt_node
      - nav2_distance_controller_bt_node
      - nav2_speed_controller_bt_node
      - nav2_recovery_node_bt_node
      - nav2_pipeline_sequence_bt_node
      - nav2_round_robin_node_bt_node
      - nav2_transform_available_condition_bt_node
      - nav2_time_expired_condition_bt_node
      - nav2_distance_traveled_condition_bt_node
      - nav2_single_trigger_bt_node
    error_code_names:
      - compute_path_error_code
      - follow_path_error_code
      # - smoother_error_code, navigate_to_pose_error_code, navigate_through_poses_
↪error_code, etc
```

4.7.2 Behavior Tree XML Nodes

The nav2_behavior_tree package provides several navigation-specific nodes that are pre-registered and can be included in Behavior Trees.

Check this introduction to learn how behavior trees work and the difference between actions, conditions, controls and decorators.

Consider checking out the *Groot - Interacting with Behavior Trees* tutorial for using Groot to visualize and modify behavior trees.

Action Plugins

Wait

Invokes the Wait ROS 2 action server, which is implemented by the nav2_behaviors module. This action is used in nav2 Behavior Trees as a recovery behavior.

Input Ports

wait_duration

Type	Default
double	1.0

Description Wait time (s).

server_name

Type	Default
string	N/A

Description Action server name.

server_timeout

Type	Default
double	10

Description Action server timeout (ms).

Example

```
<Wait wait_duration="1.0" server_name="wait_server" server_timeout="10"/>
```

Spin

Invokes the Spin ROS 2 action server, which is implemented by the nav2_behaviors module. It performs an in-place rotation by a given angle. This action is used in nav2 Behavior Trees as a recovery behavior.

Input Ports

spin_dist

Type	Default
double	1.57

Description Spin distance (radians).

time_allowance

Type	Default
double	10.0

Description Time to envoke behavior for, if exceeds considers it a stuck condition or failure case (seconds).

server_name

Type	Default
string	N/A

Description Action server name.

server_timeout

Type	Default
double	10

Description Action server timeout (ms).

is_recovery

Type	Default
bool	True

Description True if the action is being used as a recovery.

Output Ports

error_code_id

Type	Default
uint16	N/A

Description Spin error code. See `Spin` action message for the enumerated set of error codes.

Example

```
<Spin spin_dist="1.57" server_name="spin" server_timeout="10" is_recovery="true"␣
↪error_code_id="{spin_error_code}"/>
```

BackUp

Invokes the BackUp ROS 2 action server, which causes the robot to back up by a specific displacement. It performs an linear translation by a given distance. This is used in nav2 Behavior Trees as a recovery behavior. The nav2_behaviors module implements the BackUp action server.

Input Ports

backup_dist

Type	Default
double	-0.15

Description Total distance to backup (m).

backup_speed

Type	Default
double	0.025

Description Backup speed (m/s).

time_allowance

Type	Default
double	10.0

Description Time to envoke behavior for, if exceeds considers it a stuck condition or failure case (seconds).

server_name

Type	Default
string	N/A

Description Action server name.

server_timeout

Type	Default
double	10

Description Action server timeout (ms).

Output Ports

error_code_id

Type	Default
uint16	N/A

Description Backup error code. See `BackUp` action message for the enumerated set of error codes.

Example

```
<BackUp backup_dist="-0.2" backup_speed="0.05" server_name="backup_server" server_
↪timeout="10" error_code_id="{backup_error_code}"/>
```

DriveOnHeading

Invokes the DriveOnHeading ROS 2 action server, which causes the robot to drive on the current heading by a specific displacement. It performs a linear translation by a given distance. The nav2_behaviors module implements the DriveOnHeading action server.

Input Ports

dist_to_travel

Type	Default
double	0.15

Description Distance to travel (m).

speed

Type	Default
double	0.025

Description Speed at which to travel (m/s).

time_allowance

Type	Default
double	10.0

Description Time to envoke behavior for, if exceeds considers it a stuck condition or failure case (seconds).

server_name

Type	Default
string	N/A

Description Action server name.

server_timeout

Type	Default
double	10

Description Action server timeout (ms).

Output Ports

error_code_id

Type	Default
uint16	N/A

Description Drive on heading error code. See `DriveOnHeading` action message for the enumerated set of error codes.

Example

```
<DriveOnHeading dist_to_travel="0.2" speed="0.05" server_name="backup_server" server_
↪timeout="10" error_code_id="{drive_on_heading_error_code}"/>
```

AssistedTeleop

Invokes the AssistedTeleop ROS 2 action server, which filters teleop twist commands to prevent collisions. This is used in nav2 Behavior Trees as a recovery behavior or a regular behavior. The nav2_behaviors module implements the AssistedTeleop action server.

Input Ports

is_recovery

Type	Default
double	false

Description If true increment the recovery counter.

time_allowance

Type	Default
double	10.0

Description Time to envoke behavior for, if exceeds considers it a stuck condition or failure case (seconds).

server_name

Type	Default
string	N/A

Description Action server name.

server_timeout

Type	Default
double	10

Description Action server timeout (ms).

error_code_id

Type	Default
uint16	N/A

Description Assisted teleop error code. See `AssistedTeleop` action message for the enumerated set of error codes.

Example

```
<AssistedTeleop is_recovery="false" server_name="assisted_teleop_server" server_
↪timeout="10" error_code_id="{assisted_teleop_error_code}"/>
```

ComputePathToPose

Invokes the ComputePathToPose ROS 2 action server, which is implemented by the nav2_planner module. The server address can be remapped using the `server_name` input port.

Input Ports

start

Type	Default
geometry_msgs::msg::PoseStamped	N/A

Description Start pose. Optional. Only used if not left empty. Takes in a blackboard variable, e.g. "{start}".

goal

Type	Default
geometry_msgs::msg::PoseStamped	N/A

Description Goal pose. Takes in a blackboard variable, e.g. "{goal}".

planner_id

Type	Default
string	N/A

Description Mapped name to the planner plugin type to use, e.g. GridBased.

server_name

Type	Default
string	N/A

Description Action server name.

server_timeout

Type	Default
double	10

Description Action server timeout (ms).

Output Ports

path

Type	Default
nav_msgs::msg::Path	N/A

Description Path created by action server. Takes in a blackboard variable, e.g. “{path}”.

error_code_id

Type	Default
uint16	N/A

Description Compute path to pose error code. See `ComputePathToPose` action message for the enumerated set of error codes.

Example

```
<ComputePathToPose goal="{goal}" path="{path}" planner_id="GridBased" server_name=
↪"ComputePathToPose" server_timeout="10" error_code_id="{compute_path_error_code}"/>
```

FollowPath

Invokes the FollowPath ROS 2 action server, which is implemented by the controller plugin modules loaded. The server address can be remapped using the `server_name` input port.

Input Ports

path

Type	Default
string	N/A

Description Takes in a blackboard variable containing the path to follow, eg. “{path}”.

controller_id

Type	Default
string	N/A

Description Mapped name of the controller plugin type to use, e.g. FollowPath.

goal_checker_id

Type	Default
string	N/A

Description Mapped name of the goal checker plugin type to use, e.g. SimpleGoalChecker.

server_name

Type	Default
string	N/A

Description Action server name.

server_timeout

Type	Default
double	10

Description Action server timeout (ms).

Output Ports

error_code_id

Type	Default
uint16	N/A

Description Follow path error code. See `FollowPath` action for the enumerated set of error code definitions.

Example

```
<FollowPath path="{path}" controller_id="FollowPath" goal_checker_id="precise_goal_
↪checker" server_name="FollowPath" server_timeout="10" error_code_id="{follow_path_
↪error_code}"/>
```

NavigateToPose

Invokes the NavigateToPose ROS 2 action server, which is implemented by the bt_navigator module.

Input Ports

goal

Type	Default
PoseStamped	N/A

Description Takes in a blackboard variable containing the goal, eg. "{goal}".

server_name

Type	Default
string	N/A

Description Action server name.

server_timeout

Type	Default
double	10

Description Action server timeout (ms).

behavior_tree

Type	Default
string	N/A

Description Behavior tree absolute path. If none is specified, NavigateToPose action server uses a default behavior tree.

Output Ports

error_code_id

Type	Default
uint16	N/A

Description The lowest error code in the list of the *error_code_names* parameter.

Example

```
<NavigateToPose goal="{goal}" server_name="NavigateToPose" server_timeout="10" error_
↪code_id="{navigate_to_pose_error_code}"
                behavior_tree="<some-path>/behavior_trees/navigate_through_poses_w_
↪replanning_and_recovery.xml"/>
```

ClearEntireCostmap

Action to call a costmap clearing server.

Input Ports

service_name

Type	Default
string	N/A

Description costmap service name responsible for clearing the costmap.

server_timeout

Type	Default
double	10

Description Action server timeout (ms).

Example

```
<ClearEntireCostmap name="ClearLocalCostmap-Subtree" service_name="local_costmap/
↪clear_entirely_local_costmap"/>
```

ClearCostmapExceptRegion

Action to call a costmap clearing except region server.

Input Ports

reset_distance

Type	Default
double	1

Description side size of the square area centered on the robot that will not be cleared on the costmap (all the rest of the costmap will)

service_name

Type	Default
string	N/A

Description costmap service name responsible for clearing the costmap.

server_timeout

Type	Default
double	10

Description Action server timeout (ms).

Example

```
<ClearCostmapExceptRegion name="ClearLocalCostmap-Subtree" service_name="local_
↪costmap/clear_except_local_costmap"/>
```

ClearCostmapAroundRobot

Action to call a costmap clearing around robot server.

Input Ports

reset_distance

Type	Default
double	1

Description side size of the square area centered on the robot that will be cleared on the costmap (the rest of the costmap won't)

service_name

Type	Default
string	N/A

Description costmap service name responsible for clearing the costmap.

server_timeout

Type	Default
double	10

Description Action server timeout (ms).

Example

```
<ClearCostmapAroundRobot name="ClearLocalCostmap-Subtree" service_name="local_costmap/
↪clear_around_local_costmap"/>
```

ReinitializeGlobalLocalization

Used to trigger global relocalization using AMCL in case of severe delocalization or kidnapped robot problem.

Input Ports

service_name

Type	Default
string	N/A

Description Service name.

server_timeout

Type	Default
double	10

Description Server timeout (ms).

Example

```
<ReinitializeGlobalLocalization service_name="reinitialize_global_localization"/>
```

TruncatePath

A custom control node, which modifies a path making it shorter. It removes parts of the path closer than a distance to the goal pose. The resulting last pose of the path orientates the robot to the original goal pose.

Input Ports

input_path

Type	Default
nav_msgs/Path	N/A

Description The original path to be truncated.

distance

Type	Default
double	1.0

Description The distance to the original goal for truncating the path.

Ouput Ports

output_path

Type	Default
nav_msgs/Path	N/A

Description The resulting truncated path.

Example

```
<TruncatePath distance="1.0" input_path="{path}" output_path="{truncated_path}"/>
```

TruncatePathLocal

A custom control node, which modifies a path making it shorter. It removes parts of the path which are more distant than specified forward/backward distance around robot

Input Ports

input_path

Type	Default
nav_msgs/Path	N/A

Description The original path to be truncated.

distance_forward

Type	Default
double	8.0

Description The trimming distance in forward direction.

distance_backward

Type	Default
double	4.0

Description The trimming distance in backward direction.

robot_frame

Type	Default
string	"base_link"

Description Robot base frame id.

transform_tolerance

Type	Default
double	0.2

Description Robot pose lookup tolerance.

pose

Type	Default
geometry_msgs/PoseStamped	N/A

Description Manually specified pose to be used alternatively to current robot pose.

angular_distance_weight

Type	Default
double	0.0

Description Weight of angular distance relative to positional distance when finding which path pose is closest to robot. Not applicable on paths without orientations assigned.

max_robot_pose_search_dist

Type	Default
double	infinity

Description Maximum forward integrated distance along the path (starting from the last detected pose) to bound the search for the closest pose to the robot. When set to infinity (default), whole path is searched every time.

Ouput Ports

output_path

Type	Default
nav_msgs/Path	N/A

Description The resulting truncated path.

Example

```
<TruncatePathLocal input_path="{path}" output_path="{path_local}" distance_forward="3.
↪5" distance_backward="2.0" robot_frame="base_link"/>
```

PlannerSelector

It is used to select the planner that will be used by the planner server. It subscribes to the `planner_selector` topic to receive command messages with the name of the planner to be used. It is commonly used before of the ComputePathToPoseAction. The `selected_planner` output port is passed to `planner_id` input port of the ComputePathToPoseAction. If none is provided on the topic, the `default_planner` is used.

Any publisher to this topic needs to be configured with some QoS defined as `reliable` and `transient local`.

Input Ports

topic_name

Type	Default
string	planner_selector

Description The name of the topic used to received select command messages. This is used to support multiple PlannerSelector nodes.

default_planner

Type	Default
string	N/A

Description The default value for the selected planner if no message is received from the input topic.

Output Ports

selected_planner

Type	Default
string	N/A

Description The output selected planner id. This selected_planner string is usually passed to the ComputePathToPose behavior via the planner_id input port.

Example

```
<PlannerSelector selected_planner="{selected_planner}" default_planner="GridBased"␣
↪topic_name="planner_selector"/>
```

ControllerSelector

It is used to select the Controller that will be used by the Controller server. It subscribes to the `controller_selector` topic to receive command messages with the name of the Controller to be used. It is commonly used before of the FollowPathAction. The `selected_controller` output port is passed to `controller_id` input port of the FollowPathAction. If none is provided on the topic, the `default_controller` is used.

Any publisher to this topic needs to be configured with some QoS defined as `reliable` and `transient local`.

Input Ports

topic_name

Type	Default
string	controller_selector

Description The name of the topic used to received select command messages. This is used to support multiple ControllerSelector nodes.

default_controller

Type	Default
string	N/A

Description The default value for the selected Controller if no message is received from the input topic.

Output Ports

selected_controller

Type	Default
string	N/A

Description The output selected Controller id. This selected_controller string is usually passed to the FollowPath behavior via the controller_id input port.

Example

```
<ControllerSelector selected_controller="{selected_controller}" default_controller=
↪"FollowPath" topic_name="controller_selector"/>
```

SmootherSelector

It is used to select the Smoother that will be used by the Smoother server. It subscribes to the `smoother_selector` topic to receive command messages with the name of the Smoother to be used. It is commonly used before of the FollowPathAction. If none is provided on the topic, the `default_smoother` is used.

Any publisher to this topic needs to be configured with some QoS defined as `reliable` and `transient local`.

Input Ports

topic_name

Type	Default
string	smoother_selector

Description The name of the topic used to received select command messages. This is used to support multiple SmootherSelector nodes.

default_smoother

Type	Default
string	N/A

Description The default value for the selected Smoother if no message is received from the input topic.

Output Ports

selected_smoother

Type	Default
string	N/A

Description The output selected Smoother id.

Example

```
<SmootherSelector selected_smoother="{selected_smoother}" default_smoother=
↪"SimpleSmoother" topic_name="smoother_selector"/>
```

GoalCheckerSelector

It is used to select the GoalChecker that will be used by the goal_checker server. It subscribes to the `goal_checker_selector` topic to receive command messages with the name of the GoalChecker to be used. It is commonly used before of the FollowPathAction. The `selected_goal_checker` output port is passed to `goal_checker_id` input port of the FollowPathAction. If none is provided on the topic, the `default_goal_checker` is used.

Any publisher to this topic needs to be configured with some QoS defined as `reliable` and `transient local`.

Input Ports

topic_name

Type	Default
string	goal_checker_selector

Description The name of the topic used to received select command messages. This is used to support multiple GoalCheckerSelector nodes.

default_goal_checker

Type	Default
string	N/A

Description The default value for the selected GoalChecker if no message is received from the input topic.

Output Ports

selected_goal_checker

Type	Default
string	N/A

Description The output selected GoalChecker id. This selected_goal_checker string is usually passed to the FollowPath behavior via the goal_checker_id input port.

Example

```
<GoalCheckerSelector selected_goal_checker="{selected_goal_checker}" default_goal_
↪checker="precise_goal_checker" topic_name="goal_checker_selector"/>
```

NavigateThroughPoses

Invokes the NavigateThroughPoses ROS 2 action server, which is implemented by the bt_navigator module.

Input Ports

goals

Type	Default
vector<geometry_msgs::msg::PoseStamped>	N/A

Description Goal poses. Takes in a blackboard variable, e.g. "{goals}".

server_name

Type	Default
string	N/A

Description Action server name.

server_timeout

Type	Default
double	10

Description Action server timeout (ms).

behavior_tree

Type	Default
string	N/A

Description Behavior tree absolute path. If none is specified, NavigateThroughPoses action server uses a default behavior tree.

Output Ports

error_code_id

Type	Default
uint16	N/A

Description The lowest error code in the list of the *error_code_names* parameter.

Example

```
<NavigateThroughPoses goals="{goals}" server_name="NavigateThroughPoses" server_
↪timeout="10" error_code_id="{navigate_through_poses_error_code}"
                     behavior_tree="<some-path>/behavior_trees/navigate_through_
↪poses_w_replanning_and_recovery.xml"/>
```

ComputePathThroughPoses

Invokes the ComputePathThroughPoses ROS 2 action server, which is implemented by the nav2_planner module. The server address can be remapped using the `server_name` input port.

Input Ports

start

Type	Default
geometry_msgs::msg::PoseStamped	N/A

Description Start pose. Optional. Only used if not left empty. Takes in a blackboard variable, e.g. "{start}".

goals

Type	Default
vector<geometry_msgs::msg::PoseStamped>	N/A

Description Goal poses. Takes in a blackboard variable, e.g. "{goals}".

planner_id

Type	Default
string	N/A

Description Mapped name to the planner plugin type to use, e.g. GridBased.

server_name

Type	Default
string	N/A

Description Action server name.

server_timeout

Type	Default
double	10

Description Action server timeout (ms).

Output Ports

path

Type	Default
nav_msgs::msg::Path	N/A

Description Path created by action server. Takes in a blackboard variable, e.g. "{path}".

error_code_id

Type	Default
uint16	N/A

Description Compute path through poses error code. See `ComputePathThroughPoses` action message for the enumerated set of error codes.

Example

```
<ComputePathThroughPoses goals="{goals}" path="{path}" planner_id="GridBased" server_
↪name="ComputePathThroughPoses" server_timeout="10" error_code_id="{compute_path_
↪error_code}"/>
```

RemovePassedGoals

Looks over the input port `goals` and removes any point that the robot is in close proximity to or has recently passed. This is used to cull goal points that have been passed from `ComputePathToPoses` to enable replanning to only the current task goals.

Input Ports

radius

Type	Default
double	0.5

Description The radius (m) in proximity to the viapoint for the BT node to remove from the list as having passed.

global_frame

Type	Default
string	"map"

Description Reference frame.

robot_base_frame

Type	Default
string	"base_link"

Description Robot base frame.

input_goals

Type	Default
geometry_msgs::msg::PoseStamped	N/A

Description A vector of goals to check if it passed any in the current iteration.

Output Ports

output_goals

Type	Default
geometry_msgs::msg::PoseStamped	N/A

Description A vector of goals with goals removed in proximity to the robot

Example

```
<RemovePassedGoals radius="0.6" input_goals="{goals}" output_goals="{goals}"/>
```

CancelControl

Used to cancel the goals given to the controllers' action server. The server address can be remapped using the `server_name` input port.

Input Ports

service_name

Type	Default
string	N/A

Description Service name.

server_timeout

Type	Default
double	10

Description Server timeout (ms).

Example

```
<CancelControl server_name="FollowPath" server_timeout="10"/>
```

CancelBackUp

Used to cancel the backup action that is part of the behavior server. The server address can be remapped using the `server_name` input port.

Input Ports

service_name

Type	Default
string	N/A

Description Service name, if not using default of `backup` due to remapping.

server_timeout

Type	Default
double	10

Description Server timeout (ms).

Example

```
<CancelBackUp server_name="BackUp" server_timeout="10"/>
```

CancelSpin

Used to cancel the spin action that is part of the behavior server. The server address can be remapped using the `server_name` input port.

Input Ports

service_name

Type	Default
string	N/A

Description Service name, if not using default of `spin` due to remapping.

server_timeout

Type	Default
double	10

Description Server timeout (ms).

Example

```
<CancelSpin server_name="Spin" server_timeout="10"/>
```

CancelWait

Used to cancel the wait action that is part of the behavior server. The server address can be remapped using the `server_name` input port.

Input Ports

service_name

Type	Default
string	N/A

Description Service name, if not using default of `wait` due to remapping.

server_timeout

Type	Default
double	10

Description Server timeout (ms).

Example

```
<CancelWait server_name="Wait" server_timeout="10"/>
```

CancelDriveOnHeading

Used to cancel the drive on heading action that is part of the behavior server. The server address can be remapped using the `server_name` input port.

Input Ports

service_name

Type	Default
string	N/A

Description Service name, if not using default of `drive_on_heading` due to remapping.

server_timeout

Type	Default
double	10

Description Server timeout (ms).

Example

```
<CancelDriveOnHeading server_name="drive_on_heading" server_timeout="10"/>
```

CancelAssistedTeleop

Used to cancel the AssistedTeleop action that is part of the behavior server. The server address can be remapped using the `server_name` input port.

Input Ports

service_name

Type	Default
string	N/A

Description Service name, if not using default of `assisted_teleop` due to remapping.

server_timeout

Type	Default
double	10

Description Server timeout (ms).

Example

```
<CancelAssistedTeleop server_name="assisted_teleop" server_timeout="10"/>
```

SmoothPath

Invokes the SmoothPath action API in the smoother server to smooth a given path plan.

Input Ports

unsmoothed_path

Type	Default
string	N/A

Description The blackboard variable or hard-coded input path to smooth

max_smoothing_duration

Type	Default
double	3.0

Description Maximum time to smooth for (seconds)

check_for_collisions

Type	Default
bool	false

Description Whether to check the output smoothed path for collisions.

smoother_id

Type	Default
string	N/A

Description The smoother plugin ID to use for smoothing in the smoother server

Output Ports

smoothed_path

Type	Default
string	N/A

Description The output blackboard variable to assign the smoothed path to

smoothing_duration

Type	Default
double	N/A

Description The actual duration used for smoothing

was_completed

Type	Default
bool	N/A

Description Indicates if the smoothing process was completed. Will return `false` if `check_for_collisions` is set to `true` and a collision is detected.

error_code_id

Type	Default
uint16	N/A

Description Follow smoother error code. See `SmoothPath` action for the enumerated set of error code definitions.

Example

```
<SmoothPath unsmoothed_path="{path}" smoothed_path="{path}" max_smoothing_duration="3.
↪0" smoother_id="simple_smoother" check_for_collisions="false" smoothing_duration="
↪{smoothing_duration_used}" was_completed="{smoothing_completed}" error_code_id="
↪{smoothing_path_error_code}"/>
```

Condition Plugins

GoalReached

Checks the distance to the goal, if the distance to goal is less than the pre-defined threshold, the tree returns SUCCESS, otherwise it returns FAILURE.

Parameter

transform_tolerance Defined and declared in *Behavior-Tree Navigator*.

goal_reached_tol

Type	Default
double	0.25

Description Tolerance of accepting pose as the goal (m).

Example

```
bt_navigator:
  ros__parameters:
    # other bt_navigator parameters
    transform_tolerance: 0.1
    goal_reached_tol: 0.25
```

Input Ports

goal

Type	Default
string	N/A

Description Destination to check. Takes in a blackboard variable, e.g. "{goal}".

global_frame

Type	Default
string	"map"

Description Reference frame.

robot_base_frame

Type	Default
string	"base_link"

Description Robot base frame.

Example

```
<GoalReached goal="{goal}" global_frame="map" robot_base_frame="base_link"/>
```

TransformAvailable

Checks if a TF transform is available. Returns failure if it cannot be found. Once found, it will always return success. Useful for initial condition checks.

Input Ports

child

Type	Default
string	""

Description Child frame for transform.

parent

Type	Default
string	""

Description Parent frame for transform.

Example

```
<TransformAvailable parent="odom" child="base_link"/>
```

DistanceTraveled

Node that returns success when a configurable distance has been traveled.

Parameters

transform_tolerance Defined and declared in *Behavior-Tree Navigator*.

Example

```
bt_navigator:
  ros__parameters:
    # other bt_navigator parameters
    transform_tolerance: 0.1
```

Input Ports

distance

Type	Default
double	1.0

Description The distance that must travel before returning success (m).

global_frame

Type	Default
string	"map"

Description Reference frame.

robot_base_frame

Type	Default
string	"base_link"

Description Robot base frame.

Example

```
<DistanceTraveled distance="0.8" global_frame="map" robot_base_frame="base_link"/>
```

GoalUpdated

Checks if the global navigation goal has changed in the blackboard. Returns failure if the goal is the same, if it changes, it returns success.

Example

```
<GoalUpdated/>
```

GloballyUpdatedGoal

Checks if the global navigation goal has changed in the blackboard. Returns failure if the goal is the same, if it changes, it returns success.

This node differs from the GoalUpdated by retaining the state of the current goal/goals throughout each tick of the BehaviorTree such that it will update on any "global" change to the goal.

Example

```
<GlobalUpdatedGoal/>
```

InitialPoseReceived

Node that returns success when the initial pose is sent to AMCL via */initial_pose* `.

Example

```
<InitialPoseReceived/>
```

IsStuck

Determines if the robot is not progressing towards the goal. If the robot is stuck and not progressing, the condition returns SUCCESS, otherwise it returns FAILURE.

Example

```
<IsStuck/>
```

TimeExpired

Node that returns success when a time duration has passed

seconds

Type	Default
double	1.0

Description The time passed to return success (s).

Example

```
<TimeExpired seconds="1.0"/>
```

IsBatteryLow

Checks if battery is low by subscribing to a `sensor_msgs/BatteryState` topic and checking if battery percentage/voltage is below a specified minimum value. By default percentage (in range 0 to 1) is used to check for low battery. Set the `is_voltage` parameter to *true* to use voltage. Returns SUCCESS when battery percentage/voltage is lower than the specified value, FAILURE otherwise.

Example

```
<IsBatteryLow min_battery="0.5" battery_topic="/battery_status" is_voltage="false"/>
```

IsPathValid

Checks to see if the global path is valid. If there is a obstacle along the path, the condition returns FAILURE, otherwise it returns SUCCESS.

Example

```
<IsPathValid server_timeout="10" path="{path}"/>
```

PathExpiringTimer

Checks if the timer has expired. Returns success if the timer has expired, otherwise it returns failure. The timer will reset if the path gets updated.

Example

```
<PathExpiringTimer seconds="15" path="{path}"/>
```

AreErrorCodesPresent

Checks the if the provided error code matches any error code within a set.

If the active error code is a match, the node returns `SUCCESS`. Otherwise, it returns `FAILURE`.

Input Ports

error_code

Type	Default
unsigned short	N/A

Description The active error code to compare against.

error_codes_to_check

Type	Default
std::set<unsigned short>	N/A

Description The set of error codes you wish to compare against the active error code.

Example

Error codes to check are defined in another port.

```
<AreErrorCodesPresent error_code="{error_code}" error_codes_to_check="{error_codes_to_
↪check}"/>
```

Error codes to check are defined to be 101, 107 and 119.

```
<AreErrorCodesPresent error_code="{error_code}" error_codes_to_check="101,107,119"/>
```

WouldAControllerRecoveryHelp

Checks if the active controller server error code is UNKNOWN, PATIENCE_EXCEEDED, FAILED_TO_MAKE_PROGRESS, or NO_VALID_CONTROL.

If the active error code is a match, the node returns `SUCCESS`. Otherwise, it returns `FAILURE`.

Input Port

error_code

Type	Default
unsigned short	N/A

Description The active error code to compare against. This should match the controller server error code.

Example

```
<WouldAControllerRecoveryHelp error_code="{follow_path_error_code}"/>
```

WouldAPlannerRecoveryHelp

Checks if the active controller server error code is UNKNOWN, NO_VALID_CONTROL, or TIMEOUT.

If the active error code is a match, the node returns `SUCCESS`. Otherwise, it returns `FAILURE`.

Input Port

error_code

Type	Default
unsigned short	N/A

Description The active error code to compare against. This should match the planner server error code.

Example

```
<WouldAPlannerRecoveryHelp error_code="{compute_path_to_pose_error_code}"/>
```

WouldASmootherRecoveryHelp

Checks if the active controller server error code is UNKNOWN, TIMEOUT, FAILED_TO_SMOOTH_PATH, or SMOOTHED_PATH_IN_COLLISION.

If the active error code is a match, the node returns `SUCCESS`. Otherwise, it returns `FAILURE`.

Input Port

error_code

Type	Default
unsigned short	N/A

Description The active error code to compare against. This should match the smoother server error code.

Example

```
<WouldASmootherRecoveryHelp error_code="{smoother_error_code}"/>
```

IsBatteryCharging

Checks if the battery is charging by subscribing to a `sensor_msgs/BatteryState` topic and checking if the power_supply_status is `POWER_SUPPLY_STATUS_CHARGING`. Returns SUCCESS in that case, FAILURE otherwise.

Example

```
<IsBatteryCharging battery_topic="/battery_status"/>
```

Control Plugins

PipelineSequence

Ticks the first child till it succeeds, then ticks the first and second children till the second one succeeds. It then ticks the first, second, and third children until the third succeeds, and so on, and so on. If at any time a child returns RUNNING, that doesn't change the behavior. If at any time a child returns FAILURE, that stops all children and returns FAILURE overall.

Example

```
<PipelineSequence>
    <!--Add tree components here--->
</PipelineSequence>
```

RoundRobin

Custom control flow node used to create a round-robbin behavior for children BT nodes.

Example

```
<RoundRobin>
    <!--Add tree components here--->
</RoundRobin>
```

RecoveryNode

The RecoveryNode is a control flow node with two children. It returns SUCCESS if and only if the first child returns SUCCESS. The second child will be executed only if the first child returns FAILURE. If the second child SUCCEEDS, then the first child will be executed again. The user can specify how many times the recovery actions should be taken before returning FAILURE. In nav2, the RecoveryNode is included in Behavior Trees to implement recovery actions upon failures.

Input Ports

number_of_retries

Type	Default
int	1

Description Number of retries.

Example

```
<RecoveryNode number_of_retries="1">
    <!--Add tree components here--->
</RecoveryNode>
```

Decorator Plugins

RateController

A node that throttles the tick rate for its child. The tick rate can be supplied to the node as a parameter. The node returns RUNNING when it is not ticking its child. Currently, in the navigation stack, the `RateController` is used to adjust the rate at which the `ComputePathToPose` and `GoalReached` nodes are ticked.

Input Ports

hz

Type	Default
double	10.0

Description Rate to throttle an action or a group of actions.

Example

```
<RateController hz="1.0">
    <!--Add tree components here--->
</RateController>
```

DistanceController

A node that controls the tick rate for its child based on the distance traveled. The distance to be traveled before replanning can be supplied to the node as a parameter. The node returns RUNNING when it is not ticking its child. Currently, in the navigation stack, the `DistanceController` is used to adjust the rate at which the `ComputePathToPose` and `GoalReached` nodes are ticked.

Input Ports

distance

Type	Default
double	1.0

Description The distance travelled to trigger an action such as planning a path (m).

global_frame

Type	Default
string	“map”

Description Reference frame.

robot_base_frame

Type	Default
string	“base_link”

Description Robot base frame.

Example

```
<DistanceController distance="0.5" global_frame="map" robot_base_frame="base_link">
  <!--Add tree components here--->
</DistanceController>
```

SpeedController

A node that controls the tick rate for its child based on current robot speed. The maximum and minimum replanning rates can be supplied to the node as parameters along with maximum and minimum speed. The node returns RUNNING when it is not ticking its child. Currently, in the navigation stack, the `SpeedController` is used to adjust the rate at which the `ComputePathToPose` and `GoalReached` nodes are ticked.

Input Ports

min_rate

Type	Default
double	0.1

Description The minimum rate at which child node can be ticked (hz).

max_rate

Type	Default
double	1.0

Description The maximum rate at which child node can be ticked (hz).

min_speed

Type	Default
double	0.0

Description The minimum robot speed below which the child node is ticked at minimum rate (m/s).

max_speed

Type	Default
double	0.5

Description The maximum robot speed above which the child node is ticked at maximum rate (m/s).

filter_duration

Type	Default
double	0.3

Description Duration (secs) over which robot velocity should be smoothed.

Example

```
<SpeedController min_rate="0.1" max_rate="1.0" min_speed="0.0" max_speed="0.5" filter_
↪duration="0.3">
  <!--Add tree components here--->
</SpeedController>
```

GoalUpdater

A custom control node, which updates the goal pose. It subscribes to a topic in which it can receive an updated goal pose to use instead of the one commanded in action. It is useful for dynamic object following tasks.

Parameters

goal_updater_topic

Type	Default
string	"goal_update"

Description The topic to receive the updated goal pose

Input Ports

input_goal

Type	Default
geometry_msgs/PoseStamped	N/A

Description The original goal pose

output_goal

Type	Default
geometry_msgs/PoseStamped	N/A

Description The resulting updated goal. If no goal received by subscription, it will be the input_goal

Example

```
<GoalUpdater input_goal="{goal}" output_goal="{updated_goal}">
  <!--Add tree components here--->
</GoalUpdater>
```

PathLongerOnApproach

This node checks if the newly generated global path is significantly larger than the old global path in the user-defined robot's goal proximity and triggers their corresponding children. This allows users to enact special behaviors before a robot attempts to execute a path significantly longer than the prior path when close to a goal (e.g. going around an dynamic obstacle that may just need a few seconds to move out of the way).

Input Ports

path

Type	Default
nav_msgs::msg::Path	N/A

Description Path created by action server. Takes in a blackboard variable, e.g. "{path}".

prox_len

Type	Default
double	3.0

Description Proximity length (m) for the path to be longer on approach.

length_factor

Type	Default
double	2.0

Description Length multiplication factor to check if the path is significantly longer.

Example

```
<PathLongerOnApproach path="{path}" prox_len="3.0" length_factor="2.0">
  <!--Add tree components here--->
</PathLongerOnApproach>
```

SingleTrigger

This node triggers its child only once and returns FAILURE for every succeeding tick.

Example

```
<SingleTrigger>
  <!--Add tree components here--->
</SingleTrigger>
```

Example

This Behavior Tree replans the global path periodically at 1 Hz and it also has recovery actions.

```xml
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <RecoveryNode number_of_retries="6" name="NavigateRecovery">
      <PipelineSequence name="NavigateWithReplanning">
        <RateController hz="1.0">
          <RecoveryNode number_of_retries="1" name="ComputePathToPose">
            <ComputePathToPose goal="{goal}" path="{path}" planner_id="GridBased"/>
            <ReactiveFallback name="ComputePathToPoseRecoveryFallback">
              <GoalUpdated/>
              <ClearEntireCostmap name="ClearGlobalCostmap-Context" service_name=
↪"global_costmap/clear_entirely_global_costmap"/>
            </ReactiveFallback>
          </RecoveryNode>
        </RateController>
        <RecoveryNode number_of_retries="1" name="FollowPath">
          <FollowPath path="{path}" controller_id="FollowPath"/>
          <ReactiveFallback name="FollowPathRecoveryFallback">
            <GoalUpdated/>
            <ClearEntireCostmap name="ClearLocalCostmap-Context" service_name="local_
↪costmap/clear_entirely_local_costmap"/>
          </ReactiveFallback>
        </RecoveryNode>
      </PipelineSequence>
      <ReactiveFallback name="RecoveryFallback">
        <GoalUpdated/>
        <RoundRobin name="RecoveryActions">
          <Sequence name="ClearingActions">
            <ClearEntireCostmap name="ClearLocalCostmap-Subtree" service_name="local_
↪costmap/clear_entirely_local_costmap"/>
            <ClearEntireCostmap name="ClearGlobalCostmap-Subtree" service_name=
↪"global_costmap/clear_entirely_global_costmap"/>
          </Sequence>
          <Spin spin_dist="1.57"/>
          <Wait wait_duration="5"/>
          <BackUp backup_dist="0.15" backup_speed="0.025"/>
        </RoundRobin>
      </ReactiveFallback>
    </RecoveryNode>
  </BehaviorTree>
</root>
```

4.7.3 Costmap 2D

Source code on Github.

The Costmap 2D package implements a 2D grid-based costmap for environmental representations and a number of sensor processing plugins. It is used in the planner and controller servers for creating the space to check for collisions or higher cost areas to negotiate around.

Costmap2D ROS Parameters

always_send_full_costmap

Type	Default
bool	False

Description Whether to send full costmap every update, rather than updates.

footprint_padding

Type	Default
double	0.01

Description Amount to pad footprint (m).

footprint

Type	Default
vector<double>	"[]"

Description Ordered set of footprint points passed in as a string, must be closed set. For example, the following defines a square base with side lengths of 0.2 meters *footprint: "[[0.1, 0.1], [0.1, -0.1], [-0.1, -0.1], [-0.1, 0.1]]"*.

global_frame

Type	Default
string	"map"

Description Reference frame.

height

Type	Default
int	5

Description Height of costmap (m).

width

Type	Default
int	5

Description Width of costmap (m).

lethal_cost_threshold

Type	Default
int	100

Description Minimum cost of an occupancy grid map to be considered a lethal obstacle.

map_topic

Type	Default
string	"map"

Description Topic of map from map_server or SLAM.

observation_sources

Type	Default
string	“”

Description List of sources of sensors as a string, to be used if not specified in plugin specific configurations. Ex. “static_layer stvl_layer”

origin_x

Type	Default
double	0.0

Description X origin of the costmap relative to width (m).

origin_y

Type	Default
double	0.0

Description Y origin of the costmap relative to height (m).

publish_frequency

Type	Default
double	1.0

Description Frequency to publish costmap to topic.

resolution

Type	Default
double	0.1

Description Resolution of 1 pixel of the costmap, in meters.

robot_base_frame

Type	Default
string	“base_link”

Description Robot base frame.

robot_radius

Type	Default
double	0.1

Description Robot radius to use, if footprint coordinates not provided.

rolling_window

Type	Default
bool	False

Description Whether costmap should roll with robot base frame.

track_unknown_space

Type	Default
bool	False

Description If false, treats unknown space as free space, else as unknown space.

transform_tolerance

Type	Default
double	0.3

Description TF transform tolerance.

trinary_costmap

Type	Default
bool	True

Description If occupancy grid map should be interpreted as only 3 values (free, occupied, unknown) or with its stored values.

unknown_cost_value

Type	Default
int	255

Description Cost of unknown space if tracking it.

update_frequency

Type	Default
double	5.0

Description Costmap update frequency.

use_maximum

Type	Default
bool	False

Description whether when combining costmaps to use the maximum cost or override.

plugins

Type	Default
vector<string>	{“static_layer”, “obstacle_layer”, “inflation_layer”}

Description List of mapped plugin names for parameter namespaces and names.

Note Each plugin namespace defined in this list needs to have a `plugin` parameter defining the type of plugin to be loaded in the namespace.

Example:

```
local_costmap:
  ros__parameters:
    plugins: ["obstacle_layer", "voxel_layer", "inflation_layer"]
    obstacle_layer:
      plugin: "nav2_costmap_2d::ObstacleLayer"
    voxel_layer:
      plugin: "nav2_costmap_2d::VoxelLayer"
    inflation_layer:
      plugin: "nav2_costmap_2d::InflationLayer"
```

filters

Type	Default
vector<string>	{}

Description List of mapped costmap filter names for parameter namespaces and names.

Note Costmap filters are also loadable plugins just as ordinary costmap layers. This separation is made to avoid plugin and filter interference and places these filters on top of the combined layered costmap. As with plugins, each costmap filter namespace defined in this list needs to have a `plugin` parameter defining the type of filter plugin to be loaded in the namespace.

Example:

```
local_costmap:
  ros__parameters:
    filters: ["keepout_filter", "speed_filter"]
    keepout_filter:
      plugin: "nav2_costmap_2d::KeepoutFilter"
    speed_filter:
      plugin: "nav2_costmap_2d::SpeedFilter"
```

Default Plugins

When the `plugins` parameter is not overridden, the following default plugins are loaded:

Namespace	Plugin
“static_layer”	“nav2_costmap_2d::StaticLayer”
“obstacle_layer”	“nav2_costmap_2d::ObstacleLayer”
“inflation_layer”	“nav2_costmap_2d::InflationLayer”

Plugin Parameters

Static Layer Parameters

`<static layer>` is the corresponding plugin name selected for this type.

`<static layer>`.enabled

Type	Default
bool	True

Description Whether it is enabled.

`<static layer>`.subscribe_to_updates

Type	Default
bool	False

Description Subscribe to static map updates after receiving first.

`<static layer>`.map_subscribe_transient_local

Type	Default
bool	True

Description QoS settings for map topic.

`<static layer>`.transform_tolerance

Type	Default
double	0.0

Description TF tolerance.

`<static layer>`.map_topic

Type	Default
string	“”

Description Map topic to subscribe to. If left empty the map topic will default to the global *map_topic* parameter in *costmap_2d_ros*.

Inflation Layer Parameters

`<inflation layer>` is the corresponding plugin name selected for this type.

`<inflation layer>`.enabled

Type	Default
bool	True

Description Whether it is enabled.

`<inflation layer>`.inflation_radius

Type	Default
double	0.55

Description Radius to inflate costmap around lethal obstacles.

`<inflation layer>`.cost_scaling_factor

Type	Default
double	10.0

Description Exponential decay factor across inflation radius.

<inflation layer>.inflate_unknown

Type	Default
bool	False

Description Whether to inflate unknown cells as if lethal.

<inflation layer>.inflate_around_unknown

Type	Default
bool	False

Description Whether to inflate unknown cells.

Obstacle Layer Parameters

`<obstacle layer>` is the corresponding plugin name selected for this type.

`<data source>` is the corresponding observation source name for that sources parameters.

<obstacle layer>.enabled

Type	Default
bool	True

Description Whether it is enabled.

<obstacle layer>.footprint_clearing_enabled

Type	Default
bool	True

Description Clear any occupied cells under robot footprint.

<obstacle layer>.max_obstacle_height

Type	Default
double	2.0

Description Maximum height to add return to occupancy grid.

<obstacle layer>.combination_method

Type	Default
int	1

Description Enum for method to add data to master costmap. Must be 0, 1 or 2, default to 1 (see below).

0 - Overwrite: Overwrite master costmap with every valid observation.

1 - Max: Sets the new value to the maximum of the master_grid's value and this layer's value. This is the default.

2 - MaxWithoutUnknownOverwrite: Sets the new value to the maximum of the master_grid's value and this layer's value. If the master value is NO_INFORMATION, it is NOT overwritten. It can be used to make sure that the static map is the dominant source of information, and prevent the robot to go through places that are not present in the static map.

`<obstacle layer>`.observation_sources

Type	Default
vector<string>	{""}

Description namespace of sources of data.

`<obstacle layer>`. `<data source>`.topic

Type	Default
string	""

Description Topic of data.

`<obstacle layer>`. `<data source>`.sensor_frame

Type	Default
string	""

Description Frame of sensor, to use if not provided by message. If empty, uses message frame_id.

`<obstacle layer>`. `<data source>`.observation_persistence

Type	Default
double	0.0

Description How long to store messages in a buffer to add to costmap before removing them (s).

`<obstacle layer>`. `<data source>`.expected_update_rate

Type	Default
double	0.0

Description Expected rate to get new data from sensor.

`<obstacle layer>`. `<data source>`.data_type

Type	Default
string	"LaserScan"

Description Data type of input, LaserScan or PointCloud2.

`<obstacle layer>`. `<data source>`.min_obstacle_height

Type	Default
double	0.0

Description Minimum height to add return to occupancy grid.

`<obstacle layer>. <data source>`.max_obstacle_height

Type	Default
double	0.0

Description Maximum height to add return to occupancy grid.

`<obstacle layer>. <data source>`.inf_is_valid

Type	Default
bool	False

Description Are infinite returns from laser scanners valid measurements to raycast.

`<obstacle layer>. <data source>`.marking

Type	Default
bool	True

Description Whether source should mark in costmap.

`<obstacle layer>. <data source>`.clearing

Type	Default
bool	False

Description Whether source should raytrace clear in costmap.

`<obstacle layer>. <data source>`.obstacle_max_range

Type	Default
double	2.5

Description Maximum range to mark obstacles in costmap.

`<obstacle layer>. <data source>`.obstacle_min_range

Type	Default
double	0.0

Description Minimum range to mark obstacles in costmap.

`<obstacle layer>. <data source>`.raytrace_max_range

Type	Default
double	3.0

Description Maximum range to raytrace clear obstacles from costmap.

`<obstacle layer>. <data source>`.raytrace_min_range

Type	Default
double	0.0

Description Minimum range to raytrace clear obstacles from costmap.

Voxel Layer Parameters

`<voxel layer>` is the corresponding plugin name selected for this type.

`<data source>` is the corresponding observation source name for that sources parameters.

`<voxel layer>`.enabled

Type	Default
bool	True

Description Whether it is enabled.

`<voxel layer>`.footprint_clearing_enabled

Type	Default
bool	True

Description Clear any occupied cells under robot footprint.

`<voxel layer>`.max_obstacle_height

Type	Default
double	2.0

Description Maximum height to add return to occupancy grid.

`<voxel layer>`.z_voxels

Type	Default
int	10

Description Number of voxels high to mark, maximum 16.

`<voxel layer>`.origin_z

Type	Default
double	0.0

Description Where to start marking voxels (m).

`<voxel layer>`.z_resolution

Type	Default
double	0.2

Description Resolution of voxels in height (m).

`<voxel layer>`.unknown_threshold

Type	Default
int	15

Description Minimum number of empty voxels in a column to mark as unknown in 2D occupancy grid.

`<voxel layer>`.mark_threshold

Type	Default
int	0

Description Minimum number of voxels in a column to mark as occupied in 2D occupancy grid.

`<voxel layer>`.combination_method

Type	Default
int	1

Description Enum for method to add data to master costmap. Must be 0, 1 or 2, default to 1 (see below).

0 - Overwrite: Overwrite master costmap with every valid observation.

1 - Max: Sets the new value to the maximum of the master_grid's value and this layer's value.
This is the default.

2 - MaxWithoutUnknownOverwrite: Sets the new value to the maximum of the master_grid's
value and this layer's value. If the master value is NO_INFORMATION, it is NOT overwritten.
It can be used to make sure that the static map is the dominant source of information, and
prevent the robot to go through places that are not present in the static map.

`<voxel layer>`.publish_voxel_map

Type	Default
bool	False

Description Whether to publish 3D voxel grid for debug, computationally expensive.

`<voxel layer>`.observation_sources

Type	Default
vector<string>	{""}

Description namespace of sources of data.

`<voxel layer>`. `<data source>`.topic

Type	Default
string	""

Description Topic of data.

`<voxel layer>`. `<data source>`.sensor_frame

Type	Default
string	""

Description Frame of sensor, to use if not provided by message. If empty, uses message frame_id.

`<voxel layer>`. `<data source>`.observation_persistence

Type	Default
double	0.0

Description How long to store messages in a buffer to add to costmap before removing them (s).

`<voxel layer>`. `<data source>`.expected_update_rate

Type	Default
double	0.0

Description Expected rate to get new data from sensor.

`<voxel layer>`. `<data source>`.data_type

Type	Default
string	"LaserScan"

Description Data type of input, LaserScan or PointCloud2.

`<voxel layer>`. `<data source>`.min_obstacle_height

Type	Default
double	0.0

Description Minimum height to add return to occupancy grid.

`<voxel layer>`. `<data source>`.max_obstacle_height

Type	Default
double	0.0

Description Maximum height to add return to occupancy grid.

`<voxel layer>`. `<data source>`.inf_is_valid

Type	Default
bool	False

Description Are infinite returns from laser scanners valid measurements to raycast.

`<voxel layer>`. `<data source>`.marking

Type	Default
bool	True

Description Whether source should mark in costmap.

`<voxel layer>`. `<data source>`.clearing

Type	Default
bool	False

Description Whether source should raytrace clear in costmap.

`<voxel layer>`. `<data source>`.obstacle_max_range

Type	Default
double	2.5

Description Maximum range to mark obstacles in costmap.

`<voxel layer>`. `<data source>`.obstacle_min_range

Type	Default
double	0.0

Description Minimum range to mark obstacles in costmap.

`<voxel layer>`. `<data source>`.raytrace_max_range

Type	Default
double	3.0

Description Maximum range to raytrace clear obstacles from costmap.

`<voxel layer>`. `<data source>`.raytrace_min_range

Type	Default
double	0.0

Description Minimum range to raytrace clear obstacles from costmap.

Range Sensor Parameters

`<range layer>` is the corresponding plugin name selected for this type.

`<range layer>`.enabled

Type	Default
bool	True

Description Whether it is enabled.

`<range layer>`.topics

Type Default	
vector<string>	[“”]

Description Range topics to subscribe to.

`<range layer>`.phi

Type	Default
double	1.2

Description Phi value.

`<range layer>`.inflate_cone

Type	Default
double	1.0

Description Inflate the triangular area covered by the sensor (percentage).

`<range layer>`.no_readings_timeout

Type	Default
double	0.0

Description If zero, this parameter has no effect. Otherwise if the layer does not receive sensor data for this amount of time, the layer will warn the user and the layer will be marked as not current.

`<range layer>`.clear_threshold

Type	Default
double	0.2

Description Probability below which cells are marked as free.

`<range layer>`.mark_threshold

Type	Default
double	0.8

Description Probability above which cells are marked as occupied.

`<range layer>`.clear_on_max_reading

Type	Default
bool	False

Description Whether to clear the sensor readings on max range.

`<range layer>`.input_sensor_type

Type	Default
string	ALL

Description Input sensor type is either ALL (automatic selection), VARIABLE (min range != max range), or FIXED (min range == max range).

Denoise Layer Parameters

`<denoise layer>` is the corresponding plugin name selected for this type.

`<denoise layer>`.enabled

Type	Default
bool	True

Description Whether it is enabled.

`<denoise layer>`.minimal_group_size

Type	Default
int	2

Description The minimum number of adjacent obstacles that should not be discarded as noise.

If 1 or less, all obstacles will be kept.
If 2, standalone obstacles (without neighbors in adjacent cells) will be removed.
If N, obstacles groups smaller than N will be removed.

`<denoise layer>`.group_connectivity_type

Type	Default
int	8

Description Obstacles connectivity type (is the way in which obstacles relate to their neighbors). Must be 4 or 8.

4 - adjacent obstacles are connected horizontally and vertically.
8 - adjacent obstacles are connected horizontally, vertically and diagonally.

Example

```
local_costmap:
  local_costmap:
    ros__parameters:
      ...
      plugins: ["voxel_layer", "denoise_layer", "inflation_layer"]
      ...
      denoise_layer:
        plugin: "nav2_costmap_2d::DenoiseLayer"
        enabled: true
        minimal_group_size: 2
        group_connectivity_type: 8
```

Costmap Filters Parameters

Keepout Filter Parameters

Keepout Filter - is a Costmap Filter that enforces robot to avoid keepout areas or stay on preferred lanes, by updating corresponding costmap layer using filter mask information.

Note: As Costmap Filters does not have the inflation layer applied to them (since inflation is not sensible for a speed or other non-occupation zone type), it may be beneficial to add a separate inflation layer into the vector of filters when using only a keepout zone. Some planners (e.g. Smac Feasible) will use the cost of the center point for a collision checking optimization before doing full SE2 footprint checks. Without inflation, the planner will not respect the Keepout Zone on it extremities – but will still respect it for the robot centers. If you wish to have a Keepout Zone for any part of the robot base while using a feasible planner, please enable the inflation layer.

<*filter name*>: is the corresponding plugin name selected for this type.

`<filter name>`.enabled

Type	Default
bool	True

Description Whether it is enabled.

`<filter name>`.filter_info_topic

Type	Default
string	N/A

Description Name of the incoming CostmapFilterInfo topic having filter-related information. Published by Costmap Filter Info Server along with filter mask topic. For more details about Map and Costmap Filter Info servers configuration please refer to the *Map Server / Saver* configuration page.

`<filter name>`.transform_tolerance

Type	Default
double	0.1

Description Time with which to post-date the transform that is published, to indicate that this transform is valid into the future. Used when filter mask and current costmap layer are in different frames.

Example

```
global_costmap:
  global_costmap:
    ros__parameters:
      ...
      plugins: ["static_layer", "obstacle_layer", "inflation_layer"]
      filters: ["keepout_filter"]
      ...
      keepout_filter:
        plugin: "nav2_costmap_2d::KeepoutFilter"
        enabled: True
        filter_info_topic: "/costmap_filter_info"
        transform_tolerance: 0.1
...
local_costmap:
  local_costmap:
    ros__parameters:
      ...
      plugins: ["voxel_layer", "inflation_layer"]
      filters: ["keepout_filter"]
      ...
      keepout_filter:
        plugin: "nav2_costmap_2d::KeepoutFilter"
        enabled: True
        filter_info_topic: "/costmap_filter_info"
        transform_tolerance: 0.1
```

Speed Filter Parameters

Speed Filter - is a Costmap Filter that restricting maximum velocity of robot. The areas where robot should slow down and values of maximum allowed velocities are encoded at filter mask. Filter mask published by Map Server, goes in a pair with filter info topic published by Costmap Filter Info Server. Speed Filter itself publishes a speed restricting messages which are targeted for a Controller in order to make the robot to not exceed the required velocity.

<*filter name*>: is the corresponding plugin name selected for this type.

`<filter name>`.enabled

Type	Default
bool	True

Description Whether it is enabled.

`<filter name>`.filter_info_topic

Type	Default
string	N/A

Description Name of the incoming CostmapFilterInfo topic having filter-related information. Published by Costmap Filter Info Server along with filter mask topic. For more details about Map and Costmap Filter Info servers configuration please refer to the *Map Server / Saver* configuration page.

`<filter name>`.speed_limit_topic

Type	Default
string	"speed_limit"

Description Topic to publish speed limit to. The messages have the following fields' meaning:

- `percentage`: speed limit is expressed in percentage if `true` or in absolute values in `false` case. This parameter is set depending on `type` field of `CostmapFilterInfo` message.
- `speed_limit`: non-zero values show maximum allowed speed expressed in a percent of maximum robot speed or in absolute value depending on `percentage` value. Zero value means no speed restriction (independently on `percentage`). `speed_limit` is being linearly converted from `OccupancyGrid` filter mask value as: `speed_limit = base + multiplier * mask_value`, where `base` and `multiplier` coefficients are taken from `CostmapFilterInfo` message.

  Note `speed_limit` expressed in a percent should belong to `(0.0 .. 100.0]` range.

This topic will be used by a Controller Server. Please refer to *Controller Server* configuration page to set it appropriately.

`<filter name>`.transform_tolerance

Type	Default
double	0.1

Description Time with which to post-date the transform that is published, to indicate that this transform is valid into the future. Used when filter mask and current costmap layer are in different frames.

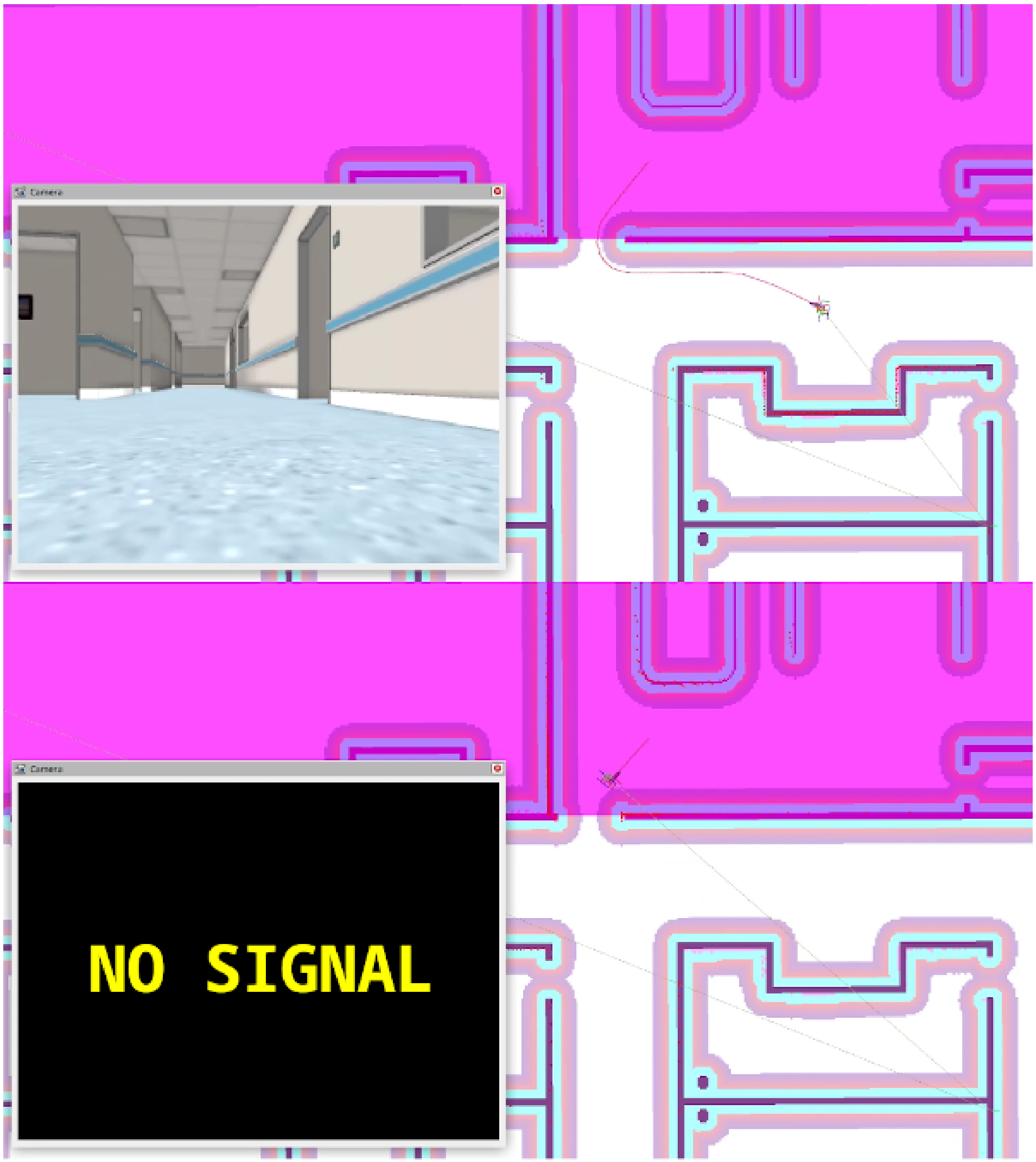
Example

```
global_costmap:
  global_costmap:
    ros__parameters:
    ...
    plugins: ["static_layer", "obstacle_layer", "inflation_layer"]
    filters: ["speed_filter"]
    ...
    speed_filter:
      plugin: "nav2_costmap_2d::SpeedFilter"
      enabled: True
      filter_info_topic: "/costmap_filter_info"
      speed_limit_topic: "/speed_limit"
      transform_tolerance: 0.1
```

Binary Filter Parameters

Binary Filter - is a Costmap Filter that publishes a boolean topic, flipping binary state when the encoded filter space value (corresponding to the filter mask point where the robot is) is higher than given threshold. It then flips back when lower or equal.

Filter space value is being calculated as: `Fv = base + multiplier * mask_value`, where `base` and `multiplier` are being published in a filter info by Costmap Filter Info Server. The example of usage are include: camera operating on/off to turn off cameras in sensitive areas, headlights switch on/off for moving indoors to outdoors, smart house light triggering, etc.



<*filter name*>: is the corresponding plugin name selected for this type.

`<filter name>`.enabled

Type	Default
bool	True

Description Whether it is enabled.

`<filter name>`.filter_info_topic

Type	Default
string	N/A

Description Name of the incoming CostmapFilterInfo topic having filter-related information. Published by Costmap Filter Info Server along with filter mask topic. For more details about Map and Costmap Filter Info servers configuration please refer to the *Map Server / Saver* configuration page.

`<filter name>`.transform_tolerance

Type	Default
double	0.1

Description Time with which to post-date the transform that is published, to indicate that this transform is valid into the future. Used when filter mask and current costmap layer are in different frames.

`<filter name>`.default_state

Type	Default
bool	false

Description Default state of binary filter.

`<filter name>`.binary_state_topic

Type	Default
string	“binary_state”

Description Topic of `std_msgs::msg::Bool` type to publish binary filter state to.

`<filter name>`.flip_threshold

Type	Default
double	50.0

Descrioption Threshold for binary state flipping. Filter values higher than this threshold, will set binary state to non-default.

Example

```
global_costmap:
  global_costmap:
    ros__parameters:
    ...
    plugins: ["static_layer", "obstacle_layer", "inflation_layer"]
    filters: ["binary_filter"]
    ...
    binary_filter:
      plugin: "nav2_costmap_2d::BinaryFilter"
      enabled: True
      filter_info_topic: "/costmap_filter_info"
      transform_tolerance: 0.1
      default_state: False
```

(continues on next page)

(continued from previous page)

```
      binary_state_topic: "/binary_state"
      flip_threshold: 50.0
```

Example

```
global_costmap:
  global_costmap:
    ros__parameters:
      footprint_padding: 0.03
      update_frequency: 1.0
      publish_frequency: 1.0
      global_frame: map
      robot_base_frame: base_link
      use_sim_time: True
      robot_radius: 0.22 # radius set and used, so no footprint points
      resolution: 0.05
      plugins: ["static_layer", "obstacle_layer", "voxel_layer", "inflation_layer"]
      obstacle_layer:
        plugin: "nav2_costmap_2d::ObstacleLayer"
        enabled: True
        observation_sources: scan
        footprint_clearing_enabled: true
        max_obstacle_height: 2.0
        combination_method: 1
        scan:
          topic: /scan
          obstacle_max_range: 2.5
          obstacle_min_range: 0.0
          raytrace_max_range: 3.0
          raytrace_min_range: 0.0
          max_obstacle_height: 2.0
          min_obstacle_height: 0.0
          clearing: True
          marking: True
          data_type: "LaserScan"
          inf_is_valid: false
      voxel_layer:
        plugin: "nav2_costmap_2d::VoxelLayer"
        enabled: True
        footprint_clearing_enabled: true
        max_obstacle_height: 2.0
        publish_voxel_map: True
        origin_z: 0.0
        z_resolution: 0.05
        z_voxels: 16
        max_obstacle_height: 2.0
        unknown_threshold: 15
        mark_threshold: 0
        observation_sources: pointcloud
        combination_method: 1
        pointcloud:  # no frame set, uses frame from message
          topic: /intel_realsense_r200_depth/points
          max_obstacle_height: 2.0
          min_obstacle_height: 0.0
          obstacle_max_range: 2.5
```

(continues on next page)

(continued from previous page)

```
          obstacle_min_range: 0.0
          raytrace_max_range: 3.0
          raytrace_min_range: 0.0
          clearing: True
          marking: True
          data_type: "PointCloud2"
      static_layer:
        plugin: "nav2_costmap_2d::StaticLayer"
        map_subscribe_transient_local: True
        enabled: true
        subscribe_to_updates: true
        transform_tolerance: 0.1
      inflation_layer:
        plugin: "nav2_costmap_2d::InflationLayer"
        enabled: true
        inflation_radius: 0.55
        cost_scaling_factor: 1.0
        inflate_unknown: false
        inflate_around_unknown: true
      always_send_full_costmap: True


local_costmap:
  local_costmap:
    ros__parameters:
      update_frequency: 5.0
      publish_frequency: 2.0
      global_frame: odom
      robot_base_frame: base_link
      use_sim_time: True
      rolling_window: true
      width: 3
      height: 3
      resolution: 0.05
```

4.7.4 Lifecycle Manager

Source code on Github.

The Lifecycle Manager module implements the method for handling the lifecycle transition states for the stack in a deterministic way. It will take in a set of ordered nodes to transition one-by-one into the configurating and activate states to run the stack. It will then bring down the stack into the finalized state in the opposite order. It will also create bond connections with the servers to ensure they are still up and transition down all nodes if any are non-responsive or crashed.

Parameters

node_names

Type	Default
vector<string>	N/A

Description Ordered list of node names to bringup through lifecycle transition.

autostart

Type	Default
bool	false

Description Whether to transition nodes to active state on startup.

bond_timeout

Type	Default
double	4.0

Description Timeout to transition down all lifecycle nodes of this manager if a server is non-responsive, in seconds. Set to `0` to deactivate. Recommended to be always larger than 0.3s for all-local node discovery. Note: if a server cleanly exits the manager will immediately be notified.

attempt_respawn_reconnection

Type	Default
bool	true

Description Whether to try to reconnect to servers that go down, presumably because respawn is set to `true` to re-create crashed nodes. While default to `true`, reconnections will not be made unless respawn is set to true in your launch files or your watchdog systems will bring up the server externally.

bond_respawn_max_duration

Type	Default
double	10.0

Description When a server crashes or becomes non-responsive, the lifecycle manager will bring down all nodes for safety. This is the duration of which the lifecycle manager will attempt to reconnect with the failed server(s) during to recover and re-activate the system. If this passes, it will stop attempts and will require a manual re-activation once the problem is manually resolved. Units: seconds.

Example

```
lifecycle_manager:
  ros__parameters:
    autostart: true
    node_names: ['controller_server', 'planner_server', 'behavior_server', 'bt_
↪navigator', 'waypoint_follower']
    bond_timeout: 4.0
    attempt_respawn_reconnection: true
    bond_respawn_max_duration: 10.0
```

4.7.5 Planner Server

Source code on Github.

The Planner Server implements the server for handling the planner requests for the stack and host a map of plugin implementations. It will take in a goal and a planner plugin name to use and call the appropriate plugin to compute a path to the goal.

Parameters

planner_plugins

Type	Default
vector<string>	['GridBased']

Description List of Mapped plugin names for parameters and processing requests.

Note Each plugin namespace defined in this list needs to have a `plugin` parameter defining the type of plugin to be loaded in the namespace.

Example:

```
planner_server:
  ros__parameters:
    planner_plugins: ["GridBased"]
    GridBased:
      plugin: "nav2_navfn_planner/NavfnPlanner"
```

expected_planner_frequency

Type	Default
double	[20.0]

Description Expected planner frequency. If the current frequency is less than the expected frequency, display the warning message.

Default Plugins

When the `planner_plugins` parameter is not overridden, the following default plugins are loaded:

Namespace	Plugin
"GridBased"	"nav2_navfn_planner/NavfnPlanner"

Example

```
planner_server:
  ros__parameters:
    expected_planner_frequency: 20.0
    planner_plugins: ['GridBased']
    GridBased:
      plugin: 'nav2_navfn_planner/NavfnPlanner'
```

4.7.6 NavFn Planner

Source code on Github.

The Navfn Planner plugin implements a wavefront Dijkstra or A* expanded planner.

`<name>` is the corresponding planner plugin ID selected for this type.

Parameters

`<name>`.tolerance

Type	Default
double	0.5

Description Tolerance in meters between requested goal pose and end of path.

`<name>`.use_astar

Type	Default
bool	False

Description Whether to use A*. If false, uses Dijkstra's expansion.

`<name>`.allow_unknown

Type	Default
bool	True

Description Whether to allow planning in unknown space.

`<name>`.use_final_approach_orientation

Type	Default
bool	false

Description If true, the last pose of the path generated by the planner will have its orientation set to the approach orientation, i.e. the orientation of the vector connecting the last two points of the path

Example

```
planner_server:
  ros__parameters:
    planner_plugins: ['GridBased']
    GridBased:
      plugin: 'nav2_navfn_planner/NavfnPlanner'
      use_astar: True
      allow_unknown: True
      tolerance: 1.0
```

4.7.7 Smac Planner

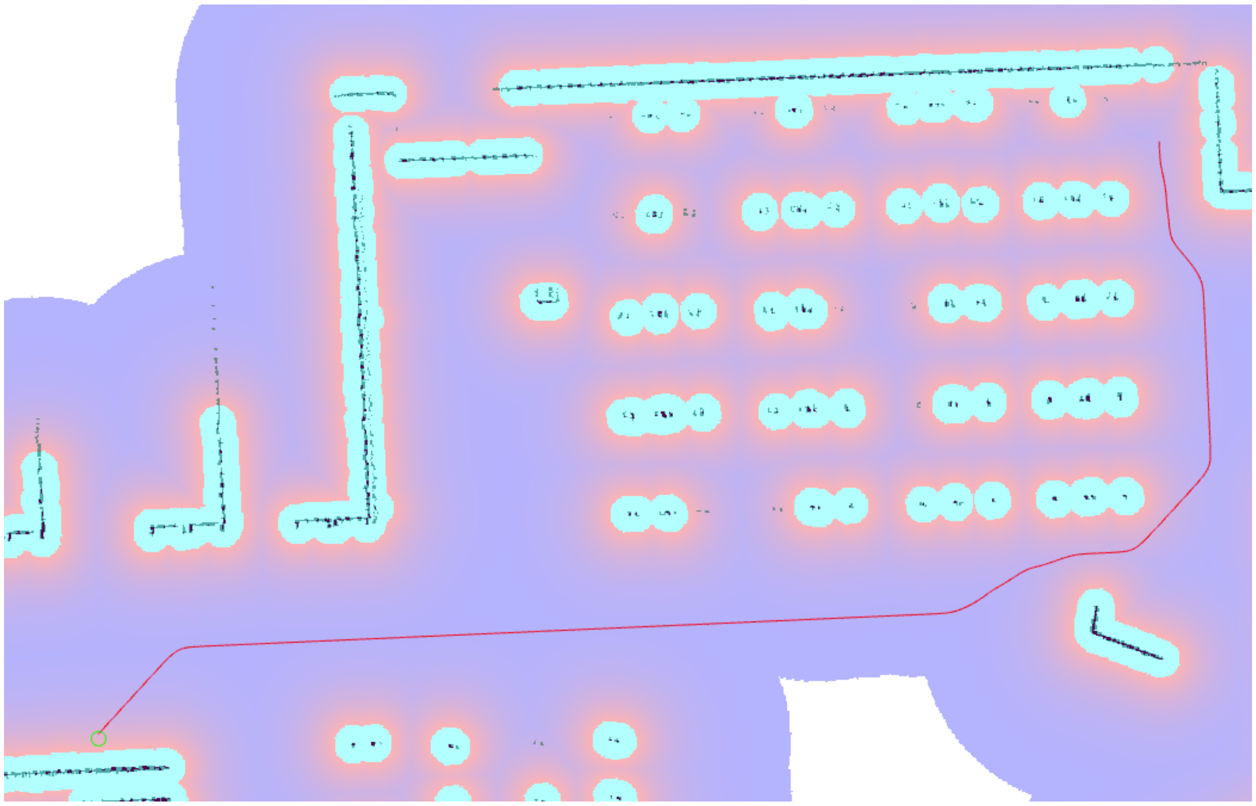
Source code and `README` with design, explanations, and metrics can be found on Github. A brief explanation can be found below, but the `README` contains the most detailed overview of the framework and planner implementations.

The Smac Planner plugin implements three A* based planning algorithms: 2D A*, Hybrid-A*, and State Lattice path planners. It is important to know that in June 2021 and December 2021, the package received a several **major** updates both improving path quality and run-times by 2-3x.

Provided Plugins

The plugins listed below are inside the `nav2_smac_planner` package. See the pages for individual configuration information.

Smac 2D Planner



`<name>` is the corresponding planner plugin ID selected for this type.

Parameters

<name>.tolerance

Type	Default
double	0.125

Description Tolerance in meters between requested goal pose and end of path.

<name>.downsample_costmap

Type	Default
bool	False

Description Whether to downsample costmap to another resolution for search.

<name>.downsampling_factor

Type	Default
int	1

Description Multiplier factor to downsample costmap by (e.g. if 5cm costmap at 2 `downsample_factor`, 10cm output).

<name>.allow_unknown

Type	Default
bool	True

Description Whether to allow traversing/search in unknown space.

<name>.max_iterations

Type	Default
int	1000000

Description Maximum number of search iterations before failing to limit compute time, disabled by -1.

<name>.max_on_approach_iterations

Type	Default
int	1000

Description Maximum number of iterations after the search is within `tolerance` before returning approximate path with best heuristic if exact path is not found.

<name>.max_planning_time

Type	Default
double	2.0

Description Maximum planning time in seconds.

<name>.cost_travel_multiplier

Type	Default
double	2.0

Description Cost multiplier to apply to search to steer away from high cost areas. Larger values will place in the center of aisles more exactly (if non-*FREE* cost potential field exists) but take slightly longer to compute. To optimize for speed, a value of 1.0 is reasonable. A reasonable tradeoff value is 2.0. A value of 0.0 effective disables steering away from obstacles and acts like a naive binary search A*.

<name>.use_final_approach_orientation

Type	Default
bool	false

Description If true, the last pose of the path generated by the planner will have its orientation set to the approach orientation, i.e. the orientation of the vector connecting the last two points of the path

<name>.smoother.max_iterations

Type	Default
int	1000

Description The maximum number of iterations the smoother has to smooth the path, to bound potential computation.

<name>.smoother.w_smooth

Type	Default
double	0.3

Description Weight for smoother to apply to smooth out the data points

<name>.smoother.w_data

Type	Default
double	0.2

Description Weight for smoother to apply to retain original data information

<name>.smoother.tolerance

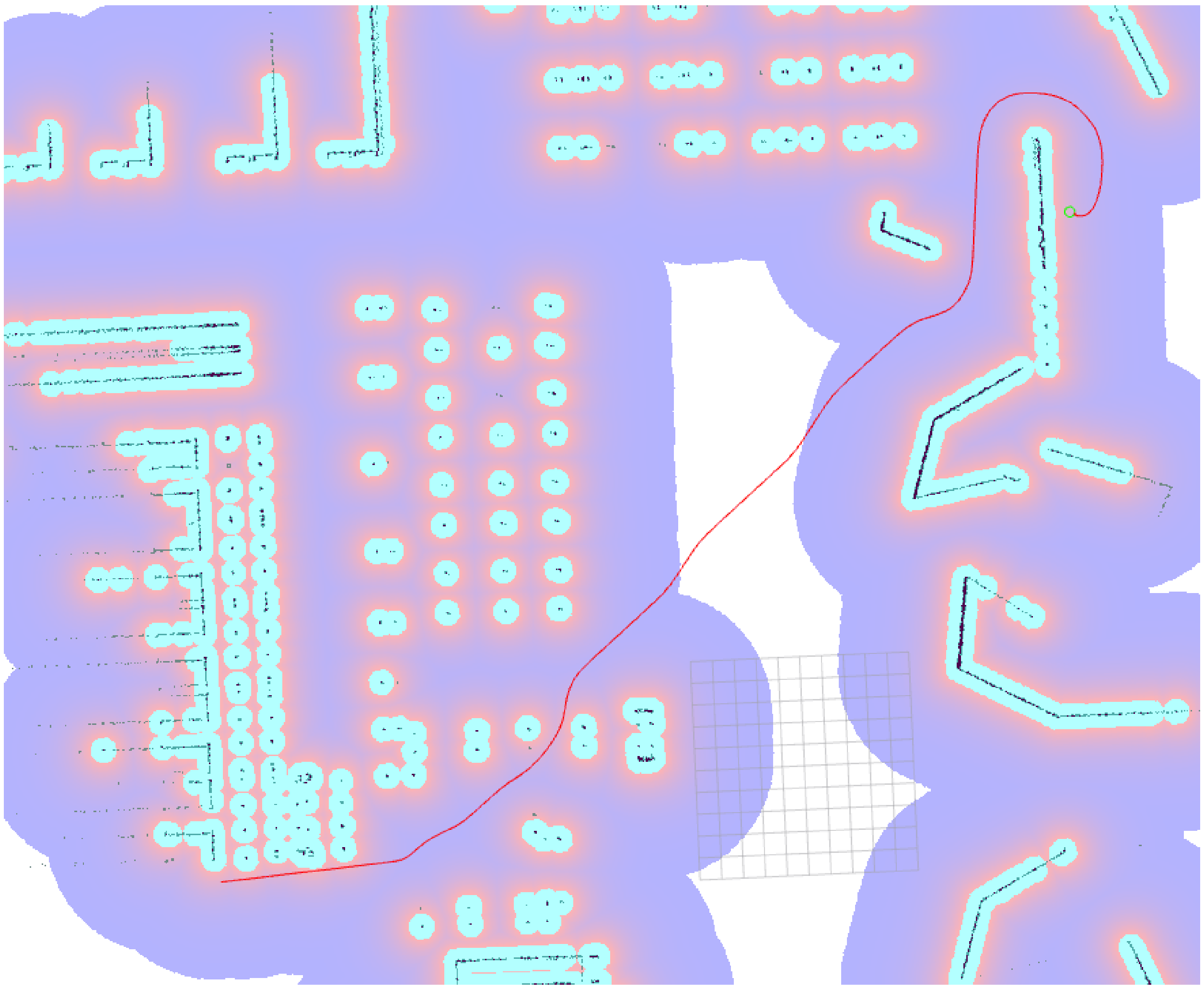
Type	Default
double	1e-10

Description Parameter tolerance change amount to terminate smoothing session

Example

```
planner_server:
  ros__parameters:
    planner_plugins: ["GridBased"]
    use_sim_time: True

    GridBased:
      plugin: "nav2_smac_planner/SmacPlanner2D"
      tolerance: 0.125                      # tolerance for planning if unable to␣
↪reach exact pose, in meters
      downsample_costmap: false             # whether or not to downsample the map
      downsampling_factor: 1                # multiplier for the resolution of the␣
↪costmap layer (e.g. 2 on a 5cm costmap would be 10cm)
      allow_unknown: true                   # allow traveling in unknown space
      max_iterations: 1000000               # maximum total iterations to search for␣
↪before failing (in case unreachable), set to -1 to disable
      max_on_approach_iterations: 1000      # maximum number of iterations to attempt␣
↪to reach goal once in tolerance
      max_planning_time: 2.0                # max time in s for planner to plan,␣
↪smooth
      cost_travel_multiplier: 2.0           # Cost multiplier to apply to search to␣
↪steer away from high cost areas. Larger values will place in the center of aisles␣
↪more exactly (if non-`FREE` cost potential field exists) but take slightly longer to␣
↪compute. To optimize for speed, a value of 1.0 is reasonable. A reasonable tradeoff␣
↪value is 2.0. A value of 0.0 effective disables steering away from obstacles and␣
↪acts like a naive binary search A*.
      use_final_approach_orientation: false # Whether to set the final path pose at␣
↪the goal's orientation to the requested orientation (false) or in line with the␣
↪approach angle so the robot doesn't rotate to heading (true)
      smoother:
        max_iterations: 1000
        w_smooth: 0.3
        w_data: 0.2
        tolerance: 1e-10
```

Smac Hybrid-A* Planner



`<name>` is the corresponding planner plugin ID selected for this type.

Parameters

`<name>`.downsample_costmap

Type	Default
bool	False

Description Whether to downsample costmap to another resolution for search.

`<name>`.downsampling_factor

Type	Default
int	1

Description Multiplier factor to downsample costmap by (e.g. if 5cm costmap at 2 `downsample_factor`, 10cm output).

<name>.allow_unknown

Type	Default
bool	True

Description Whether to allow traversing/search in unknown space.

<name>.tolerance

Type	Default
double	0.25

Description If an exact path cannot be found, the tolerance (as measured by the heuristic cost-to-goal) that would be acceptable to diverge from the requested pose.

<name>.max_iterations

Type	Default
int	1000000

Description Maximum number of search iterations before failing to limit compute time, disabled by -1.

<name>.max_on_approach_iterations

Type	Default
int	1000

Description Maximum number of iterations once a visited node is within the goal tolerances to continue to try to find an exact match before returning the best path solution within tolerances. Negative values convert to infinite.

<name>.max_planning_time

Type	Default
double	5.0

Description Maximum planning time in seconds.

<name>.analytic_expansion_ratio

Type	Default
double	3.5

Description Planner will attempt to complete an analytic expansions in a frequency proportional to this value and the minimum heuristic.

<name>.analytic_expansion_max_length

Type	Default
double	3.0

Description If the length is too far, reject this expansion. This prevents shortcutting of search with its penalty functions far out from the goal itself (e.g. so we don't reverse half-way across open maps or cut through high cost zones). This should never be smaller than 4-5x the minimum turning radius being used, or planning times will begin to spike.

`<name>`.motion_model_for_search

Type	Default
string	“DUBIN”

Description Motion model enum string to search with. For Hybrid-A* node, default is “DUBIN”. Options for SE2 are DUBIN or REEDS_SHEPP.

`<name>`.angle_quantization_bins

Type	Default
int	72

Description Number of angular bins to use for SE2 search. This can be any even number, but a good baseline is 64 or 72 (for 5 degree increments).

`<name>`.minimum_turning_radius

Type	Default
double	0.4

Description Minimum turning radius in meters of vehicle. Also used in the smoother to compute maximum curvature.

`<name>`.reverse_penalty

Type	Default
double	2.0

Description Heuristic penalty to apply to SE2 node if searching in reverse direction. Only used in `REEDS_SHEPP` motion model.

`<name>`.change_penalty

Type	Default
double	0.0

Description Heuristic penalty to apply to SE2 node if changing direction (e.g. left to right) in search. Disabled by default after change to guarantee admissibility of the Hybrid-A* planner.

`<name>`.non_straight_penalty

Type	Default
double	1.20

Description Heuristic penalty to apply to SE2 node if searching in non-straight direction.

`<name>`.cost_penalty

Type	Default
double	2.0

Description Heuristic penalty to apply to SE2 node for cost at pose. Allows Hybrid-A* to be cost aware.

`<name>`.retrospective_penalty

Type	Default
double	0.015

Description Heuristic penalty to apply to SE2 node penalty. Causes Hybrid-A* to prefer later maneuvers before earlier ones along the path. Saves search time since earlier (shorter) branches are not expanded until it is necessary. Must be >= 0.0 and <= 1.0. Must be *0.0* to be fully admissible.

`<name>`.lookup_table_size

Type	Default
double	20.0

Description Size of the dubin/reeds-sheep distance window to cache, in meters.

`<name>`.debug_visualizations

Type	Default
bool	false

Description Whether to publish expansions on the `/expansions` topic as an array of poses (the orientation has no meaning) and the path's footprints on the `/planned_footprints` topic. WARNING: heavy to compute and to display, for debug only as it degrades the performance.

`<name>`.cache_obstacle_heuristic

Type	Default
bool	false

Description Cache the obstacle map dynamic programming distance expansion heuristic between subsiquent replannings of the same goal location. Dramatically speeds up replanning performance (40x) if costmap is largely static.

`<name>`.smooth_path

Type	Default
bool	true

Description If true, does simple and fast smoothing post-processing to the path from search

`<name>`.smoother.max_iterations

Type	Default
int	1000

Description The maximum number of iterations the smoother has to smooth the path, to bound potential computation.

`<name>`.smoother.w_smooth

Type	Default
double	0.3

Description Weight for smoother to apply to smooth out the data points

<name>.smoother.w_data

Type	Default
double	0.2

Description Weight for smoother to apply to retain original data information

<name>.smoother.tolerance

Type	Default
double	1e-10

Description Parameter tolerance change amount to terminate smoothing session

<name>.smoother.do_refinement

Type	Default
bool	true

Description Performs extra refinement smoothing runs. Essentially, this recursively calls the smoother using the output from the last smoothing cycle to further smooth the path for macro-trends. This typically improves quality especially in the Hybrid-A* planner due to the extra "wobbling" it can have due to the very small primitive lengths but may cause the path to get slightly closer to some obstacles.

<name>.smoother.refinement_num

Type	Default
int	2

Description Number of times to recursively attempt to smooth, must be `>= 1`.

Example

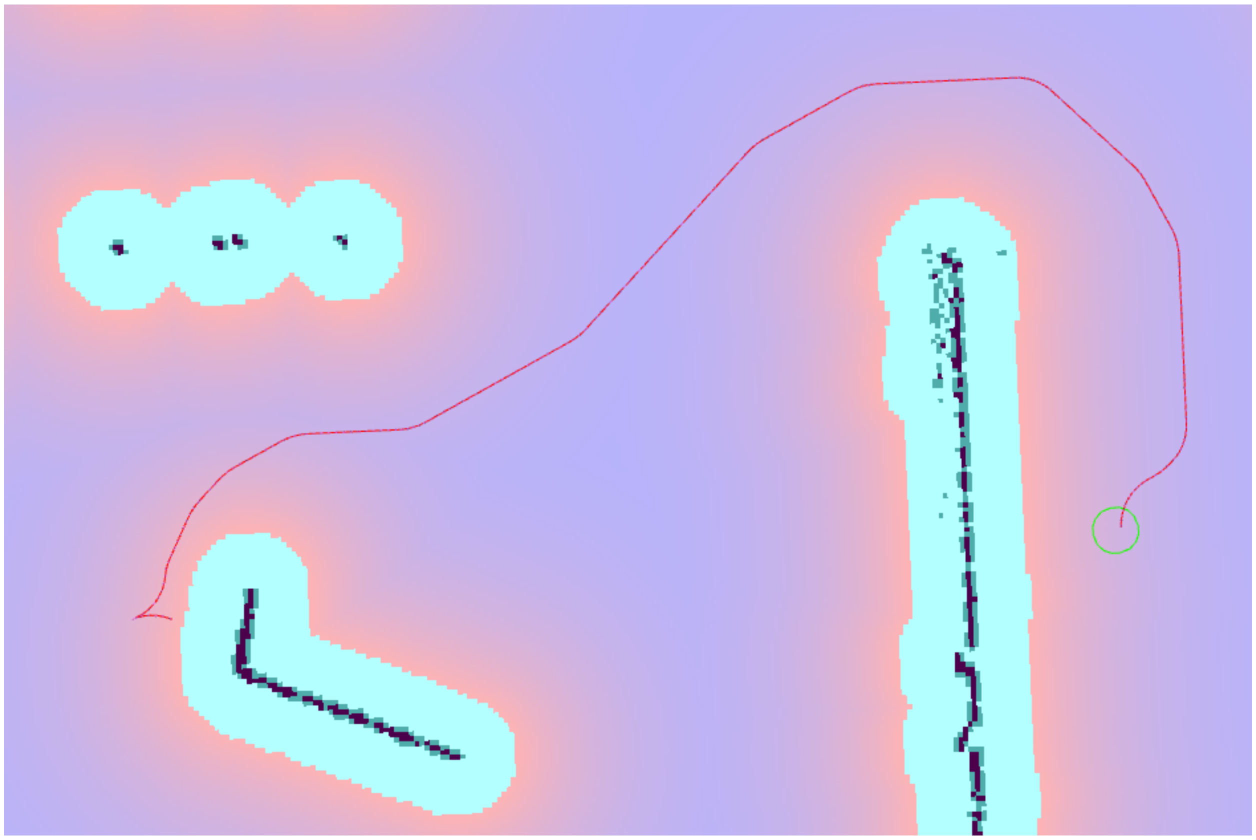
```
planner_server:
  ros__parameters:
    planner_plugins: ["GridBased"]
    use_sim_time: True

    GridBased:
      plugin: "nav2_smac_planner/SmacPlannerHybrid"
      downsample_costmap: false           # whether or not to downsample the map
      downsampling_factor: 1              # multiplier for the resolution of the␣
↪costmap layer (e.g. 2 on a 5cm costmap would be 10cm)
      tolerance: 0.25                     # dist-to-goal heuristic cost (distance)␣
↪for valid tolerance endpoints if exact goal cannot be found.
      allow_unknown: true                 # allow traveling in unknown space
      max_iterations: 1000000             # maximum total iterations to search for␣
↪before failing (in case unreachable), set to -1 to disable
      max_on_approach_iterations: 1000    # Maximum number of iterations after within␣
↪tolerances to continue to try to find exact solution
      max_planning_time: 5.0              # max time in s for planner to plan, smooth
      motion_model_for_search: "DUBIN"    # Hybrid-A* Dubin, Redds-Shepp
      angle_quantization_bins: 72         # Number of angle bins for search
      analytic_expansion_ratio: 3.5       # The ratio to attempt analytic expansions␣
↪during search for final approach.
```

(continues on next page)

(continued from previous page)

```
      analytic_expansion_max_length: 3.0  # For Hybrid/Lattice nodes: The maximum␣
↪length of the analytic expansion to be considered valid to prevent unsafe␣
↪shortcutting
      minimum_turning_radius: 0.40        # minimum turning radius in m of path /␣
↪vehicle
      reverse_penalty: 2.0                # Penalty to apply if motion is reversing,␣
↪must be => 1
      change_penalty: 0.0                 # Penalty to apply if motion is changing␣
↪directions (L to R), must be >= 0
      non_straight_penalty: 1.2           # Penalty to apply if motion is non-
↪straight, must be => 1
      cost_penalty: 2.0                   # Penalty to apply to higher cost areas␣
↪when adding into the obstacle map dynamic programming distance expansion heuristic.␣
↪This drives the robot more towards the center of passages. A value between 1.3 - 3.
↪5 is reasonable.
      retrospective_penalty: 0.015
      lookup_table_size: 20.0             # Size of the dubin/reeds-sheep distance␣
↪window to cache, in meters.
      cache_obstacle_heuristic: false     # Cache the obstacle map dynamic␣
↪programming distance expansion heuristic between subsiquent replannings of the same␣
↪goal location. Dramatically speeds up replanning performance (40x) if costmap is␣
↪largely static.
      debug_visualizations: false         # For Hybrid nodes: Whether to publish␣
↪expansions on the /expansions topic as an array of poses (the orientation has no␣
↪meaning) and the path's footprints on the /planned_footprints topic. WARNING: heavy␣
↪to compute and to display, for debug only as it degrades the performance.
      smooth_path: True                   # If true, does a simple and quick␣
↪smoothing post-processing to the path

      smoother:
        max_iterations: 1000
        w_smooth: 0.3
        w_data: 0.2
        tolerance: 1.0e-10
        do_refinement: true
        refinement_num: 2
```

Smac State Lattice Planner



`<name>` is the corresponding planner plugin ID selected for this type.

Note: State Lattice does not have the costmap downsampler due to the minimum control sets being tied with map resolutions on generation. The minimum turning radius is also not a parameter in State Lattice since this was specified at the minimum control set pre-computation phase. See the Smac Planner package to generate custom control sets for your vehicle or use one of our pre-generated examples.

The image above you can see the reverse expansion enabled, such that the robot can back into a tight requested spot close to an obstacle.

Parameters

`<name>`.allow_unknown

Type	Default
bool	True

Description Whether to allow traversing/search in unknown space.

`<name>`.tolerance

Type	Default
double	0.25

Description If an exact path cannot be found, the tolerance (as measured by the heuristic cost-to-goal) that would be acceptable to diverge from the requested pose in distance-to-goal.

`<name>`.max_iterations

Type	Default
int	1000000

Description Maximum number of search iterations before failing to limit compute time, disabled by -1.

`<name>`.max_on_approach_iterations

Type	Default
int	1000

Description Maximum number of iterations once a visited node is within the goal tolerances to continue to try to find an exact match before returning the best path solution within tolerances.

`<name>`.max_planning_time

Type	Default
double	5.0

Description Maximum planning time in seconds.

`<name>`.analytic_expansion_ratio

Type	Default
double	3.5

Description SE2 node will attempt to complete an analytic expansion with frequency proportional to this value and the minimum heuristic. Negative values convert to infinite.

`<name>`.analytic_expansion_max_length

Type	Default
double	3.0

Description If the length is too far, reject this expansion. This prevents shortcutting of search with its penalty functions far out from the goal itself (e.g. so we don't reverse half-way across open maps or cut through high cost zones). This should never be smaller than 4-5x the minimum turning radius being used, or planning times will begin to spike.

`<name>`.reverse_penalty

Type	Default
double	2.0

Description Heuristic penalty to apply to SE2 node if searching in reverse direction. Only used in `allow_reverse_expansion = true`.

`<name>`.change_penalty

Type	Default
double	0.05

Description Heuristic penalty to apply to SE2 node if changing direction (e.g. left to right) in search.

`<name>`.non_straight_penalty

Type	Default
double	1.05

Description Heuristic penalty to apply to SE2 node if searching in non-straight direction.

`<name>`.cost_penalty

Type	Default
double	2.0

Description Heuristic penalty to apply to SE2 node for cost at pose. Allows State Lattice to be cost aware.

`<name>`.rotation_penalty

Type	Default
double	5.0

Description Penalty to apply for rotations in place, if minimum control set contains in-place rotations. This should always be set sufficiently high to weight against in-place rotations unless strictly necessary for obstacle avoidance or there may be frequent discontinuities in the plan where the plan requests the robot to rotate in place to short-cut an otherwise smooth forward-moving path for marginal path distance savings.

`<name>`.retrospective_penalty

Type	Default
double	0.015

Description Heuristic penalty to apply to SE2 node penalty. Causes State Lattice to prefer later maneuvers before earlier ones along the path. Saves search time since earlier (shorter) branches are not expanded until it is necessary. Must be >= 0.0 and <= 1.0. Must be *0.0* to be fully admissible.

`<name>`.lattice_filepath

Type	Default
string	“”

Description The filepath to the state lattice minimum control set graph, this will default to a 16 bin, 0.5m turning radius control set located in `test/` for basic testing and evaluation (opposed to Hybrid-A*’s default of 0.5m).

`<name>`.lookup_table_size

Type	Default
double	20.0

Description Size of the dubin/reeds-sheep distance window to cache, in meters.

`<name>`.cache_obstacle_heuristic

Type	Default
bool	false

Description Cache the obstacle map dynamic programming distance expansion heuristic between subsiquent replannings of the same goal location. Dramatically speeds up replanning performance (40x) if costmap is largely static.

`<name>`.allow_reverse_expansion

Type	Default
bool	false

Description If true, allows the robot to use the primitives to expand in the mirrored opposite direction of the current robot's orientation (to reverse).

`<name>`.smooth_path

Type	Default
bool	true

Description If true, does simple and fast smoothing post-processing to the path from search

`<name>`.smoother.max_iterations

Type	Default
int	1000

Description The maximum number of iterations the smoother has to smooth the path, to bound potential computation.

`<name>`.smoother.w_smooth

Type	Default
double	0.3

Description Weight for smoother to apply to smooth out the data points

`<name>`.smoother.w_data

Type	Default
double	0.2

Description Weight for smoother to apply to retain original data information

`<name>`.smoother.tolerance

Type	Default
double	1e-10

Description Parameter tolerance change amount to terminate smoothing session

`<name>`.smoother.do_refinement

Type	Default
bool	true

Description Performs extra refinement smoothing runs. Essentially, this recursively calls the smoother using the output from the last smoothing cycle to further smooth the path for macro-trends. This typically improves quality especially in the Hybrid-A* planner but can be helpful on the state lattice planner to reduce the "blocky" movements in State Lattice caused by the limited number of headings.

`<name>`.smoother.refinement_num

Type	Default
int	2

Description Number of times to recursively attempt to smooth, must be `>= 1`.

Example

```
planner_server:
  ros__parameters:
    planner_plugins: ["GridBased"]
    use_sim_time: True

    GridBased:
      plugin: "nav2_smac_planner/SmacPlannerLattice"
      allow_unknown: true                 # Allow traveling in unknown space
      tolerance: 0.25                     # dist-to-goal heuristic cost (distance)␣
↪for valid tolerance endpoints if exact goal cannot be found.
      max_iterations: 1000000             # Maximum total iterations to search for␣
↪before failing (in case unreachable), set to -1 to disable
      max_on_approach_iterations: 1000    # Maximum number of iterations after within␣
↪tolerances to continue to try to find exact solution
      max_planning_time: 5.0              # Max time in s for planner to plan, smooth
      analytic_expansion_ratio: 3.5       # The ratio to attempt analytic expansions␣
↪during search for final approach.
      analytic_expansion_max_length: 3.0  # For Hybrid/Lattice nodes: The maximum␣
↪length of the analytic expansion to be considered valid to prevent unsafe␣
↪shortcutting
      reverse_penalty: 2.0                # Penalty to apply if motion is reversing,␣
↪must be => 1
      change_penalty: 0.05                # Penalty to apply if motion is changing␣
↪directions (L to R), must be >= 0
      non_straight_penalty: 1.05          # Penalty to apply if motion is non-
↪straight, must be => 1
      cost_penalty: 2.0                   # Penalty to apply to higher cost areas␣
↪when adding into the obstacle map dynamic programming distance expansion heuristic.␣
↪This drives the robot more towards the center of passages. A value between 1.3 - 3.
↪5 is reasonable.
      rotation_penalty: 5.0               # Penalty to apply to in-place rotations,␣
↪if minimum control set contains them
      retrospective_penalty: 0.015
      lattice_filepath: ""                # The filepath to the state lattice graph
      lookup_table_size: 20.0             # Size of the dubin/reeds-sheep distance␣
↪window to cache, in meters.
      cache_obstacle_heuristic: false     # Cache the obstacle map dynamic␣
↪programming distance expansion heuristic between subsiquent replannings of the same␣
↪goal location. Dramatically speeds up replanning performance (40x) if costmap is␣
↪largely static.
      allow_reverse_expansion: false      # If true, allows the robot to use the␣
↪primitives to expand in the mirrored opposite direction of the current robot's␣
↪orientation (to reverse).
```

(continues on next page)

(continued from previous page)

```
      smooth_path: True                     # If true, does a simple and quick␣
↪smoothing post-processing to the path
      smoother:
        max_iterations: 1000
        w_smooth: 0.3
        w_data: 0.2
        tolerance: 1.0e-10
        do_refinement: true
        refinement_num: 2
```

Description

The `nav2_smac_planner` package contains an optimized templated A* search algorithm used to create multiple A*-based planners for multiple types of robot platforms. It uses template node types to develop different search-based planners.

We support circular differential-drive and omni-directional drive robots using the `SmacPlanner2D` planner which implements a cost-aware A* planner. We support car-like (ackermann) and legged vehicles using the `SmacPlannerHybrid` plugin which implements a Hybrid-A* planner. We support non-circular, arbitrary shaped, any model vehicles using the `SmacPlannerLattice` plugin which implements a State Lattice planner (e.g. omni, diff, ackermann, legged, custom). It contains control sets and generators for ackermann, legged, differential drive and omnidirectional vehicles, but you may provide your own for another robot type or to have different planning behaviors.

The last two plugins are both **kinematically feasible** and support **reversing**. They have performance similar to its 2D counter parts like 2D-A* and NavFn via highly optimized heuristic functions and efficient programming. An example of the 3 planners can be seen below, planning a roughly 75 m path.

- Hybrid-A* computed the path in 144ms
- State Lattice computed the path in 113ms
- 2D A* computed the path in 243ms
- For reference: NavFn compute the path in 146ms, including some nasty path discontinuity artifacts

Usual planning times are below 100ms for some environments, occasionally approaching up to 200ms. The performance of all 3 planners is roughly comparable with naive 2D search algorithms that have long been mainstays of the ROS Navigation ecosystem, but also achieving kinematic feasibility, support for reversing, and using modern state of the art techniques.

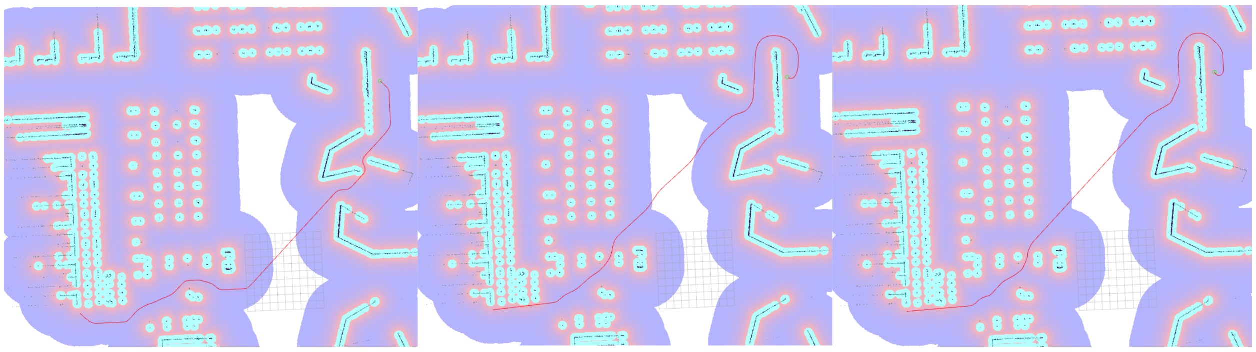
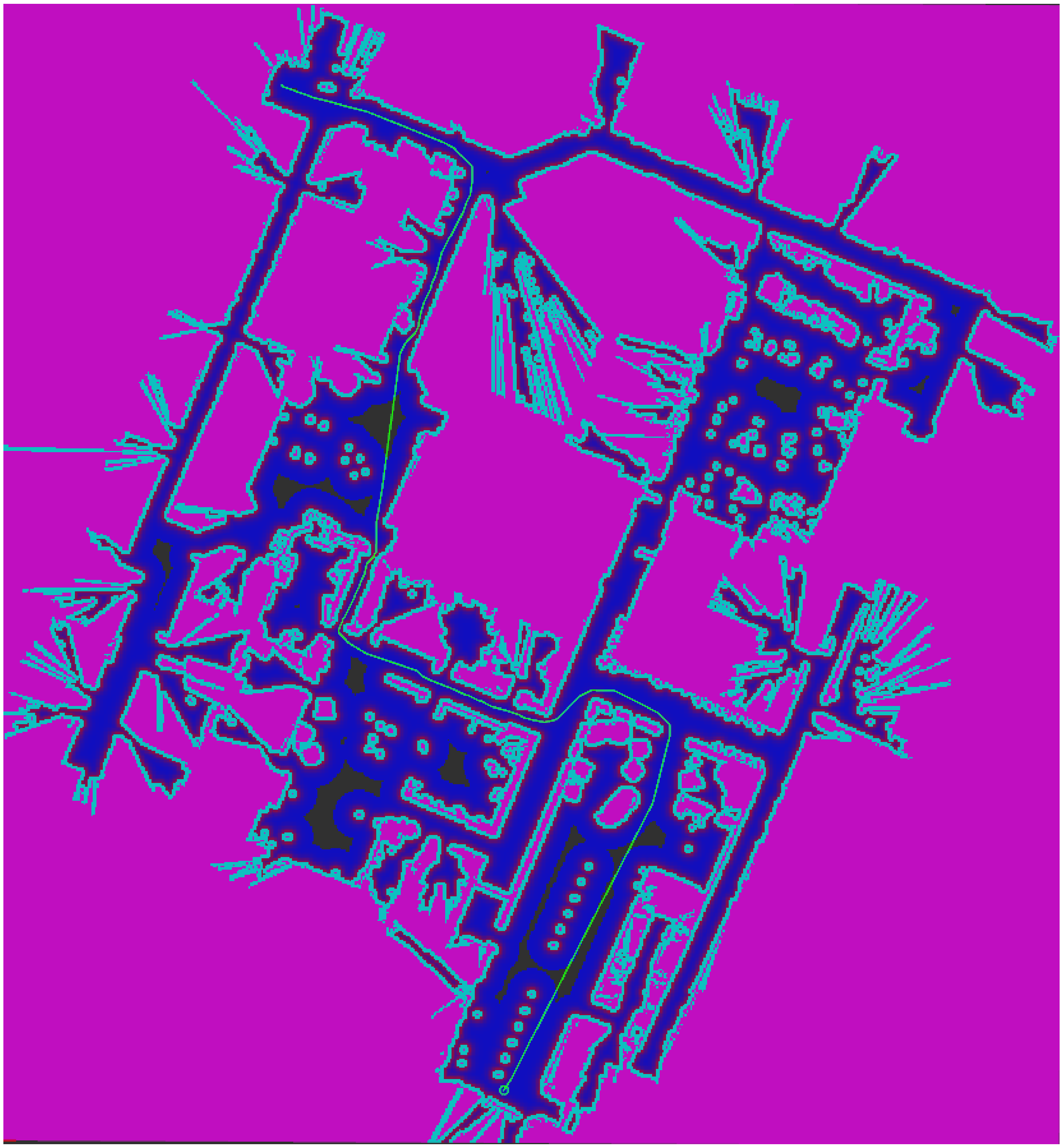


Figure 4.7: 2D A* (Panel 1), Hybrid-A* (Panel 2), State Lattice (Panel 3)

4.7.8 Theta Star Planner

Theta Star Planner implements the Theta* path planner meant to plan any-angled paths using A*.

For the below example the planner took ~46ms (averaged value) to compute the path of 87.5m -



`<name>` is the corresponding planner plugin ID selected for this type.

Parameters

The parameters of the planner are:

`<name>`.how_many_corners

Type	Default
int	8

Description To choose between 4-connected (up, down, left, right) and 8-connected (all the adjacent cells) graph expansions, the accepted values are 4 and 8

`<name>` .w_euc_cost

Type	Default
double	1.0

Description Weight applied on the length of the path.

`<name>`.w_traversal_cost

Type	Default
double	2.0

Description It tunes how harshly the nodes of high cost are penalised. From the above g(neigh) equation you can see that the cost-aware component of the cost function forms a parabolic curve, thus this parameter would, on increasing its value, make that curve steeper allowing for a greater differentiation (as the delta of costs would increase, when the graph becomes steep) among the nodes of different costs.

`<name>`.use_final_approach_orientation

Type	Default
bool	false

Description If true, the last pose of the path generated by the planner will have its orientation set to the approach orientation, i.e. the orientation of the vector connecting the last two points of the path

`<name>`.allow_unknown

Type	Default
bool	True

Description Whether to allow planning in unknown space.

Note: Do go through the README file available on this repo's link to develop a better understanding of how you could tune this planner. This planner requires you to tune the *cost_scaling_factor* parameter of your costmap too, to get good results.

Example

```
planner_server:
ros__parameters:
  expected_planner_frequency: 20.0
  use_sim_time: True
  planner_plugins: ["GridBased"]
  GridBased:
    plugin: "nav2_theta_star_planner/ThetaStarPlanner"
    how_many_corners: 8
    w_euc_cost: 1.0
    w_traversal_cost: 2.0
    w_heuristic_cost: 1.0
```

4.7.9 Controller Server

Source code on Github.

The Controller Server implements the server for handling the controller requests for the stack and host a map of plugin implementations. It will take in path and plugin names for controller, progress checker and goal checker to use and call the appropriate plugins.

Parameters

controller_frequency

Type	Default
double	20.0

Description Frequency to run controller (Hz).

controller_plugins

Type	Default
vector<string>	[‘FollowPath’]

Description List of mapped names for controller plugins for processing requests and parameters.

Note Each plugin namespace defined in this list needs to have a `plugin` parameter defining the type of plugin to be loaded in the namespace.

Example:

```
controller_server:
  ros__parameters:
    controller_plugins: ["FollowPath"]
    FollowPath:
      plugin: "dwb_core::DWBLocalPlanner"
```

progress_checker_plugins

Type	Default
vector<string>	[“progress_checker”]

Description Mapped name for progress checker plugin for checking progress made by robot. Formerly `progress_checker_plugin` for Humble and older with a single string plugin.

Note The plugin namespace defined needs to have a `plugin` parameter defining the type of plugin to be loaded in the namespace.

Example:

```
controller_server:
  ros__parameters:
    progress_checker_plugins: ["progress_checker"] # progress_checker_
↪plugin: "progress_checker" For Humble and older
    progress_checker:
      plugin: "nav2_controller::SimpleProgressChecker"
```

goal_checker_plugins

Type	Default
string	‘goal_checker’

Description Mapped name for goal checker plugin for checking goal is reached.

Note The plugin namespace defined needs to have a `plugin` parameter defining the type of plugin to be loaded in the namespace.

Example:

```
controller_server:
  ros__parameters:
    goal_checker_plugins: ["goal_checker"]
    goal_checker:
      plugin: "nav2_controller::SimpleGoalChecker"
```

min_x_velocity_threshold

Type	Default
double	0.0001

Description The controller server filters the velocity portion of the odometry messages received before sending them to the controller plugin. Odometry values below this threshold (in m/s) will be set to 0.0.

min_y_velocity_threshold

Type	Default
double	0.0001

Description The controller server filters the velocity portion of the odometry messages received before sending them to the controller plugin. Odometry values below this threshold (in m/s) will be set to 0.0. For non-holonomic robots

min_theta_velocity_threshold

Type	Default
double	0.0001

Description The controller server filters the velocity portion of the odometry messages received before sending them to the controller plugin. Odometry values below this threshold (in rad/s) will be set to 0.0.

failure_tolerance

Type	Default
double	0.0

Description The maximum duration in seconds the called controller plugin can fail (i.e. the `computeVelocityCommands` function of the plugin throwing an exception) before the `nav2_msgs::action::FollowPath` action fails. Setting it to the special value of -1.0 makes it infinite, 0 to disable, and any positive value for the appropriate timeout.

speed_limit_topic

Type	Default
string	"speed_limit"

Description Speed limiting topic name to subscribe. This could be published by Speed Filter (please refer to *Speed Filter Parameters* configuration page). You can also use this without the Speed Filter as well if you provide an external server to publish these messages.

odom_topic

Type	Default
string	"odom"

Description Topic to get instantaneous measurement of speed from.

Provided Plugins

The plugins listed below are inside the `nav2_controller` namespace.

SimpleProgressChecker

Checks whether the robot has made progress.

Parameters

`<nav2_controller plugin>`: nav2_controller plugin name defined in the **progress_checker_plugin_id** parameter in *Controller Server*.

`<nav2_controller plugin>`.required_movement_radius

Type	Default
double	0.5

Description Minimum amount a robot must move to be progressing to goal (m).

`<nav2_controller plugin>`.movement_time_allowance

Type	Default
double	10.0

Description Maximum amount of time a robot has to move the minimum radius (s).

PoseProgressChecker

Checks whether the robot has made progress based on translation and rotation.

Parameters

`<nav2_controller plugin>`: nav2_controller plugin name defined in the **progress_checker_plugin_id** parameter in *Controller Server*.

`<nav2_controller plugin>`.required_movement_radius

Type	Default
double	0.5

Description Minimum amount a robot must move to be progressing to goal (m).

`<nav2_controller plugin>`.required_movement_angle

Type	Default
double	0.5

Description Minimum amount a robot must rotate to be progressing to goal (rad).

`<nav2_controller plugin>`.movement_time_allowance

Type	Default
double	10.0

Description Maximum amount of time a robot has to move the minimum radius or the mnimum angle (s).

SimpleGoalChecker

Checks whether the robot has reached the goal pose.

Parameters

`<nav2_controller plugin>`: nav2_controller plugin name defined in the **goal_checker_plugin_id** parameter in *Controller Server*.

`<nav2_controller plugin>`.xy_goal_tolerance

Type	Default
double	0.25

Description Tolerance to meet goal completion criteria (m).

`<nav2_controller plugin>`.yaw_goal_tolerance

Type	Default
double	0.25

Description Tolerance to meet goal completion criteria (rad).

`<nav2_controller plugin>`.stateful

Type	Default
bool	true

Description Whether to check for XY position tolerance after rotating to goal orientation in case of minor localization changes.

StoppedGoalChecker

Checks whether the robot has reached the goal pose and come to a stop.

Parameters

`<nav2_controller plugin>`: nav2_controller plugin name defined in the **goal_checker_plugin_id** parameter in *Controller Server*.

`<nav2_controller plugin>`.trans_stopped_velocity

Type	Default
double	0.25

Description Velocity below is considered to be stopped at tolerance met (m/s).

`<nav2_controller plugin>`.rot_stopped_velocity

Type	Default
double	0.25

Description Velocity below is considered to be stopped at tolerance met (rad/s).

Default Plugins

When the `progress_checker_plugins`, `goal_checker_plugin` or `controller_plugins` parameters are not overridden, the following default plugins are loaded:

Namespace	Plugin
“progress_checker”	“nav2_controller::SimpleProgressChecker”
“goal_checker”	“nav2_controller::SimpleGoalChecker”
“FollowPath”	“dwb_core::DWBLocalPlanner”

Example

```
controller_server:
  ros__parameters:
    use_sim_time: True
    controller_frequency: 20.0
    min_x_velocity_threshold: 0.001
    min_y_velocity_threshold: 0.5
    min_theta_velocity_threshold: 0.001
```

(continues on next page)

(continued from previous page)

```
    failure_tolerance: 0.3
    odom_topic: "odom"
    progress_checker_plugins: ["progress_checker"] # progress_checker_plugin:
↪"progress_checker" For Humble and older
    goal_checker_plugin: "goal_checker"
    controller_plugins: ["FollowPath"]
    progress_checker:
      plugin: "nav2_controller::SimpleProgressChecker"
      required_movement_radius: 0.5
      movement_time_allowance: 10.0
    goal_checker:
      plugin: "nav2_controller::SimpleGoalChecker"
      xy_goal_tolerance: 0.25
      yaw_goal_tolerance: 0.25
      stateful: True
    FollowPath:
      plugin: "dwb_core::DWBLocalPlanner"
```

4.7.10 DWB Controller

Source code on Github.

The DWB controller is the default controller. It is a fork of David Lu's controller modified for ROS 2.

Controller

DWB Controller

Parameters

`<dwb plugin>`: DWB plugin name defined in the **controller_plugin_ids** parameter in *Controller Server*.

`<dwb plugin>`.critics

Type	Default
vector<string>	N/A

Description List of critic plugins to use.

`<dwb plugin>`.default_critic_namespaces

Type	Default
vector<string>	["dwb_critics"]

Description Namespaces to load critics in.

`<dwb plugin>`.prune_plan

Type	Default
bool	true

Description Whether to prune the path of old, passed points.

`<dwb plugin>`.shorten_transformed_plan

Type	Default
bool	true

Description Determines whether we will pass the full plan on to the critics.

`<dwb plugin>`.prune_distance

Type	Default
double	2.0

Description Distance (m) to prune backward until.

`<dwb plugin>`.forward_prune_distance

Type	Default
double	2.0

Description Distance (m) to prune forward until. If set to `-1`, it will search the full path for the closest point, in the case of no replanning.

`<dwb plugin>`.debug_trajectory_details

Type	Default
bool	false

Description Publish debug information (on what topic???).

`<dwb plugin>`.trajectory_generator_name

Type	Default
string	"dwb_plugins::StandardTrajectoryGenerator"

Description Trajectory generator plugin name.

`<dwb plugin>`.goal_checker_name

Type	Default
string	"dwb_plugins::SimpleGoalChecker"

Description Goal checker plugin name.

`<dwb plugin>`.transform_tolerance

Type	Default
double	0.1

Description TF transform tolerance (s).

`<dwb plugin>`.short_circuit_trajectory_evaluation

Type	Default
bool	true

Description Stop evaluating scores after best score is found.

`<dwb plugin>`.path_distance_bias (Legacy)

Type	Default
double	N/A

Description Old version of PathAlign.scale, use that instead.

`<dwb plugin>`.goal_distance_bias (Legacy)

Type	Default
double	N/A

Description Old version of GoalAlign.scale, use that instead.

`<dwb plugin>`.occdist_scale (Legacy)

Type	Default
double	N/A

Description Old version of ObstacleFootprint.scale, use that instead.

`<dwb plugin>`.max_scaling_factor (Legacy)

Type	Default
double	N/A

Description Old version of ObstacleFootprint.max_scaling_factor, use that instead.

`<dwb plugin>`.scaling_speed (Legacy)

Type	Default
double	N/A

Description Old version of ObstacleFootprint.scaling_speed, use that instead.

`<dwb plugin>`.PathAlign.scale

Type	Default
double	32.0

Description Scale for path align critic, overriding local default.

`<dwb plugin>`.GoalAlign.scale

Type	Default
double	24.0

Description Scale for goal align critic, overriding local default.

`<dwb plugin>`.PathDist.scale

Type	Default
double	32.0

Description Scale for path distance critic, overriding local default.

`<dwb plugin>`.GoalDist.scale

Type	Default
double	24.0

Description Scale for goal distance critic, overriding local default.

XYTheta Iterator

Parameters

`<dwb plugin>`: DWB plugin name defined in the **controller_plugin_ids** parameter in *Controller Server*.

`<dwb plugin>`.vx_samples

Type	Default
int	20

Description Number of velocity samples in the X velocity direction.

`<dwb plugin>`.vy_samples

Type	Default
int	5

Description Number of velocity samples in the Y velocity direction.

`<dwb plugin>`.vtheta_samples

Type	Default
int	20

Description Number of velocity samples in the angular directions.

Kinematic Parameters

Parameters

`<dwb plugin>`: DWB plugin name defined in the **controller_plugin_ids** parameter in *Controller Server*.

`<dwb plugin>`.max_vel_theta

Type	Default
double	0.0

Description Maximum angular velocity (rad/s).

`<dwb plugin>`.min_speed_xy

Type	Default
double	0.0

Description Minimum translational speed (m/s).

`<dwb plugin>`.max_speed_xy

Type	Default
double	0.0

Description Maximum translational speed (m/s).

`<dwb plugin>`.min_speed_theta

Type	Default
double	0.0

Description Minimum angular speed (rad/s).

`<dwb plugin>`.min_vel_x

Type	Default
double	0.0

Description Minimum velocity X (m/s).

`<dwb plugin>`.min_vel_y

Type	Default
double	0.0

Description Minimum velocity Y (m/s).

`<dwb plugin>`.max_vel_x

Type	Default
double	0.0

Description Maximum velocity X (m/s).

`<dwb plugin>`.max_vel_y

Type	Default
double	0.0

Description Maximum velocity Y (m/s).

`<dwb plugin>`.acc_lim_x

Type	Default
double	0.0

Description Maximum acceleration X (m/s^2).

`<dwb plugin>`.acc_lim_y

Type	Default
double	0.0

Description Maximum acceleration Y (m/s^2).

<dwb plugin>.acc_lim_theta

Type	Default
double	0.0

Description Maximum acceleration rotation (rad/s^2).

<dwb plugin>.decel_lim_x

Type	Default
double	0.0

Description Maximum deceleration X (m/s^2).

<dwb plugin>.decel_lim_y

Type	Default
double	0.0

Description Maximum deceleration Y (m/s^2).

<dwb plugin>.decel_lim_theta

Type	Default
double	0.0

Description Maximum deceleration rotation (rad/s^2).

Publisher

Parameters

`<dwb plugin>`: DWB plugin name defined in the **controller_plugin_ids** parameter in *Controller Server*.

<dwb plugin>.publish_evaluation

Type	Default
bool	true

Description Whether to publish the local plan evaluation.

<dwb plugin>.publish_global_plan

Type	Default
bool	true

Description Whether to publish the global plan.

<dwb plugin>.publish_transformed_plan

Type	Default
bool	true

Description Whether to publish the global plan in the odometry frame.

`<dwb plugin>`.publish_local_plan

Type	Default
bool	true

Description Whether to publish the local planner's plan.

`<dwb plugin>`.publish_trajectories

Type	Default
bool	true

Description Whether to publish debug trajectories.

`<dwb plugin>`.publish_cost_grid_pc

Type	Default
bool	false

Description Whether to publish the cost grid.

`<dwb plugin>`.marker_lifetime

Type	Default
double	0.1

Description How long for the marker to remain.

Plugins

The plugins listed below are inside the `dwb_plugins` namespace.

LimitedAccelGenerator

`<dwb plugin>`: DWB plugin name defined in the **controller_plugin_ids** parameter in *Controller Server*.

Parameters

`<dwb plugin>`.sim_time

Type	Default
double	1.7

Description Time to simulate ahead by (s).

StandardTrajectoryGenerator

Parameters

`<dwb plugin>`: DWB plugin name defined in the **controller_plugin_ids** parameter in *Controller Server*.

`<dwb plugin>`.sim_time

Type	Default
double	1.7

Description Time to simulate ahead by (s).

`<dwb plugin>`.discretize_by_time

Type	Default
bool	false

Description If true, forward simulate by time. If False, forward simulate by linear and angular granularity.

`<dwb plugin>`.time_granularity

Type	Default
double	0.5

Description Time ahead to project.

`<dwb plugin>`.linear_granularity

Type	Default
double	0.5

Description Linear distance forward to project.

`<dwb plugin>`.angular_granularity

Type	Default
double	0.025

Description Angular distance to project.

`<dwb plugin>`.include_last_point

Type	Default
bool	true

Description Whether to include the last pose in the trajectory.

Trajectory Critics

The trajectory critics listed below are inside the `dwb_critics` namespace.

BaseObstacleCritic

Scores a trajectory based on where the path passes over the costmap. To use this properly, you must use the inflation layer in costmap to expand obstacles by the robot's radius.

Parameters

`<dwb plugin>`: DWB plugin name defined in the **controller_plugin_ids** parameter in *Controller Server*.

`<name>`: BaseObstacleCritic critic name defined in the **<dwb plugin>.critics** parameter defined in *DWB Controller*.

`<dwb plugin>.<name>`.sum_scores

Type	Default
bool	false

Description Whether to allow for scores to be summed up.

`<dwb plugin>.<name>`.scale

Type	Default
double	1.0

Description Weighed scale for critic.

GoalAlignCritic

Scores a trajectory based on how well aligned the trajectory is with the goal pose.

Parameters

`<dwb plugin>`: DWB plugin name defined in the **controller_plugin_ids** parameter in *Controller Server*.

`<name>`: GoalAlignCritic critic name defined in the **<dwb plugin>.critics** parameter defined in *DWB Controller*.

`<dwb plugin>.<name>`.forward_point_distance

Type	Default
double	0.325

Description Point in front of robot to look ahead to compute angular change from.

`<dwb plugin>.<name>`.aggregation_type

Type	Default
string	"last"

Description last, sum, or product combination methods.

`<dwb plugin>.<name>`.scale

Type	Default
double	1.0

Description Weighed scale for critic.

GoalDistCritic

Scores a trajectory based on how close the trajectory gets the robot to the goal pose.

Parameters

`<dwb plugin>`: DWB plugin name defined in the **controller_plugin_ids** parameter in *Controller Server*.

`<name>`: GoalDistCritic critic name defined in the **<dwb plugin>.critics** parameter defined in *DWB Controller*.

`<dwb plugin>.<name>`.aggregation_type

Type	Default
string	"last"

Description last, sum, or product combination methods.

`<dwb plugin>.<name>`.scale

Type	Default
double	1.0

Description Weighed scale for critic.

ObstacleFootprintCritic

Scores a trajectory based on verifying all points along the robot's footprint don't touch an obstacle marked in the costmap.

Parameters

`<dwb plugin>`: DWB plugin name defined in the **controller_plugin_ids** parameter in *Controller Server*.

`<name>`: ObstacleFootprintCritic critic name defined in the **<dwb plugin>.critics** parameter defined in *DWB Controller*.

`<dwb plugin>.<name>`.sum_scores

Type	Default
bool	false

Description Whether to allow for scores to be summed up.

<dwb plugin>.<name>.scale

Type	Default
double	1.0

Description Weighed scale for critic.

OscillationCritic

Prevents the robot from just moving backwards and forwards.

Parameters

`<dwb plugin>`: DWB plugin name defined in the **controller_plugin_ids** parameter in *Controller Server*.

`<name>`: OscillationCritic critic name defined in the **<dwb plugin>.critics** parameter defined in *DWB Controller*.

<dwb plugin>.<name>.oscillation_reset_dist

Type	Default
double	0.05

Description Minimum distance to move to reset oscillation watchdog (m).

<dwb plugin>.<name>.oscillation_reset_angle

Type	Default
double	0.2

Description Minimum angular distance to move to reset watchdog (rad).

<dwb plugin>.<name>.oscillation_reset_time

Type	Default
double	-1

Description Duration when a reset may be called. If -1, cannot be reset..

<dwb plugin>.<name>.x_only_threshold

Type	Default
double	0.05

Description Threshold to check in the X velocity direction.

<dwb plugin>.<name>.scale

Type	Default
double	1.0

Description Weighed scale for critic.

PathAlignCritic

Scores a trajectory based on how well it is aligned to the path provided by the global planner.

Parameters

`<name>`: PathAlignCritic critic name defined in the **<dwb plugin>.critics** parameter defined in *DWB Controller*.

`<dwb plugin>`.<name>.forward_point_distance

Type	Default
double	0.325

Description Point in front of robot to look ahead to compute angular change from.

`<dwb plugin>`.<name>.aggregation_type

Type	Default
string	"last"

Description last, sum, or product combination methods.

`<dwb plugin>`.<name>.scale

Type	Default
double	1.0

Description Weighed scale for critic.

PathDistCritic

Scores a trajectory based on how well it is aligned to the path provided by the global planner.

Parameters

`<name>`: PathDistCritic critic name defined in the **<dwb plugin>.critics** parameter defined in *DWB Controller*.

`<dwb plugin>`.<name>.aggregation_type

Type	Default
string	"last"

Description last, sum, or product combination methods.

`<dwb plugin>`.<name>.scale

Type	Default
double	1.0

Description Weighed scale for critic.

PreferForwardCritic

Scores trajectories that move the robot forwards more highly.

Parameters

`<dwb plugin>`: DWB plugin name defined in the **controller_plugin_ids** parameter in *Controller Server*.

`<name>`: PreferForwardCritic critic name defined in the **<dwb plugin>.critics** parameter defined in *DWB Controller*.

`<dwb plugin>.<name>.penalty`

Type	Default
double	1.0

Description Penalty to apply to backward motion.

`<dwb plugin>.<name>.strafe_x`

Type	Default
double	0.1

Description Minimum X velocity before penalty.

`<dwb plugin>.<name>.strafe_theta`

Type	Default
double	0.2

Description Minimum angular velocity before applying penalty.

`<dwb plugin>.<name>.theta_scale`

Type	Default
double	10.0

Description Weight for angular velocity component.

`<dwb plugin>.<name>.scale`

Type	Default
double	1.0

Description Weighed scale for critic.

RotateToGoalCritic

Only allows the robot to rotate to the goal orientation when it is sufficiently close to the goal location.

Parameters

`<dwb plugin>`: DWB plugin name defined in the **controller_plugin_ids** parameter in *Controller Server*.

`<name>`: RotateToGoalCritic critic name defined in the **<dwb plugin>.critics** parameter defined in *DWB Controller*.

<dwb plugin>.xy_goal_tolerance

Type	Default
double	0.25

Description Tolerance to meet goal completion criteria (m).

<dwb plugin>.trans_stopped_velocity

Type	Default
double	0.25

Description Velocity below is considered to be stopped at tolerance met (rad/s).

<dwb plugin>.<name>.slowing_factor

Type	Default
double	5.0

Description Factor to slow robot motion by while rotating to goal.

<dwb plugin>.<name>.lookahead_time

Type	Default
double	-1

Description If > 0, amount of time to look forward for a collision for..

<dwb plugin>.<name>.scale

Type	Default
double	1.0

Description Weighed scale for critic.

TwirlingCritic

Prevents holonomic robots from spinning as they make their way to the goal.

Parameters

`<dwb plugin>`: DWB plugin name defined in the **controller_plugin_ids** parameter in *Controller Server*.

`<name>`: TwirlingCritic critic name defined in the **<dwb plugin>.critics** parameter defined in *DWB Controller*.

<dwb plugin>.<name>.scale

Type	Default
double	1.0

Description Weighed scale for critic.

Example

```yaml
controller_server:
  ros__parameters:
    # controller server parameters (see Controller Server for more info)
    use_sim_time: True
    controller_frequency: 20.0
    min_x_velocity_threshold: 0.001
    min_y_velocity_threshold: 0.5
    min_theta_velocity_threshold: 0.001
    progress_checker_plugins: ["progress_checker"] # progress_checker_plugin:
↪"progress_checker" For Humble and older
    goal_checker_plugins: ["goal_checker"]
    controller_plugins: ["FollowPath"]
    progress_checker:
      plugin: "nav2_controller::SimpleProgressChecker"
      required_movement_radius: 0.5
      movement_time_allowance: 10.0
    goal_checker:
      plugin: "nav2_controller::SimpleGoalChecker"
      xy_goal_tolerance: 0.25
      yaw_goal_tolerance: 0.25
      stateful: True
    # DWB controller parameters
    FollowPath:
      plugin: "dwb_core::DWBLocalPlanner"
      debug_trajectory_details: True
      min_vel_x: 0.0
      min_vel_y: 0.0
      max_vel_x: 0.26
      max_vel_y: 0.0
      max_vel_theta: 1.0
      min_speed_xy: 0.0
      max_speed_xy: 0.26
      min_speed_theta: 0.0
      acc_lim_x: 2.5
      acc_lim_y: 0.0
      acc_lim_theta: 3.2
      decel_lim_x: -2.5
      decel_lim_y: 0.0
      decel_lim_theta: -3.2
      vx_samples: 20
      vy_samples: 5
      vtheta_samples: 20
      sim_time: 1.7
      linear_granularity: 0.05
      angular_granularity: 0.025
      transform_tolerance: 0.2
      xy_goal_tolerance: 0.25
      trans_stopped_velocity: 0.25
      short_circuit_trajectory_evaluation: True
      stateful: True
      critics: ["RotateToGoal", "Oscillation", "BaseObstacle", "GoalAlign", "PathAlign
↪", "PathDist", "GoalDist"]
      BaseObstacle.scale: 0.02
```

(continues on next page)

(continued from previous page)

```
      PathAlign.scale: 32.0
      GoalAlign.scale: 24.0
      PathAlign.forward_point_distance: 0.1
      GoalAlign.forward_point_distance: 0.1
      PathDist.scale: 32.0
      GoalDist.scale: 24.0
      RotateToGoal.scale: 32.0
      RotateToGoal.slowing_factor: 5.0
      RotateToGoal.lookahead_time: -1.0
```

4.7.11 Regulated Pure Pursuit

Source code on Github.

The Regulated Pure Pursuit controller implements a variation on the Pure Pursuit controller that specifically targeting service / industrial robot needs. It regulates the linear velocities by curvature of the path to help reduce overshoot at high speeds around blind corners allowing operations to be much more safe. It also better follows paths than any other variation currently available of Pure Pursuit. It also has heuristics to slow in proximity to other obstacles so that you can slow the robot automatically when nearby potential collisions. It also implements the Adaptive lookahead point features to be scaled by velocities to enable more stable behavior in a larger range of translational speeds.

See the package's `README` for more complete information.

If you use the Regulated Pure Pursuit Controller algorithm or software from this repository, please cite this work in your papers:

- S. Macenski, S. Singh, F. Martin, J. Gines, Regulated Pure Pursuit for Robot Path Tracking. Autonomous Robots, 2023.

Regulated Pure Pursuit Parameters

desired_linear_vel

Type	Default
double	0.5

Description The desired maximum linear velocity (m/s) to use.

lookahead_dist

Type	Default
double	0.6

Description The lookahead distance (m) to use to find the lookahead point when `use_velocity_scaled_lookahead_dist` is `false`.

min_lookahead_dist

Type	Default
double	0.3

Description The minimum lookahead distance (m) threshold when `use_velocity_scaled_lookahead_dist` is `true`.

max_lookahead_dist

Type	Default
double	0.9

Description The maximum lookahead distance (m) threshold when `use_velocity_scaled_lookahead_dist` is `true`.

lookahead_time

Type	Default
double	1.5

Description The time (s) to project the velocity by when `use_velocity_scaled_lookahead_dist` is `true`. Also known as the lookahead gain.

rotate_to_heading_angular_vel

Type	Default
double	1.8

Description If `use_rotate_to_heading` is `true`, this is the angular velocity to use.

transform_tolerance

Type	Default
double	0.1

Description The TF transform tolerance (s).

use_velocity_scaled_lookahead_dist

Type	Default
bool	false

Description Whether to use the velocity scaled lookahead distances or constant `lookahead_distance`.

min_approach_linear_velocity

Type	Default
double	0.05

Description The minimum velocity (m/s) threshold to apply when approaching the goal to ensure progress. Must be `> 0.01`.

approach_velocity_scaling_dist

Type	Default
double	0.6

Description The distance (m) left on the path at which to start slowing down. Should be less than the half the costmap width.

use_collision_detection

Type	Default
bool	true

Description Whether to enable collision detection.

max_allowed_time_to_collision_up_to_carrot

Type	Default
double	1.0

Description The time (s) to project a velocity command forward to check for collisions when `use_collision_detection` is `true`. Pre-`Humble`, this was `max_allowed_time_to_collision`.

use_regulated_linear_velocity_scaling

Type	Default
bool	true

Description Whether to use the regulated features for path curvature (e.g. slow on high curvature paths).

use_cost_regulated_linear_velocity_scaling

Type	Default
bool	true

Description Whether to use the regulated features for proximity to obstacles (e.g. slow in close proximity to obstacles).

regulated_linear_scaling_min_radius

Type	Default
double	0.90

Description The turning radius (m) for which the regulation features are triggered when `use_regulated_linear_velocity_scaling` is `true`. Remember, sharper turns have smaller radii.

regulated_linear_scaling_min_speed

Type	Default
double	0.25

Description The minimum speed (m/s) for which any of the regulated heuristics can send, to ensure process is still achievable even in high cost spaces with high curvature. Must be `> 0.1`.

use_fixed_curvature_lookahead

Type	Default
bool	false

Description Whether to use a fixed lookahead distance to compute curvature from. Since a lookahead distance may be set to vary on velocity, it can introduce a reference cycle that can be problematic for large lookahead distances.

curvature_lookahead_dist

Type	Default
double	0.6

Description Distance to look ahead on the path to detect curvature.

use_rotate_to_heading

Type	Default
bool	true

Description Whether to enable rotating to rough heading and goal orientation when using holonomic planners. Recommended on for all robot types that can rotate in place.

Note: both `use_rotate_to_heading` and `allow_reversing` cannot be set to `true` at the same time as it would result in ambiguous situations.

allow_reversing

Type	Default
bool	false

Description Enables the robot to drive in the reverse direction, when the path planned involves reversing (which is represented by orientation cusps). Variants of the smac_planner comes with the support of reversing. Checkout the *Smac Planner* to know more.

Note: both `use_rotate_to_heading` and `allow_reversing` cannot be set to `true` at the same time as it would result in ambiguous situations.

rotate_to_heading_min_angle

Type	Default
double	0.785

Description The difference in the path orientation and the starting robot orientation (radians) to trigger a rotate in place, if `use_rotate_to_heading` is `true`.

max_angular_accel

Type	Default
double	3.2

Description Maximum allowable angular acceleration (rad/s/s) while rotating to heading, if `use_rotate_to_heading` is `true`.

max_robot_pose_search_dist

Type	Default
double	Local costmap max extent (max(width, height) / 2)

Description Upper bound on integrated distance along the global plan to search for the closest pose to the robot pose. This should be left as the default unless there are paths with loops and intersections that do not leave the local costmap, in which case making this value smaller is necessary to prevent shortcutting. If set to `-1`, it will use the maximum distance possible to search every point on the path for the nearest path point.

use_interpolation

Type	Default
bool	true

Description Enable linear interpolation between poses for lookahead point selection. Leads to smoother commanded linear and angular velocities.

Example

```
controller_server:
  ros__parameters:
    use_sim_time: True
    controller_frequency: 20.0
    min_x_velocity_threshold: 0.001
    min_y_velocity_threshold: 0.5
    min_theta_velocity_threshold: 0.001
    progress_checker_plugins: ["progress_checker"] # progress_checker_plugin:
↪"progress_checker" For Humble and older
    goal_checker_plugins: ["goal_checker"]
    controller_plugins: ["FollowPath"]

    progress_checker:
      plugin: "nav2_controller::SimpleProgressChecker"
      required_movement_radius: 0.5
      movement_time_allowance: 10.0
    goal_checker:
      plugin: "nav2_controller::SimpleGoalChecker"
      xy_goal_tolerance: 0.25
      yaw_goal_tolerance: 0.25
      stateful: True
    FollowPath:
      plugin: "nav2_regulated_pure_pursuit_controller::RegulatedPurePursuitController"
      desired_linear_vel: 0.5
      lookahead_dist: 0.6
      min_lookahead_dist: 0.3
      max_lookahead_dist: 0.9
      lookahead_time: 1.5
      rotate_to_heading_angular_vel: 1.8
      transform_tolerance: 0.1
      use_velocity_scaled_lookahead_dist: false
      min_approach_linear_velocity: 0.05
      approach_velocity_scaling_dist: 0.6
      use_collision_detection: true
      max_allowed_time_to_collision_up_to_carrot: 1.0
      use_regulated_linear_velocity_scaling: true
      use_fixed_curvature_lookahead: false
      curvature_lookahead_dist: 0.25
      use_cost_regulated_linear_velocity_scaling: false
      regulated_linear_scaling_min_radius: 0.9
      regulated_linear_scaling_min_speed: 0.25
      use_rotate_to_heading: true
      allow_reversing: false
      rotate_to_heading_min_angle: 0.785
      max_angular_accel: 3.2
      max_robot_pose_search_dist: 10.0
```

(continues on next page)

(continued from previous page)

```
    use_interpolation: false
```

4.7.12 Model Predictive Path Integral Controller

Source code on Github.

The MPPI Controller implements a Model Predictive Path Integral Controller. The new Nav2 MPPI Controller is a predictive controller - a successor to TEB and pure path tracking MPC controllers. It uses a sampling based approach to select optimal trajectories, optimizing between successive iterations. It contains plugin-based objective functions for customization and extension for various behaviors and behavioral attributes.

It works currently with Differential, Omnidirectional, and Ackermann robots. This controller is measured to run at 50+ Hz on a modest Intel processor (4th gen i5).

The MPPI algorithm is an MPC variant that finds a control velocity for the robot using an iterative approach. Using the previous time step's best control solution and the robot's current state, a set of randomly sampled perturbations from a Gaussian distribution are applied. These noised controls are forward simulated to generate a set of trajectories within the robot's motion model. Next, these trajectories are scored using a set of plugin-based critic functions to find the best trajectory in the batch. The output scores are used to set the best control with a soft max function. This process is then repeated a number of times and returns a converged solution. This solution is then used as the basis of the next time step's initial control.

A powerful result of this work is the ability to utilize objective functions which are not require to be convex nor differentiable, providing greater designer flexibility in behavior.

See the package's `README` for more complete information.

MPPI Parameters

motion_model

Type	Default
string	"DiffDrive"

Description The desired motion model to use for trajectory planning. Options are `DiffDrive`, `Omni`, or `Ackermann`. Differential drive robots may use forward/reverse and angular velocities; Omni add in lateral motion; and Ackermann adds minimum curvature constraints.

critics

Type	Default
string vector	N/A

Description A vector of critic plugin functions to use, without `mppi::critic::` namespace which will be automatically added on loading.

iteration_count

Type	Default
int	1

Description Iteration count in the MPPI algorithm. Recommended to remain as 1 and instead prefer larger batch sizes.

batch_size

Type	Default
int	1000

Description Count of randomly sampled candidate trajectories from current optimal control sequence in a given iteration. 1000 @ 50 Hz or 2000 @ 30 Hz seems to produce good results.

time_steps

Type	Default
int	56

Description Number of time steps (points) in candidate trajectories

model_dt

Type	Default
double	0.05

Description Length of each time step's `dt` timestep, in seconds. `time_steps * model_dt` is the prediction horizon.

vx_std

Type	Default
double	0.2

Description Sampling standard deviation for Vx

vy_std

Type	Default
double	0.2

Description Sampling standard deviation for Vy

wz_std

Type	Default
double	0.2

Description Sampling standard deviation for Wz (angular velocity)

vx_max

Type	Default
double	0.5

Description Target maximum forward velocity (m/s).

vy_max

Type	Default
double	0.5

Description Target maximum lateral velocity, if using `Omni` motion model (m/s).

vx_min

Type	Default
double	-0.35

Description Maximum reverse velocity (m/s).

wz_max

Type	Default
double	1.9

Description Maximum rotational velocity (rad/s).

temperature

Type	Default
double	0.3

Description Selectiveness of trajectories by their costs (The closer this value to 0, the "more" we take in consideration controls with less cost), 0 mean use control with best cost, huge value will lead to just taking mean of all trajectories without cost consideration.

gamma

Type	Default
double	0.015

Description A trade-off between smoothness (high) and low energy (low). This is a complex parameter that likely won't need to be changed from the default. See Section 3D-2 in "Information Theoretic Model Predictive Control: Theory and Applications to Autonomous Driving" for detailed information.

visualize

Type	Default
bool	false

Description Whether to publish debuggin trajectories for visualization. This can slow down the controller substantially (e.g. 1000 batches of 56 size every 30hz is alot of data).

retry_attempt_limit

Type	Default
int	1

Description Number of attempts to find feasible trajectory on failure for soft-resets before reporting total failure.

reset_period

Type	Default
double	1.0

Description Required time of inactivity to reset optimizer (only in Humble due to backport ABI policies).

Trajectory Visualization

trajectory_step

Type	Default
int	5

Description The step between trajectories to visualize to downsample candidate trajectory pool.

time_step

Type	Default
int	3

Description The step between points on trajectories to visualize to downsample trajectory density.

Path Handler

transform_tolerance

Type	Default
double	0.1

Description Time tolerance for data transformations with TF (s).

prune_distance

Type	Default
double	1.5

Description Distance ahead of nearest point on path to robot to prune path to (m).

max_robot_pose_search_dist

Type	Default
double	Costmap size / 2

Description Max integrated distance ahead of robot pose to search for nearest path point in case of path looping.

enforce_path_inversion

Type	Default
bool	false

Description If true, it will prune paths containing cusping points for segments changing directions (e.g. path inversions) such that the controller will be forced to change directions at or very near the planner's requested inversion point. This is targeting Smac Planner users with feasible paths who need their robots to switch directions where specifically requested.

inversion_xy_tolerance

Type	Default
double	0.2

Description Cartesian proximity (m) to path inversion point to be considered "achieved" to pass on the rest of the path after path inversion.

inversion_yaw_tolerance

Type	Default
double	0.4

Description Angular proximity (radians) to path inversion point to be considered "achieved" to pass on the rest of the path after path inversion. 0.4 rad = 23 deg.

Ackermann Motion Model

min_turning_r

Type	Default
double	0.2

Description The minimum turning radius possible for the vehicle platform (m).

Constraint Critic

cost_weight

Type	Default
double	4.0

Description Weight to apply to critic term.

cost_power

Type	Default
int	1

Description Power order to apply to term.

Goal Angle Critic

cost_weight

Type	Default
double	3.0

Description Weight to apply to critic term.

cost_power

Type	Default
int	1

Description Power order to apply to term.

threshold_to_consider

Type	Default
double	0.5

Description Minimal distance (m) between robot and goal above which angle goal cost considered.

Goal Critic

cost_weight

Type	Default
double	5.0

Description Weight to apply to critic term.

cost_power

Type	Default
int	1

Description Power order to apply to term.

threshold_to_consider

Type	Default
double	1.4

Description Minimal distance (m) between robot and goal above which goal distance cost considered. It is wise to start with this as being the same as your prediction horizon to have a clean hand-off with the path follower critic.

Obstacles Critic

critical_weight

Type	Default
double	20.0

Description Weight to apply to critic for near collisions closer than `collision_margin_distance` to prevent near collisions **only** as a method of virtually inflating the footprint. This should not be used to generally influence obstacle avoidance away from critical collisions.

repulsion_weight

Type	Default
double	1.5

Description Weight to apply to critic for generally preferring routes in lower cost space. This is separated from the critical term to allow for fine tuning of obstacle behaviors with path alignment for dynamic scenes without impacting actions which may directly lead to near-collisions. This is applied within the `inflation_radius` distance from obstacles.

cost_power

Type	Default
int	1

Description Power order to apply to term.

consider_footprint

Type	Default
bool	false

Description Whether to use point cost (if robot is circular or low compute power) or compute SE2 footprint cost.

collision_cost

Type	Default
double	10000.0

Description Cost to apply to a true collision in a trajectory.

collision_margin_distance

Type	Default
double	0.10

Description Margin distance (m) from collision to apply severe penalty, similar to footprint inflation. Between 0.05-0.2 is reasonable. Note that it will highly influence the controller not to enter spaces more confined than this, so ensure this parameter is set lower than the narrowest you expect the robot to need to traverse through.

near_goal_distance

Type	Default
double	0.50

Description Distance (m) near goal to stop applying preferential obstacle term to allow robot to smoothly converge to goal pose in close proximity to obstacles.

cost_scaling_factor

Type	Default
double	10.0

Description Exponential decay factor across inflation radius. This should be the same as for your inflation layer (Humble only)

inflation_radius

Type	Default
double	0.55

Description Radius to inflate costmap around lethal obstacles. This should be the same as for your inflation layer (Humble only)

Path Align Critic

cost_weight

Type	Default
double	10.0

Description Weight to apply to critic term.

cost_power

Type	Default
int	1

Description Power order to apply to term.

threshold_to_consider

Type	Default
double	0.5

Description Distance (m) between robot and goal to **stop** considering path alignment and allow goal critics to take over.

offset_from_furthest

Type	Default
int	20

Description Checks that the candidate trajectories are sufficiently far along their way tracking the path to apply the alignment critic. This ensures that path alignment is only considered when actually tracking the path, preventing awkward initialization motions preventing the robot from leaving the path to achieve the appropriate heading.

max_path_occupancy_ratio

Type	Default
double	0.07

Description Maximum proportion of the path that can be occupied before this critic is not considered to allow the obstacle and path follow critics to avoid obstacles while following the path's intent in presence of dynamic objects in the scene. Between 0-1 for 0-100%.

use_path_orientations

Type	Default
bool	false

Description Whether to consider path's orientations in path alignment, which can be useful when paired with feasible smac planners to incentivize directional changes only where/when the smac planner requests them. If you want the robot to deviate and invert directions where the controller sees fit, keep as false. If your plans do not contain orientation information (e.g. navfn), keep as false.

Path Angle Critic

cost_weight

Type	Default
double	2.0

Description Weight to apply to critic term.

cost_power

Type	Default
int	1

Description Power order to apply to term.

threshold_to_consider

Type	Default
double	0.5

Description Distance (m) between robot and goal to **stop** considering path angles and allow goal critics to take over.

offset_from_furthest

Type	Default
int	20

Description Number of path points after furthest one any trajectory achieves to compute path angle relative to.

max_angle_to_furthest

Type	Default
double	1.2

Description Angular distance (rad) between robot and goal above which path angle cost starts being considered

mode

Type	Default
int	0

Description Enum type for mode of operations for the path angle critic depending on path input types and behavioral desires. 0: Forward Preference, penalizes high path angles relative to the robot's orientation to incentivize turning towards the path. 1: No directional preference, penalizes high path angles relative to the robot's orientation or mirrored orientation (e.g. reverse), which ever is less, when a particular direction of travel is not preferable. 2: Consider feasible path orientation, when using a feasible path whereas the path points have orientation information (e.g. Smac Planners), consider the path's requested direction of travel to penalize path angles such that the robot will follow the path in the requested direction.

Path Follow Critic

cost_weight

Type	Default
double	5.0

Description Weight to apply to critic term.

cost_power

Type	Default
int	1

Description Power order to apply to term.

threshold_to_consider

Type	Default
double	1.4

Description Distance (m) between robot and goal to **stop** considering path following and allow goal critics to take over. It is wise to start with this as being the same as your prediction horizon to have a clean hand-off with the goal critic.

offset_from_furthest

Type	Default
int	6

Description Number of path points after furthest one any trajectory achieves to drive path tracking relative to.

Prefer Forward Critic

cost_weight

Type	Default
double	5.0

Description Weight to apply to critic term.

cost_power

Type	Default
int	1

Description Power order to apply to term.

threshold_to_consider

Type	Default
double	0.5

Description Distance (m) between robot and goal to **stop** considering preferring forward and allow goal critics to take over.

Twirling Critic

cost_weight

Type	Default
double	10.0

Description Weight to apply to critic term.

cost_power

Type	Default
int	1

Description Power order to apply to term.

Example

```
controller_server:
  ros__parameters:
    controller_frequency: 30.0
    FollowPath:
      plugin: "nav2_mppi_controller::MPPIController"
      time_steps: 56
      model_dt: 0.05
      batch_size: 2000
      vx_std: 0.2
      vy_std: 0.2
      wz_std: 0.4
      vx_max: 0.5
      vx_min: -0.35
      vy_max: 0.5
      wz_max: 1.9
      iteration_count: 1
      prune_distance: 1.7
      transform_tolerance: 0.1
      temperature: 0.3
      gamma: 0.015
      motion_model: "DiffDrive"
      visualize: false
      reset_period: 1.0 # (only in Humble)
      TrajectoryVisualizer:
        trajectory_step: 5
        time_step: 3
      AckermannConstrains:
        min_turning_r: 0.2
      critics: ["ConstraintCritic", "ObstaclesCritic", "GoalCritic", "GoalAngleCritic
↪", "PathAlignCritic", "PathFollowCritic", "PathAngleCritic", "PreferForwardCritic"]
      ConstraintCritic:
        enabled: true
        cost_power: 1
        cost_weight: 4.0
      GoalCritic:
        enabled: true
        cost_power: 1
```

(continues on next page)

(continued from previous page)

```
        cost_weight: 5.0
        threshold_to_consider: 1.4
      GoalAngleCritic:
        enabled: true
        cost_power: 1
        cost_weight: 3.0
        threshold_to_consider: 0.5
      PreferForwardCritic:
        enabled: true
        cost_power: 1
        cost_weight: 5.0
        threshold_to_consider: 0.5
      ObstaclesCritic:
        enabled: true
        cost_power: 1
        repulsion_weight: 1.5
        critical_weight: 20.0
        consider_footprint: false
        collision_cost: 10000.0
        collision_margin_distance: 0.1
        near_goal_distance: 0.5
        inflation_radius: 0.55 # (only in Humble)
        cost_scaling_factor: 10.0 # (only in Humble)
      PathAlignCritic:
        enabled: true
        cost_power: 1
        cost_weight: 14.0
        max_path_occupancy_ratio: 0.05
        trajectory_point_step: 3
        threshold_to_consider: 0.5
        offset_from_furthest: 20
        use_path_orientations: false
      PathFollowCritic:
        enabled: true
        cost_power: 1
        cost_weight: 5.0
        offset_from_furthest: 5
        threshold_to_consider: 1.4
      PathAngleCritic:
        enabled: true
        cost_power: 1
        cost_weight: 2.0
        offset_from_furthest: 4
        threshold_to_consider: 0.5
        max_angle_to_furthest: 1.0
        mode: 0
      # TwirlingCritic:
      #   enabled: true
      #   twirling_cost_power: 1
      #   twirling_cost_weight: 10.0
```

Notes to Users

General Words of Wisdom

The `model_dt` parameter generally should be set to the duration of your control frequency. So if your control frequency is 20hz, this should be `0.05`. However, you may also set it lower **but not larger**.

Visualization of the trajectories using `visualize` uses compute resources to back out trajectories for visualization and therefore slows compute time. It is not suggested that this parameter is set to `true` during a deployed use, but is a useful debug instrument while tuning the system, but use sparingly. Visualizing 2000 batches @ 56 points at 30 hz is *alot*.

The most common parameters you might want to start off changing are the velocity profiles (`vx_max`, `vx_min`, `wz_max`, and `vy_max` if holonomic) and the `motion_model` to correspond to your vehicle. Its wise to consider the `prune_distance` of the path plan in proportion to your maximum velocity and prediction horizon. The only deeper parameter that will likely need to be adjusted for your particular settings is the Obstacle critics' `repulsion_weight` since the tuning of this is proprtional to your inflation layer's radius. Higher radii should correspond to reduced `repulsion_weight` due to the penalty formation (e.g. `inflation_radius - min_dist_to_obstacle`). If this penalty is too high, the robot will slow significantly when entering cost-space from non-cost space or jitter in narrow corridors. It is noteworthy, but likely not necessary to be changed, that the Obstacle critic may use the full footprint information if `consider_footprint = true`, though comes at an increased compute cost.

Otherwise, the parameters have been closely pre-tuned by your friendly neighborhood navigator to give you a decent starting point that hopefully you only need to retune for your specific desired behavior lightly (if at all). Varying costmap parameters or maximum speeds are the actions which require the most attention, as described below:

Prediction Horizon, Costmap Sizing, and Offsets

As this is a predictive planner, there is some relationship between maximum speed, prediction times, and costmap size that users should keep in mind while tuning for their application. If a controller server costmap is set to 3.0m in size, that means that with the robot in the center, there is 1.5m of information on either side of the robot. When your prediction horizon (`time_steps * model_dt`) at maximum speed (`vx_max`) is larger than this, then your robot will be artificially limited in its maximum speeds and behavior by the costmap limitation. For example, if you predict forward 3 seconds (60 steps @ 0.05s per step) at 0.5m/s maximum speed, the **minimum** required costmap radius is 1.5m - or 3m total width.

The same applies to the Path Follow and Align offsets from furthest. In the same example if the furthest point we can consider is already at the edge of the costmap, then further offsets are thresholded because they're unusable. So its important while selecting these parameters to make sure that the theoretical offsets can exist on the costmap settings selected with the maximum prediction horizon and velocities desired. Setting the threshold for consideration in the path follower + goal critics as the same as your prediction horizon can make sure you have clean hand-offs between them, as the path follower will otherwise attempt to slow slightly once it reaches the final goal pose as its marker.

The Path Follow critic cannot drive velocities greater than the projectable distance of that velocity on the available path on the rolling costmap. The Path Align critic *offset_from_furthest* represents the number of path points a trajectory passes through while tracking the path. If this is set either absurdly low (e.g. 5) it can trigger when a robot is simply trying to start path tracking causing some suboptimal behaviors and local minima while starting a task. If it is set absurdly high (e.g. 50) relative to the path resolution and costmap size, then the critic may never trigger or only do so when at full-speed. A balance here is wise. A selection of this value to be ~30% of the maximum velocity distance projected is good (e.g. if a planner produces points every 2.5cm, 60 can fit on the 1.5m local costmap radius. If the max speed is 0.5m/s with a 3s prediction time, then 20 points represents 33% of the maximum speed projected over the prediction horizon onto the path). When in doubt, `prediction_horizon_s * max_speed / path_resolution / 3.0` is a good baseline.

Obstacle, Inflation Layer, and Path Following

There also exists a relationship between the costmap configurations and the Obstacle critic configurations. If the Obstacle critic is not well tuned with the costmap parameters (inflation radius, scale) it can cause the robot to wobble significantly as it attempts to take finitely lower-cost trajectories with a slightly lower cost in exchange for jerky motion. It may also perform awkward maneuvers when in free-space to try to maximize time in a small pocket of 0-cost over a more natural motion which involves moving into some low-costed region. Finally, it may generally refuse to go into costed space at all when starting in a free 0-cost space if the gain is set disproportionately higher than the Path Follow scoring to encourage the robot to move along the path. This is due to the critic cost of staying in free space becoming more attractive than entering even lightly costed space in exchange for progression along the task.

Thus, care should be taken to select weights of the obstacle critic in conjunction with the costmap inflation radius and scale so that a robot does not have such issues. How I (Steve, your friendly neighborhood navigator) tuned this was to first create the appropriate obstacle critic behavior desirable in conjunction with the inflation layer parameters. Its worth noting that the Obstacle critic converts the cost into a distance from obstacles, so the nature of the distribution of costs in the inflation isn't overly significant. However, the inflation radius and the scale will define the cost at the end of the distribution where free-space meets the lowest cost value within the radius. So testing for quality behavior when going over that threshold should be considered.

As you increase or decrease your weights on the Obstacle, you may notice the aforementioned behaviors (e.g. won't overcome free to non-free threshold). To overcome them, increase the FollowPath critic cost to increase the desire for the trajectory planner to continue moving towards the goal. Make sure to not overshoot this though, keep them balanced. A desirable outcome is smooth motion roughly in the center of spaces without significant close interactions with obstacles. It shouldn't be perfectly following a path yet nor should the output velocity be wobbling jaggedly.

Once you have your obstacle avoidance behavior tuned and matched with an appropriate path following penalty, tune the Path Align critic to align with the path. If you design exact-path-alignment behavior, its possible to skip the obstacle critic step as highly tuning the system to follow the path will give it less ability to deviate to avoid obstacles (though it'll slow and stop). Tuning the critic weight for the Obstacle critic high will do the job to avoid near-collisions but the repulsion weight is largely unnecessary to you. For others wanting more dynamic behavior, it *can* be beneficial to slowly lower the weight on the obstacle critic to give the path alignment critic some more room to work. If your path was generated with a cost-aware planner (like all provided by Nav2) and providing paths sufficiently far from obstacles for your satisfaction, the impact of a slightly reduced Obstacle critic with a Path Alignment critic will do you well. Not over-weighting the path align critic will allow the robot to deviate from the path to get around dynamic obstacles in the scene or other obstacles not previous considered during path planning. It is subjective as to the best behavior for your application, but it has been shown that MPPI can be an exact path tracker and/or avoid dynamic obstacles very fluidly and everywhere in between. The defaults provided are in the generally right regime for a balanced initial trade-off.

4.7.13 Rotation Shim Controller

Source code on Github.

The `nav2_rotation_shim_controller` will check the rough heading difference with respect to the robot and a newly received path. If within a threshold, it will pass the request onto the `primary_controller` to execute the task. If it is outside of the threshold, this controller will rotate the robot in place towards that path heading. Once it is within the tolerance, it will then pass off control-execution from this rotation shim controller onto the primary controller plugin. At this point, the robot's main plugin will take control for a smooth hand off into the task.

The `RotationShimController` is most suitable for:

- Robots that can rotate in place, such as differential and omnidirectional robots.
- Preference to rotate in place when starting to track a new path that is at a significantly different heading than the robot's current heading – or when tuning your controller for its task makes tight rotations difficult.

- Using planners that are non-kinematically feasible, such as NavFn, Theta*, or Smac 2D (Feasible planners such as Smac Hybrid-A* and State Lattice will start search from the robot's actual starting heading, requiring no rotation since their paths are guaranteed drivable by physical constraints).

See the package's `README` for more complete information.

Rotation Shim Controller Parameters

angular_dist_threshold

Type	Default
double	0.785

Description Maximum angular distance, in radians, away from the path heading to trigger rotation until within.

forward_sampling_distance

Type	Default
double	0.5

Description Forward distance, in meters, along path to select a sampling point to use to approximate path heading

rotate_to_heading_angular_vel

Type	Default
double	1.8

Description Angular rotational velocity, in rad/s, to rotate to the path heading

primary_controller

Type	Default
string	N/A

Description Internal controller plugin to use for actual control behavior after rotating to heading

max_angular_accel

Type	Default
double	3.2

Description Maximum angular acceleration for rotation to heading (rad/s/s)

simulate_ahead_time

Type	Default
double	1.0

Description Time in seconds to forward simulate a rotation command to check for collisions. If a collision is found, forwards control back to the primary controller plugin.

Example

```
controller_server:
  ros__parameters:
    use_sim_time: True
    controller_frequency: 20.0
    min_x_velocity_threshold: 0.001
    min_y_velocity_threshold: 0.5
    min_theta_velocity_threshold: 0.001
    progress_checker_plugins: ["progress_checker"] # progress_checker_plugin:
↪"progress_checker" For Humble and older
    goal_checker_plugins: ["goal_checker"]
    controller_plugins: ["FollowPath"]

    progress_checker:
      plugin: "nav2_controller::SimpleProgressChecker"
      required_movement_radius: 0.5
      movement_time_allowance: 10.0
    goal_checker:
      plugin: "nav2_controller::SimpleGoalChecker"
      xy_goal_tolerance: 0.25
      yaw_goal_tolerance: 0.25
      stateful: True
    FollowPath:
      plugin: "nav2_rotation_shim_controller::RotationShimController"
      primary_controller: "nav2_regulated_pure_pursuit_
↪controller::RegulatedPurePursuitController"
      angular_dist_threshold: 0.785
      forward_sampling_distance: 0.5
      rotate_to_heading_angular_vel: 1.8
      max_angular_accel: 3.2
      simulate_ahead_time: 1.0

      # Primary controller params can be placed here below
      # ...
```

4.7.14 Map Server / Saver

Source code on Github.

The Map Server implements the server for handling the map load requests for the stack and host a map topic. It also implements a map saver server which will run in the background and save maps based on service requests. There exists a map saver CLI similar to ROS 1 as well for a single map save.

Map Saver Parameters

save_map_timeout

Type	Default
int	2.0

Description Timeout to attempt saving the map (seconds).

free_thresh_default

Type	Default
double	0.25

Description Free space maximum probability threshold value for occupancy grid.

occupied_thresh_default

Type	Default
double	0.65

Description Occupied space minimum probability threshhold value for occupancy grid.

Map Server Parameters

yaml_filename

Type	Default
string	N/A

Description Path to map yaml file. Note from Rolling + Iron-Turtle forward: This parameter can set either from the yaml file or using the launch configuration parameter `map`. If we set it on launch commandline / launch configuration default, we override the yaml default. If you would like the specify your map file in yaml, remove the launch default so it is not overridden in Nav2's default launch files. Before Iron: `yaml_filename` must be set in the yaml (even if a bogus value) so that our launch scripts can overwrite it with launch values.

topic_name

Type	Default
string	"map"

Description Topic to publish loaded map to.

frame_id

Type	Default
string	"map"

Description Frame to publish loaded map in.

Costmap Filter Info Server Parameters

type

Type	Default
int	0

Description Type of costmap filter used. This is an enum for the type of filter this should be interpreted as. We provide the following pre-defined types:

- 0: keepout zones / preferred lanes filter
- 1: speed filter, speed limit is specified in % of maximum speed
- 2: speed filter, speed limit is specified in absolute value (m/s)

- 3: binary filter

filter_info_topic

Type	Default
string	"costmap_filter_info"

Description Topic to publish costmap filter information to.

mask_topic

Type	Default
string	"filter_mask"

Description Topic to publish filter mask to. The value of this parameter should be in accordance with `topic_name` parameter of Map Server tuned to filter mask publishing.

base

Type	Default
double	0.0

Description Base of `OccupancyGrid` mask value -> filter space value linear conversion which is being proceeded as: `filter_space_value = base + multiplier * mask_value`

multiplier

Type	Default
double	1.0

Description Multiplier of `OccupancyGrid` mask value -> filter space value linear conversion which is being proceeded as: `filter_space_value = base + multiplier * mask_value`

Example

```
map_server:
  ros__parameters:
    yaml_filename: "turtlebot3_world.yaml"
    topic_name: "map"
    frame_id: "map"

map_saver:
  ros__parameters:
    save_map_timeout: 5.0
    free_thresh_default: 0.25
    occupied_thresh_default: 0.65

costmap_filter_info_server:
  ros__parameters:
    type: 1
    filter_info_topic: "costmap_filter_info"
    mask_topic: "filter_mask"
    base: 0.0
    multiplier: 0.25
```

4.7.15 AMCL

Source code on Github.

AMCL implements the server for taking a static map and localizing the robot within it using an Adaptive Monte-Carlo Localizer.

Parameters

alpha1

Type	Default
double	0.2

Description Expected process noise in odometry's rotation estimate from rotation.

alpha2

Type	Default
double	0.2

Description Expected process noise in odometry's rotation estimate from translation.

alpha3

Type	Default
double	0.2

Description Expected process noise in odometry's translation estimate from translation.

alpha4

Type	Default
double	0.2

Description Expected process noise in odometry's translation estimate from rotation.

alpha5

Type	Default
double	0.2

Description For Omni models only: translation noise.

base_frame_id

Type	Default
string	"base_footprint"

Description Robot base frame.

beam_skip_distance

Type	Default
double	0.5

Description Ignore beams that most particles disagree with in Likelihood field model. Maximum distance to consider skipping for (m).

beam_skip_error_threshold

Type	Default
double	0.9

Description Percentage of beams after not matching map to force full update due to bad convergance.

beam_skip_threshold

Type	Default
double	0.3

Description Percentage of beams required to skip.

do_beamskip

Type	Default
bool	False

Description Whether to do beam skipping in Likelihood field model.

global_frame_id

Type	Default
string	“map”

Description The name of the coordinate frame published by the localization system.

lambda_short

Type	Default
double	0.1

Description Exponential decay parameter for z_short part of model.

laser_likelihood_max_dist

Type	Default
double	2.0

Description Maximum distance to do obstacle inflation on map, for use in likelihood_field model.

laser_max_range

Type	Default
double	100.0

Description Maximum scan range to be considered, -1.0 will cause the laser’s reported maximum range to be used.

laser_min_range

Type	Default
double	-1.0

Description Minimum scan range to be considered, -1.0 will cause the laser's reported minimum range to be used.

laser_model_type

Type	Default
string	"likelihood_field"

Description Which model to use, either beam, likelihood_field, or likelihood_field_prob. Same as likelihood_field but incorporates the beamskip feature, if enabled.

set_initial_pose

Type	Default
bool	False

Description Causes AMCL to set initial pose from the initial_pose* parameters instead of waiting for the initial_pose message.

initial_pose

Type	Default
Pose2D	{x: 0.0, y: 0.0, z: 0.0, yaw: 0.0}

Description X, Y, Z, and yaw coordinates of initial pose (meters and radians) of robot base frame in global frame.

max_beams

Type	Default
int	60

Description How many evenly-spaced beams in each scan to be used when updating the filter.

max_particles

Type	Default
int	2000

Description Maximum allowed number of particles.

min_particles

Type	Default
int	500

Description Minimum allowed number of particles.

odom_frame_id

Type	Default
string	"odom"

Description Which frame to use for odometry.

pf_err

Type	Default
double	0.05

Description Particle Filter population error.

pf_z

Type	Default
double	0.99

Description Particle filter population density.

recovery_alpha_fast

Type	Default
double	0.0

Description Exponential decay rate for the fast average weight filter, used in deciding when to recover by adding random poses. A good value might be 0.1.

recovery_alpha_slow

Type	Default
double	0.0

Description Exponential decay rate for the slow average weight filter, used in deciding when to recover by adding random poses. A good value might be 0.001.

resample_interval

Type	Default
int	1

Description Number of filter updates required before resampling.

robot_model_type

Type	Default
string	"nav2_amcl::DifferentialMotionModel"

Description The fully-qualified type of the plugin class. Options are "nav2_amcl::DifferentialMotionModel" and "nav2_amcl::OmniMotionModel". Users can also provide their own custom motion model plugin type.

Note for users of galactic and earlier The models are selectable by string key (valid options: "differential", "omnidirectional") rather than plugins.

save_pose_rate

Type	Default
double	0.5

Description Maximum rate (Hz) at which to store the last estimated pose and covariance to the parameter server, in the variables ~initial_pose_* and ~initial_cov_*. This saved pose will be used on subsequent runs to initialize the filter (-1.0 to disable).

sigma_hit

Type	Default
double	0.2

Description Standard deviation for Gaussian model used in z_hit part of the model.

tf_broadcast

Type	Default
bool	True

Description Set this to false to prevent amcl from publishing the transform between the global frame and the odometry frame.

transform_tolerance

Type	Default
double	1.0

Description Time with which to post-date the transform that is published, to indicate that this transform is valid into the future.

update_min_a

Type	Default
double	0.2

Description Rotational movement required before performing a filter update.

update_min_d

Type	Default
double	0.25

Description Translational movement required before performing a filter update.

z_hit

Type	Default
double	0.5

Description Mixture weight for z_hit part of model, sum of all used z weight must be 1. Beam uses all 4, likelihood model uses z_hit and z_rand..

z_max

Type	Default
double	0.05

Description Mixture weight for z_max part of model, sum of all used z weight must be 1. Beam uses all 4, likelihood model uses z_hit and z_rand.

z_rand

Type	Default
double	0.5

Description Mixture weight for z_rand part of model, sum of all used z weight must be 1. Beam uses all 4, likelihood model uses z_hit and z_rand..

z_short

Type	Default
double	0.005

Description Mixture weight for z_short part of model, sum of all used z weight must be 1. Beam uses all 4, likelihood model uses z_hit and z_rand.

always_reset_initial_pose

Type	Default
bool	False

Description Requires that AMCL is provided an initial pose either via topic or initial_pose* parameter (with parameter set_initial_pose: true) when reset. Otherwise, by default AMCL will use the last known pose to initialize.

scan_topic

Type	Default
string	scan

Description Laser scan topic to subscribe to.

map_topic

Type	Default
string	map

Description Map topic to subscribe to.

first_map_only

Type	Default
bool	False

Description Allows AMCL to accept maps more than once on the map_topic. This is especially useful when you're using the *LoadMap* service in *map_server*. Prior to Humble, this is `first_map_only_`.

Example

```yaml
amcl:
  ros__parameters:
    alpha1: 0.2
    alpha2: 0.2
    alpha3: 0.2
    alpha4: 0.2
    alpha5: 0.2
    base_frame_id: "base_footprint"
    beam_skip_distance: 0.5
    beam_skip_error_threshold: 0.9
```

(continues on next page)

(continued from previous page)

```
    beam_skip_threshold: 0.3
    do_beamskip: false
    global_frame_id: "map"
    lambda_short: 0.1
    laser_likelihood_max_dist: 2.0
    laser_max_range: 100.0
    laser_min_range: -1.0
    laser_model_type: "likelihood_field"
    max_beams: 60
    max_particles: 2000
    min_particles: 500
    odom_frame_id: "odom"
    pf_err: 0.05
    pf_z: 0.99
    recovery_alpha_fast: 0.0
    recovery_alpha_slow: 0.0
    resample_interval: 1
    robot_model_type: "nav2_amcl::DifferentialMotionModel"
    save_pose_rate: 0.5
    sigma_hit: 0.2
    tf_broadcast: true
    transform_tolerance: 1.0
    update_min_a: 0.2
    update_min_d: 0.25
    z_hit: 0.5
    z_max: 0.05
    z_rand: 0.5
    z_short: 0.05
    scan_topic: scan
    map_topic: map
    set_initial_pose: false
    always_reset_initial_pose: false
    first_map_only: false
    initial_pose:
      x: 0.0
      y: 0.0
      z: 0.0
      yaw: 0.0
```

4.7.16 Behavior Server

Source code on Github.

The Behavior Server implements the server for handling recovery behavior requests and hosting a vector of plugins implementing various C++ behaviors. It is also possible to implement independent behavior servers for each custom behavior, but this server will allow multiple behaviors to share resources such as costmaps and TF buffers to lower incremental costs for new behaviors.

Note: the wait recovery behavior has no parameters, the duration to wait is given in the action request. Note: pre-Rolling/Humble this was the Recovery server, not behavior server. Launch files, behaviors and tests were all renamed.

Behavior Server Parameters

local_costmap_topic

Type	Default
string	"local_costmap/costmap_raw"

Description Raw costmap topic for collision checking on the local costmap.

global_costmap_topic

Type	Default
string	"global_costmap/costmap_raw"

Description Raw costmap topic for collision checking on the global costmap.

local_footprint_topic

Type	Default
string	"local_costmap/published_footprint"

Description Topic for footprint in the local costmap frame.

global_footprint_topic

Type	Default
string	"global_costmap/published_footprint"

Description Topic for footprint in the global costmap frame.

cycle_frequency

Type	Default
double	10.0

Description Frequency to run behavior plugins.

transform_tolerance

Type	Default
double	0.1

Description TF transform tolerance.

local_frame

Type	Default
string	"odom"

Description Local reference frame.

global_frame

Type	Default
string	"map"

Description Global reference frame.

robot_base_frame

Type	Default
string	“base_link”

Description Robot base frame.

behavior_plugins

Type	Default
vector<string>	{“spin”, “back_up”, “drive_on_heading”, “wait”}

Description List of plugin names to use, also matches action server names.

Note Each plugin namespace defined in this list needs to have a `plugin` parameter defining the type of plugin to be loaded in the namespace.

Example:

```
behavior_server:
  ros__parameters:
    behavior_plugins: ["spin", "backup", "drive_on_heading", "wait"]
    spin:
      plugin: "nav2_behaviors/Spin"
    backup:
      plugin: "nav2_behaviors/BackUp"
    drive_on_heading:
      plugin: "nav2_behaviors/DriveOnHeading"
    wait:
      plugin: "nav2_behaviors/Wait"
```

Default Plugins

When the `behavior_plugins` parameter is not overridden, the following default plugins are loaded:

Namespace	Plugin
“spin”	“nav2_behaviors/Spin”
“backup”	“nav2_behaviors/BackUp”
“drive_on_heading”	“nav2_behaviors/DriveOnHeading”
“wait”	“nav2_behaviors/Wait”

Spin Behavior Parameters

Spin distance is given from the action request

simulate_ahead_time

Type	Default
double	2.0

Description Time to look ahead for collisions (s).

max_rotational_vel

Type	Default
double	1.0

Description Maximum rotational velocity (rad/s).

min_rotational_vel

Type	Default
double	0.4

Description Minimum rotational velocity (rad/s).

rotational_acc_lim

Type	Default
double	3.2

Description maximum rotational acceleration (rad/s^2).

BackUp Behavior Parameters

Backup distance, speed and time_allowance is given from the action request.

simulate_ahead_time

Type	Default
double	2.0

Description Time to look ahead for collisions (s).

DriveOnHeading Behavior Parameters

DriveOnHeading distance, speed and time_allowance is given from the action request.

simulate_ahead_time

Type	Default
double	2.0

Description Time to look ahead for collisions (s).

AssistedTeleop Behavior Parameters

AssistedTeleop time_allowance is given in the action request

projection_time

Type	Default
double	1.0

Description Time to look ahead for collisions (s).

simulation_time_step

Type	Default
double	0.1

Description Time step for projections (s).

cmd_vel_teleop

Type	Default
string	cmd_vel_teleop

Description Topic to listen for teleop messages.

Example

```
behavior_server:
  ros__parameters:
    local_costmap_topic: local_costmap/costmap_raw
    local_footprint_topic: local_costmap/published_footprint
    global_costmap_topic: global_costmap/costmap_raw
    global_footprint_topic: global_costmap/published_footprint
    cycle_frequency: 10.0
    behavior_plugins: ["spin", "backup", "drive_on_heading", "wait", "assisted_teleop
↪"]
    spin:
      plugin: "nav2_behaviors/Spin"
    backup:
      plugin: "nav2_behaviors/BackUp"
    drive_on_heading:
      plugin: "nav2_behaviors/DriveOnHeading"
    wait:
      plugin: "nav2_behaviors/Wait"
    assisted_teleop:
      plugin: "nav2_behaviors/AssistedTeleop"
    local_frame: odom
    global_frame: map
    robot_base_frame: base_link
    transform_timeout: 0.1
    simulate_ahead_time: 2.0
    max_rotational_vel: 1.0
    min_rotational_vel: 0.4
    rotational_acc_lim: 3.2
```

4.7.17 Smoother Server

Source code on Github.

The Smoother Server implements the server for handling smooth path requests and hosting a vector of plugins implementing various C++ smoothers. The server exposes an action interface for smoothing with multiple smoothers that share resources such as costmaps and TF buffers.

Smoother Server Parameters

costmap_topic

Type	Default
string	"global_costmap/costmap_raw"

Description Raw costmap topic for collision checking.

footprint_topic

Type	Default
string	"global_costmap/published_footprint"

Description Topic for footprint in the costmap frame.

transform_tolerance

Type	Default
double	0.1

Description TF transform tolerance.

robot_base_frame

Type	Default
string	"base_link"

Description Robot base frame.

smoother_plugins

Type	Default
vector<string>	{"nav2_smoother::SimpleSmoother"}

Description List of plugin names to use, also matches action server names.

Note Each plugin namespace defined in this list needs to have a `plugin` parameter defining the type of plugin to be loaded in the namespace.

Example:

```
smoother_server:
  ros__parameters:
    smoother_plugins: ["simple_smoother", "curvature_smoother"]
    curvature_smoother:
      plugin: "nav2_ceres_costaware_smoother/CeresCostawareSmoother"
    simple_smoother:
      plugin: "nav2_smoother::SimpleSmoother"
```

Example

```yaml
smoother_server:
  ros__parameters:
    costmap_topic: global_costmap/costmap_raw
    footprint_topic: global_costmap/published_footprint
    robot_base_frame: base_link
    transform_timeout: 0.1
    smoother_plugins: ["simple_smoother"]
    simple_smoother:
      plugin: "nav2_smoother::SimpleSmoother"
      tolerance: 1.0e-10
      do_refinement: True
```

4.7.18 Simple Smoother

Source code on Github.

The Simple Smoother is a Smoother Server plugin that will take in an input path and smooth it using a simple and fast smoothing technique. It weights the initial path points and the smoothed path points to create a balanced result where the path retains its high level characteristics but reduces oscillations or jagged features.

It is recommended this is paired ONLY with infeasible (e.g. 2D) planners, since this algorithm will break any kinematically feasible conditions. It is recommended users use the Constrained Smoother plugin instead with feasible plans.

Simple Smoother Parameters

tolerance

Type	Default
double	1.0e-10

Description Change in parameter values across path to terminate smoothing

do_refinement

Type	Default
bool	True

Description Whether to smooth the smoothed path recursively to refine the quality further

refinement_num

Type	Default
int	2

Description Number of times to recursively attempt to smooth, must be `>= 1`.

max_its

Type	Default
int	1000

Description Maximum number of iterations to attempt smoothing before termination

w_data

Type	Default
double	0.2

Description Weight to apply to path data given (bounds it)

w_smooth

Type	Default
double	0.3

Description Weight to apply to smooth the path (smooths it)

Example

```yaml
smoother_server:
  ros__parameters:
    costmap_topic: global_costmap/costmap_raw
    footprint_topic: global_costmap/published_footprint
    robot_base_frame: base_link
    transform_timeout: 0.1
    smoother_plugins: ["simple_smoother"]
    simple_smoother:
      plugin: "nav2_smoother::SimpleSmoother"
      tolerance: 1.0e-10
      do_refinement: True
      refinement_num: 2
      max_its: 1000
      w_data: 0.2
      w_smooth: 0.3
```

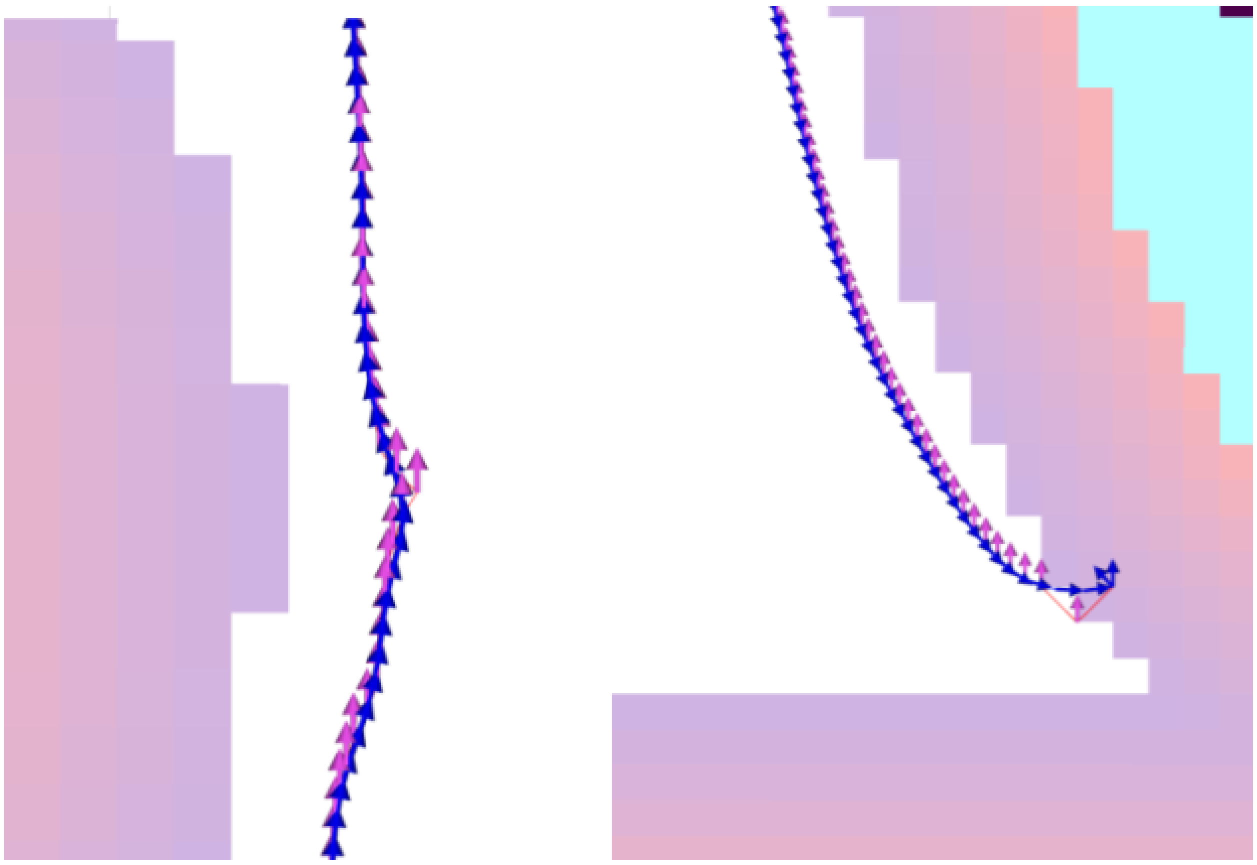
4.7.19 Savitzky-Golay Smoother

Source code on Github.

The Savitzky-Golay Smoother is a Smoother Server plugin that will take in an input path and smooth it using a simple and fast smoothing technique based on Savitzky Golay Filters. It uses a digital signal processing technique designed to reduce noise distorting a reference signal, in this case, a path.

It is useful for all types of planners, but particularly in NavFn to remove tiny artifacts that can occur near the end of paths or Theta* to slightly soften the transition between Line of Sight line segments **without** modifying the primary path. It is very fast (<< 1ms) so is a recommended default for planners that may result in slight discontinuities. However, it will not smooth out larger scale discontinuities, oscillations, or improve smoothness. For those, use one of the other provided smoother plugins. It also provides estimated orientation vectors of the path points after smoothing.

This algorithm is deterministic and low-parameter. In the below image, some odd points from NavFn's gradient descent are smoothed out by the smoother in the middle and end of a given path, while otherwise retaining the exact character of the path.



Savitzky-Golay Smoother Parameters

do_refinement

Type	Default
bool	True

Description Whether to smooth the smoothed results `refinement_num` times to get an improved result.

refinement_num

Type	Default
int	2

Description Number of times to recursively smooth a segment

Example

```
smoother_server:
  ros__parameters:
    costmap_topic: global_costmap/costmap_raw
    footprint_topic: global_costmap/published_footprint
    robot_base_frame: base_link
    transform_timeout: 0.1
    smoother_plugins: ["savitzky_golay_smoother"]
    savitzky_golay_smoother:
      plugin: "nav2_smoother::SavitzkyGolaySmoother"
      do_refinement: True
      refinement_num: 2
```

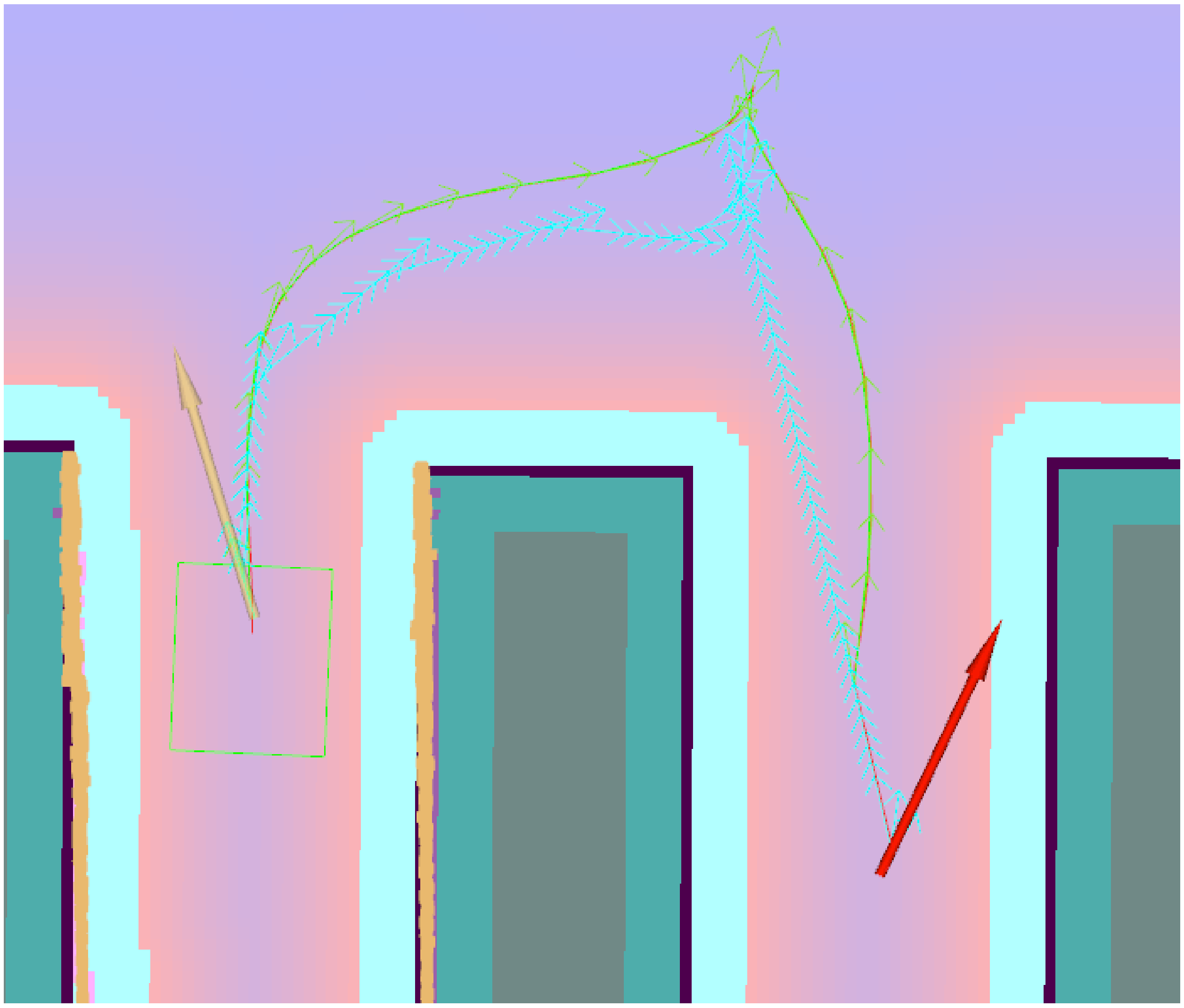
4.7.20 Constrained smoother

Source code on Github.

A smoother plugin for nav2_smoother based on the original deprecated smoother in nav2_smac_planner and put into operational state by RoboTech Vision. Suitable for applications which need planned global path to be pushed away from obstacles and/or for Reeds-Shepp motion models.

Important note: Constrained smoother uses a rather heavy optimization algorithm and thus is suggested to be used on a periodically truncated path. TruncatePathLocal BT Node can be used for achieving a proper path length and DistanceController BT Node can be used for achieving periodicity.

Following image depicts how Constrained Smoother can improve quality of an input path (cyan, generated by an outdated version of Smac Planner, intentionally not configured optimally to highlight the power of the smoother), increasing its smoothness and distance from obstacles. Resulting path is marked by green color. Note: last few path poses are not smoothed since TruncatePathLocal is used on this path.



Smoother Server Parameters

reversing_enabled

Type	Default
bool	true

Description Whether to detect forward/reverse direction and cusps. Should be set to false for paths without orientations assigned

path_downsampling_factor

Type	Default
int	1

Description Every n-th node of the path is taken for optimization. Useful for speed-up

path_upsampling_factor

Type	Default
int	1

Description Upsampling factor for refining. 0 - path remains downsampled (see `path_downsampling_factor`), 1 - path is upsampled back to original granularity using cubic bezier, 2… - more upsampling

keep_start_orientation

Type	Default
bool	true

Description Whether to prevent the start orientation from being smoothed

keep_goal_orientation

Type	Default
bool	true

Description Whether to prevent the goal orientation from being smoothed

minimum_turning_radius

Type	Default
double	0.4

Description Minimum turning radius the robot can perform. Can be set to 0.0 (or w_curve can be set to 0.0 with the same effect) for diff-drive/holonomic robots

w_curve

Type	Default
double	30.0

Description Weight to enforce minimum_turning_radius

w_dist

Type	Default
double	0.0

Description Weight to bind path to original as optional replacement for cost weight

w_smooth

Type	Default
double	2000000.0

Description Weight to maximize smoothness of path

w_cost

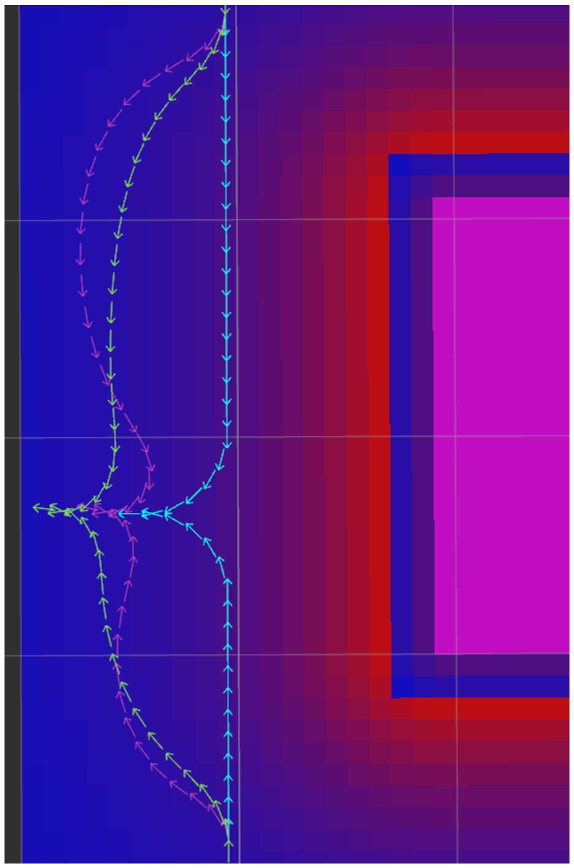
Type	Default
double	0.015

Description Weight to steer robot away from collision and cost

w_cost_cusp_multiplier

Type	Default
double	3.0

Description Option to use higher weight during forward/reverse direction change, helping optimizer to converge or add an extra obstacle avoidance at these problematic segments. Following image depicts improvement of the path with `w_cost_cusp_multiplier` (green) compared to one without it (purple). Original path has cyan color.



cusp_zone_length

Type	Default
double	2.5

Description Length of the section around cusp in which nodes use `w_cost_cusp_multiplier` (w_cost rises gradually inside the zone towards the cusp point, whose costmap weight eqals w_cost*w_cost_cusp_multiplier)

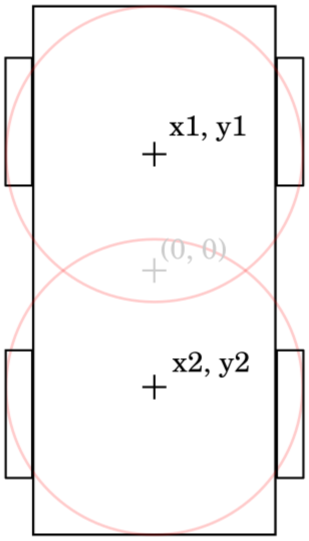
cost_check_points

Type	Default
array of double	[]

Description Points in robot frame to grab costmap values from. Format: [x1, y1, weight1, x2, y2, weight2, . . .].

IMPORTANT: Requires much higher number of optimizer iterations to actually improve the path. Use only if you really need it (highly elongated/asymmetric robots)

Following image depicts how two cost check points can be used to improve cost awareness of a rectangular robot.



optimizer.max_iterations

Type	Default
int	100

Description Maximum number of optimizer iterations

optimizer.debug_optimizer

Type	Default
bool	false

Description Whether to print optimizer debug info

optimizer.linear_solver_type

Type	Default
string	“SPARSE_NORMAL_CHOLESKY”

Description Linear solver type to be used by optimizer. Valid values are `SPARSE_NORMAL_CHOLESKY` and `DENSE_QR`

optimizer.gradient_tol

Type	Default
bool	1e-10

Description Gradient tolerance optimization termination criterion

optimizer.fn_tol

Type	Default
bool	1e-7

Description Function tolerance optimization termination criterion

optimizer.param_tol

Type	Default
bool	1e-15

Description Parameter tolerance optimization termination criterion

Example

```
smoother_server:
  ros__parameters:
    use_sim_time: True
    smoother_plugins: ["SmoothPath"]

    SmoothPath:
      plugin: "nav2_constrained_smoother/ConstrainedSmoother"
      reversing_enabled: true       # whether to detect forward/reverse direction and␣
↪cusps. Should be set to false for paths without orientations assigned
      path_downsampling_factor: 3   # every n-th node of the path is taken. Useful␣
↪for speed-up
      path_upsampling_factor: 1     # 0 - path remains downsampled, 1 - path is␣
↪upsampled back to original granularity using cubic bezier, 2... - more upsampling
      keep_start_orientation: true  # whether to prevent the start orientation from␣
↪being smoothed
      keep_goal_orientation: true   # whether to prevent the gpal orientation from␣
↪being smoothed
      minimum_turning_radius: 0.40  # minimum turning radius the robot can perform.␣
↪Can be set to 0.0 (or w_curve can be set to 0.0 with the same effect) for diff-
↪drive/holonomic robots
      w_curve: 30.0                 # weight to enforce minimum_turning_radius
      w_dist: 0.0                   # weight to bind path to original as optional␣
↪replacement for cost weight
      w_smooth: 2000000.0           # weight to maximize smoothness of path
      w_cost: 0.015                 # weight to steer robot away from collision and␣
↪cost

      # Parameters used to improve obstacle avoidance near cusps (forward/reverse␣
↪movement changes)
      w_cost_cusp_multiplier: 3.0   # option to use higher weight during forward/
↪reverse direction change which is often accompanied with dangerous rotations
      cusp_zone_length: 2.5         # length of the section around cusp in which␣
↪nodes use w_cost_cusp_multiplier (w_cost rises gradually inside the zone towards␣
↪the cusp point, whose costmap weight eqals w_cost*w_cost_cusp_multiplier)

      # Points in robot frame to grab costmap values from. Format: [x1, y1, weight1,␣
↪x2, y2, weight2, ...]
      # IMPORTANT: Requires much higher number of iterations to actually improve the␣
↪path. Uncomment only if you really need it (highly elongated/asymmetric robots)
      # cost_check_points: [-0.185, 0.0, 1.0]

      optimizer:
        max_iterations: 70            # max iterations of smoother
        debug_optimizer: false        # print debug info
        gradient_tol: 5e3
```

(continues on next page)

(continued from previous page)

```
        fn_tol: 1.0e-15
        param_tol: 1.0e-20
```

4.7.21 Velocity Smoother

Source code on Github.

The `nav2_velocity_smoother` is a package containing a lifecycle-component node for smoothing velocities sent by Nav2 to robot controllers. The aim of this package is to implement velocity, acceleration, and deadband smoothing from Nav2 to reduce wear-and-tear on robot motors and hardware controllers by smoothing out the accelerations/jerky movements that might be present with some local trajectory planners' control efforts.

See the package's README for more information.

Velocity Smoother Parameters

smoothing_frequency

Type	Default
double	20.0

Description Frequency (Hz) to use the last received velocity command to smooth by velocity, acceleration, and deadband constraints. If set approximately to the rate of your local trajectory planner, it should smooth by acceleration constraints velocity commands. If set much higher, it will interpolate and provide a smooth set of commands to the hardware controller.

scale_velocities

Type	Default
bool	false

Description Whether or not to adjust other components of velocity proportionally to a component's required changes due to acceleration limits. This will try to adjust all components to follow the same direction, but still enforces acceleration limits to guarantee compliance, even if it means deviating off commanded trajectory slightly.

feedback

Type	Default
string	"OPEN_LOOP"

Description Type of feedback to use for the current state of the robot's velocity. In `OPEN_LOOP`, it will use the last commanded velocity as the next iteration's current velocity. When acceleration limits are set appropriately, this is a good assumption. In `CLOSED_LOOP`, it will use the odometry from the `odom` topic to estimate the robot's current speed. In closed loop mode, it is important that the odometry is high rate and low latency, relative to the smoothing frequency.

max_velocity

Type	Default
vector<double>	[0.5, 0.0, 2.5]

Description Maximum velocities (m/s) in `[x, y, theta]` axes.

min_velocity

Type	Default
vector<double>	[-0.5, 0.0, -2.5]

Description Minimum velocities (m/s) in `[x, y, theta]` axes. This is **signed** and thus must be **negative** to reverse. Note: rotational velocities negative direction is a right-hand turn, so this should always be negative regardless of reversing preference.

deadband_velocity

Type	Default
vector<double>	[0.0, 0.0, 0.0]

Description Minimum velocities (m/s) to send to the robot hardware controllers, to prevent small commands from damaging hardware controllers if that speed cannot be achieved due to stall torque.

velocity_timeout

Type	Default
double	1.0

Description Timeout (s) after which the velocity smoother will send a zero-ed out `Twist` command and stop publishing.

max_accel

Type	Default
vector<double>	[2.5, 0.0, 3.2]

Description Maximum acceleration to apply to each axis `[x, y, theta]`.

max_decel

Type	Default
vector<double>	[-2.5, 0.0, -3.2]

Description Minimum acceleration to apply to each axis `[x, y, theta]`. This is **signed** and thus these should generally all be **negative**.

odom_topic

Type	Default
string	“odom”

Description Topic to find robot odometry, if in `CLOSED_LOOP` operational mode.

odom_duration

Type	Default
double	0.1

Description Time (s) to buffer odometry commands to estimate the robot speed, if in `CLOSED_LOOP` operational mode.

Example

```
velocity_smoother:
  ros__parameters:
    smoothing_frequency: 20.0
    scale_velocities: false
    feedback: "OPEN_LOOP"
    max_velocity: [0.5, 0.0, 2.5]
    min_velocity: [-0.5, 0.0, -2.5]
    deadband_velocity: [0.0, 0.0, 0.0]
    velocity_timeout: 1.0
    max_accel: [2.5, 0.0, 3.2]
    max_decel: [-2.5, 0.0, -3.2]
    odom_topic: "odom"
    odom_duration: 0.1
```

4.7.22 Collision Monitor

The Collision Monitor is a node providing an additional level of robot safety. It performs several collision avoidance related tasks using incoming data from the sensors, bypassing the costmap and trajectory planners, to monitor for and prevent potential collisions at the emergency-stop level.

This is analogous to safety sensor and hardware features; take in laser scans from a real-time certified safety scanner, detect if there is to be an imminent collision in a configurable bounding box, and either emergency-stop the certified robot controller or slow the robot to avoid such collision. However, this node is done at the CPU level with any form of sensor. As such, this does not provide hard real-time safety certifications, but uses the same types of techniques with the same types of data for users that do not have safety-rated laser sensors, safety-rated controllers, or wish to use any type of data input (e.g. pointclouds from depth or stereo or range sensors).

This is a useful and integral part of large heavy industrial robots, or robots moving with high velocities, around people or other dynamic agents (e.g. other robots) as a safety mechanism for high-response emergency stopping. The costmaps / trajectory planners will handle most situations, but this is to handle obstacles that virtually appear out of no where (from the robot's perspective) or approach the robot at such high speed it needs to immediately stop to prevent collision.

See the package's `README` for more complete information.

Features

The Collision Monitor uses polygons relative the robot's base frame origin to define "zones". Data that fall into these zones trigger an operation depending on the model being used. A given instance of the Collision Monitor can have many zones with different models at the same time. When multiple zones trigger at once, the most aggressive one is used (e.g. stop > slow 50% > slow 10%).

The following models of safety behaviors are employed by Collision Monitor:

- **Stop model**: Define a zone and a point threshold. If `min_points` or more obstacle points appear inside this area, stop the robot until the obstacles will disappear.
- **Slowdown model**: Define a zone around the robot and slow the maximum speed for a `slowdown_ratio`, if `min_points` or more points will appear inside the area.
- **Limit model**: Define a zone around the robot and restricts the maximum linear and angular velocities to `linear_limit` and `angular_limit` values accordingly, if `min_points` or more points will appear inside the area.

- **Approach model**: Using the current robot speed, estimate the time to collision to sensor data. If the time is less than `time_before_collision` seconds (0.5, 2, 5, etc…), the robot will slow such that it is now at least `time_before_collision` seconds to collision. The effect here would be to keep the robot always `time_before_collision` seconds from any collision.

The zones around the robot can take the following shapes:

- Arbitrary user-defined polygon relative to the robot base frame, which can be static in a configuration file or dynamically changing via a topic interface.
- Robot footprint polygon, which is used in the approach behavior model only. Will use the static user-defined polygon or the footprint topic to allow it to be dynamically adjusted over time.
- Circle: is made for the best performance and could be used in the cases where the zone or robot footprint could be approximated by round shape.

All shapes (`Polygon` and `Circle`) are derived from base `Polygon` class, so without loss of generality they would be called as "polygons". Subscribed footprint is also having the same properties as other polygons, but it is being obtained a footprint topic for the Approach Model.

The data may be obtained from different data sources:

- Laser scanners (`sensor_msgs::msg::LaserScan` messages)
- PointClouds (`sensor_msgs::msg::PointCloud2` messages)
- IR/Sonars (`sensor_msgs::msg::Range` messages)

Parameters

base_frame_id

Type	Default
string	"base_footprint"

Description: Robot base frame.

odom_frame_id

Type	Default
string	"odom"

Description: Which frame to use for odometry.

cmd_vel_in_topic

Type	Default
string	"cmd_vel_raw"

Description: Input `cmd_vel` topic with desired robot velocity.

cmd_vel_out_topic

Type	Default
string	"cmd_vel"

Description: Output `cmd_vel` topic with output produced by Collision Monitor velocities.

state_topic

Type	Default
string	“”

Description: Output the currently activated polygon action type and name. Optional parameter. No publisher will be created if it is unspecified.

transform_tolerance

Type	Default
double	0.1

Description Time with which to post-date the transform that is published, to indicate that this transform is valid into the future.

source_timeout

Type	Default
double	2.0

Description: Maximum time interval in which source data is considered as valid.

base_shift_correction

Type	Default
bool	True

Description: Whether to correct source data towards to base frame movement, considering the difference between current time and latest source time. If enabled, produces more accurate sources positioning in the robot base frame, at the cost of slower performance. This will cause average delays for `~1/(2*odom_rate)` per each `cmd_vel` calculation cycle. However, disabling this option for better performance is not recommended for the fast moving robots, where during the typical rate of data sources, robot could move unacceptably far. Thus reasonable odometry rates are recommended (~100 hz).

stop_pub_timeout

Type	Default
double	1.0

Description: Timeout, after which zero-velocity ceases to be published. It could be used for other overrode systems outside Nav2 are trying to bring the robot out of a state close to a collision, or to allow a standing robot to go into sleep mode.

polygons

Type	Default
vector<string>	N/A

Description: List of zones (stop/slowdown/limit bounding boxes, footprint, approach circle, etc…). Causes an error, if not specialized.

observation_sources

Type	Default
vector<string>	N/A

Description: List of data sources (laser scanners, pointclouds, etc…). Causes an error, if not specialized.

Polygons parameters

`<polygon name>` is the corresponding polygon name ID selected for this type.

`<polygon_name>`.type

Type	Default
string	N/A

Description: Type of polygon shape. Available values are `polygon`, `circle`. Causes an error, if not specialized.

`<polygon_name>`.points

Type	Default
vector<double>	N/A

Description: Polygon vertexes, listed in `{p1.x, p1.y, p2.x, p2.y, p3.x, p3.y, ...}` format (e.g. `{0.5, 0.25, 0.5, -0.25, 0.0, -0.25, 0.0, 0.25}` for the square in the front). Used for `polygon` type. Minimum 3 points for a triangle polygon. If not specified, the collision monitor will use dynamic polygon subscription to `polygon_sub_topic` for points in the `stop`/`slowdown`/`limit` action types, or footprint subscriber to `footprint_topic` for `approach` action type.

`<polygon_name>`.polygon_sub_topic

Type	Default
string	N/A

Description: Topic to listen the polygon points from. Applicable only for `polygon` type and `stop`/`slowdown`/`limit` action types. Causes an error, if not specified **and** points are also not specified. If both `points` and `polygon_sub_topic` are specified, the static `points` takes priority.

`<polygon_name>`.footprint_topic

Type	Default
string	"local_costmap/published_footprint"

Description: Topic to listen the robot footprint from. Applicable only for `polygon` type and `approach` action type. If both `points` and `footprint_topic` are specified, the static `points` takes priority.

`<polygon_name>`.radius

Type	Default
double	N/A

Description: Circle radius. Used for `circle` type. Causes an error, if not specialized.

`<polygon_name>`.action_type

Type	Default
string	N/A

Description: Zone behavior model. Available values are `stop`, `slowdown`, `limit`, `approach`. Causes an error, if not specialized.

`<polygon_name>`.min_points

Type	Default
int	4

Description: Minimum number of data readings within a zone to trigger the action. Former `max_points` parameter for Humble, that meant the maximum number of data readings within a zone to not trigger the action). `min_points` is equal to `max_points + 1` value.

`<polygon_name>`.slowdown_ratio

Type	Default
double	0.5

Description: Robot slowdown (share of its actual speed). Applicable for `slowdown` action type.

`<polygon_name>`.linear_limit

Type	Default
double	0.5

Description: Robot linear speed limit. Applicable for `limit` action type.

`<polygon_name>`.angular_limit

Type	Default
double	0.5

Description: Robot angular speed limit. Applicable for `limit` action type.

`<polygon_name>`.time_before_collision

Type	Default
double	2.0

Description: Time before collision in seconds. Maximum simulation time used in collision prediction. Higher values mean lower performance. Applicable for `approach` action type.

`<polygon_name>`.simulation_time_step

Type	Default
double	0.1

Description: Time iteration step for robot movement simulation during collision prediction. Higher values mean lower prediction accuracy but better performance. Applicable for `approach` action type.

`<polygon_name>`.visualize

Type	Default
bool	False

Description: Whether to publish the polygon in a separate topic.

`<polygon_name>`.polygon_pub_topic

Type	Default
string	<polygon_name>

Description: Topic name to publish a polygon to. Used only if `visualize` is true.

Observation sources parameters

`<source name>` is the corresponding data source name ID selected for this type.

`<source name>`.type

Type	Default
string	"scan"

Description: Type of polygon shape. Could be `scan`, `pointcloud` or `range`.

`<source name>`.topic

Type	Default
string	"scan"

Description: Topic to listen the source data from.

`<source name>`.min_height

Type	Default
double	0.05

Description: Minimum height the PointCloud projection to 2D space started from. Applicable for `pointcloud` type.

`<source name>`.max_height

Type	Default
double	0.5

Description: Maximum height the PointCloud projection to 2D space ended with. Applicable for `pointcloud` type.

`<source name>`.obstacles_angle

Type	Default
double	PI / 180 (1 degree)

Description: Angle increment (in radians) between nearby obstacle points at the range arc. Two outermost points from the field of view are not taken into account (they will always exist regardless of this value). Applicable for `range` type.

Example

Here is an example of configuration YAML for the Collision Monitor. For more information how to bring-up your own Collision Monitor node, please refer to the *Using Collision Monitor* tutorial.

```yaml
collision_monitor:
  ros__parameters:
    base_frame_id: "base_footprint"
    odom_frame_id: "odom"
    cmd_vel_in_topic: "cmd_vel_raw"
    cmd_vel_out_topic: "cmd_vel"
    state_topic: "collision_monitor_state"
    transform_tolerance: 0.5
    source_timeout: 5.0
    base_shift_correction: True
    stop_pub_timeout: 2.0
    polygons: ["PolygonStop", "PolygonSlow", "FootprintApproach"]
    PolygonStop:
      type: "circle"
      radius: 0.3
      action_type: "stop"
      min_points: 4  # max_points: 3 for Humble
      visualize: True
      polygon_pub_topic: "polygon_stop"
    PolygonSlow:
      type: "polygon"
      points: [1.0, 1.0, 1.0, -1.0, -0.5, -1.0, -0.5, 1.0]
      action_type: "slowdown"
      min_points: 4  # max_points: 3 for Humble
      slowdown_ratio: 0.3
      visualize: True
      polygon_pub_topic: "polygon_slowdown"
    PolygonLimit:
      type: "polygon"
      points: [0.5, 0.5, 0.5, -0.5, -0.5, -0.5, -0.5, 0.5]
      action_type: "limit"
      min_points: 4  # max_points: 3 for Humble
      linear_limit: 0.4
      angular_limit: 0.5
      visualize: True
      polygon_pub_topic: "polygon_limit"
    FootprintApproach:
      type: "polygon"
      action_type: "approach"
      footprint_topic: "/local_costmap/published_footprint"
      time_before_collision: 2.0
      simulation_time_step: 0.02
      min_points: 6  # max_points: 5 for Humble
      visualize: False
    observation_sources: ["scan", "pointcloud"]
    scan:
      type: "scan"
      topic: "/scan"
    pointcloud:
      type: "pointcloud"
      topic: "/intel_realsense_r200_depth/points"
      min_height: 0.1
      max_height: 0.5
```

4.7.23 Waypoint Follower

Source code on Github.

The Waypoint Follower module implements a way of doing waypoint following using the NavigateToPose action server. It will take in a set of ordered waypoints to follow and then try to navigate to them in order. It also hosts a waypoint task executor plugin which can be used to perform custom behavior at a waypoint like waiting for user instruction, taking a picture, or picking up a box. If a waypoint is not achievable, the `stop_on_failure` parameter will determine whether to continue to the next point or stop.

Parameters

stop_on_failure

Type	Default
bool	true

Description Whether to fail action task if a single waypoint fails. If false, will continue to next waypoint.

loop_rate

Type	Default
int	20

Description Rate to check for results from current navigation task.

waypoint_task_executor_plugin

Type	Default
string	'wait_at_waypoint'

Description A plugin to define tasks to be executed when robot arrives to a waypoint.

Note The plugin namespace defined needs to have a `plugin` parameter defining the type of plugin to be loaded in the namespace.

Example:

```yaml
waypoint_follower:
  ros__parameters:
    waypoint_task_executor_plugin: "wait_at_waypoint"
    wait_at_waypoint:
      plugin: "nav2_waypoint_follower::WaitAtWaypoint"
      enabled: True
      waypoint_pause_duration: 0
```

Provided Plugins

The plugins listed below are inside the `nav2_waypoint_follower` namespace.

WaitAtWaypoint

Lets robot to pause for a specified amount of time after reaching each waypoints.

Parameters

`<nav2_waypoint_follower plugin>`: nav2_waypoint_follower plugin name defined in the **waypoint_task_executor_plugin_id** parameter in *Waypoint Follower*.

`<nav2_waypoint_follower plugin>`.enabled

Type	Default
bool	true

Description Whether waypoint_task_executor plugin is enabled.

`<nav2_waypoint_follower plugin>`.waypoint_pause_duration

Type	Default
int	0

Description Amount of time in milliseconds, for robot to sleep/wait after each waypoint is reached. If zero, robot will directly continue to next waypoint.

PhotoAtWaypoint

Run-time plugin that takes photos at waypoint arrivals when using waypoint follower node. Saves the taken photos to specified directory. The name for taken photos are determined by the waypoint index and timestamp(seconds). For instance `/home/atas/0_1602582820.png` is an sample taken image, where `0_1602582820` is the file name determined by waypoint index and time stamp. The leading digit in file name implies the waypoint index and the rest of digits at righthand side imples the time stamp when the photo was taken.

Parameters

`<nav2_waypoint_follower plugin>`: nav2_waypoint_follower plugin name defined in the **waypoint_task_executor_plugin_id** parameter in *Waypoint Follower*.

`<nav2_waypoint_follower plugin>`.enabled

Type	Default
bool	true

Description Whether waypoint_task_executor plugin is enabled.

`<nav2_waypoint_follower plugin>`.camera_image_topic_name

Type	Default
string	"/camera/color/image_raw"

Description Camera image topic name to susbcribe

<nav2_waypoint_follower plugin>.save_images_dir

Type	Default
string	"/tmp/waypoint_images"

Description Path to directory to save taken photos at waypoint arrivals.

<nav2_waypoint_follower plugin>.image_format

Type	Default
string	"png"

Description Desired image format.

InputAtWaypoint

Lets robot to wait for external input, with timeout, at a waypoint.

Parameters

`<nav2_waypoint_follower plugin>`: nav2_waypoint_follower plugin name defined in the **waypoint_task_executor_plugin_id** parameter in *Waypoint Follower*.

<nav2_waypoint_follower plugin>.enabled

Type	Default
bool	true

Description Whether waypoint_task_executor plugin is enabled.

<nav2_waypoint_follower plugin>.timeout

Type	Default
double	10.0

Description Amount of time in seconds to wait for user input before moving on to the next waypoint.

<nav2_waypoint_follower plugin>.input_topic

Type	Default
string	"input_at_waypoint/input"

Description Topic input is published to to indicate to move to the next waypoint, in *std_msgs/Empty*.

Default Plugin

Namespace	Plugin
“wait_at_waypoint”	“nav2_waypoint_follower::WaitAtWaypoint”

Example

```yaml
waypoint_follower:
  ros__parameters:
    loop_rate: 20
    stop_on_failure: false
    waypoint_task_executor_plugin: "wait_at_waypoint"
      wait_at_waypoint:
        plugin: "nav2_waypoint_follower::WaitAtWaypoint"
        enabled: True
        waypoint_pause_duration: 0
```

4.8 Tuning Guide

This guide is meant to assist users in tuning their navigation system. While *Configuration Guide* is the home of the list of parameters for all of Nav2, it doesn’t contain much *color* for how to tune a system using the most important of them. The aim of this guide is to give more advice in how to setup your system beyond a first time setup, which you can find at *First-Time Robot Setup Guide*. This will by no means cover all of the parameters (so please, do review the configuration guides for the packages of interest), but will give some helpful hints and tips.

This tuning guide is a perpetual work in progress. If you see some insights you have missing, please feel free to file a ticket or pull request with the wisdom you would like to share. This is an open section supported by the generosity of Nav2 users and maintainers. Please consider paying it forward.

4.8.1 Inflation Potential Fields

Many users and ecosystem navigation configuration files the maintainers find are really missing the point of the inflation layer. While it’s true that you can simply inflate a small radius around the walls to weight against critical collisions, the *true* value of the inflation layer is creating a consistent potential field around the entire map.

Some of the most popular tuning guides for ROS Navigation / Nav2 even call this out specifically that there’s substantial benefit to creating a gentle potential field across the width of the map - after inscribed costs are applied - yet very few users do this in practice.

This habit actually results in paths produced by NavFn, Theta*, and Smac Planner to be somewhat suboptimal. They really want to look for a smooth potential field rather than wide open 0-cost spaces in order to stay in the middle of spaces and deal with close-by moving obstacles better. It will allow search to be weighted towards freespace far before the search algorithm runs into the obstacle that the inflation is caused by, letting the planner give obstacles as wide of a berth as possible.

So it is the maintainers’ recommendation, as well as all other cost-aware search planners available in ROS, to increase your inflation layer cost scale and radius in order to adequately produce a smooth potential across the entire map. For very large open spaces, its fine to have 0-cost areas in the middle, but for halls, aisles, and similar; **please create a smooth potential to provide the best performance**.

4.8.2 Robot Footprint vs Radius

Nav2 allows users to specify the robot's shape in 2 ways: a geometric `footprint` or the radius of a circle encompassing the robot. In ROS (1), it was pretty reasonable to always specify a radius approximation of the robot, since the global planning algorithms didn't use it and the local planners / costmaps were set up with the circular assumption baked in.

However, in Nav2, we now have multiple planning and controller algorithms that make use of the full SE2 footprint. If your robot is non-circular, it is recommended that you give the planners and controllers the actual, geometric footprint of your robot. This will allow the planners and controllers to plan or create trajectories into tighter spaces. For example, if you have a very long but skinny robot, the circular assumption wouldn't allow a robot to plan into a space only a little wider than your robot, since the robot would not fit length-wise.

The kinematically feasible planners (e.g. Smac Hybrid-A*, Smac State Lattice) will use the SE2 footprint for collision checking to create kinematically feasible plans, if provided with the actual footprint. As of December, 2021 all of the controller plugins support full footprint collision checking to ensure safe path tracking. If you provide a footprint of your robot, it will be used to make sure trajectories are valid and it is recommended you do so. It will prevent a number of "stuck robot" situations that could have been easily avoided.

If your robot is truly circular, continue to use the `robot_radius` parameter. The three valid reasons for a non-circular robot to use the radius instead:

- The robot is very small relative to the environment (e.g. RC car in a warehouse)
- The robot has such limited compute power, using SE2 footprint checking would add too much computational burden (e.g. embedded micro-processors)
- If you plan to use a holonomic planner (e.g. Theta*, Smac 2D-A*, or NavFn), you may continue to use the circular footprint, since these planners are not kinematically feasible and will not make use of the SE2 footprint anyway.

4.8.3 Rotate in Place Behavior

Using the *Rotation Shim Controller*, a robot will simply rotate in place before starting to track a holonomic path. This allows a developer to tune a controller plugin to be optimized for path tracking and give you clean rotations, out of the box.

This was added due to quirks in some existing controllers whereas tuning the controller for a task can make it rigid – or the algorithm simply doesn't rotate in place when working with holonomic paths (if that's a desirable trait). The result is an awkward, stuttering, or whipping around behavior when your robot's initial and path heading's are significantly divergent. Giving a controller a better starting point to start tracking a path makes tuning the controllers significantly easier and creates more intuitive results for on-lookers (in one maintainer's opinion).

Note: If using a non-holonomic, kinematically feasible planner (e.g. Smac Hybrid-A*, Smac State Lattice), this is not a necessary behavioral optimization. This class of planner will create plans that take into account the robot's starting heading, not requiring any rotation behaviors.

This behavior is most optimially for:

- Robots that can rotate in place, such as differential and omnidirectional robots.
- Preference to rotate in place when starting to track a new path that is at a significantly different heading than the robot's current heading – or when tuning your controller for its task makes tight rotations difficult.

4.8.4 Planner Plugin Selection

Nav2 provides a number of planning plugins out of the box. For a first-time setup, see *Setting Up Navigation Plugins* for a more verbose breakdown of algorithm styles within Nav2, and *Navigation Plugins* for a full accounting of the current list of plugins available (which may be updated over time).

In general though, the following table is a good guide for the optimal planning plugin for different types of robot bases:

Plugin Name	Supported Robot Types
NavFn Planner	Circular Differential, Circular Omnidirectional
Smac Planner 2D	
Theta Star Planner	
Smac Hybrid-A* Planner	Non-circular or Circular Ackermann, Non-circular or Circular Legged
Smac Lattice Planner	Non-circular Differential, Non-circular Omnidirectional, Arbitrary

If you are using a non-circular robot with very limited compute, it may be worth assessing the benefits of using one of the holonomic planners (e.g. particle assumption planners). It is the recommendation of the maintainers to start using one of the more advanced algorithms appropriate for your platform *first*, but to scale back the planner if need be. The run-time of the feasible planners are typically on par (or sometimes faster) than their holonomic counterparts, so don't let the more recent nature of them fool you.

Since the planning problem is primarily driven by the robot type, the table accurately summarizes the advice to users by the maintainers. Within the circular robot regime, the choice of planning algorithm is dependent on application and desirable behavior. NavFn will typically make broad, sweeping curves; Theta* prefers straight lines and supports them at any angle; and Smac 2D is essentially a classical A* algorithm with cost-aware penalties.

Note: These are simply the default and available plugins from the community. For a specific application / platform, you may also choose to use none of these and create your own, and that's the intention of the Nav2 framework. See the *Writing a New Planner Plugin* tutorial for more details. If you're willing to contribute this work back to the community, please file a ticket or contact a maintainer! They'd love to hear from you.

4.8.5 Controller Plugin Selection

Nav2 provides a number of controller plugins out of the box. For a first-time setup, see *Setting Up Navigation Plugins* for a more verbose breakdown of algorithm styles within Nav2, and *Navigation Plugins* for a full accounting of the current list of plugins available (which may be updated over time).

In general though, the following table is a good first-order description of the controller plugins available for different types of robot bases:

Plugin Name	Supported Robot Types	Task
DWB controller	Differential, Omnidirectional	Dynamic obstacle avoidance
MPPI Controller	Differential, Omnidirectional, Ackermann, Legged	Dynamic obstacle avoidance
RPP controller	Differential, Ackermann, Legged	Exact path following
Rotation Shim	Differential, Omnidirectional	Rotate to rough heading

All of the above controllers can handle both circular and arbitrary shaped robots in configuration.

Regulated Pure Pursuit is good for exact path following and is typically paired with one of the kinematically feasible planners (eg State Lattice, Hybrid-A*, etc) since those paths are known to be drivable given hard physical constraints. However, it can also be applied to differential drive robots who can easily pivot to match any holonomic path. This is the plugin of choice if you simply want your robot to follow the path, rather exactly, without any dynamic obstacle avoidance or deviation. It is simple and geometric, as well as slowing the robot in the presence of near-by obstacles *and* while making sharp turns.

DWB and MPPI are both options that will track paths, but also diverge from the path if there are dynamic obstacles present (in order to avoid them). DWB does this through scoring multiple trajectories on a set of critics. These trajectories are also generated via plugins that can be replaced, but support out of the box Omni and Diff robot types within the valid velocity and acceleration restrictions. These critics are plugins that can be selected at run-time and contain weights that may be tuned to create the desired behavior, such as minimizing path distance, minimizing distance to the goal or headings, and other action penalties that can be designed. This does require a bit of tuning for a given platform, application, and desired behavior, but it is possible to tune DWB to do nearly any single thing well.

MPPI on the other hand implements an optimization based approach, using randomly perturbed samples of the previous optimal trajectory to maximize a set of plugin-based objective functions. In that regard, it is similar to DWB however MPPI is a far more modern and advanced technique that will deal with dynamic agents in the environment and create intelligent behavior due to the optimization based trajectory planning, rather then DWB's constant action model. MPPI however does have moderately higher compute costs, but it is highly recommended to go this route and has received considerable development resources and attention due to its power. This typically works pretty well out of the box, but to tune for specific behaviors, you may have to retune some of the parameters. The README.md file for this package contains details on how to tune it efficiently.

Finally, the Rotation Shim Plugin helps assist plugins like TEB and DWB (among others) to rotate the robot in place towards a new path's heading before starting to track the path. This allows you to tune your local trajectory planner to operate with a desired behavior without having to worry about being able to rotate on a dime with a significant deviation in angular distance over a very small euclidean distance. Some controllers when heavily tuned for accurate path tracking are constrained in their actions and don't very cleanly rotate to a new heading. Other controllers have a 'spiral out' behavior because their sampling requires some translational velocity, preventing it from simply rotating in place. This helps alleviate that problem and makes the robot rotate in place very smoothly.

Note: These are simply the default and available plugins from the community. For a specific robot platform / company, you may also choose to use none of these and create your own. See the *Writing a New Controller Plugin* tutorial for more details. If you're willing to contribute this work back to the community, please file a ticket or contact a maintainer! They'd love to hear from you.

4.8.6 Caching Obstacle Heuristic in Smac Planners

Smac's Hybrid-A* and State Lattice Planners provide an option, `cache_obstacle_heuristic`. This can be used to cache the heuristic to use between replannings to the same goal pose, which can increase the speed of the planner **significantly** (40-300% depending on many factors). The obstacle heuristic is used to steer the robot into the middle of spaces, respecting costs, and drives the kinematically feasible search down the corridors towards a valid solution. Think of it like a 2D cost-aware search to "prime" the planner about where it should go when it needs to expend more effort in the fully feasible search / SE2 collision checking.

This is useful to speed up performance to achieve better replanning speeds. However, if you cache this heuristic, it will not be updated with the most current information in the costmap to steer search. During planning, the planner will still make use of the newest cost information for collision checking, *thusly this will not impact the safety of the path*. However, it may steer the search down newly blocked corridors or guide search towards areas that may have new dynamic obstacles in them, which can slow things down significantly if entire solution spaces are blocked.

Therefore, it is the recommendation of the maintainers to enable this only when working in largely static (e.g. not many moving things or changes, not using live sensor updates in the global costmap, etc) environments when planning across large spaces to singular goals. Between goal changes to Nav2, this heuristic will be updated with the most current set of information, so it is not very helpful if you change goals very frequently.

4.8.7 Nav2 Launch Options

Nav2's launch files are made to be very configurable. Obviously for any serious application, a user should use `nav2_bringup` as the basis of their navigation launch system, but should be moved to a specific repository for a users' work. A typical thing to do is to have a `<robot_name>_nav` configuration package containing the launch and parameter files.

Within `nav2_bringup`, there is a main entryfile `tb3_simulation_launch.py`. This is the main file used for simulating the robot and contains the following configurations:

- `slam` : Whether or not to use AMCL or SLAM Toolbox for localization and/or mapping. Default `false` to AMCL.
- `map` : The filepath to the map to use for navigation. Defaults to `map.yaml` in the package's `maps/` directory.
- `world` : The filepath to the world file to use in simulation. Defaults to the `worlds/` directory in the package.
- `params_file` : The main navigation configuration file. Defaults to `nav2_params.yaml` in the package's `params/` directory.
- `autostart` : Whether to autostart the navigation system's lifecycle management system. Defaults to `true` to transition up the Nav2 stack on creation to the activated state, ready for use.
- `use_composition` : Whether to launch each Nav2 server into individual processes or in a single composed node, to leverage savings in CPU and memory. Default `true` to use single process Nav2.
- `use_respawn` : Whether to allow server that crash to automatically respawn. When also configured with the lifecycle manager, the manager will transition systems back up if already activated and went down due to a crash. Only works in non-composed bringup since all of the nodes are in the same process / container otherwise.
- `use_sim_time` : Whether to set all the nodes to use simulation time, needed in simulation. Default `true` for simulation.
- `rviz_config_file` : The filepath to the rviz configuration file to use. Defaults to the `rviz/` directory's file.
- `use_simulator` : Whether or not to start the Gazebo simulator with the Nav2 stack. Defaults to `true` to launch Gazebo.
- `use_robot_state_pub` : Whether or not to start the robot state publisher to publish the robot's URDF transformations to TF2. Defaults to `true` to publish the robot's TF2 transformations.
- `use_rviz` : Whether or not to launch rviz for visualization. Defaults to `true` to show rviz.
- `headless` : Whether or not to launch the Gazebo front-end alongside the background Gazebo simulation. Defaults to `true` to display the Gazebo window.
- `namespace` : The namespace to launch robots into, if need be.
- `use_namespace` : Whether or not to launch robots into this namespace. Default `false` and uses global namespace for single robot.
- `robot_name` : The name of the robot to launch.
- `robot_sdf` : The filepath to the robot's gazebo configuration file containing the Gazebo plugins and setup to simulate the robot system.
- `x_pose`, `y_pose`, `z_pose`, `roll`, `pitch`, `yaw` : Parameters to set the initial position of the robot in the simulation.

4.8.8 Other Pages We'd Love To Offer

If you are willing to chip in, some ideas are in https://github.com/ros-planning/navigation.ros.org/issues/204, but we'd be open to anything you think would be insightful!

4.9 Nav2 Behavior Trees

4.9.1 Introduction To Nav2 Specific Nodes

Warning:

Vocabulary can be a large point of confusion here when first starting out.

- A `Node` when discussing BTs is entirely different than a `Node` in the ROS 2 context
- An `ActionNode` in the context of BTs is not necessarily connected to an Action Server in the ROS 2 context (but often it is)

There are quite a few custom Nav2 BT nodes that are provided to be used in the Nav2 specific fashion. Some commonly used Nav2 nodes will be described below. The full list of custom BT nodes can be found in the nav2_behavior_tree plugins folder. The configuration guide can also be quite useful.

Action Nodes

- ComputePathToPose - ComputePathToPose Action Server Client (Planner Interface)
- FollowPath - FollowPath Action Server Client (Controller Interface)
- Spin, Wait, Backup - Behaviors Action Server Client
- ClearCostmapService - ClearCostmapService Server Clients

Upon completion, these action nodes will return `SUCCESS` if the action server believes the action has been completed correctly, `RUNNING` when still running, and will return `FAILURE` otherwise. Note that in the above list, the *ClearCostmapService* action node is *not* an action server client, but a service client.

Condition Nodes

- GoalUpdated - Checks if the goal on the goal topic has been updated
- GoalReached - Checks if the goal has been reached
- InitialPoseReceived - Checks to see if a pose on the `intial_pose` topic has been received
- isBatteryLow - Checks to see if the battery is low by listening on the battery topic

The above list of condition nodes can be used to probe particular aspects of the system. Typically they will return `SUCCESS` if the condition is true and `FAILURE` otherwise. The key condition that is used in the default Nav2 BT is `GoalUpdated` which is checked asynchronously within particular subtrees. This condition node allows for the behavior described as "If the goal has been updated, then we must replan". Condition nodes are typically paired with ReactiveFallback nodes.

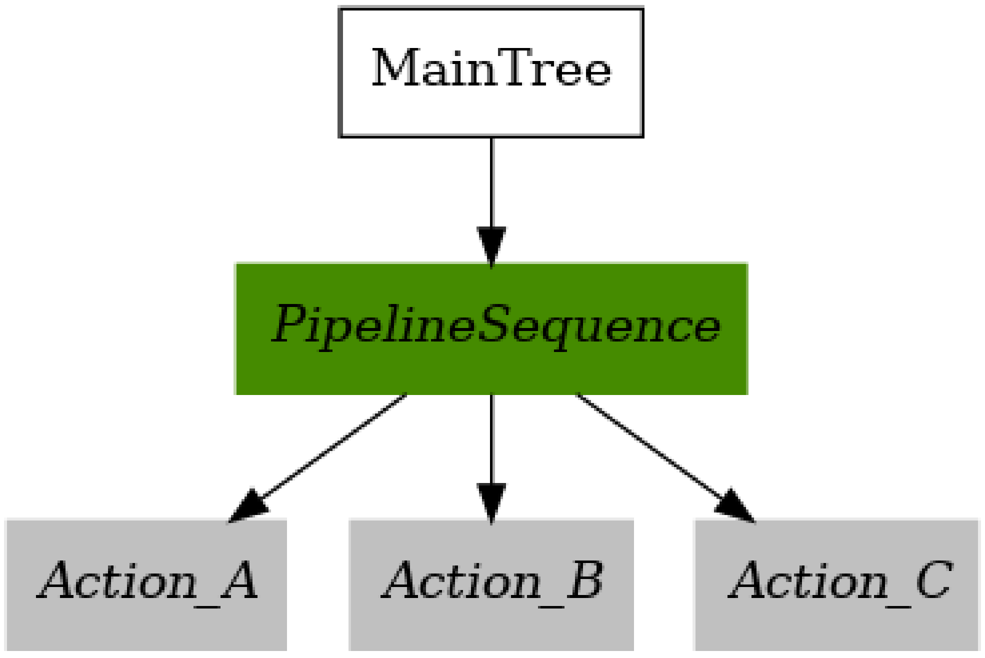
Decorator Nodes

- Distance Controller - Will tick children nodes every time the robot has traveled a certain distance
- Rate Controller - Controls the ticking of its child node at a constant frequency. The tick rate is an exposed port
- Goal Updater - Will update the goal of children nodes via ports on the BT
- Single Trigger - Will only tick its child node once, and will return `FAILURE` for all subsequent ticks
- Speed Controller - Controls the ticking of its child node at a rate proportional to the robot's speed

Control: PipelineSequence

The `PipelineSequence` control node re-ticks previous children when a child returns `RUNNING`. This node is similar to the `Sequence` node, with the additional property that the children prior to the "current" are re-ticked, (resembling the flow of water in a pipe). If at any point a child returns `FAILURE`, all children will be halted and the parent node will also return `FAILURE`. Upon `SUCCESS` of the **last node** in the sequence, this node will halt and return `SUCCESS`.

To explain this further, here is an example BT that uses PipelineSequence.



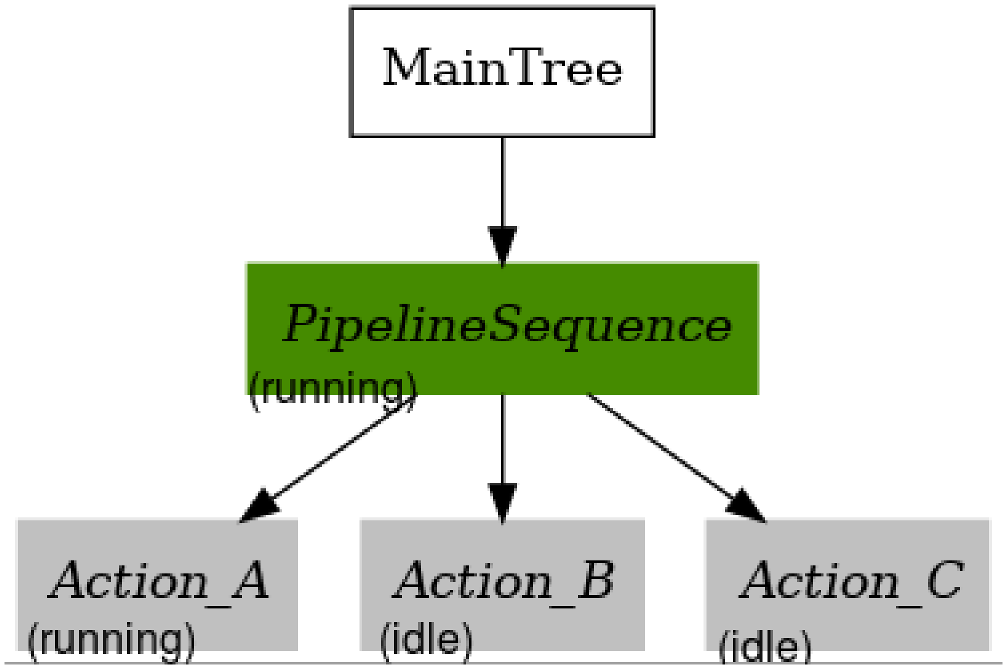
```
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <PipelineSequence>
```

(continues on next page)

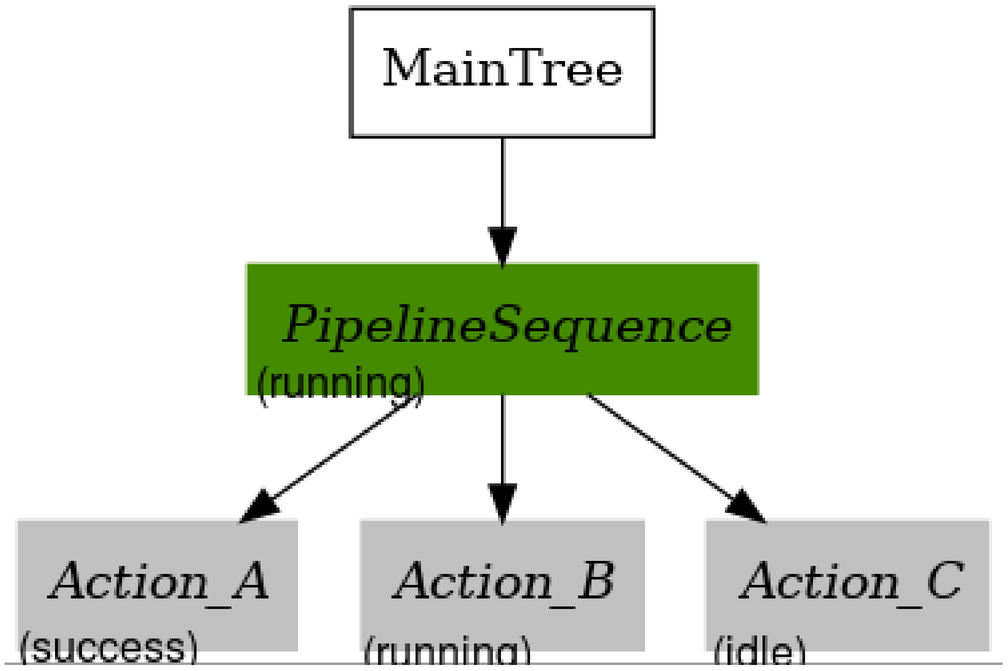
(continued from previous page)

```
            <Action_A/>
            <Action_B/>
            <Action_C/>
        </PipelineSequence>
    </BehaviorTree>
</root>
```

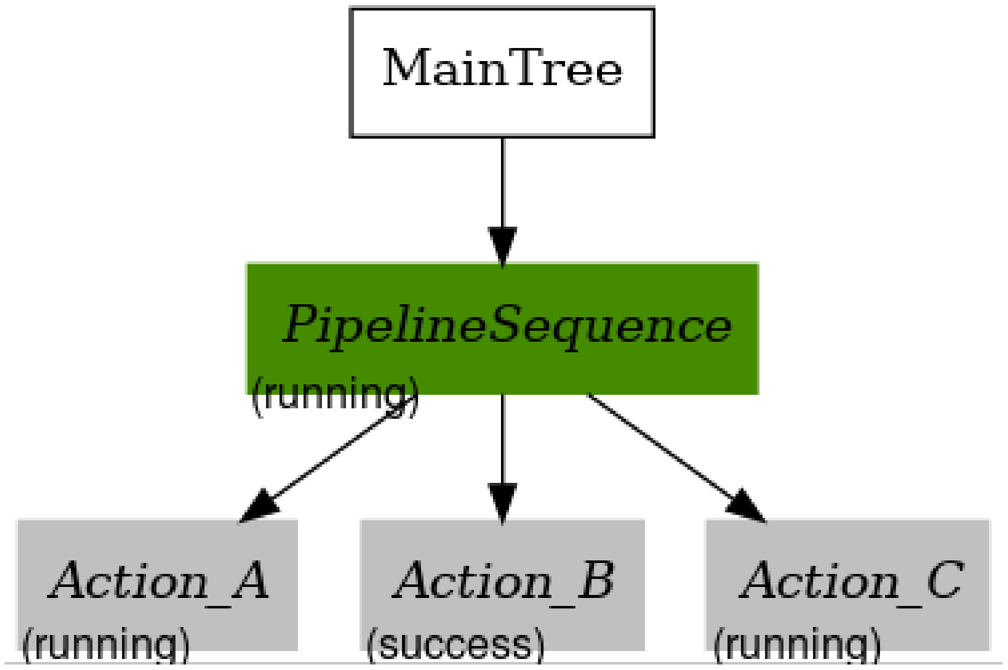
1. `Action_A`, `Action_B`, and `Action_C` are all `IDLE`.
2. When the parent PipelineSequence is first ticked, let's assume `Action_A` returns `RUNNING`. The parent node will now return `RUNNING` and no other nodes are ticked.



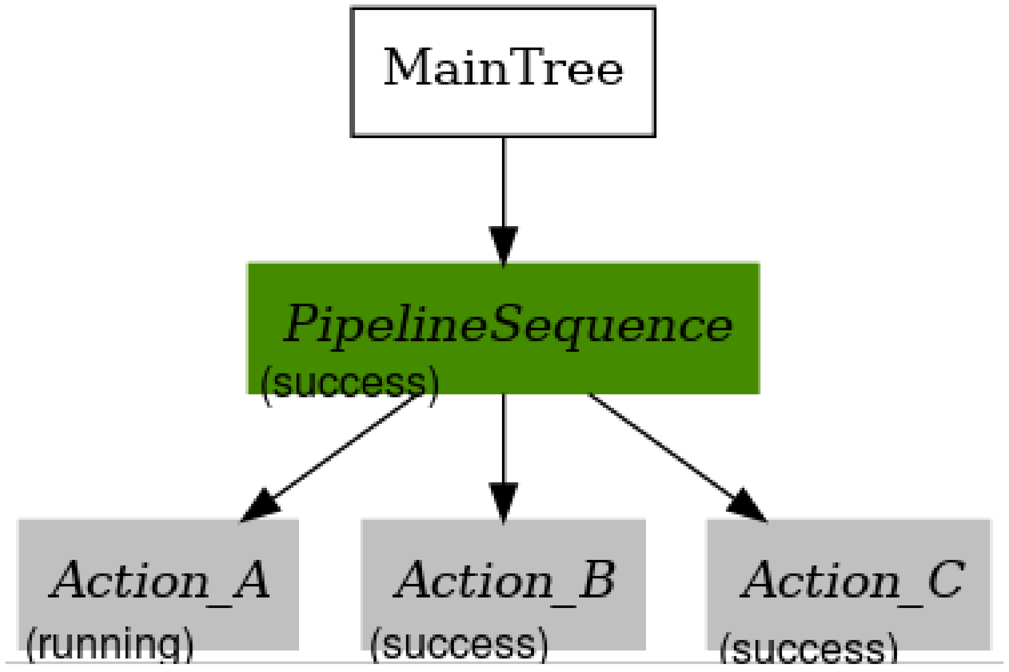
3. Now, let's assume `Action_A` returns `SUCCESS`, `Action_B` will now get ticked and will return `RUNNING`. `Action_C` has not yet been ticked so will return `IDLE`.



4. `Action_A` gets ticked again and returns `RUNNING`, and `Action_B` gets re-ticked and returns `SUCCESS` and therefore the BT goes on to tick `Action_C` for the first time. Let's assume `Action_C` returns `RUNNING`. The retick-ing of `Action_A` is what makes PipelineSequence useful.



5. All actions in the sequence will be re-ticked. Let's assume `Action_A` still returns `RUNNING`, where as `Action_B` returns `SUCCESS` again, and `Action_C` now returns `SUCCESS` on this tick. The sequence is now complete, and therefore `Action_A` is halted, even though it was still `RUNNING`.



Recall that if `Action_A`, `Action_B`, or `Action_C` returned `FAILURE` at any point of time, the parent would have returned `FAILURE` and halted any children as well.

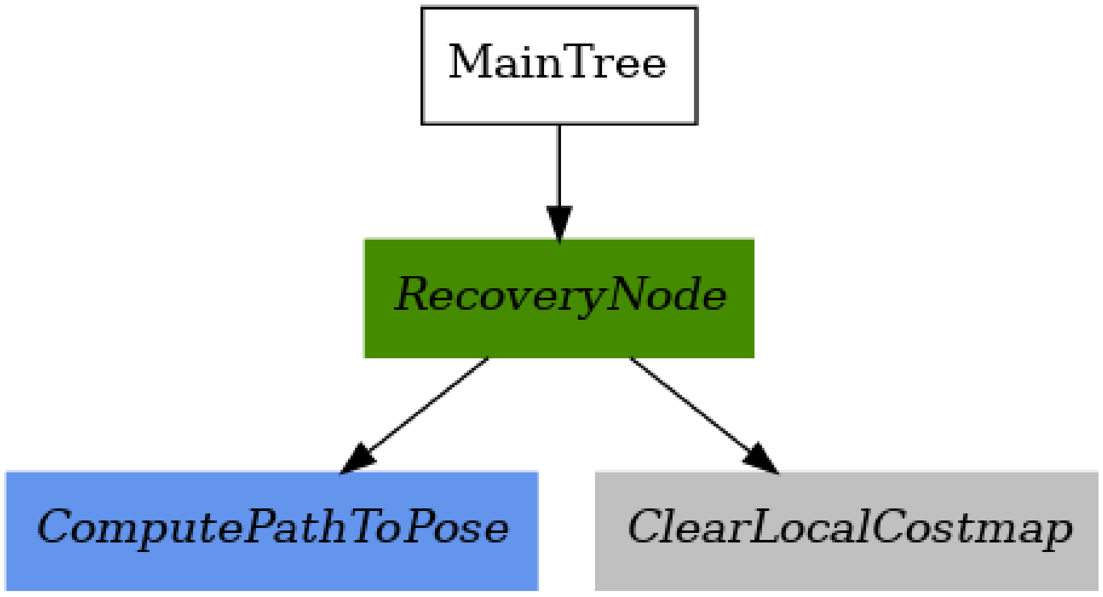
For additional details regarding the `PipelineSequence` please see the PipelineSequence configuration guide.

Control: Recovery

The Recovery control node has only two children and returns `SUCCESS` if and only if the first child returns `SUCCESS`. If the first child returns `FAILURE`, the second child will be ticked. This loop will continue until either:

- The first child returns `SUCCESS` (which results in `SUCCESS` of the parent node)
- The second child returns `FAILURE` (which results in `FAILURE` of the parent node)
- The `number_of_retries` input parameter is violated

This node is usually used to link together an action, and a recovery action as the name suggests. The first action will typically be the "main" behavior, and the second action will be something to be done in case of `FAILURE` of the main behavior. Often, the ticking of the second child action will promote the chance the first action will succeed.



```
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <RecoveryNode number_of_retries="1">
            <ComputePathToPose/>
            <ClearLocalCostmap/>
        </RecoveryNode>
    </BehaviorTree>
</root>
```

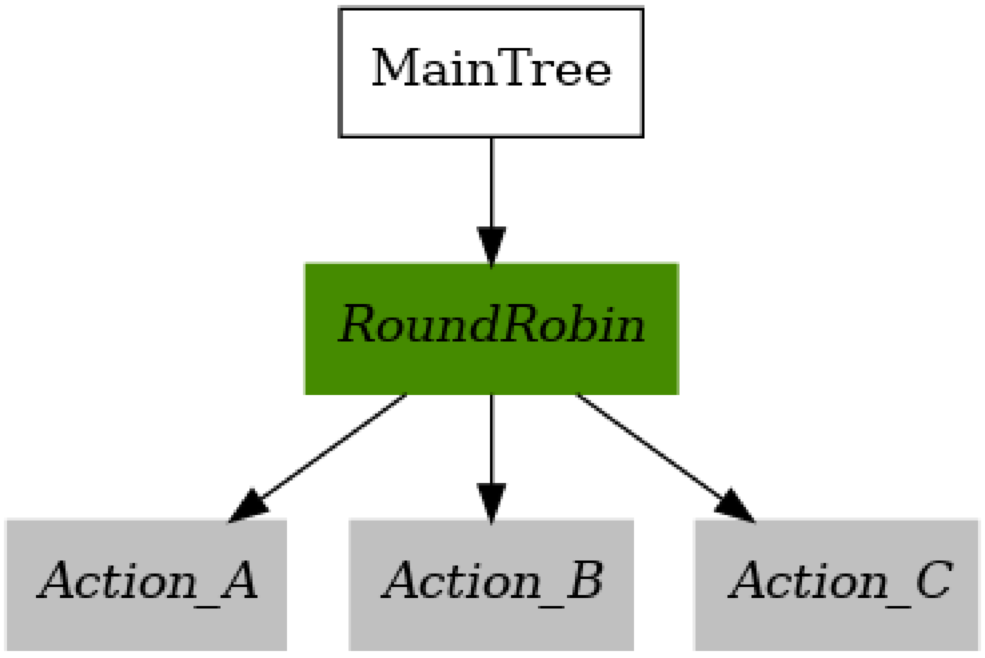
In the above example, let's assume `ComputePathToPose` fails. `ClearLocalCostmap` will be ticked in response, and return `SUCCESS`. Now that we have cleared the costmap, let's say the robot is correctly able to compute the path and `ComputePathToPose` now returns `SUCCESS`. Then, the parent RecoveryNode will also return `SUCCESS` and the BT will be complete.

For additional details regarding the `RecoveryNode` please see the RecoveryNode configuration guide.

Control: RoundRobin

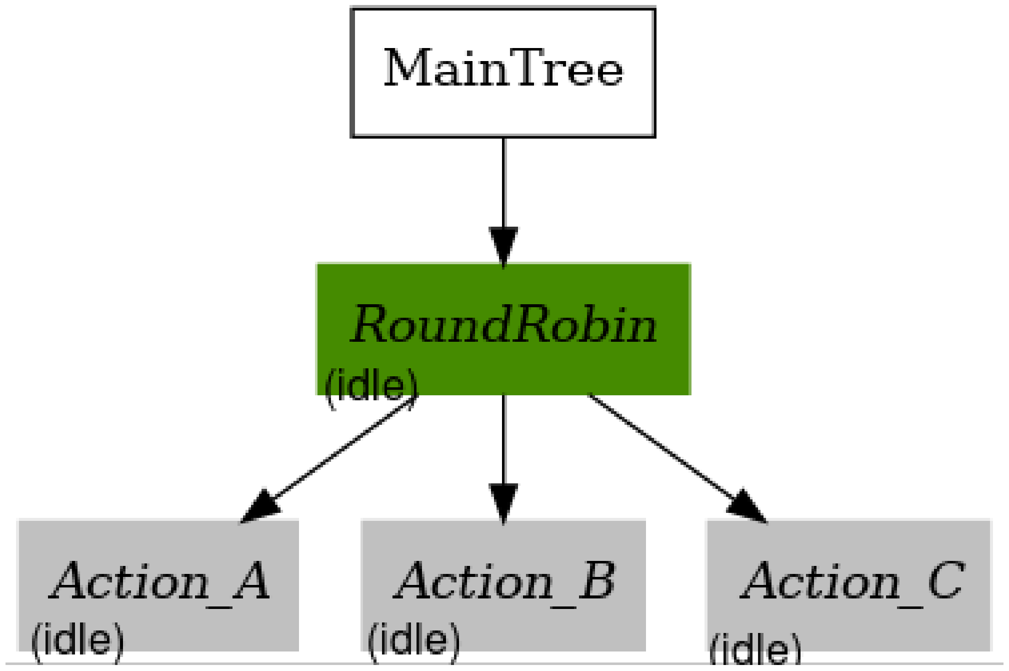
The RoundRobin control node ticks its children in a round robin fashion until a child returns `SUCCESS`, in which the parent node will also return `SUCCESS`. If all children return `FAILURE` so will the parent RoundRobin.

Here is an example BT we will use to walk through the concept.

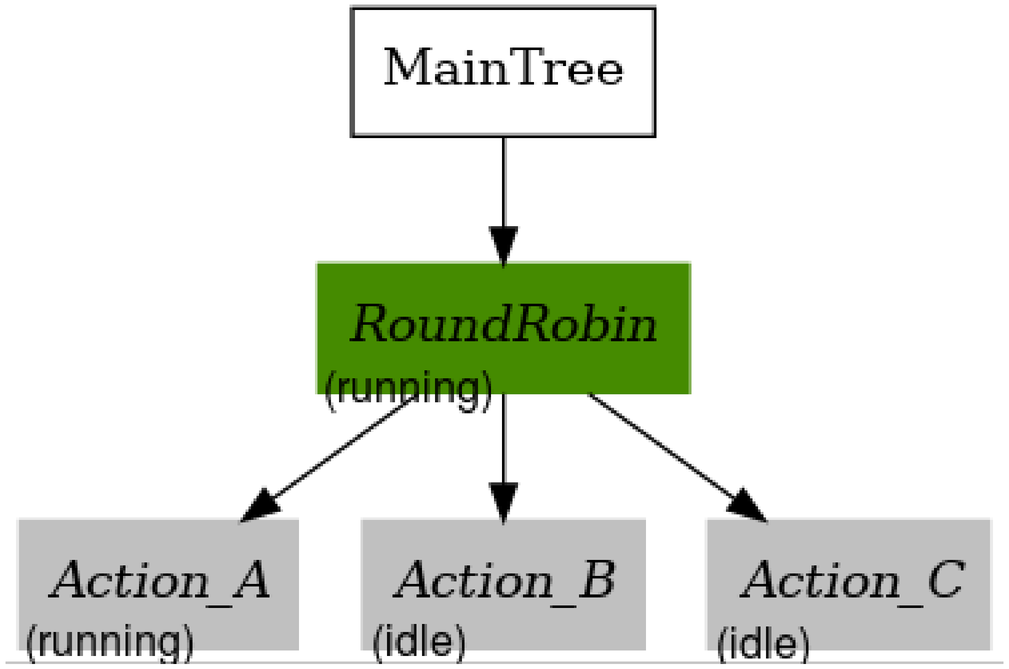


```
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <RoundRobin>
            <Action_A/>
            <Action_B/>
            <Action_C/>
        </RoundRobin>
    </BehaviorTree>
</root>
```

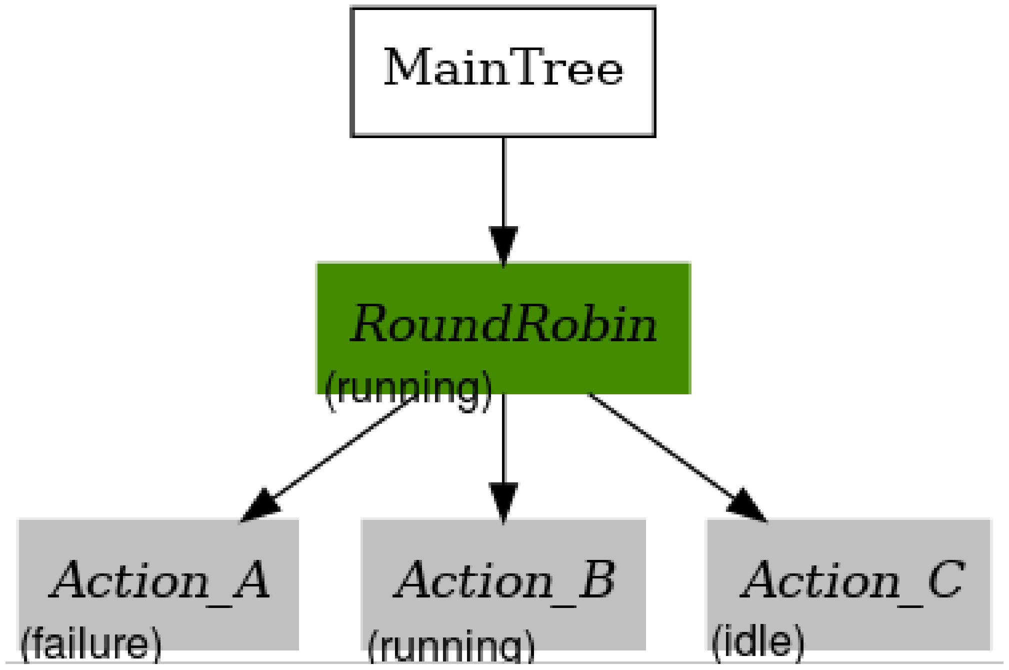
1. All the nodes start at `IDLE`



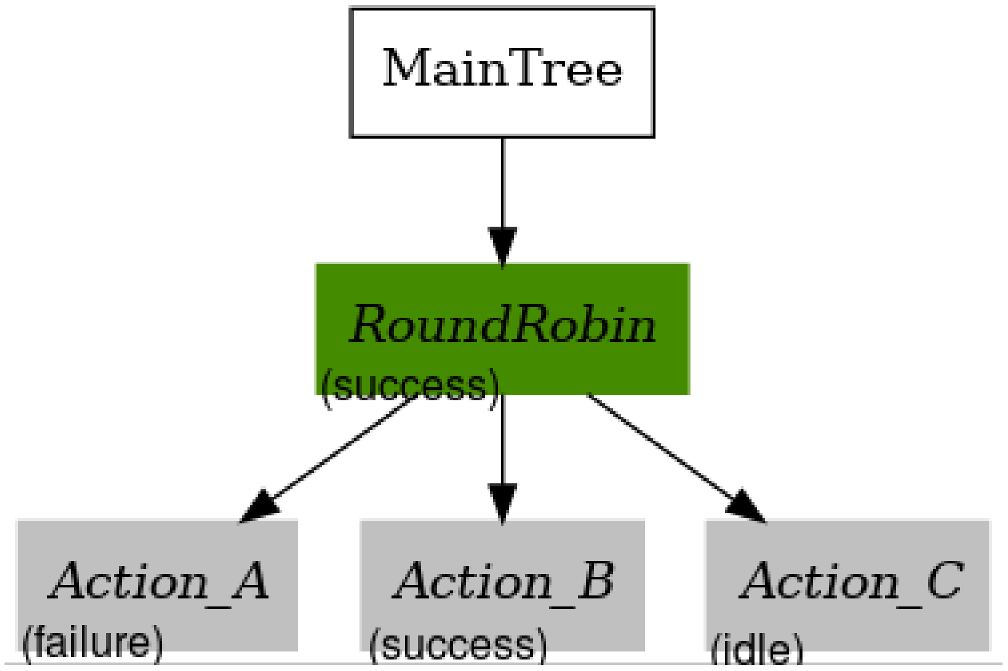
2. Upon tick of the parent node, the first child (`Action_A`) is ticked. Let's assume on tick the child returns `RUNNING`. In this case, no other children are ticked and the parent node returns `RUNNING` as well.



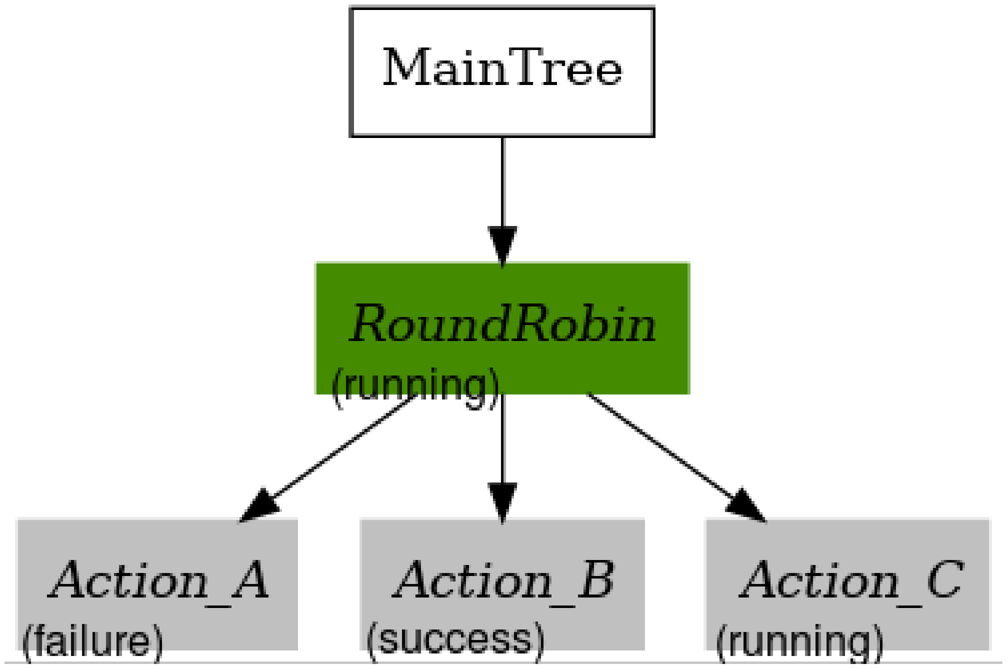
3. Upon the next tick, let's assume that `Action_A` returns `FAILURE`. This means that `Action_B` will get ticked next, and `Action_C` remains unticked. Let's assume `Action_B` returns `RUNNING` this time. That means the parent RoundRobin node will also return `RUNNING`.



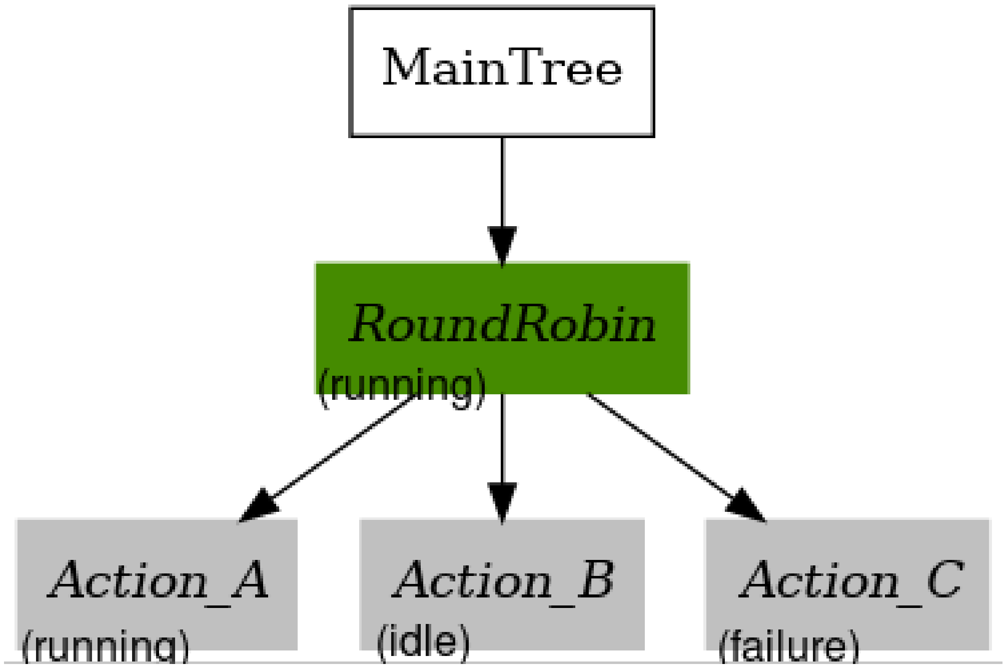
4. Upon this next tick, let's assume that `Action_B` returns `SUCCESS`. The parent RoundRobin will now halt all children and return `SUCCESS`. The parent node retains this state information, and will tick `Action_C` upon the next tick rather than start from `Action_A` like Step 2 did.



5. On this tick, let's assume `Action_C` returns `RUNNING`, and so does the parent RoundRobin. No other nodes are ticked.



6. On this last tick, let's assume `Action_C` returns `FAILURE`. The parent will circle and tick `Action_A` again. `Action_A` returns `RUNNING` and so will the parent RoundRobin node. This pattern will continue indefinitely unless all children return `FAILURE`.



For additional details regarding the `RecoveryNode` please see the RoundRobin configuration guide.

4.9.2 Detailed Behavior Tree Walkthrough

- *Overview*
- *Prerequisites*
- *Navigate To Pose With Replanning and Recovery*
- *Navigation Subtree*
- *Recovery Subtree*

Overview

This document serves as a reference guide to the main behavior tree (BT) used in Nav2.

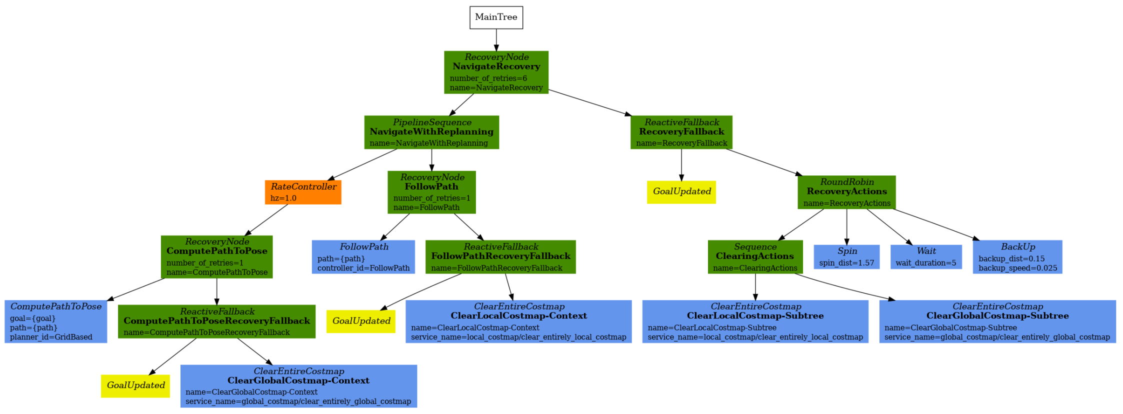
There are many example behavior trees provided in `nav2_bt_navigator/behavior_trees`, but these sometimes have to be re-configured based on the application of the robot. The following document will walk through the current main default BT `navigate_to_pose_w_replanning_and_recovery.xml` in great detail.

Prerequisites

- Become familiar with the concept of a behavior tree before continuing with this walkthrough
  - Read the short explanation in navigation concepts
  - Read the general tutorial and guide (not Nav2 specific) on the BehaviorTree CPP V3 website. Specifically, the "Learn the Basics" section on the BehaviorTree CPP V3 website explains the basic generic nodes that will be used that this guide will build upon.
- Become familiar with the custom Nav2 specific BT nodes

Navigate To Pose With Replanning and Recovery

The following section will describe in detail the concept of the main and default BT currently used in Nav2, `navigate_to_pose_w_replanning_and_recovery.xml`. This behavior tree replans the global path periodically at 1 Hz and it also has recovery actions.


MainTree
RecoveryNode NavigateRecovery number_of_retries=6 name=NavigateRecovery
PipelineSequence NavigateWithReplanning name=NavigateWithReplanning
ReactiveFallback RecoveryFallback name=RecoveryFallback
RateController hz=1.0
RecoveryNode FollowPath number_of_retries=1 name=FollowPath
GoalUpdated
RoundRobin RecoveryActions name=RecoveryActions
RecoveryNode ComputePathToPose number_of_retries=1 name=ComputePathToPose
FollowPath path={path} controller_id=FollowPath
ReactiveFallback FollowPathRecoveryFallback name=FollowPathRecoveryFallback
Sequence ClearingActions name=ClearingActions
Spin spin_dist=1.57
Wait wait_duration=5
BackUp backup_dist=0.15 backup_speed=0.025
ComputePathToPose goal={goal} path={path} planner_id=GridBased
ReactiveFallback ComputePathToPoseRecoveryFallback name=ComputePathToPoseRecoveryFallback
GoalUpdated
ClearEntireCostmap ClearLocalCostmap-Context name=ClearLocalCostmap-Context service_name=local_costmap/clear_entirely_local_costmap
ClearEntireCostmap ClearLocalCostmap-Subtree name=ClearLocalCostmap-Subtree service_name=local_costmap/clear_entirely_local_costmap
ClearEntireCostmap ClearGlobalCostmap-Subtree name=ClearGlobalCostmap-Subtree service_name=global_costmap/clear_entirely_global_costmap
GoalUpdated
ClearEntireCostmap ClearGlobalCostmap-Context name=ClearGlobalCostmap-Context service_name=global_costmap/clear_entirely_global_costmap

BTs are primarily defined in XML. The tree shown above is represented in XML as follows.

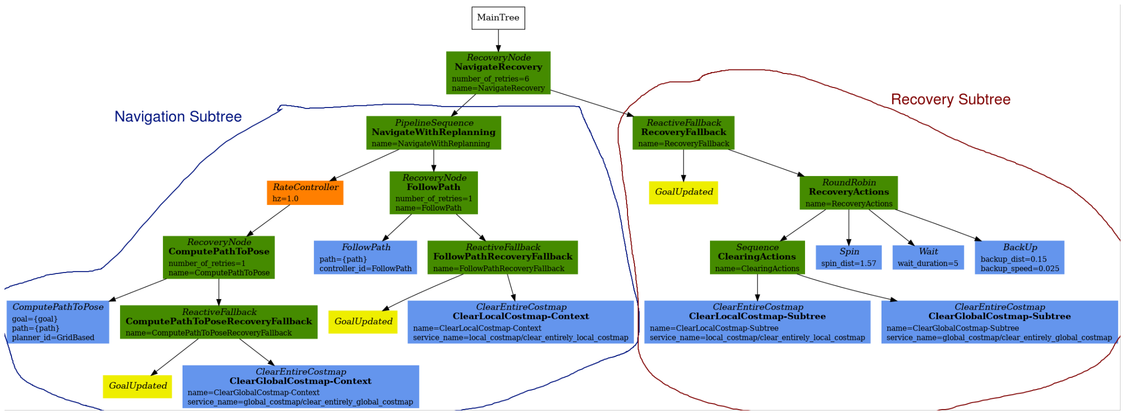
```xml
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <RecoveryNode number_of_retries="6" name="NavigateRecovery">
            <PipelineSequence name="NavigateWithReplanning">
                <RateController hz="1.0">
```

(continues on next page)

(continued from previous page)

```
                    <RecoveryNode number_of_retries="1" name="ComputePathToPose">
                        <ComputePathToPose goal="{goal}" path="{path}" planner_id=
↪"GridBased"/>
                        <ReactiveFallback name="ComputePathToPoseRecoveryFallback">
                            <GoalUpdated/>
                            <ClearEntireCostmap name="ClearGlobalCostmap-Context"␣
↪service_name="global_costmap/clear_entirely_global_costmap"/>
                        </ReactiveFallback>
                    </RecoveryNode>
                </RateController>
                <RecoveryNode number_of_retries="1" name="FollowPath">
                    <FollowPath path="{path}" controller_id="FollowPath"/>
                    <ReactiveFallback name="FollowPathRecoveryFallback">
                        <GoalUpdated/>
                        <ClearEntireCostmap name="ClearLocalCostmap-Context" service_
↪name="local_costmap/clear_entirely_local_costmap"/>
                    </ReactiveFallback>
                </RecoveryNode>
            </PipelineSequence>
            <ReactiveFallback name="RecoveryFallback">
                <GoalUpdated/>
                <RoundRobin name="RecoveryActions">
                    <Sequence name="ClearingActions">
                        <ClearEntireCostmap name="ClearLocalCostmap-Subtree" service_
↪name="local_costmap/clear_entirely_local_costmap"/>
                        <ClearEntireCostmap name="ClearGlobalCostmap-Subtree" service_
↪name="global_costmap/clear_entirely_global_costmap"/>
                    </Sequence>
                    <Spin spin_dist="1.57"/>
                    <Wait wait_duration="5"/>
                    <BackUp backup_dist="0.15" backup_speed="0.025"/>
                </RoundRobin>
            </ReactiveFallback>
        </RecoveryNode>
    </BehaviorTree>
</root>
```

This is likely still a bit overwhelming, but this tree can be broken into two smaller subtrees that we can focus on one at a time. These smaller subtrees are the children of the top-most `RecoveryNode`. From this point forward the `NavigateWithReplanning` subtree will be referred to as the `Navigation` subtree, and the `RecoveryFallback` subtree will be known as the `Recovery` subtree. This can be represented in the following way:



The `Navigation` subtree mainly involves actual navigation behavior:

- calculating a path
- following a path
- contextual recovery behaviors for each of the above primary navigation behaviors

The `Recovery` subtree includes behaviors for system level failures or items that were not easily dealt with internally.

The overall BT will (hopefully) spend most of its time in the `Navigation` subtree. If either of the two main behaviors in the `Navigation` subtree fail (path calculation or path following), contextual recoveries will be attempted.

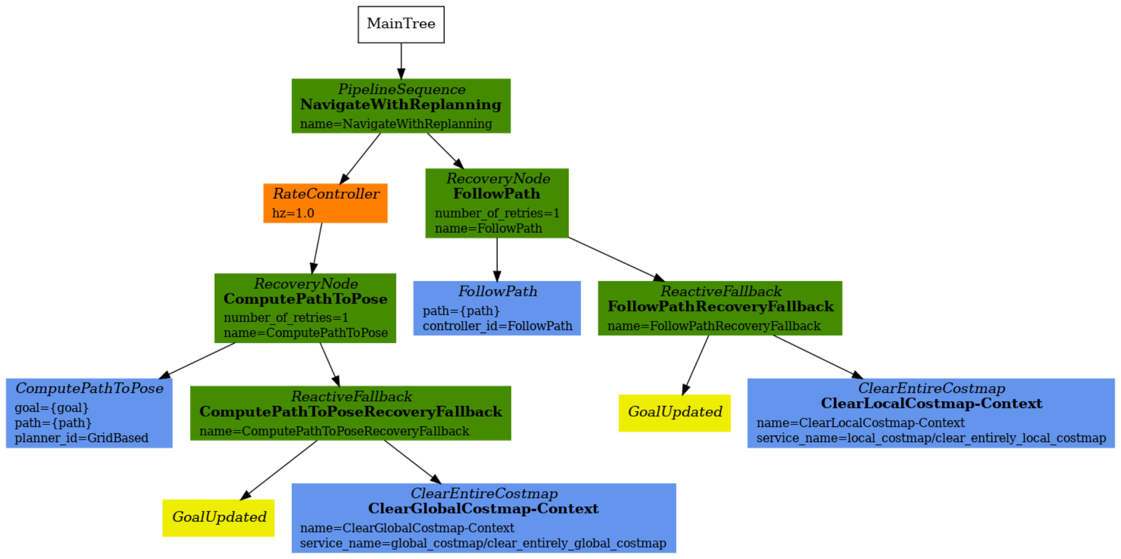
If the contextual recoveries were still not enough, the `Navigation` subtree will return `FAILURE`. The system will move on to the `Recovery` subtree to attempt to clear any system level navigation failures.

This happens until the `number_of_retries` for the parent `RecoveryNode` is exceeded (which by default is 6).

```
<RecoveryNode number_of_retries="6" name="NavigateRecovery">
```

Navigation Subtree

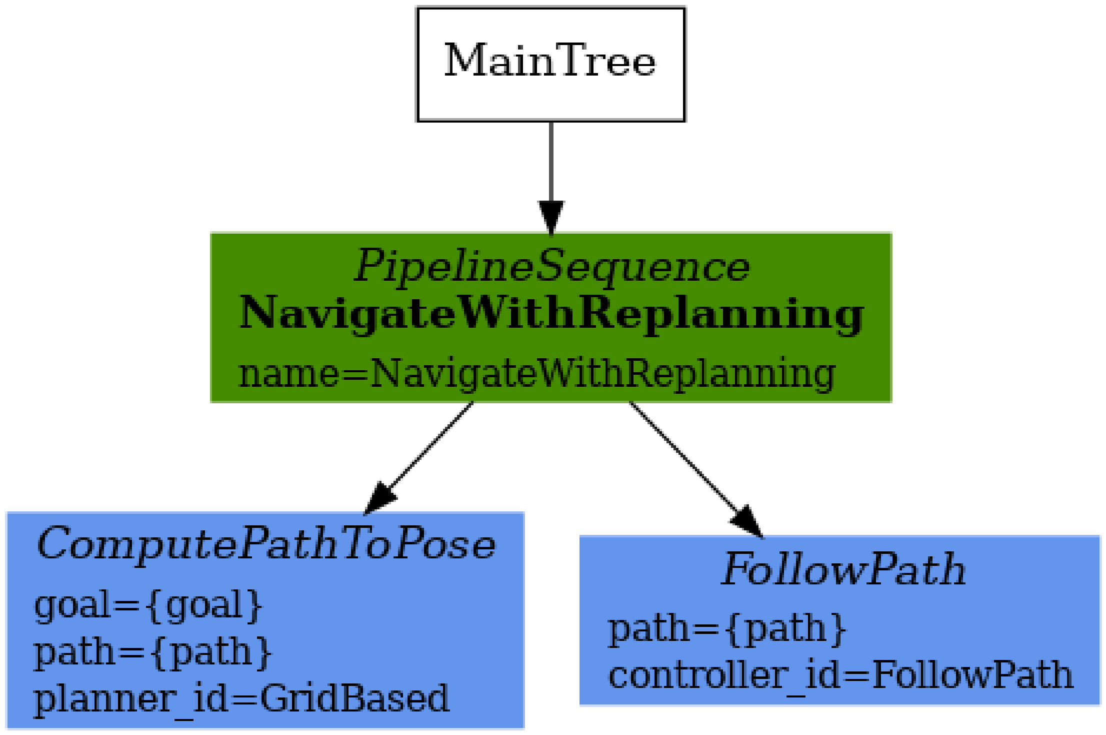
Now that we have gone over the control flow between the `Navigation` subtree and the `Recovery` subtree, let's focus on the Navigation subtree.



The XML of this subtree is as follows:

```xml
<PipelineSequence name="NavigateWithReplanning">
    <RateController hz="1.0">
        <RecoveryNode number_of_retries="1" name="ComputePathToPose">
            <ComputePathToPose goal="{goal}" path="{path}" planner_id="GridBased"/>
            <ReactiveFallback name="ComputePathToPoseRecoveryFallback">
                <GoalUpdated/>
                <ClearEntireCostmap name="ClearGlobalCostmap-Context" service_name=
↪"global_costmap/clear_entirely_global_costmap"/>
            </ReactiveFallback>
        </RecoveryNode>
    </RateController>
    <RecoveryNode number_of_retries="1" name="FollowPath">
        <FollowPath path="{path}" controller_id="FollowPath"/>
        <ReactiveFallback name="FollowPathRecoveryFallback">
            <GoalUpdated/>
            <ClearEntireCostmap name="ClearLocalCostmap-Context" service_name="local_
↪costmap/clear_entirely_local_costmap"/>
        </ReactiveFallback>
    </RecoveryNode>
</PipelineSequence>
```

This subtree has two primary actions `ComputePathToPose` and `FollowPath`. If either of these two actions fail, they will attempt to clear the failure contextually. The crux of the tree can be represented with only one parent and two children nodes like this:

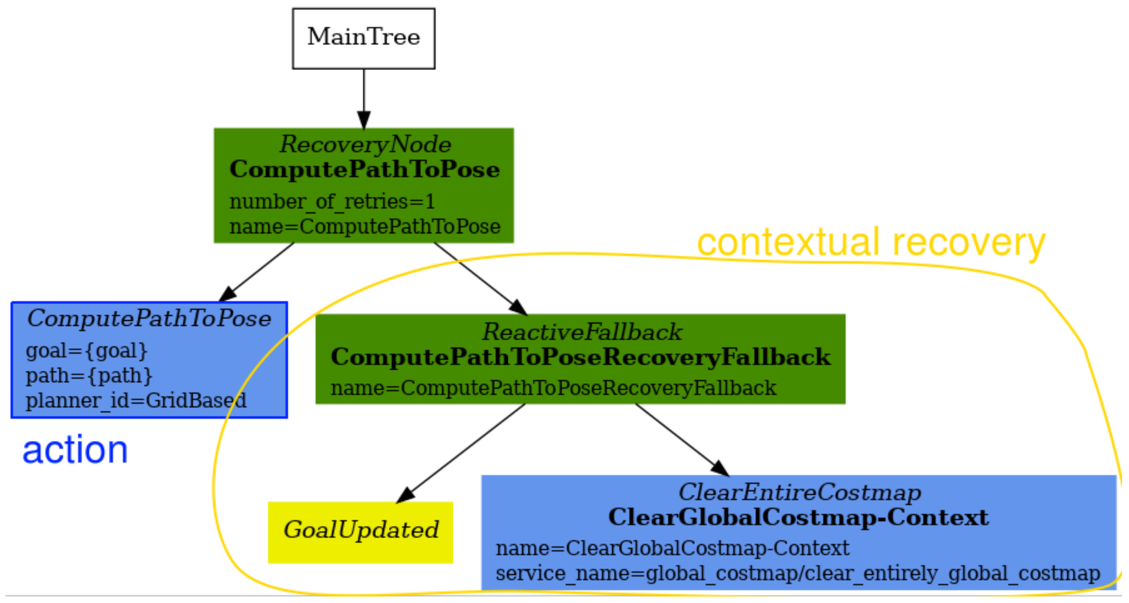


The parent `PipelineSequence` node allows the `ComputePathToPose` to be ticked, and once that succeeds, `FollowPath` to be ticked. While the `FollowPath` subtree is being ticked, the `ComputePathToPose` subtree will be ticked as well. This allows for the path to be recomputed as the robot moves around.

Both the `ComputePathToPose` and the `FollowPath` follow the same general structure.

- Do the action
- If the action fails, try to see if we can contextually recover

The below is the `ComputePathToPose` subtree:



The parent `RecoveryNode` controls the flow between the action, and the contextual recovery subtree. The contextual recoveries for both `ComputePathToPose` and `FollowPath` involve checking if the goal has been updated, and involves clearing the relevant costmap.

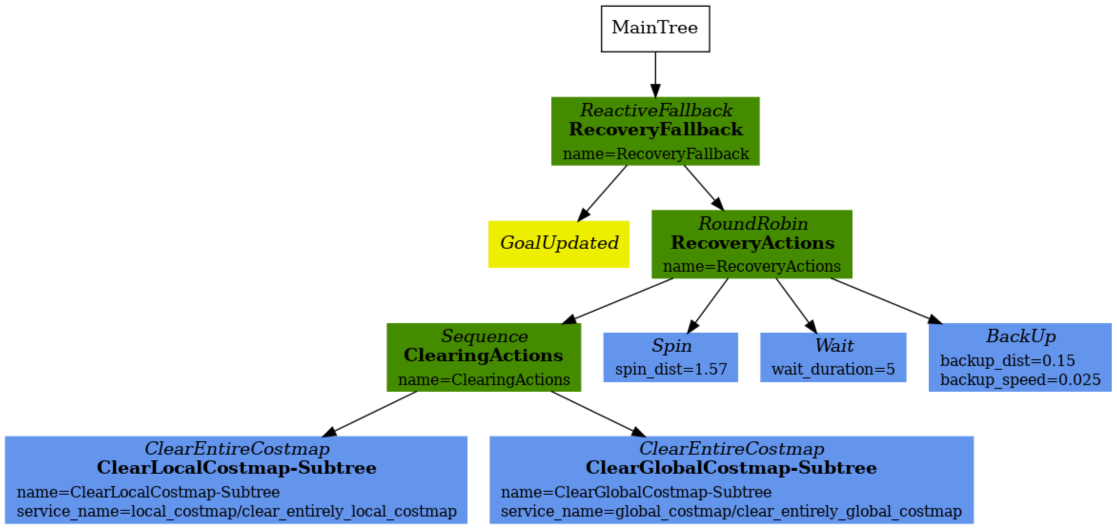
Consider changing the `number_of_retries` parameter in the parent `RecoveryNode` control node if your application can tolerate more attempts at contextual recoveries before moving on to system-level recoveries.

The only differences in the BT subtree of `ComputePathToPose` and `FollowPath` are outlined below:

- **The action node in the subtree:**
  - The `ComputePathToPose` subtree centers around the `ComputePathToPose` action.
  - The `FollowPath` subtree centers around the `FollowPath` action.
- **The `RateController` that decorates the `ComputePathToPose` subtree** The `RateController` decorates the `ComputePathToPose` subtree to keep planning at the specified frequency. The default frequency for this BT is 1 hz. This is done to prevent the BT from flooding the planning server with too many useless requests at the tree update rate (100Hz). Consider changing this frequency to something higher or lower depending on the application and the computational cost of calculating the path. There are other decorators that can be used instead of the `RateController`. Consider using the `SpeedController` or `DistanceController` decorators if appropriate.
- **The costmap that is being cleared within the contextual recovery:**
  - The `ComputePathToPose` subtree clears the global costmap. The global costmap is the relevant costmap in the context of the planner
  - The `FollowPath` subtree clears the local costmap. The local costmap is the relevant costmap in the context of the controller

Recovery Subtree

The Recovery subtree is the second big "half" of the Nav2 default navigate_to_pose_w_replanning_and_recovery.xml tree. In short, this subtree is triggered when the Navigation subtree returns FAILURE and controls the recoveries at the system level (in the case the contextual recoveries in the Navigation subtree were not sufficient).



And the XML snippet:

```
<ReactiveFallback name="RecoveryFallback">
    <GoalUpdated/>
    <RoundRobin name="RecoveryActions">
        <Sequence name="ClearingActions">
            <ClearEntireCostmap name="ClearLocalCostmap-Subtree" service_name="local_
↪costmap/clear_entirely_local_costmap"/>
            <ClearEntireCostmap name="ClearGlobalCostmap-Subtree" service_name=
↪"global_costmap/clear_entirely_global_costmap"/>
        </Sequence>
        <Spin spin_dist="1.57"/>
        <Wait wait_duration="5"/>
        <BackUp backup_dist="0.15" backup_speed="0.025"/>
    </RoundRobin>
</ReactiveFallback>
```

The top most parent, ReactiveFallback controls the flow between the rest of the system wide recoveries, and asynchronously checks if a new goal has been received. If at any point the goal gets updated, this subtree will halt all children and return SUCCESS. This allows for quick reactions to new goals and preempt currently executing recoveries. This should look familiar to the contextual recovery portions of the Navigation subtree. This is a common BT pattern to handle the situation "Unless 'this condition' happens, Do action A".

These condition nodes can be extremely powerful and are typically paired with `ReactiveFallback`. It can be easy to imagine wrapping this whole `navigate_to_pose_w_replanning_and_recovery` tree in a `ReactiveFallback` with a `isBatteryLow` condition – meaning the `navigate_to_pose_w_replanning_and_recovery` tree will execute *unless* the battery becomes low (and then entire a different subtree for docking to recharge).

If the goal is never updated, the behavior tree will go on to the `RoundRobin` node. These are the default four system-level recoveries in the BT are:

- A sequence that clears both costmaps (local, and global)
- `Spin` action
- `Wait` action
- `BackUp` action

Upon `SUCCESS` of any of the four children of the parent `RoundRobin`, the robot will attempt to renavigate in the `Navigation` subtree. If this renavigation was not successful, the next child of the `RoundRobin` will be ticked.

For example, let's say the robot is stuck and the `Navigation` subtree returns `FAILURE`: (for the sake of this example, let's assume that the goal is never updated).

1. The Costmap clearing sequence in the `Recovery` subtree is attempted, and returns `SUCCESS`. The robot now moves to `Navigation` subtree again
2. Let's assume that clearing both costmaps was not sufficient, and the `Navigation` subtree returns `FAILURE` once again. The robot now ticks the `Recovery` subtree
3. In the `Recovery` subtree, the `Spin` action will be ticked. If this returns `SUCCESS`, then the robot will return to the main `Navigation` subtree *BUT* let's assume that the `Spin` action returns `FAILURE`. In this case, the tree will *remain* in the `Recovery` subtree
4. Let's say the next action, `Wait` returns `SUCCESS`. The robot will then move on to the `Navigation` subtree
5. Assume the `Navigation` subtree returns `FAILURE` (clearing the costmaps, attempting a spin, and waiting were *still* not sufficient to recover the system. The robot will move onto the `Recovery` subtree and attempt the `BackUp` action. Let's say that the robot attempts the `BackUp` action and was able to successfully complete the action. The `BackUp` action node returns `SUCCESS` and so now we move on to the Navigation subtree again.
6. In this hypothetical scenario, let's assume that the `BackUp` action allowed the robot to successfully navigate in the `Navigation` subtree, and the robot reaches the goal. In this case, the overall BT will still return `SUCCESS`.

If the `BackUp` action was not sufficient enough to allow the robot to become un-stuck, the above logic will go on indefinitely until the `number_of_retries` in the parent of the `Navigate` subtree and `Recovery` subtree is exceeded, or if all the system-wide recoveries in the `Recovery` subtree return `FAILURE` (this is unlikely, and likely points to some other system failure).

4.9.3 Navigate To Pose

This behavior tree implements a significantly more mature version of the behavior tree on *Nav2 Behavior Trees*. It navigates from a starting point to a single point goal in freespace. It contains both use of custom recovery behaviors in specific sub-contexts as well as a global recovery subtree for system-level failures. It also provides the opportunity for users to retry tasks multiple times before returning a failed state.

The `ComputePathToPose` and `FollowPath` BT nodes both also specify their algorithms to utilize. By convention we name these by the style of algorithms that they are (e.g. not `DWB` but rather `FollowPath`) such that a behavior tree or application developer need not worry about the technical specifics. They just want to use a path following controller.

In this behavior tree, we attempt to retry the entire navigation task 6 times before returning to the caller that the task has failed. This allows the navigation system ample opportunity to try to recovery from failure conditions or wait for transient issues to pass, such as crowding from people or a temporary sensor failure.

In nominal execution, this will replan the path at every second and pass that path onto the controller, similar to the behavior tree in *Nav2 Behavior Trees*. However, this time, if the planner fails, it will trigger contextually aware recovery behaviors in its subtree, clearing the global costmap. Additional recovery behaviors can be added here for additional context-specific recoveries, such as trying another algorithm.

Similarly, the controller has similar logic. If it fails, it also attempts a costmap clearing of the local costmap impacting the controller. It is worth noting the `GoalUpdated` node in the reactive fallback. This allows us to exit recovery conditions when a new goal has been passed to the navigation system through a preemption. This ensures that the navigation system will be very responsive immediately when a new goal is issued, even when the last goal was in an attempted recovery.

If these contextual recoveries fail, this behavior tree enters the recovery subtree. This subtree is reserved for system-level failures to help resolve issues like the robot being stuck or in a bad spot. This subtree also has the `GoalUpdated` BT node it ticks every iteration to ensure responsiveness of new goals. Next, the recovery subtree will the recoveries: costmap clearing operations, spinning, waiting, and backing up. After each of the recoveries in the subtree, the main navigation subtree will be reattempted. If it continues to fail, the next recovery in the recovery subtree is ticked.

While this behavior tree does not make use of it, the `PlannerSelector`, `ControllerSelector`, and `GoalCheckerSelector` behavior tree nodes can also be helpful. Rather than hardcoding the algorithm to use (`GridBased` and `FollowPath`), these behavior tree nodes will allow a user to dynamically change the algorithm used in the navigation system via a ROS topic. It may be instead advisable to create different subtree contexts using condition nodes with specified algorithms in their most useful and unique situations. However, the selector nodes can be a useful way to change algorithms from an external application rather than via internal behavior tree control flow logic. It is better to implement changes through behavior tree methods, but we understand that many professional users have external applications to dynamically change settings of their navigators.

```xml
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <RecoveryNode number_of_retries="6" name="NavigateRecovery">
      <PipelineSequence name="NavigateWithReplanning">
        <RateController hz="1.0">
          <RecoveryNode number_of_retries="1" name="ComputePathToPose">
            <ComputePathToPose goal="{goal}" path="{path}" planner_id="GridBased"/>
            <ReactiveFallback name="ComputePathToPoseRecoveryFallback">
              <GoalUpdated/>
              <ClearEntireCostmap name="ClearGlobalCostmap-Context" service_name=
↪"global_costmap/clear_entirely_global_costmap"/>
            </ReactiveFallback>
          </RecoveryNode>
        </RateController>
        <RecoveryNode number_of_retries="1" name="FollowPath">
          <FollowPath path="{path}" controller_id="FollowPath"/>
          <ReactiveFallback name="FollowPathRecoveryFallback">
            <GoalUpdated/>
            <ClearEntireCostmap name="ClearLocalCostmap-Context" service_name="local_
↪costmap/clear_entirely_local_costmap"/>
          </ReactiveFallback>
        </RecoveryNode>
      </PipelineSequence>
      <ReactiveFallback name="RecoveryFallback">
        <GoalUpdated/>
        <RoundRobin name="RecoveryActions">
          <Sequence name="ClearingActions">
            <ClearEntireCostmap name="ClearLocalCostmap-Subtree" service_name="local_
↪costmap/clear_entirely_local_costmap"/>
```

(continues on next page)

(continued from previous page)

```
            <ClearEntireCostmap name="ClearGlobalCostmap-Subtree" service_name=
↪"global_costmap/clear_entirely_global_costmap"/>
          </Sequence>
          <Spin spin_dist="1.57"/>
          <Wait wait_duration="5"/>
          <BackUp backup_dist="0.15" backup_speed="0.025"/>
        </RoundRobin>
      </ReactiveFallback>
    </RecoveryNode>
  </BehaviorTree>
</root>
```

4.9.4 Navigate Through Poses

This behavior tree implements a navigation behavior from a starting point, through many intermediary hard pose constraints, to a final goal in freespace. It contains both use of custom behaviors for recovery in specific sub-contexts as well as a global recovery subtree for system-level failures. It also provides the opportunity for users to retry tasks multiple times before returning a failed state.

The `ComputePathThroughPoses` and `FollowPath` BT nodes both also specify their algorithms to utilize. By convention we name these by the style of algorithms that they are (e.g. not `DWB` but rather `FollowPath`) such that a behavior tree or application developer need not worry about the technical specifics. They just want to use a path following controller.

In this behavior tree, we attempt to retry the entire navigation task 6 times before returning to the caller that the task has failed. This allows the navigation system ample opportunity to try to recovery from failure conditions or wait for transient issues to pass, such as crowding from people or a temporary sensor failure.

In nominal execution, this will replan the path at every 3 seconds and pass that path onto the controller, similar to the behavior tree in *Nav2 Behavior Trees*. The planner though is now `ComputePathThroughPoses` taking a vector, `goals`, rather than a single pose `goal` to plan to. The `RemovePassedGoals` node is used to cull out `goals` that the robot has passed on its path. In this case, it is set to remove a pose from the poses when the robot is within `0.5` of the goal and it is the next goal in the list. This is implemented such that replanning can be computed after the robot has passed by some of the intermediary poses and not continue to try to replan through them in the future. This time, if the planner fails, it will trigger contextually aware recoveries in its subtree, clearing the global costmap. Additional recoveries can be added here for additional context-specific recoveries, such as trying another algorithm.

Similarly, the controller has similar logic. If it fails, it also attempts a costmap clearing of the local costmap impacting the controller. It is worth noting the `GoalUpdated` node in the reactive fallback. This allows us to exit recovery conditions when a new goal has been passed to the navigation system through a preemption. This ensures that the navigation system will be very responsive immediately when a new goal is issued, even when the last goal was in an attempted recovery.

If these contextual recoveries fail, this behavior tree enters the recovery subtree. This subtree is reserved for system-level failures to help resolve issues like the robot being stuck or in a bad spot. This subtree also has the `GoalUpdated` BT node it ticks every iteration to ensure responsiveness of new goals. Next, the recovery subtree will tick the costmap clearing operations, spinning, waiting, and backing up. After each of the recoveries in the subtree, the main navigation subtree will be reattempted. If it continues to fail, the next recovery in the recovery subtree is ticked.

While this behavior tree does not make use of it, the `PlannerSelector`, `ControllerSelector`, and `GoalCheckerSelector` behavior tree nodes can also be helpful. Rather than hardcoding the algorithm to use (`GridBased` and `FollowPath`), these behavior tree nodes will allow a user to dynamically change the algorithm used in the navigation system via a ROS topic. It may be instead advisable to create different subtree contexts using condition nodes with specified algorithms in their most useful and unique situations. However, the selector nodes can be a useful way to change algorithms from an external application rather than via internal behavior tree control flow logic. It is better to implement changes through behavior tree methods, but we understand that many professional users have external applications to dynamically change settings of their navigators.

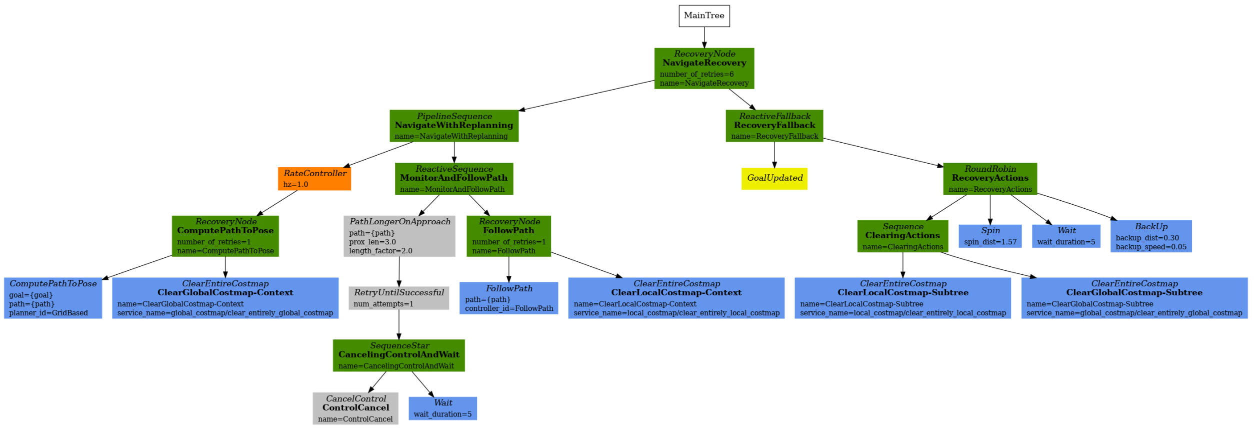
```
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <RecoveryNode number_of_retries="6" name="NavigateRecovery">
      <PipelineSequence name="NavigateWithReplanning">
        <RateController hz="0.333">
          <RecoveryNode number_of_retries="1" name="ComputePathThroughPoses">
            <ReactiveSequence>
              <RemovePassedGoals input_goals="{goals}" output_goals="{goals}" radius=
↪"0.5"/>
              <ComputePathThroughPoses goals="{goals}" path="{path}" planner_id=
↪"GridBased"/>
            </ReactiveSequence>
            <ReactiveFallback name="ComputePathThroughPosesRecoveryFallback">
              <GoalUpdated/>
              <ClearEntireCostmap name="ClearGlobalCostmap-Context" service_name=
↪"global_costmap/clear_entirely_global_costmap"/>
            </ReactiveFallback>
          </RecoveryNode>
        </RateController>
        <RecoveryNode number_of_retries="1" name="FollowPath">
          <FollowPath path="{path}" controller_id="FollowPath"/>
          <ReactiveFallback name="FollowPathRecoveryFallback">
            <GoalUpdated/>
            <ClearEntireCostmap name="ClearLocalCostmap-Context" service_name="local_
↪costmap/clear_entirely_local_costmap"/>
          </ReactiveFallback>
        </RecoveryNode>
      </PipelineSequence>
      <ReactiveFallback name="RecoveryFallback">
        <GoalUpdated/>
        <RoundRobin name="RecoveryActions">
          <Sequence name="ClearingActions">
            <ClearEntireCostmap name="ClearLocalCostmap-Subtree" service_name="local_
↪costmap/clear_entirely_local_costmap"/>
            <ClearEntireCostmap name="ClearGlobalCostmap-Subtree" service_name=
↪"global_costmap/clear_entirely_global_costmap"/>
          </Sequence>
          <Spin spin_dist="1.57"/>
          <Wait wait_duration="5"/>
          <BackUp backup_dist="0.15" backup_speed="0.025"/>
        </RoundRobin>
      </ReactiveFallback>
    </RecoveryNode>
  </BehaviorTree>
</root>
```

4.9.5 Navigate To Pose and Pause Near Goal-Obstacle

Note: As a prerequisite, we encourage the users to go through the Behavior Tree documentation, which explains about different behaviors nodes used in these trees such as `ReactiveSequence`, `SequenceStar` and `RetryUntilSucessfull`.

This behavior tree is a soft extension to the *Navigate To Pose*. Apart from the functionalities of *Navigate To Pose*, this behavior tree allows the robot to efficiently handle an obstacle (e.g. forklift, person, or other temporary obstacles) close to the goal by pausing the robot's navigation and wait for a user-specified time to check if the obstacle has cleared. If the obstacle has moved during the waiting time, the robot will continue to the goal taking the shorter path. If the obstacle has not moved during the waiting time or the waiting time expires, then the robot will use the longer path around to reach the final goal location. Ultimately, for a given task, this behavior tree aids in solving the problem of long cycle time, which is caused because of the long path generated due to the temporary obstacles present close to the goal location.

The behavior tree is depicted in the image below. From the image, it can be noted that there is an additional branch in the Navigation Subtree known as `MonitorAndFollowPath`. This branch is created with the intention for the users to perform any kind of monitoring behavior that their robot should exhibit. In this particular BT, the monitoring branch is exclusively utilized by `PathLongerOnApproach` BT node for checking if the global planner has decided to plan a significantly longer path for the robot on approaching the user-specified goal proximity. If there is no significantly longer path, the monitor node goes into the `FollowPath` recovery node, which then generates the necessary control commands.



Once there is a significantly longer path, the child node for the `PathLongerOnApproach` node ticks. The child node is a `RetryUntilSuccesfull` decorator node, which inturns have a `SequenceStar` node as its child. Firstly, the `SequenceStar` node cancels the controller server by ticking the `CancelControl` node. The cancellation of the controller server halts the further navigation of the robot. Next, the `SequenceStar` node ticks the `Wait` node, which enables the robot to wait for the given user-specified time. Here we need to note that, the `MonitorAndFollowPath` is a `ReactiveSequence` node, therefore the `PathLongerOnApproach` node needs to return SUCCESS, before the `FollowPath` node can be ticked once again.

In the below GIF, it can be seen that the robot is approaching the goal location, but it found an obstacle in the goal proximity, because of which the global planner, plans a longer path around. This is the point where the `PathLongerOnApproach` ticks and ticks its children, consequently cancelling the `controller_server` and waiting to see if the obstacle clears up. In the below scenario, the obstacles do not clear, causing the robot to take the longer path.

Alternatively, if the obstacles are cleared, then there is a shorter path generated by the global planner. Now, the `PathLongerOnApproach` returns SUCCESS, that cause the `FollowPath` to continue with the robot navigation.

Apart from the above scenarios, we also need to note that, the robot will take the longer path to the goal location if the obstacle does not clear up in the given user-specific wait time.

In conclusion, this particular BT would serve, both as an example and ready-to-use BT for an organizational specific application, that wishes to optimize its process cycle time.

```
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <RecoveryNode number_of_retries="6" name="NavigateRecovery">
      <PipelineSequence name="NavigateWithReplanning">
        <RateController hz="1.0">
          <RecoveryNode number_of_retries="1" name="ComputePathToPose">
            <ComputePathToPose goal="{goal}" path="{path}" planner_id="GridBased"/>
            <ClearEntireCostmap name="ClearGlobalCostmap-Context" service_name=
↪"global_costmap/clear_entirely_global_costmap"/>
          </RecoveryNode>
        </RateController>
        <ReactiveSequence name="MonitorAndFollowPath">
          <PathLongerOnApproach path="{path}" prox_len="3.0" length_factor="2.0">
            <RetryUntilSuccessful num_attempts="1">
              <SequenceStar name="CancelingControlAndWait">
                <CancelControl name="ControlCancel"/>
                <Wait wait_duration="5"/>
              </SequenceStar>
            </RetryUntilSuccessful>
          </PathLongerOnApproach>
          <RecoveryNode number_of_retries="1" name="FollowPath">
            <FollowPath path="{path}" controller_id="FollowPath"/>
            <ClearEntireCostmap name="ClearLocalCostmap-Context" service_name="local_
↪costmap/clear_entirely_local_costmap"/>
          </RecoveryNode>
        </ReactiveSequence>
      </PipelineSequence>
      <ReactiveFallback name="RecoveryFallback">
        <GoalUpdated/>
        <RoundRobin name="RecoveryActions">
          <Sequence name="ClearingActions">
            <ClearEntireCostmap name="ClearLocalCostmap-Subtree" service_name="local_
↪costmap/clear_entirely_local_costmap"/>
            <ClearEntireCostmap name="ClearGlobalCostmap-Subtree" service_name=
↪"global_costmap/clear_entirely_global_costmap"/>
          </Sequence>
          <Spin spin_dist="1.57"/>
          <Wait wait_duration="5"/>
          <BackUp backup_dist="0.30" backup_speed="0.05"/>
        </RoundRobin>
      </ReactiveFallback>
    </RecoveryNode>
  </BehaviorTree>
</root>
```

A complete demo of this BT can be seen in the video below:

4.9.6 Navigate To Pose With Consistent Replanning And If Path Becomes Invalid

This behavior tree implements a significantly more mature version of the behavior tree on *Nav2 Behavior Trees*. It navigates from a starting point to a single point goal in freespace. It contains both use of custom recoveries in specific sub-contexts as well as a global recovery subtree for system-level failures. It also provides the opportunity for users to retry tasks multiple times before returning a failed state.

The `ComputePathToPose` and `FollowPath` BT nodes both also specify their algorithms to utilize. By convention we name these by the style of algorithms that they are (e.g. not `DWB` but rather `FollowPath`) such that a behavior tree or application developer need not worry about the technical specifics. They just want to use a path following controller.

In this behavior tree, we attempt to retry the entire navigation task 6 times before returning to the caller that the task has failed. This allows the navigation system ample opportunity to try to recovery from failure conditions or wait for transient issues to pass, such as crowding from people or a temporary sensor failure.

In nominal execution, replanning can be triggered by an a invalid previous path, a new goal or if a new path has not been created for 10 seconds. If the planner or controller fails, it will trigger contextually aware recoveries in its subtree. Currently, the recoveries will clear the global costmap if the planner fails and clear the local costmap if the controller fails. Additional context-specific recoveries can be added to these subtrees.

If these contextual recoveries fail, this behavior tree enters the recovery subtree. This subtree is reserved for system-level failures to help resolve issues like the robot being stuck or in a bad spot. This subtree has the `GoalUpdated` BT node which ticks every iteration to ensure responsiveness of new goals. Next, the recovery subtree will attempt the following recoveries: costmap clearing operations, spinning, waiting, and backing up. After each of the recoveries in the subtree, the main navigation subtree will be reattempted. If it continues to fail, the next recovery in the recovery subtree is ticked.

While this behavior tree does not make use of it, the `PlannerSelector`, `ControllerSelector`, and `GoalCheckerSelector` behavior tree nodes can also be helpful. Rather than hardcoding the algorithm to use (`GridBased` and `FollowPath`), these behavior tree nodes will allow a user to dynamically change the algorithm used in the navigation system via a ROS topic. It may be instead advisable to create different subtree contexts using condition nodes with specified algorithms in their most useful and unique situations. However, the selector nodes can be a useful way to change algorithms from an external application rather than via internal behavior tree control flow logic. It is better to implement changes through behavior tree methods, but we understand that many professional users have external applications to dynamically change settings of their navigators.

```xml
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <RecoveryNode number_of_retries="6" name="NavigateRecovery">
      <PipelineSequence name="NavigateWithReplanning">
        <RateController hz="2.0">
          <RecoveryNode number_of_retries="1" name="ComputePathToPose">
            <Fallback>
              <ReactiveSequence>
                <Inverter>
                  <PathExpiringTimer seconds="10" path="{path}"/>
                </Inverter>
                <Inverter>
                  <GlobalUpdatedGoal/>
                </Inverter>
                <IsPathValid path="{path}"/>
              </ReactiveSequence>
              <ComputePathToPose goal="{goal}" path="{path}" planner_id="GridBased"/>
            </Fallback>
            <ClearEntireCostmap name="ClearGlobalCostmap-Context" service_name=
↪"global_costmap/clear_entirely_global_costmap"/>
          </RecoveryNode>
```

(continues on next page)

(continued from previous page)

```
        </RateController>
        <RecoveryNode number_of_retries="1" name="FollowPath">
          <FollowPath path="{path}" controller_id="FollowPath"/>
          <ClearEntireCostmap name="ClearLocalCostmap-Context" service_name="local_
↪costmap/clear_entirely_local_costmap"/>
        </RecoveryNode>
      </PipelineSequence>
      <ReactiveFallback name="RecoveryFallback">
        <GoalUpdated/>
        <RoundRobin name="RecoveryActions">
          <Sequence name="ClearingActions">
            <ClearEntireCostmap name="ClearLocalCostmap-Subtree" service_name="local_
↪costmap/clear_entirely_local_costmap"/>
            <ClearEntireCostmap name="ClearGlobalCostmap-Subtree" service_name=
↪"global_costmap/clear_entirely_global_costmap"/>
          </Sequence>
          <Spin spin_dist="1.57"/>
          <Wait wait_duration="5"/>
          <BackUp backup_dist="0.30" backup_speed="0.05"/>
        </RoundRobin>
      </ReactiveFallback>
    </RecoveryNode>
  </BehaviorTree>
</root>
```

4.9.7 Follow Dynamic Point

This behavior tree implements a navigation behavior from a starting point, attempting to follow a dynamic point over time. This "dynamic point" could be a person, another robot, a virtual carrot, anything. The only requirement is that the pose you'd like to follow is published to the topic outlined in the `GoalUpdater` BT node.

In this tree, we replan at 1 hz just as we did in *Navigate To Pose* using the `ComputePathToPose` node. However, this time when we replan, we update the `goal` based on the newest information in on the updated goal topic. After we plan a path to this dynamic point, we use the `TruncatePath` node to remove path points from the end of the path near the dynamic point. This behavior tree node is useful so that the robot always remains at least `distance` away from the obstacle, even if it stops. It also smooths out any off path behavior involved with trying to path plan towards a probably occupied space in the costmap.

After the new path to the dynamic point is computed and truncated, it is again passed to the controller via the `FollowPath` node. However, note that it is under a `KeepRunningUntilFailure` decorator node ensuring the controller continues to execute until a failure mode. This behavior tree will execute infinitely in time until the navigation request is preempted or cancelled.

```
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <PipelineSequence name="NavigateWithReplanning">
      <RateController hz="1.0">
        <Sequence>
          <GoalUpdater input_goal="{goal}" output_goal="{updated_goal}">
            <ComputePathToPose goal="{updated_goal}" path="{path}" planner_id=
↪"GridBased"/>
          </GoalUpdater>
         <TruncatePath distance="1.0" input_path="{path}" output_path="{truncated_
↪path}"/>
        </Sequence>
```

(continues on next page)

(continued from previous page)

```
      </RateController>
      <KeepRunningUntilFailure>
        <FollowPath path="{truncated_path}" controller_id="FollowPath"/>
      </KeepRunningUntilFailure>
    </PipelineSequence>
  </BehaviorTree>
</root>
```

4.9.8 Odometry Calibration

This behavior tree drives the robot in a CCW square three times using the DriveOnHeading and Spin behaviors. The robot will traverse each side of the square at 0.2 (m/s) for 2 meters before making a 90 degree turn. This is a primitive experiment to measure odometric accuracy and can be used and repeated to tune parameters related to odometry to improve quality.

```
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <Repeat num_cycles="3">
      <Sequence name="Drive in a square">
        <DriveOnHeading dist_to_travel="2.0" speed="0.2" time_allowance="12"/>
        <Spin spin_dist="1.570796" is_recovery="false"/>
        <DriveOnHeading dist_to_travel="2.0" speed="0.2" time_allowance="12"/>
        <Spin spin_dist="1.570796" is_recovery="false"/>
        <DriveOnHeading dist_to_travel="2.0" speed="0.2" time_allowance="12"/>
        <Spin spin_dist="1.570796" is_recovery="false"/>
        <DriveOnHeading dist_to_travel="2.0" speed="0.2" time_allowance="12"/>
        <Spin spin_dist="1.570796" is_recovery="false"/>
      </Sequence>
    </Repeat>
  </BehaviorTree>
</root>
```

Nav2 is an incredibly reconfigurable project. It allows users to set many different plugin types, across behavior trees, core algorithms, status checkers, and more! This section highlights some of the example behavior tree xml files provided by default in the project to do interesting tasks. It should be noted that these can be modified for your specific application, or used as a guide to building your own application-specific behavior tree. These are some exemplary examples of how you can reconfigure your navigation behavior significantly by using behavior trees. Other behavior trees are provided by Nav2 in the `nav2_bt_navigator` package, but this section highlights the important ones.

A **very** basic, but functional, navigator can be seen below.

```
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <PipelineSequence name="NavigateWithReplanning">
      <DistanceController distance="1.0">
        <ComputePathToPose goal="{goal}" path="{path}"/>
      </DistanceController>
      <FollowPath path="{path}"/>
    </PipelineSequence>
  </BehaviorTree>
</root>
```

This behavior tree will simply plan a new path to `goal` every 1 meter (set by `DistanceController`) using

`ComputePathToPose`. If a new path is computed on the `path` blackboard variable, `FollowPath` will take this `path` and follow it using the server's default algorithm.

This tree contains:

- No recovery methods
- No retries on failure
- No selected planner or controller algorithms
- No nodes to contextually change settings for optimal performance
- No integration with automatic door, elevator, or other APIs
- No user provided custom BT nodes
- No subtrees for other behaviors like docking, following, etc.

All of this, and more, can be set and configured for your customized navigation logic in Nav2.

4.10 Navigation Plugins

There are a number of plugin interfaces for users to create their own custom applications or algorithms with. Namely, the costmap layer, planner, controller, behavior tree, and behavior plugins. A list of all known plugins are listed here below for ROS 2 Navigation. If you know of a plugin, or you have created a new plugin, please consider submitting a pull request with that information.

This file can be found and editted under `sphinx_docs/plugins/index.rst`. For tutorials on creating your own plugins, please see *Writing a New Costmap2D Plugin*, *Writing a New Behavior Tree Plugin*, *Writing a New Controller Plugin*, *Writing a New Planner Plugin*, *Writing a New Behavior Plugin*, or *Writing a New Navigator Plugin*.

4.10.1 Behavior-Tree Navigators

Plugin Name	Creator	Description
NavigateToPoseNavigator	Steve Macenski	Point-to-point navigation via a behavior tree action server
NavigateThroughPosesNavigator	Steve Macenski	Point-through-points navigation via a behavior tree action server

4.10.2 Costmap Layers

Plugin Name	Creator	Description
Voxel Layer	Eitan Marder-Eppstein	Maintains persistant 3D voxel layer using depth and laser sensor readings and raycasting to clear free space
Range Layer	David Lu	Uses a probabalistic model to put data from sensors that publish range msgs on the costmap
Static Layer	Eitan Marder-Eppstein	Gets static `map` and loads occupancy information into costmap
Inflation Layer	Eitan Marder-Eppstein	Inflates lethal obstacles in costmap with exponential decay
Obstacle Layer	Eitan Marder-Eppstein	Maintains persistent 2D costmap from 2D laser scans with raycasting to clear free space
Spatio-Temporal Voxel Layer	Steve Macenski	Maintains temporal 3D sparse volumetric voxel grid with decay through sensor models
Non-Persistent Voxel Layer	Steve Macenski	Maintains 3D occupancy grid consisting only of the most sets of measurements
Denoise Layer	Andrey Ryzhikov	Filters noise-induced standalone obstacles or small obstacles groups

4.10.3 Costmap Filters

Plugin Name	Creator	Description
Keepout Filter	Alexey Merzlyakov	Maintains keep-out/safety zones and preferred lanes for moving
Speed Filter	Alexey Merzlyakov	Limits maximum velocity of robot in speed restriction areas
Binary Filter	Alexey Merzlyakov	Enables binary (boolean) mask behavior to trigger actions.

4.10.4 Controllers

Plugin Name	Creator	Description	Drivetrain support
DWB Controller	David Lu!!	A highly configurable DWA implementation with plugin interfaces	Differential, Omnidirectional, Legged
TEB Controller	Christoph Rösmann	A MPC-like controller suitable for ackermann, differential, and holonomic robots.	**Ackermann**, Legged, Omnidirectional, Differential
Regulated Pure Pursuit	Steve Macenski	A service / industrial robot variation on the pure pursuit algorithm with adaptive features.	**Ackermann**, Legged, Differential
MPPI Controller	Steve Macenski Aleksei Budyakov	A predictive MPC controller with modular & custom cost functions that can accomplish many tasks.	Differential, Omni, **Ackermann**
Rotation Shim Controller	Steve Macenski	A “shim” controller to rotate to path heading before passing to main controller for tracking.	Differential, Omni, model rotate in place

4.10.5 Planners

Plugin Name	Creator	Description	Drivetrain support
NavFn Planner	Eitan Marder-Eppstein & Kurt Konolige	A navigation function using A* or Dijkstras expansion, assumes 2D holonomic particle	Differential, Omnidirectional, Legged
SmacPlannerHybrid (formerly *SmacPlanner*)	Steve Macenski	A SE2 Hybrid-A* implementation using either Dubin or Reeds-shepp motion models with smoother and multi-resolution query. Cars, car-like, and ackermann vehicles. Kinematically feasible.	**Ackermann**, Differential, Omnidirectional, Legged
SmacPlanner2D	Steve Macenski	A 2D A* implementation Using either 4 or 8 connected neighborhoods with smoother and multi-resolution query	Differential, Omnidirectional, Legged
SmacPlannerLattice	Steve Macenski	An implementation of State Lattice Planner using pre-generated minimum control sets for kinematically feasible planning with any type of vehicle imaginable. Includes generator script for Ackermann, diff, omni, and legged robots.	Differential, Omnidirectional, Ackermann, Legged, Arbitrary / Custom
ThetaStarPlanner	Anshumaan Singh	An implementaion of Theta* using either 4 or 8 connected neighborhoods, assumes the robot as a 2D holonomic particle	Differential, Omnidirectional

4.10.6 Smoothers

Plugin Name	Creator	Description
Simple Smoother	Steve Macenski	A simple path smoother for infeasible (e.g. 2D) planners
Constrained Smoother	Matej Vargovcik & Steve Macenski	A path smoother using a constraints problem solver to optimize various criteria such as smoothness or distance from obstacles, maintaining minimum turning radius
Savitzky-Golay Smoother	Steve Macenski	A path smoother using a Savitzky-Golay filter to smooth the path via digital signal processing to remove noise from the path.

4.10.7 Behaviors

Plugin Name	Creator	Description
Clear Costmap	Eitan Marder-Eppstein	A service to clear the given costmap in case of incorrect perception or robot is stuck
Spin	Steve Macenski	Rotate behavior of configurable angles to clear out free space and nudge robot out of potential local failures
Back Up	Brian Wilcox	Back up behavior of configurable distance to back out of a situation where the robot is stuck
Wait	Steve Macenski	Wait behavior with configurable time to wait in case of time based obstacle like human traffic or getting more sensor data
Drive On Heading	Joshua Wallace	Drive on heading behavior with configurable distance to drive
Assisted Teleop	Joshua Wallace	AssistedTeleop behavior that scales teleop commands to prevent collisions.

4.10.8 Waypoint Task Executors

Plugin Name	Creator	Description
WaitAtWaypoint	Fetullah Atas	A plugin to execute a wait behavior on waypoint arrivals.
PhotoAtWaypoint	Fetullah Atas	A plugin to take and save photos to specified directory on waypoint arrivals.
InputAtWaypoint	Steve Macenski	A plugin to wait for user input before moving onto the next waypoint.

4.10.9 Goal Checkers

Plugin Name	Creator	Description
SimpleGoalChecker	David Lu!!	A plugin check whether robot is within translational distance and rotational distance of goal.
StoppedGoalChecker	David Lu!!	A plugin check whether robot is within translational distance , rotational distance of goal, and velocity threshold.

4.10.10 Progress Checkers

Plugin Name	Creator	Description
SimpleProgressChecker	David Lu!!	A plugin to check whether the robot was able to move a minimum distance in a given time to make progress towards a goal
PoseProgressChecker	Guillaume Doisy	A plugin to check whether the robot was able to move a minimum distance or angle in a given time to make progress towards a goal

4.10.11 Behavior Tree Nodes

Action Plugin Name	Creator	Description
Back Up Action	Michael Jeronimo	Calls backup behavior action
Drive On Heading Action	Joshua Wallace	Calls drive on heading behavior action
Assisted Teleop Action	Joshua Wallace	Calls assisted teleop behavior action
Clear Entire Costmap Service	Carl Delsey	Calls clear entire costmap service
Clear Costmap Except Region Service	Guillaume Doisy	Calls clear costmap except region service
Clear Costmap Around Robot Service	Guillaume Doisy	Calls clear costmap around robot service
Compute Path to Pose Action	Michael Jeronimo	Calls Nav2 planner server
Smooth Path Action	Matej Vargovcik	Calls Nav2 smoother server
Follow Path Action	Michael Jeronimo	Calls Nav2 controller server
Navigate to Pose Action	Michael Jeronimo	BT Node for other BehaviorTree.CPP BTs to call Navigation2 as a subtree action
Reinitalize Global Localization Service	Carl Delsey	Reinitialize AMCL to a new pose
Spin Action	Carl Delsey	Calls spin behavior action
Wait Action	Steve Macenski	Calls wait behavior action
Truncate Path	Francisco Martín	Modifies a path making it shorter
Truncate Path Local	Matej Vargovcik	Extracts a path section around robot
Planner Selector	Pablo Iñigo Blasco	Selects the global planner based on a topic input, otherwises uses a default planner id
Controller Selector	Pablo Iñigo Blasco	Selects the controller based on a topic input, otherwises uses a default controller id
Goal Checker Selector	Pablo Iñigo Blasco	Selects the goal checker based on a topic input, otherwises uses a default goal checker id
Smoother Selector	Owen Hooper	Selects the smoother based on a topic input, otherwises uses a default smoother id
Navigate Through Poses	Steve Macenski	BT Node for other BehaviorTree.CPP BTs to call Nav2's NavThroughPoses action
Remove Passed Goals	Steve Macenski	Removes goal poses passed or within a tolerance for culling old viapoints from path re-planning
Compute Path Through Poses	Steve Macenski	Computes a path through a set of poses rather than a single end goal pose using the planner plugin specified
Cancel Control Action	Pradheep Padmanabhan	Cancels Nav2 controller server
Cancel BackUp Action	Pradheep Padmanabhan	Cancels backup behavior action
Cancel Spin Action	Pradheep Padmanabhan	Cancels spin behavior action
Cancel Wait Action	Pradheep Padmanabhan	Cancels wait behavior action
Cancel Drive on Heading Action	Joshua Wallace	Cancels drive on heading behavior action
Cancel Assisted Teleop Action	Joshua Wallace	Cancels assisted teleop behavior action

Condition Plugin Name	Creator	Description
Goal Reached Condition	Carl Delsey	Checks if goal is reached within tol.
Goal Updated Condition	Aitor Miguel Blanco	Checks if goal is preempted.
Globally Updated Goal Condition	Joshua Wallace	Checks if goal is preempted in the global BT context
Initial Pose received Condition	Carl Delsey	Checks if initial pose has been set
Is Stuck Condition	Michael Jeronimo	Checks if robot is making progress or stuck
Transform Available Condition	Steve Macenski	Checks if a TF transformation is available. When succeeds returns success for subsequent calls.
Distance Traveled Condition	Sarthak Mittal	Checks is robot has traveled a given distance.
Time Expired Condition	Sarthak Mittal	Checks if a given time period has passed.
Is Battery Low Condition	Sarthak Mittal	Checks if battery percentage is below a specified value.
Is Path Valid Condition	Joshua Wallace	Checks if a path is valid by making sure there are no LETHAL obstacles along the path.
Path Expiring Timer	Joshua Wallace	Checks if the timer has expired. The timer is reset if the path gets updated.
Are Error Codes Present	Joshua Wallace	Checks if the specified error codes are present.
Would A Controller Recovery Help	Joshua Wallace	Checks if a controller recovery could help clear the controller server error code.
Would A Planner Recovery Help	Joshua Wallace	Checks if a planner recovery could help clear the planner server error code.
Would A Smoother Recovery Help	Joshua Wallace	Checks if a Smoother recovery could help clear the smoother server error code.
Is Battery Charging Condition	Alberto Tudela	Checks if the battery is charging.

Decorator Plugin Name	Creator	Description
Rate Controller	Michael Jeronimo	Throttles child node to a given rate
Distance Controller	Sarthak Mittal	Ticks child node based on the distance traveled by the robot
Speed Controller	Sarthak Mittal	Throttles child node to a rate based on current robot speed.
Goal Updater	Francisco Martín	Updates the goal received via topic subscription.
Single Trigger	Steve Macenski	Triggers nodes/subtrees below only a single time per BT run.
PathLongerOnApproach	Pradheep Padmanabhan	Triggers child nodes if the new global path is significantly larger than the old global path on approach to the goal

Control Plugin Name	Creator	Description
Pipeline Sequence	Carl Delsey	A variant of a sequence node that will re-tick previous children even if another child is running
Recovery	Carl Delsey	Node must contain 2 children and returns success if first succeeds. If first fails, the second will be ticked. If successful, it will retry the first and then return its value
Round Robin	Mohammad Haghigh-ipanah	Will tick `i` th child until a result and move on to `i+1`

4.11 Migration Guides

Navigation2 guides for migration between distributions.

4.11.1 Dashing to Eloquent

Moving from ROS 2 Dashing to Eloquent, a number of stability improvements were added that we will not specifically address here.

New Packages

Navigation2 now includes a new package `nav2_waypoint_follower`. The waypoint follower is an action server that will take in a list of waypoints to follow and follow them in order. There is a parameter `stop_on_failure` whether the robot should continue to the next waypoint on a single waypoint failure, or to return fail to the action client. The waypoint follower is also a reference application for how to use the Navigation2 action server to complete a basic autonomy task.

Navigation2 now supports new algorithms for control and SLAM. The Timed-Elastic Band (TEB) controller was implemented and can be found here. It is its own controller plugin that can be used instead of the DWB controller. Nav2 also supports SLAM Toolbox as the default SLAM implementation for ROS 2. This replaces the use of Cartographer.

New Plugins

Eloquent introduces back in pluginlib plugins to the navigation stack. `nav2_core` defines the plugin header interfaces to be used to implement controller, planner, recovery, and goal checker plugins. All algorithms (NavFn, DWB, recoveries) were added as plugin interfaces and the general packages for servers were created. `nav2_planner` is the action server for planning that hosts a plugin for the planner. `nav2_controller` is the action server for controller that hosts a plugin for the controller. `nav2_recovery` is the action server for recovery that hosts a plugin for recovery.

New recovery plugins were added including backup, which will take in a distance to back up, if collision-free. Additionally, the wait recovery was added that will wait a configurable period of time before trying to navigate again. This plugin is especially helpful for time-dependent obstacles or pausing navigation for a scene to become less dynamic.

Many new behavior tree nodes were added. These behavior tree nodes are hard-coded in the behavior tree engine. Behavior tree cpp v3 supports plugins and will be converted in the next release.

Navigation2 Architectural Changes

The `nav2_world_model` package was removed. The individual `nav2_planner` and `nav2_controller` servers now host their relevent costmaps. This was done to reduce network traffic and ensure up-to-date information for the safety-critical elements of the system. As above mentions, plugins were introduced into the stack and these servers each host plugins for navigation, control, and costmap layers.

Map server was substantially refactored but the external API remains the same. It now uses the SDL library for image loading.

TF-based positioning is now used for pose-estimation everywhere in the stack. Prior, some elements of the navigation stack only updated its pose from the `/amcl_pose` topic publishing at an irregular rate. This is obviously low-accuracy and high-latency. All positioning is now based on the TF tree from the global frame to the robot frame.

Prior to Eloquent, there were no ROS 2 action servers and clients available. Navigation2, rather, used an interface we called Tasks. Eloquent now contains actions and a simple action server interface was created and is used now throughout the stack. Tasks were removed.

4.11.2 Eloquent to Foxy

Moving from ROS 2 Eloquent to Foxy, a number of stability improvements were added that we will not specifically address here. We will specifically mention, however, the reduction in terminal noise. TF2 transformation timeout errors and warnings on startup have been largely removed or throttled to be more tractable. Additionally, message filters filling up resulting in messages being dropped were resolved in costmap 2d.

General

The lifecycle manager was split into 2 unique lifecycle managers. They are the `navigation_lifecycle_manager` and `localization_lifecycle_manager`. This gives each process their own manager to allow users to switch between SLAM and localization without effecting Navigation. It also reduces the redundant code in `nav2_bringup`.

The lifecycle manager also now contains `Bond` connections to each lifecycle server. This means that if a server crashes or exits, the lifecycle manager will be constantly checking and transition down its lifecycle nodes for safety. This acts as a watchdog during run-time to complement the lifecycle manager's transitioning up and down from active states. See this PR for details.

A fix to the BT navigator was added to remove a rare issue where it may crash due to asynchronous issues. As a result, a behavior tree is created for each navigation request rather than resetting an existing tree. The creation of this tree will add a small amount of latency. Proposals to reduce this latency will be considered before the next release.

Server Updates

All plugin servers (controller, planner, recovery) now supports the use of multiple plugins. This can be done by loading a map of plugins, mapping the name of the plugin to its intended use-case. Each server defines a parameter where the list of names for the plugins to be loaded can be defined.

Server Name	Plugin Parameter
Controller Server	progress_checker_plugin
Controller Server	goal_checker_plugin
Controller Server	controller_plugins
Planner Server	planner_plugins
Recovery Server	recovery_plugins
Costmap Node	plugins

The type of plugin to be mapped to a particular name has to be defined using the `plugin` parameter in the plugin namespace for each name defined in the server plugin list. Each name in the plugin parameter list is expected to have the `plugin` parameter defined.

An example: `controller_server` defines the parameter `controller_plugins` where a list of plugin names can be defined:

```
controller_server:
  ros__parameters:
    controller_plugins: ["FollowPath", "DynamicFollowPath"]
    FollowPath:
      plugin: "dwb_core/DWBLocalPlanner"
      max_vel_x: 0.26
    DynamicFollowPath:
      plugin: "teb_local_planner/TEBLocalPlanner"
      max_vel_x: 0.5
```

`FollowPath` controller is of type `dwb_local_planner/DWBLocalPlanner` and `DynamicFollowPath` of type `teb_local_planner/TEBLocalPlanner`. Each plugin will load the parameters in their namespace, e.g. `FollowPath.max_vel_x`, rather than globally in the server namespace. This will allow multiple plugins of the same type with different parameters and reduce conflicting parameter names.

DWB Contains new parameters as an update relative to the ROS 1 updates, see here for more information. Additionally, the controller and planner interfaces were updated to include a `std::string name` parameter on initialization. This was added to the interfaces to allow the plugins to know the namespace it should load its parameters in. E.g. for a controller to find the parameter `FollowPath.max_vel_x`, it must be given its name, `FollowPath` to get this parameter. All plugins will be expected to look up parameters in the namespace of its given name.

New Plugins

Many new behavior tree nodes were added. These behavior tree nodes are now BT plugins and dynamically loadable at run-time using behavior tree cpp v3. The default behavior trees have been upgraded to stop the recovery behaviours and trigger a replanning when the navigation goal is preempted. See `nav2_behavior_tree` for a full listing, or *Navigation Plugins* for the current list of behavior tree plugins and their descriptions. These plugins are set as default in the `nav2_bt_navigator` but may be overridden by the `bt_plugins` parameter to include your specific plugins.

Original GitHub tickets:

- DistanceController
- SpeedController
- GoalUpdatedCondition
- DistanceTraveledCondition
- TimeExpiredCondition
- UpdateGoal
- TruncatePath
- IsBatteryLowCondition
- ProgressChecker
- GoalChecker

Map Server Re-Work

map_saver was re-worked and divided into 2 parts: CLI and server. CLI part is a command-line tool that listens incoming map topic, saves map once into a file and finishes its work. This part is remained to be almost untouched: CLI executable was renamed from map_saver to map_saver_cli without changing its functionality. Server is a new part. It spins in the background and can be used to save map continuously through a save_map service. By each service request it tries to listen incoming map topic, receive a message from it and write obtained map into a file.

map_server was dramatically simplified and cleaned-up. OccGridLoader was merged with MapServer class as it is intended to work only with one OccupancyGrid type of messages in foreseeable future.

Map Server now has new map_io dynamic library. All functions saving/loading OccupancyGrid messages were moved from map_server and map_saver here. These functions could be easily called from any part of external ROS 2 code even if Map Server node was not started.

map_loader was completely removed from nav2_util. All its functionality already present in map_io. Please use it in your code instead.

Please refer to the original GitHub ticket and Map Server README for more information.

New Particle Filter Messages

New particle filter messages for particle clouds were added to include the particle weights along with their poses. nav2_msgs/Particle defines a single particle with a pose and a weight in a particle cloud. nav2_msgs/ParticleCloud defines a set of particles, each with a pose and a weight.

AMCL now publishes its particle cloud as a nav2_msgs/ParticleCloud instead of a geometry_msgs/PoseArray.

See here for more information.

Selection of Behavior Tree in each navigation action

The NavigateToPose action allows now to select in the action request the behavior tree to be used by bt_navigator for carrying out the navigation action through the string behavior_tree field. This field indicates the absolute path of the xml file that will be used to use to carry out the action. If no file is specified, leaving this field empty, the default behavior tree specified in the default_bt_xml_filename parameter will be used.

This functionality has been discussed in the ticket #1780, and carried out in the pull request #1784.

FollowPoint Capability

A new behavior tree followpoint.xml has added. This behavior tree makes a robot follow a dynamically generated point, keeping a certain distance from the target. This can be used for moving target following maneuvers.

This functionality has been discussed in the ticket #1660, and carried out in the pull request #1859.

New Costmap Layer

The range sensor costmap has not been ported to navigation2 as `nav2_costmap_2d::RangeSensorLayer"`. It uses the same probabilistic model as the ROS1 layer as well as much of the same interface. Documentation on parameters has been added to docs/parameters and the navigation.ros.org under `Configuration Guide`.

4.11.3 Foxy to Galactic

Moving from ROS 2 Foxy to Galactic, a number of stability improvements were added that we will not specifically address here.

NavigateToPose Action Feedback updates

The NavigateToPose action feedback has two improvements:

- `distance_remaining` now integrates path poses to report accurate distance remaining to go. Previously, this field reported the euclidean distance between the current pose and the goal pose.
- Addition of `estimated_time_remaining` field. This field reports the estimated time remaining by dividing the remaining distance by the current speed.

NavigateToPose BT-node Interface Changes

The NavigateToPose input port has been changed to PoseStamped instead of Point and Quaternion.

See *NavigateToPose* for more information.

NavigateThroughPoses and ComputePathThroughPoses Actions Added

The `NavigateThroughPoses` action has been added analog to the `NavigateToPose`. Rather than going to a single position, this Action will allow a user to specify a number of hard intermediary pose constraints between the start and final pose to plan through. The new `ComputePathThroughPoses` action has been added to the `planner_server` to process these requests through `N goal_poses`.

The `ComputePathThroughPoses` action server will take in a set of `N` goals to achieve, plan through each pose and concatenate the output path for use in navigation. The controller and navigator know nothing about the semantics of the generated path, so the robot will not stop or slow on approach to these goals. It will rather continue through each pose as it were any other point on the path continuously. When paired with the `SmacPlanner`, this feature can be used to generate **completely kinematically feasible trajectories through pose constraints**.

If you wish to stop at each goal pose, consider using the waypoint follower instead, which will stop and allow a user to optionally execute a task plugin at each pose.

ComputePathToPose BT-node Interface Changes

The `start` input port has been added to optionally allow the request of a path from `start` to `goal` instead of from the current position of the robot to `goal`.

See *ComputePathToPose* for more information.

ComputePathToPose Action Interface Changes

- The goal pose field `pose` was changed to `goal`.
- The PoseStamped field `start` has been added.
- The bool field `use_start` has been added.

These two additional fields have been added to optionally allow, when `use_start` is true, the request of a path from `start` to `goal` instead of from the current position of the robot to `goal`. Corresponding changes have been done of the Planner Server.

BackUp BT-node Interface Changes

The `backup_dist` and `backup_speed` input ports should both be positive values indicating the distance to go backward respectively the speed with which the robot drives backward.

BackUp Recovery Interface Changes

`speed` in a backup recovery goal should be positive indicating the speed with which to drive backward. `target.x` in a backup recovery goal should be positive indicating the distance to drive backward. In both cases negative values are silently inverted.

Nav2 Controllers and Goal Checker Plugin Interface Changes

As of this PR 2247, the `controller` plugins will now be given a pointer to the current goal checker in use of the navigation task in `computeAndPublishVelocity()`. This is geared to enabling controllers to have access to predictive checks for goal completion as well as access to the state information of the goal checker plugin.

The `goal_checker` plugins also have the change of including a `getTolerances()` method. This method allows a goal checker holder to access the tolerance information of the goal checker to consider at the goal. Each field of the `pose` and `velocity` represents the maximum allowable error in each dimension for a goal to be considered completed. In the case of a translational tolerance (combined X and Y components), each the X and Y will be populated with the tolerance value because it is the **maximum** tolerance in the dimension (assuming the other has no error). If the goal checker does not contain any tolerances for a dimension, the `numeric_limits<double> lowest()` value is utilized in its place.

FollowPath goal_checker_id attribute

For example: you could use for some specific navigation motion a more precise goal checker than the default one that it is used in usual motions.

```
<FollowPath path="{path}" controller_id="FollowPath" goal_checker_id="precise_goal_
↪checker" server_name="FollowPath" server_timeout="10"/>
```

- The previous usage of the `goal_checker_plugin` parameter to declare the controller_server goal_checker is now obsolete and removed.
- The controller_server parameters now support the declaration of a list of goal checkers `goal_checker_plugins` mapped to unique identifier names, such as is the case with `FollowPath` and `GridBased` for the controller and planner plugins, respectively.
- The specification of the selected goal checker is mandatory when more than one checker is defined in the controller_server parameter configuration. If only one goal_checker is configured in the controller_server it is selected by default even if no goal_checker is specified.

Below it is shown an example of goal_checker configuration of the controller_server node.

```
controller_server:
  ros__parameters:
      goal_checker_plugins: ["general_goal_checker", "precise_goal_checker"]
      precise_goal_checker:
          plugin: "nav2_controller::SimpleGoalChecker"
          xy_goal_tolerance: 0.25
        yaw_goal_tolerance: 0.25
      general_goal_checker:
          plugin: "nav2_controller::SimpleGoalChecker"
          xy_goal_tolerance: 0.25
```

Groot Support

Live Monitoring and Editing of behavior trees with Groot is now possible. Switching bt-xmls on the fly through a new goal request is also included. This is all done without breaking any APIs. Enabled by default.

New Plugins

`nav2_waypoint_follower` has an action server that takes in a list of waypoints to follow and follow them in order. In some cases we might want robot to perform some tasks/behaviours at arrivals of these waypoints. In order to perform such tasks, a generic plugin interface *WaypointTaskExecutor* has been added to `nav2_core`. Users can inherit from this interface to implement their own plugin to perform more specific tasks at waypoint arrivals for their needs.

Several example implementations are included in `nav2_waypoint_follower`. `WaitAtWaypoint` and `PhotoAtWaypoint` plusings are included in `nav2_waypoint_follower` as run-time loadable plugins. `WaitAtWaypoint` simply lets robot to pause for a specified amount of time in milliseconds, at waypoint arrivals. While `PhotoAtWaypoint` takes photos at waypoint arrivals and saves the taken photos to specified directory, the format for taken photos also can be configured through parameters. All major image formats such as `png`, `jpeg`, `jpg` etc. are supported, the default format is `png`.

Loading a plugin of this type is done through `nav2_bringup/params/nav2_param.yaml`, by specifying plugin's name, type and it's used parameters.

```
waypoint_follower:
  ros__parameters:
    loop_rate: 20
    stop_on_failure: false
    waypoint_task_executor_plugin: "wait_at_waypoint"
      wait_at_waypoint:
        plugin: "nav2_waypoint_follower::WaitAtWaypoint"
        enabled: True
        waypoint_pause_duration: 0
```

Original GitHub tickets:

- WaypointTaskExecutor
- WaitAtWaypoint
- PhotoAtWaypoint
- InputAtWaypoint

Costmap Filters

A new concept interacting with spatial-dependent objects called "Costmap Filters" appeared in Galactic (more information about this concept could be found at *Navigation Concepts* page). Costmap filters are acting as a costmap plugins, applied to a separate costmap above common plugins. In order to make a filtered costmap and change robot's behavior in annotated areas, filter plugin reads the data came from filter mask. Then this data is being linearly transformed into feature map in a filter space. It could be passability of an area, maximum speed limit in m/s, robot desired direction in degrees or anything else. Transformed feature map along with the map/costmap, sensors data and current robot position is used in plugin's algorithms to make required updates in the resulting costmap and robot's behavor.

Architecturally, costmap filters consists from `CostmapFilter` class which is a basic class incorporating much common of its inherited filter plugins:

- `KeepoutFilter`: keep-out/safety zones filter plugin.
- `SpeedFilter`: slow/speed-restricted areas filter.
- Preferred lanes in industries. This plugin is covered by `KeepoutFilter` (see discussion in corresponding PR for more details).

Each costmap filter subscribes to filter info topic (publishing by Costmap Filter Info Publisher Server) having all necessary information for loaded costmap filter and filter mask topic. `SpeedFilter` additionally publishes maximum speed restricting messages targeted for a Controller to enforce robot won't exceed given limit.

High-level design of this concept could be found here. The functionality of costmap filters is being disscussed in the ticket #1263 and carried out by PR #1882. The following tutorials: *Navigating with Keepout Zones* and *Navigating with Speed Limits* will help to easily get involved with `KeepoutFilter` and `SpeedFilter` functionalities.

SmacPlanner

A new package, `nav2_smac_planner` was added containing 4 or 8 connected 2D A*, and Dubin and Reed-shepp model hybrid-A* with smoothing, multi-resolution query, and more.

The `nav2_smac_planner` package contains an optimized templated A* search algorithm used to create multiple A*-based planners for multiple types of robot platforms. We support differential-drive and omni-directional drive robots using the `SmacPlanner2D` planner which implements a cost-aware A* planner. We support cars, car-like, and ackermann vehicles using the `SmacPlanner` plugin which implements a Hybrid-A* planner. This plugin is also useful for curvature constrained planning, like when planning robot at high speeds to make sure they don't flip over or otherwise skid out of control.

The `SmacPlanner` fully-implements the Hybrid-A* planner as proposed in Practical Search Techniques in Path Planning for Autonomous Driving, including hybrid searching, CG smoothing, analytic expansions and hueristic functions.

ThetaStarPlanner

A new package, `nav2_theta_star_planner` was added containing 4 or 8 connected Theta* implementation for 2D maps.

This package implements an optimized version of the Theta* Path Planner (specifically the Lazy Theta* P variant) to plan any-angled paths for differential-drive and omni-directional robots, while also taking into account the costmap costs. This plugin is useful for the cases where you might want to plan a path at a higher rate but without requiring extremely smooth paths around the corners which, for example, could be handled by a local planner/controller.

RegulatedPurePursuitController

A new package, `nav2_regulated_pure_pursuit_controller` was added containing a novel varient of the Pure Pursuit algorithm. It also includes configurations to enable Pure Pursuit and Adaptive Pure Pursuit variations as well.

This variation is specifically targeting service / industrial robot needs. It regulates the linear velocities by curvature of the path to help reduce overshoot at high speeds around blind corners allowing operations to be much more safe. It also better follows paths than any other variation currently available of Pure Pursuit. It also has heuristics to slow in proximity to other obstacles so that you can slow the robot automatically when nearby potential collisions. It also implements the Adaptive lookahead point features to be scaled by velocities to enable more stable behavior in a larger range of translational speeds.

There's more this does, that that's the general information. See the package's `README` for more.

Costmap2D `current_` Usage

In costmap2D, `current_` was used in ROS1 to represent whether a costmap layer was still enabled and actively processing data. It would be turned to `false` only under the situation that the expected update rate of a sensor was not met, so it was getting stale or no messages. It acts as a fail-safe for if a navigation sensor stops publishing.

In galactic, that will remain turn, however it will also add additional capabilities. It is also now set to `false` when a costmap is reset due to clearing or other navigation recoveries. That stops the robot from creating a plan or control effort until after the costmap has been updated at least once after a reset. This enables us to make sure we cannot ever create a path or control with a completely empty costmap, potentially leading to collisions, due to clearing the costmap and then immediately requesting an algorithm to run.

Standard time units in parameters

To follow the SI units outlined in REP-103 to the "T" nodes below were modified to use seconds consistently in every parameter. Under each node name you can see which parameters changed to seconds instead of using milliseconds.

- lifecycle manager
  - `bond_timeout_ms` became `bond_timeout` in seconds
- smac planner
  - `max_planning_time_ms` became `max_planning_time` in seconds
- map saver
  - `save_map_timeout` in seconds

Ray Tracing Parameters

Raytracing functionality was modified to include a minimum range parameter from which ray tracing starts to clear obstacles to avoid incorrectly clearing obstacles too close to the robot. This issue was mentioned in ROS Answers. An existing parameter `raytrace_range` was renamed to `raytrace_max_range` to reflect the functionality it affects. The renamed parameters and the plugins that they belong to are mentioned below. The changes were introduced in this pull request.

- obstacle_layer plugin
  - `raytrace_min_range` controls the minimum range from which ray tracing clears obstacles from the costmap

    - `raytrace_max_range` controls the maximum range to which ray tracing clears obstacles from the costmap
- voxel_layer plugin
    - `raytrace_min_range` controls the minimum range from which ray tracing clears obstacles from the costmap
    - `raytrace_max_range` controls the maximum range to which ray tracing clears obstacles from the costmap

Obstacle Marking Parameters

Obstacle marking was modified to include a minimum range parameter from which obstacles are marked on the costmap to prevent addition of obstacles in the costmap due to noisy and incorrect measurements. This modification is related to the change with the raytracing parameters. The renamed parameters, newly added parameters and the plugins they belong to are given below.

- obstacle_layer plugin
    - `obstacle_min_range` controls the minimum range from which obstacle are marked on the costmap
    - `obstacle_max_range` controls the maximum range to which obstacles are marked on the costmap
- voxel_layer plugin
    - `obstacle_min_range` controls the minimum range from which obstacle are marked on the costmap
    - `obstacle_max_range` controls the maximum range to which obstacles are marked on the costmap

Recovery Action Changes

The recovery actions, `Spin` and `BackUp` were modified to correctly return `FAILURE` if the recovery action is aborted due to a potential collision. Previously, these actions incorrectly always returned `SUCCESS`. Changes to this resulted in downstream action clients, such as the default behavior tree. The changes were introduced in this pull request 1855.

Default Behavior Tree Changes

The default behavior tree (BT) `navigate_w_replanning_and_recovery.xml` has been updated to allow for replanning in between recoveries. The changes were introduced in this PR 1855. Additionally, an alternative BT `navigate_w_replanning_and_round_robin_recovery.xml` was removed due to similarity with the updated default BT.

NavFn Planner Parameters

The NavFn Planner has now its 3 parameters reconfigurable at runtime (`tolerance`, `use_astar` and `allow_unknown`). The changes were introduced in this pull request 2181.

New ClearCostmapExceptRegion and ClearCostmapAroundRobot BT-nodes

The ClearEntireCostmap action node was already implemented but the ClearCostmapExceptRegion and ClearCostmapAroundRobot BT nodes calling the sister services `(local_or_global)_costmap/clear_except_(local_or_global)_costmap` and `clear_around_(local_or_global)_costmap` of Costmap 2D were missing, they are now implemented in a similar way. They both expose a `reset_distance` input port. See *ClearCostmapExceptRegion* and *ClearCostmapAroundRobot* for more. The changes were introduced in this pull request 2204.

New Behavior Tree Nodes

A new behavior tree node was added and dynamically loadable at run-time using behavior tree cpp v3. See `nav2_behavior_tree` for a full listing, or *Navigation Plugins* for the current list of behavior tree plugins and their descriptions. These plugins are set as default in the `nav2_bt_navigator` but may be overridden by the `bt_plugins` parameter to include your specific plugins.

Original GitHub tickets:

- SingleTrigger
- PlannerSelector
- ControllerSelector
- GoalCheckerSelector
- NavigateThroughPoses
- RemovePassedGoals
- ComputePathThroughPoses

Additionally, behavior tree nodes were modified to contain their own local executors to spin for actions, topics, services, etc to ensure that each behavior tree node is independent of each other (e.g. spinning in one BT node doesn't trigger a callback in another).

sensor_msgs/PointCloud to sensor_msgs/PointCloud2 Change

Due to deprecation of sensor_msgs/PointCloud the topics which were publishing sensor_msgs/PointCloud are converted to sensor_msgs/PointCloud2. The details on these topics and their respective information are listed below.

- `clearing_endpoints` topic in `voxel_layer` plugin of `nav2_costmap_2d` package
- `voxel_marked_cloud` and `voxel_unknown_cloud` topic in `costmap_2d_cloud` node of `nav2_costmap_2d` package
- `cost_cloud` topic of `publisher.cpp` of `dwb_core` package.

These changes were introduced in pull request 2263.

ControllerServer New Parameter failure_tolerance

A new parameter `failure_tolerance` was added to the Controller Server for tolerating controller plugin exceptions without failing immediately. It is analogous to `controller_patience` in ROS(1) Nav. See *Controller Server* for description. This change was introduced in this pull request 2264.

Removed BT XML Launch Configurations

The launch python configurations for CLI setting of the behavior tree XML file has been removed. Instead, you should use the yaml files to set this value. If you, however, have a `path` to the yaml file that is inconsistent in a larger deployment, you can use the `RewrittenYaml` tool in your parent launch file to remap the default XML paths utilizing the `get_shared_package_path()` directory finder (or as you were before in python3).

The use of map subscription QoS launch configuration was also removed, use parameter file. This change was introduced in this pull request 2295.

Nav2 RViz Panel Action Feedback Information

The Nav2 RViz Panel now displays the action feedback published by `nav2_msgs/NavigateToPose` and `nav2_msgs/NavigateThroughPoses` actions. Users can find information like the estimated time of arrival, distance remaining to goal, time elapsed since navigation started, and number of recoveries performed during a navigation action directly through the RViz panel. This feature was introduced in this pull request 2338.

4.11.4 Galactic to Humble

Moving from ROS 2 Galactic to Humble, a number of stability improvements were added that we will not specifically address here.

Major improvements to Smac Planners

The Smac Planner was significantly improved, of both the 2D and Hybrid-A* implementations, making the paths better, faster, and of higher quality.

- Collision checker rejects collisions faster and queries the costmap for coordinates less often
- Zero-copy collision checking object
- precompute collision checking footprint orientations so no need for trig at runtime
- Only checking full SE2 footprint when the robot is in the possibly inscribed zones
- Computing the possibly inscribed zones, or the cost over which some part of the footprint may be in collision with a boundary to check the full footprint. Else, check center cost since promised to not be in a potential collision state
- Renaming Hybrid-A* planner to SmacPlannerHybrid
- Precomputing the Reedshepp and Dubin paths offline so at runtime its just a lookup table
- Replacing the wavefront heuristic with a new, and novel, heuristic dubbed the obstacle heuristic. This computes a Dijkstra's path using Differential A* search taking into account the 8 connected space, as well as weights for the cost at the positions to guide the heuristic into the center of aisle ways. It also downsamples the costmap such that it can reduce the number of expansions by 75% and have a very small error introduced into the heuristic by being off by at most a partial fraction of a single cell distance

- Improvements to the analytic expansion algorithm to remove the possibility of loops at the end of paths, whenever possible to remove
- Improving analytic expansions to provide maximum path length to prevent skirting close to obstacles
- 2D A* travel cost and heuristic improvements to speed up planning times and also increase the path quality significantly
- Replaced smoother with a bespoke gradient descent implementation
- Abstract out common utilities of planners into a utils file
- tuned cost functions
- precomputed obstacle heuristic using dynamic programming to expand only the minimum number of nodes
- A caching heuristic setting to enable 25hz planning rates using cached obstacle heuristic values when the goal remains the same
- Leveraging the symmetry in the dubin and reeds-sheep space to reduce cache size by 50% to increase the window size available for heuristic lookup.
- Precompute primitives at all orientation bins
- SmacPlanner2D parameters are now all reconfigurable
- Both Hybrid-A* and State Lattice planners are now fully admissible
- Hybrid-A* and State Lattice have had their parameterization for path smoothing readded.
- The smoother now enables kinematically feasible boundary conditions.
- State Lattice supports turning in place primitive types
- Retrospective penalty added to speed up the planner, making it prioritize later search branches before earlier ones, which have negligible chance to improve path in vast majority of situations

The tl;dr of these improvements is:

- Plans are 2-3x as fast as they were before, well under 200ms for nearly all situations, making it as fast as NavFn and Global Planner (but now kinematically feasible). Typical planning times are sub-100ms without even making use of the caching or downsampling features.
- Paths are of significantly higher quality via improved smoothers and a novel heuristic that steers the robot towards the center of aisleways implicitly. This makes smoother paths that are also further from obstacles whenever possible.
- Using caching or downsampler parameterizations, can easily achieve path planning with sub-50ms in nearly any sized space.
- Smoother is now able to do more refinements and can create kinematically feasible boundary conditions, even while reversing.

Additional improvements were made to include a `analytic_expansion_max_length` parameter such that analytic expansions are limited in their potential length. If the length is too far, reject this expansion. This prevents unsafe shortcutting of paths into higher cost areas far out from the goal itself, let search to the work of getting close before the analytic expansion brings it home. This should never be smaller than 4-5x the minimum turning radius being used, or planning times will begin to spike.

Further, the traversal cost and heuristic cost computations were updated **requiring retuning of your penalty functions** if you have a previously existing configuration. Defaults of the algorithm were also retuned appropriately to the change for similar our of the box behavior as before (to use as a reference).

Simple (Python) Commander

This PR 2411 introduces a new package to Nav2, called the `nav2_simple_commander`. It is a set of functions in an object, `BasicNavigator`, which can be used to build Nav2-powered autonomy tasks in Python3 without concerning yourself with the Nav2, ROS 2, or Action server details. It contains a simple API taking common types (primarily `PoseStamped`) and handles all of the implementation details behind the hood. For example, this is a simple navigation task using this API:

```python
def main():
    rclpy.init()
    navigator = BasicNavigator()

    # Set our demo's initial pose
    initial_pose = PoseStamped()
    ... populate pose ...
    navigator.setInitialPose(initial_pose)

    # Wait for navigation to fully activate
    navigator.waitUntilNav2Active()

    # Go to our demos first goal pose
    goal_pose = PoseStamped()
    ... populate pose ...
    navigator.goToPose(goal_pose)

    while not navigator.isTaskComplete():
        feedback = navigator.getFeedback()
        ... do something with feedback ...

        # Basic navigation timeout
        if Duration.from_msg(feedback.navigation_time) > Duration(seconds=600.0):
            navigator.cancelNav()

    result = navigator.getResult()
    if result == TaskResult.SUCCEEDED:
        print('Goal succeeded!')
    elif result == TaskResult.CANCELED:
        print('Goal was canceled!')
    elif result == TaskResult.FAILED:
        print('Goal failed!')
```

The full API can be found in the README of the package. A number of well commented examples and demos can also be found in the package's source code at the link prior.

Reduce Nodes and Executors

In order for nav2 to make the best use of ROS 2, we need minimize the number of nodes and executors in nav2, which can improve performance.

This functionality has been discussed in the ticket #816, and carried out in

- Remove `client_node_` in `class WaypointFollower` : PR2441
- Remove `rclcpp_node_` in `class MapSaver` : PR2454
- Remove `bond_client_node_` in `class LifecycleManager` : PR2456
- Remove `node_` in `class LifecycleManagerClient` : PR2469

- Remove `rclcpp_node_` in `class ControllerServer` : PR2459, PR2479
- Remove `rclcpp_node_` in `class PlannerServer` : PR2459, PR2480
- Remove `rclcpp_node_` in `class AmclNode` : PR2483
- Remove `rclcpp_node_` and `clinet_node_` in `class Costmap2DROS` : PR2489
- Remove `rclcpp_node_` in `class LifecycleNode` : PR2993

some APIs are changed in these PRs:

- PR2489 removes arguments `client_node`, `rclcpp_node` and adds argument `callback_group` in the initialize function of class `nav2_costmap_2d::Layer`. `callback_group` is used to replace `rclcpp_node`.
- PR2993 removes argument `use_rclcpp_node `` in the constructor of class ``nav2_util::LifecycleNode`.

API Change for nav2_core

PR 2976 changes the API for `nav2_core::Controller` and `nav2_core::Smoother` by replacing the use of shared pointer references `(const shared_ptr<> &)` to shared pointers `(shared_ptr<>)`. Use of shared pointer references meant that the shared pointer counter was never incremented.

Extending the BtServiceNode to process Service-Results

This PR 2481 and PR 2992 address the ticket and this ticket and adds a virtual `on_completion()` function to the `BtServiceNode` class (can be found here). Similar to the already existing virtual `on_wait_for_result()` function, it can be overwritten in the child class to react to a respective event with some user-defined operation. The added `on_completion()` function will be called after the service interaction of the `BtServiceNode` has been successfully completed.

```
/**
* @brief Function to perform some user-defined operation upon successful
* completion of the service. Could put a value on the blackboard.
* @param response can be used to get the result of the service call in the BT Node.
* @return BT::NodeStatus Returns SUCCESS by default, user may override to return␣
↪another value
*/
virtual BT::NodeStatus on_completion(std::shared_ptr<typename ServiceT::Response>/
↪*response*/)
{
  return BT::NodeStatus::SUCCESS;
}
```

The returned `BT::NodeStatus` will set the current status of the BT-Node. Since the function has access to the results of the service, the returned node-status can depend on those service results, for example. The normal behavior of the `BtServiceNode` is not affected by introducing the `on_completion()` function, since the the default implementation still simply returns `BT::NodeStatus::SUCCESS`, if the service interaction completed successfully.

Including new Rotation Shim Controller Plugin

This PR 2718 introduces the new `nav2_rotation_shim_controller`. This controller will check the rough heading difference with respect to the robot and a newly received path. If within a threshold, it will pass the request onto the primary controller to execute. If it is outside of the threshold, this controller will rotate the robot towards that path heading. Once it is within the tolerance, it will then pass off control-execution from this rotation shim controller onto the primary controller plugin. At this point, the robot is still going to be rotating, allowing the current plugin to take control for a smooth hand off into path tracking.

The Rotation Shim Controller is suitable for:

- Robots that can rotate in place, such as differential and omnidirectional robots.
- Preference to rotate in place rather than 'spiral out' when starting to track a new path that is at a significantly different heading than the robot's current heading.
- Using planners that are non-kinematically feasible, such as NavFn, Theta*, or Smac 2D (Feasible planners such as Smac Hybrid-A* and State Lattice will start search from the robot's actual starting heading, requiring no rotation).

Spawning the robot in Gazebo

This PR 2473 deletes the pkg `nav2_gazebo_spawner` inside nav2_bringup directory. Instead of `nav2_gazebo_spawner` the Node spawn_entity.py of `gazebo_ros` is recomended to spawn the robot in gazebo. Note that

- gazebo should be started with both `libgazebo_ros_init.so` and `libgazebo_ros_factory.so` to work correctly.
- spawn_entity node could not remap /tf and /tf_static to tf and tf_static in the launch file yet, used only for multi-robot situations. This problem was overcame by adding remapping argument `<remapping>/tf:=tf</remapping>` `<remapping>/tf_static:=tf_static</remapping>` under ros2 tag in each plugin which publishs transforms in the SDF file. It is essential to differentiate the tf's of the different robot.

Recovery Behavior Timeout

Recoveries in Nav2, spin and backup, now have `time_allowance` ports in their BT nodes and request fields in their actions to specify a timeout. This helps ensure that the robot can exit a backup or spin primitive behavior in case it gets stuck or otherwise is unable to backup the full distance over a reasonable block of time.

New parameter `use_final_approach_orientation` for the 3 2D planners

Pull request 2488 adds a new parameter `use_final_approach_orientation` to the 3 2D planners (Theta*, SmacPlanner2D and NavFn), `false` by default. If `true`, the last pose of the path generated by the planner will have its orientation set to the approach orientation, i.e. the orientation of the vector connecting the last two points of the path. It allows sending the robot to a position (x,y) instead of a pose (x,y,theta) by effectively ignoring the goal orientation. For example, below, for the same goal with an orientaton pointed left of the screen, `use_final_approach_orientation=false` (left) and `use_final_approach_orientation=true` (right)

SmacPlanner2D and Theta*: fix goal orientation being ignored

This pull request 2488 fixes the issue of the goal pose orientation being ignored (the end path pose orientation was always set to 0).

SmacPlanner2D, NavFn and Theta*: fix small path corner cases

This PR 2488 ensures the planners are not failing when the distance between the start and the goal is small (i.e. when they are on the same costmap cell), and in that case the output path is constructed with a single pose.

Change and fix behavior of dynamic parameter change detection

This and this PR modify the method used to catch the changes of dynamic parameters. The motivation was to fix the issue that `void on_parameter_event_callback(const rcl_interfaces::msg::ParameterEvent::SharedPtr event)` was called for every parameter change of every node leading to unwanted parameter changes if 2 different nodes had the same parameter name.

Dynamic Parameters

Newly added dynamic parameters to:

- This PR 2592 makes most of the Costmap2DROS parameters dynamic
- This PR 2607 makes most of the Regulated Pure Pursuit parameters dynamic
- This PR 2665 makes most of the Theta * Planner parameters dynamic
- This PR 2704 makes Waypoint Follower, Planner Server, and Controller Server's params reconfigurable

BT Action Nodes Exception Changes

When BT action nodes throw exceptions due to networking or action server failures, they now return a status code of `FAILURE` to fail that particular action in the behavior tree to react to. This is in contrast to prior where the exceptions thrown from networking were sent to the root tree which would count as a task-level failure that the tree could not react to.

BT Navigator Groot Multiple Navigators

This PR 2627 creates separate parameters for groot monitoring for the NavToPose and NavThroughPoses navigator types so you can individually track the state of each behavior tree through the ZMQ publisher. This resolves a long-standing problem after we added multiple navigator types to BT Navigator that you could only view the nav to poses BT execution live. BT.CPP and Groot only support one static ZMQ stream at a time, so there is a bit of a quirk where you must locally reset Groot after switching trees in order to view the live stream of the Nav Through Poses BT, if in use. This is a state of the BT.CPP and Groot libraries and not something we can resolve withing Nav2.

There is some thought into the future regarding complete deprecation of live BT monitoring using Groot due to this quirk and the almost-certain infux of tickets on the topic. Groot will however always be supported for visualizing behavior tree XML files and modifications, simply not visualizing the BT execution live during robot navigation.

Removed Kinematic Limiting in RPP

The parameters `max_linear_accel` and `max_linear_decel` were removed along with the kinematic limiting in the controller causing instabilities. Instead, use a velocity smoother available in the ROS ecosystem if you would like similar behavior.

Added Smoother Task Server

A new task server was added which loads smoother plugins and executes them to improve quality of an existing planned path. Smoothing action can be called from a behavior tree using SmoothPath action node. PR 2569 implements and PR 2875 adds in the first of the plugins using it with a simple smoother. Other smoothers are in development and will be added in the future.

Removed Use Approach Velocity Scaling Param in RPP

The parameter `use_approach_linear_velocity_scaling` is removed in favor of always on to help in smooth transitions to the goal. This PR 2701 implements.

Refactored AMCL motion models as plugins

This PR 2642 creates plugins for the different motion models currently used in AMCL. This functionality enables users to use any custom motion model by creating it as a plugin and changing the robot_model_type parameter to the name of the plugin in nav2_params.yaml file. This helps to use custom motion models without the need to modify the AMCL source code.

Dropping Support for Live Groot Monitoring of Nav2

- https://github.com/ros-planning/navigation2/pull/2696

It was a great feature idea but never quite panned out, especially after we introduced multiple navigator types in the BT Navigator server. The issue we run into primarily is that Zero-MQ prevents users from producing multiple logger types in the same process. Since BT nav has multiple servers, the swapping between them for viewing has never had a clean hand off causing folks to file tickets or have nasty logs appear or ZMQ crashes in the background. The BT.CPP client for this doesn't allow us to have a clean shutdown process so we're left with hoping that ZMQ properly handles the situation, which it rarely does. Further, Groot only supports visualizing one type of tree at a time so for applications often switching between navigator types, its not possible to use a single groot client, causing great frustration.

So, what I propose here is to remove live monitoring of the BT from Nav2. **We can still use Groot to modify, visualize, and generally work with behavior trees**, the only thing being removed is to live view the executing behavior tree as Nav2 is currently executing it (it used to light up the boxes of the current nodes). This was of dubious value anyhow, since the tree ticks so fast its difficult to visualize and get meaningful insights into things as the system is moving so quickly.

Replanning Only if Path is Invalid

This PR 2591 creates two new condition BT node to facilitate replanning only if path becomes invalid rather than constantly replanning. These new nodes were integrated into the default BT.

Fix CostmapLayer clearArea invert param logic

This PR 2772 fixes the invert paramlogic of the CostmapLayer clearArea fonction. Hence correcting the behavior of the clearAroundRobot and clearExceptRegion services and their corresponding BT actions.

Dynamic Composition

This PR 2750 provides a optional bringup based on ROS2 dynamic composition for users. It can be used to compose all Nav2 nodes in a single process instead of launching these nodes separately, which is useful for embedded systems users that need to make optimizations due to harsh resource constraints. it's used by default, but can be disabled by using the launch argument `use_composition:=False`.

Some experiments to show performance improvement of dynamic composition, and the cpu and memory are captured by `psutil` :

CPU: Intel(R) i7-8700 (6Cores 12Threads), Memory: 32GB	cpu(%)	memory(%)
normal multiple processes	44	0.76
dynamic composition (use `component_container_isolated`)	38	0.23

The way of dynamic composition consumes lower memory(saves ~70%), and lower cpu (saves ~13%) than normal multiple processes.

BT Cancel Node

This PR 2787 caters the users with an abstract node to develop cancel behaviors for different servers present in the Nav2 stack such as the controller_server, recovery_server and so on. As a start, this PR also provides the `CancelControl` behavior to cancel the goal given to the controller_server. As an addition to the `CancelControl` This PR 2856 provides the users with the option to cancel the recoveries such as the `backup`, `spin` and `wait`.

BT PathLongerOnApproach Node

In the PR, a new Decorator BT node known as `PathLongerOnApproach` has been added to provide with the functionality to check and potentially handle longer path generated due to an obstacle in the given goal proximity. To demonstrate this functionality, a new BT `navigate_to_pose_w_replanning_goal_patience_and_recovery.xml` would serve both as an example and ready-to-use BT for a specific application that wishes to optimize their process cycle time. Demo of the developed BT can be seen below, where the robot pauses when close to a goal to see if the dynamic obstacle moves out of the way. Else, it executes the replan:

Obstacle does not clear at all, with *obstacle_clearance_time* to be 3 seconds:

Obstacle clears and you can see the robot pass through the (could have been ideally the) same path:

BT TruncatePathLocal Node

In the PR 2753, a new Action BT node named `TruncatePathLocal` has been added to extract a bounded-length path section near robot to be used e.g. for collision checking or computationally expensive smoothers

Constrained Smoother

In the PR 2753, a new Smoother named `nav2_constrained_smoother::ConstrainedSmoother` has been added to optimize various path criteria such as smoothness or distance from obstacles, maintaining minimum turning radius

Replanning at a Constant Rate and if the Path is Invalid

This PR 2804 introduces a new behavior tree that navigates to pose with consistent replanning and if the path becomes invalid. To facilitate the new behavior tree a new condition node PathExpiringTimer was introduced to trigger replanning at a consistent rate.

Euclidean Distance 2D

This PR 2865 changes Euclidean distance calculation throughout nav2 to project on to the XY plane (i.e. discard any information related to components in Z). This may potentially subtly change the way certain BT nodes, BT Navigators, controller servers, planner servers, and RPP behave if using custom plugins outside the Nav2 ecosystem.

Recovery To Behavior

This PR 2867 renames the nav2_recoveries to nav2_behaviors.

In navigation_launch.py recoveries_server -> behavior_server and nav2_recoveries -> nav2_behaviors. In nav2_params.yaml recovery_plugins -> behavior_plugins and nav2_recoveries -> nav2_behaviors.

Respawn Support in Launch and Lifecycle Manager

PR 2752 enables respawn support in Nav2. In the launch files, you may set `use_respawn` to `true` to enable respawning of servers that crash. This is only available in non-composed systems, since in composed systems, all of the nodes are under a single process and a crash anywhere will bring everything down (including the lifecycle manager itself). Even if the container was set to respawn, it would only respawn the empty container, not with all of the components loaded into it.

That PR also enables the lifecycle manager to check if a system goes down due to a crash. If so, it allows the manager to check if the server comes back online within a given timeout period. If it does, it will automatically retransition the system back up to active to continue on its task automatically.

New Nav2 Velocity Smoother

PR 2964 introduces the `nav2_velocity_smoother` for smoothing velocity commands from Nav2 to a robot controller by velocity, acceleration, and deadband constraints. See *Velocity Smoother* for more details. It is not included in the default bringup batteries included from `nav2_bringup`.

Goal Checker API Changed

PR 2965 adds an extra argument in the initialize function of the *nav2_core::GoalChecker* class. The extra argument is a costmap_ros pointer. This is used to check if the goal is in collision, so that we can avoid moving towards the goal and replanning can be initiates using some BT plugin.

Added Assisted Teleop

PR 2904 adds a new behavior for assisted teleop along with two new BT nodes AssistedTeleop and CancelAssistedTeleop.

4.11.5 Humble to Iron

Moving from ROS 2 Humble to Iron, a number of stability improvements were added that we will not specifically address here.

New Behavior-Tree Navigator Plugins

New in PR 3345, the navigator types are exposed to users as plugins that can be replaced or new navigator types added. The default behaviors of navigate to pose and navigate through poses continue to be default behavior but are now customizable with new action interface definitions. These plugins implement the `nav2_core::BehaviorTreeNavigator` base class, which must process the action request, feedback, and completion messages. The behavior tree is handled by this base class with as much general logic as possible abstracted away from users to minimize repetition.

See *Writing a New Navigator Plugin* for a tutorial about writing new navigator plugins.

Added Collision Monitor

PR 2982 adds new safety layer operating independently of Nav2 stack which ensures the robot to control the collisions with near obstacles, obtained from different sensors (LaserScan, PointCloud, IR, Sonars, etc…). See *Collision Monitor* for more details. It is not included in the default bringup batteries included from `nav2_bringup`.

Removed use_sim_time from yaml

PR #3131 makes it possible to set the use_sim_time parameter from the launch file for multiple nodes instead of individually via the yaml files. If using the Nav2 launch files, you can optionally remove the use_sim_time parameter from your yaml files and set it via a launch argument.

Run-time Speed up of Smac Planner

The core data structure of the graph implementation in the Smac Planner framework was swapped out in PR 3201 to using a specialized unordered map implementation. This speeds up the planner by 10% on trivial requests and reports up to 30% on complex plans that involve numerous rehashings.

Recursive Refinement of Smac and Simple Smoothers

The number of recursive refinements for the Simple and Smac Planner Smoothers have been exposed under the `refinement_num` parameter. Previous behavior had this hardcoded if `do_refinement = True` to `4`. Now, default is `2` to help decrease out-of-the-box over smoothing reducing in paths closer to collision than probably ideal, but old behavior can be achieved by changing this to `4`.

Simple Commander Python API

PR 3159 and follow-up PRs add in Costmap API in Python3 simple commander to receive `OccupancyGrid` messages from Nav2 and be able to work with them natively in Python3, analog to the C++ Costmap API. It also includes a line iterator and collision checking object to perform footprint or other collision checking in Python3. See the Simple Commander API for more details.

Smac Planner Start Pose Included in Path

PR 3168 adds the starting pose to the Smac Planners that was previously excluded during backtracing.

Parameterizable Collision Checking in RPP

PR 3204 adds makes collision checking for RPP optional (default on).

Expanded Planner Benchmark Tests

PR 3218 adds launch files and updated scripts for performing objective random planning tests across the planners in Nav2 for benchmarking and metric computation.

Smac Planner Path Tolerances

PR 3219 adds path tolerances to Hybrid-A* and State Lattice planners to return approximate paths if exact paths cannot be found, within a configurable tolerance aroun the goal pose.

costmap_2d_node default constructor

PR #3222 changes the constructor used by the standalone costmap node. The new constructor does not set a name and namespace internally so it can be set via the launch file.

Feedback for Navigation Failures

PR #3146 updates the global planners to throw exceptions on planning failures. These exceptions get reported back to the planner server which in turn places a error code on the Behavior Tree Navigator's blackboard for use in contextual error handling in the autonomy application.

The following errors codes are supported (with more to come as necessary): Unknown, TF Error, Start or Goal Outside of Map, Start or Goal Occupied, Timeout, or No Valid Path Found.

PR #3248 updates the compute path through poses action to report planning failures. These exceptions get reported back to the planner server which in turn places a error code on the Behavior Tree Navigator's blackboard for use in contextual error handling in the autonomy application.

The following errors codes are supported (with more to come as necessary): Unknown, TF Error, Start or Goal Outside of Map, Start or Goal Occupied, Timeout, No Valid Path Found and No Waypoints given.

PR #3227 updates the controllers to throw exceptions on failures. These exceptions get reported back to the controller server which in turn places a error code on the Behavior Tree Navigatior's blackboard for use in contextual error handling in the autonomy application.

The following error codes are supported (with more to come as necessary): Unknown, TF Error, Invalid Path, Patience Exceeded, Failed To Make Progress, or No Valid Control.

PR #3251 pipes the highest priority error code through the bt_navigator and defines the error code structure.

A new parameter for the the BT Navigator called "error_code_id_names" was added to the nav2_params.yaml to define the error codes to compare. The lowest error in the "error_code_id_names" is then returned in the action request (navigate to pose, navigate through poses waypoint follower), whereas the code enums increase the higher up in the software stack - giving higher priority to lower-level failures.

The error codes produced from the servers follow the guidelines stated below. Error codes from 0 to 9999 are reserved for nav2 while error codes from 10000-65535 are reserved for external servers. Each server has two "reserved" error codes. 0 is reserved for NONE and the first error code in the sequence is reserved for UNKNOWN.

The current implemented servers with error codes are:

- Controller Server: NONE:0, UNKNOWN:100, server error codes: 101-199
- Planner Server(compute_path_to_pose): NONE:0, UNKNOWN:201, server error codes: 201-299
- Planner Server(compute_path_through_poses): NONE:0, UNKNOWN:301, server error codes: 301-399
- Smoother Server: NONE: 0, UNKNOWN: 501, server error codes: 501-599
- Waypoint Follower Server: NONE: 0, UNKNOWN: 601, server error codes: 601-699

This pr also updates the waypoint follower server to throw exceptions on failures. These exceptions get reported back to the server which in turn places a error code on the Behavior Tree Navigator's blackboard for use in contextual error handling in the autonomy application. The following errors codes are supported (with more to come as necessary): Unknown and Task Executor Failed. See *Adding a New Nav2 Task Server* and the PR for additional information.

Costmap Filters

Costmap Filters now are have an ability to be enabled/disabled in run-time by calling `toggle_filter` service for appropriate filter (PR #3229).

Added new binary flip filter, allowing e.g. to turn off camera in sensitive areas, turn on headlights/leds/other safety things or switch operating mode when robot is inside marked on mask areas (PR #3228).

Savitzky-Golay Smoother

Adding a new smoother algorithm, the Savitzky-Golay smoother to the smoother server plugin list. See the configuration guide *Savitzky-Golay Smoother* for more details.

Changes to Map yaml file path for map_server node in Launch

PR #3174 adds a way to set the path to map yaml file for the map_server node either from the yaml file or using the launch configuration parameter `map` giving priority to the launch configuration parameter. `yaml_filename` is no longer strictly required to be present in `nav2_params.yaml`.

SmootherSelector BT Node

PR #3283 adds a BT node to set the smoother based on a topic or a default. See the configuration guide *Simple Smoother* for more details.

Publish Costmap Layers

PR #3320 adds the ability for the nav2_costmap_2d package to publish out costmap data associated with each layer.

Give Behavior Server Access to Both Costmaps

PR #3255 addes the ability for a behavior to access the local and global costmap.

To update behaviors, any reference to the global_frame must be updated to the local_frame parameter along with the `configuration` method which now takes in the local and global collision checkers. Lastly, `getResourceInfo` must be overriden to return `CostmapInfoType::LOCAL`. Other options include `GLOBAL` if the behavior useses global costmap and/or footprint) or `BOTH` if both are required. This allows us to only create and maintain the minimum amount of expensive resources.

New Model Predictive Path Integral Controller

The new Nav2 MPPI Controller is a predictive controller - a successor to TEB and pure path tracking MPC controllers - with Nav2. It uses a sampling based approach to select optimal trajectories, optimizing between successive iterations. It contains plugin-based objective functions for customization and extension for various behaviors and behavioral attributes.

See the README.md and *Model Predictive Path Integral Controller* page for more detail.

Behavior Tree Uses Error Codes

PR #3324 adds three new condition nodes to check for error codes on the blackboard set by action BT nodes which contain them.

The `AreErrorCodesPresent` condition node allows the user to specify the error code from the server along with the error codes to match against. The `WouldAControllerRecoveryHelp` checks if the active error code is UNKNOWN, PATIENCE_EXCEEDED, FAILED_TO_MAKE_PROGRESS or NO_VALID_CONTROL. If the error code is a match, the condition returns `SUCCESS`. These error code are potentially able to be cleared by a controller recovery.

The `WouldAPlannerRecoveryHelp` checks if the active error code is UNKNOWN, NO_VALID_CONTROL, or TIMEOUT. If the error code is a match, the condition returns `SUCCESS`. These error code are potentially able to be cleared by a planner recovery.

The `WouldASmootherRecoveryHelp` checks if the active error code is UNKNOWN, TIMEOUT, FAILED_TO_SMOOTH_PATH, or SMOOTHED_PATH_IN_COLLISION. If the error code is a match, the condition returns `SUCCESS`. These error code are potentially able to be cleared by a smoother recovery.

Load, Save and Loop Waypoints from the Nav2 Panel in RViz

PR #3165 provides three new functionalities for the nav2 panel in RViz, they are:

- load and save waypoints in a yaml file for waypoint following (initial pose can also be stored if required)
- loop functionality to revisit the waypoints
- pause and resume button for stopping and continuing through the waypoints

Looping functionality is not specific to the nav2 panel in RViz. Users utilizing nav2_waypoint_follower can take advantage of the changes made to the FollowWaypoint action, by specifying the desired number of loops in the action request that will be eventually sent to the nav2_waypoint_follower server.

DWB Forward vs Reverse Pruning

PR #3374 adds a new `forward_prune_distance` parameter in the DWB controller. It replaces the `prune_distance` for forward path shortening, enabled through the `shorten_transformed_plan` boolean parameter. This change allows to use different values for forward and backward path shortening.

More stable regulation on curves for long lookahead distances

PR #3414 adds a new `use_fixed_curvature_lookahead` parameter to the RPP controller. This makes slowing down on curve not dependent on the instantaneous lookahead point, but instead on a fixed distance set by the parameter `curvature_lookahead_dist`.

Publish Collision Monitor State

PR #3504 adds a new `state_topic` parameter to the CollisionMonitor. If specified, this optional parameter enables the state topic publisher. The topic reports the currently activated polygon action type and name.

Renamed ROS-parameter in Collision Monitor

PR #3513 renames `max_points` parameter to `min_points` and changes its meaning. Formerly `max_points` meant the maximum number of points inside the area still not triggering the action, while `min_points` - is a minimal number of points starting from the action to be initiated. In other words `min_points` now should be adjusted as `max_points + 1`.

New safety behavior model "limit" in Collision Monitor

PR #3519 adds a new collision monitor behavior model `limit` that restricts maximum linear and angular speed to specific values (`linear_limit` and `angular_limit`) if enough points are in the given shape.

Velocity smoother applies deceleration when timeout

PR #3512 makes the VelocitySmoother apply the deceleration when the input command timeout.

PoseProgressChecker plugin

PR #3530 adds a new `nav2_controller::PoseProgressChecker` plugin. It builds on the behavior of the `SimpleProgressChecker` by adding a new parameter `required_movement_angle`, allowing the plugin to considers that there is still progress when there is no translation movement, from the moment there is a rotation movement superior to `required_movement_angle` within the `movement_time_allowance`.

Allow multiple goal checkers and change parameter progress_checker_plugin(s) name and type

PR #3555 initializes the progress checker plugin(s) in the same way as for the goal checker and controller plugins: it is now a list of string and was renamed from `progress_checker_plugin` to `progress_checker_plugins`, and the type changed from `string` to `vector<string>`. This allows the initialization of multiple progress checkers that can be chosen from the added `progress_checker_id field` of the `FollowPath` action. Beware that it is a breaking change and that configuration files will need to be updated.

IsBatteryChargingCondition BT Node

PR #3553 adds a BT node to check if the battery is charging. See the configuration guide *IsBatteryCharging* for more details.

Behavior Server Error Codes

PR #3569 updates the behavior server plugins to provide error codes on failure.

- Spin: NONE: 0, UNKNOWN: 701, server error codes: 701-709
- BackUp: NONE: 0, UNKNOWN: 801, server error codes: 710-719
- DriveOnHeading: NONE: 0, UNKNOWN: 901, server error codes: 720-729
- AssistedTeleop: NONE: 0, UNKNOWN: 1001, server error codes: 730-739

New Denoise Costmap Layer Plugin

PR #2567 adds the new plugin for filtering noise on the costmap.

Due to errors in `Voxel Layer` or `Obstacle Layer` measurements, salt and pepper noise may appear on the *costmap*. This noise creates false obstacles that prevent the robot from finding the best path on the map. The new `Denoise Layer` plugin is designed to filter out noise-induced standalone obstacles or small obstacles groups. This plugin allows you to add layer that will filter local or global costmap. More information about `Denoise Layer` plugin and how it works could be found *here*.

SmacPlannerHybrid viz_expansions parameter

PR #3577 adds a new paremeter for visualising SmacPlannerHybrid expansions for debug purpose.

4.11.6 Iron to Jazzy

Moving from ROS 2 Iron to Jazzy, a number of stability improvements were added that we will not specifically address here.

Introduce a new Multi-Robot Bringup Launch

PR #3572 introduces a new way of bringup tb3 multi-robot that names as `cloned_tb3_simulation_launch.py` for simulation. `cloned_tb3_simulation_launch.py` enables to bring up multiple robots with same parameter that described in `nav2_multirobot_param_all.yaml`. And multiple robots are separeted by namespaces which are given as a Launch Arguments. Existing `multi_tb3_simulation_launch.py` which was utilized in previous is replaced with `unique_tb3_simulation_launch.py`, allowing for multiple unique robot instances utilizing `nav2_multirobot_params_<N>.yaml` configuration files.

New option for the Voxel and Obstacle Layers

PR #3612 adds a new MaxWithoutUnknownOverwrite option to combination_method parameter in Voxel and Obstacle Layers. This can be used to make sure that the static map is the dominant source of information, and easily prevent the robot to go through places that are not present in the static map.

Changes to MPPI Goal Critic

The MPPI Goal critic's formulation is changed to better keep up with speed on approach to goal instead of preemptively slowing too significantly. It also allows you to better use the weight to adjust the degree at which it slows more naturally. This change involves adjusting the `threshold_to_consider` to be the same as your prediction horizon (e.g. samples * dt * max speed) for both the goal critic and path follower critic to have a good hand-off between them without deceleration.

Changes to MPPI Path Angle Critic

MPPI's Path Angle critic now has a `mode` setting to adjust behavior depending on robot's desired behavioral traits. Previously, it penalized path orientations that deviated far the the robot's forward orientation to turn the robot towards sharp changes in the path. This is still default (`mode: 0`), but other modes now exist too.

`mode: 1` sets the penalization of path's relative directions by either forward orientation or the opposite for reversing to allow for true bidirectional motion when one way or another is not preferable for a symmetric robot. This uses only the path's relative points to the robot to decide which direction to incentivize.

`mode: 2` instead uses the path's orientations when a feasible path is given from the Smac Planners or the Smoother server's algorithms. This way, the globally planned orientations are followed rather than the based solely on the path's relative points. This is useful for non-circular robots in highly confined settings where there may be restricted opportunities to change directions so following the global path's orientation are required to end in the orientation you require.

Changes to MPPI Path Handling For Directionality

MPPI's Path Align Critic and Path Handler object now have options to utilize the path's orientation information to force the controller to change directions when and only when requested by a feasible planner. When `enforce_path_inversion` is `true`, the path handler will prune the path to the first time the directions change to force the controller to plan to the inversion point and then be set the rest of the path, once in tolerance. The Path Align critic also contains a parameter `use_path_orientations` which can be paired with it to incentivize aligning the path containing orientation information to better attempt to achieve path inversions where requested **and** not do them when not requested.

See MPPI's configuration guide for complete information.

Move Error Code Enumerations

PR #3693 moves the enumeration codes from the goal to the result section.

Substitution in parameter file

Enabled substitution in parameter file. For example, you can write the following

```yaml
bt_navigator:
  ros__parameters:
    default_nav_to_pose_bt_xml: $(find-pkg-share my_package)/behavior_tree/my_nav_to_
↪pose_bt.xml
```

For more information about substitutions syntax, see here

Allow Behavior Server Plugins to Access The Action Result

PR #3704 allows behavior servers plugins to access and modify the action result.

Smac Planner Debug Param Name Change

`debug_visualizations` replaces `viz_expansions` parameter in Hybrid-A* to reflect the new inclusion of footprint debug information being published as well.

4.12 Simple Commander API

4.12.1 Overview

The goal of the Nav2 Simple (Python3) Commander is to provide a "navigation as a library" capability to Python3 users. We provide an API that handles all the ROS 2 and Action Server tasks for you such that you can focus on building an application leveraging the capabilities of Nav2 (after you've configured it to your liking with your plugins of choice). We also provide you with demos and examples of API usage to build common basic capabilities in autonomous mobile robotics in the `nav2_simple_commander` package.

A simple demonstration is shown below. Note: `goToPose()`, `goThroughPoses()`, `followWaypoints()` and similar are **non-blocking** such that you can receive and process feedback in a single-threaded application. As such while waiting for a task to be completed, the `while not nav.isTaskComplete()` design is necessary to poll for changes in the navigation completion, and if not complete some tasks of interest to your application (like processing feedback, doing something with the data the robot is collecting, or checking for faults).

You may use this simple commander preempt commands of the same type (e.g. you can preempt a `goToPose()` with another `goToPose()`) but you must explicitly cancel a current command and issue a new one if switching between `goToPose()`, `goThroughPoses()`, or `followWaypoints()`.

```python
from nav2_simple_commander.robot_navigator import BasicNavigator
import rclpy

rclpy.init()
nav = BasicNavigator()

# ...

nav.setInitialPose(init_pose)
nav.waitUntilNav2Active() # if autostarted, else use lifecycleStartup()

# ...

path = nav.getPath(init_pose, goal_pose)
smoothed_path = nav.smoothPath(path)

# ...

nav.goToPose(goal_pose)
while not nav.isTaskComplete():
  feedback = nav.getFeedback()
  if feedback.navigation_duration > 600:
    nav.cancelTask()

# ...
```

(continues on next page)

(continued from previous page)

```
result = nav.getResult()
if result == TaskResult.SUCCEEDED:
    print('Goal succeeded!')
elif result == TaskResult.CANCELED:
    print('Goal was canceled!')
elif result == TaskResult.FAILED:
    print('Goal failed!')
```

4.12.2 Commander API

The methods provided by the basic navigator are shown below, with inputs and expected returns. If a server fails, it may throw an exception or return a *None* object, so please be sure to properly wrap your navigation calls in try/catch and check returns for *None* type.

Robot Navigator Method	Description
setInitialPose(initial_pose)	Sets the initial pose (`PoseStamped`) of the robot to localization.
goThroughPoses(poses, behavior_tree=”)	Requests the robot to drive through a set of poses (list of `PoseStamped`).
goToPose(pose, behavior_tree=”)	Requests the robot to drive to a pose (`PoseStamped`).
followWaypoints(poses)	Requests the robot to follow a set of waypoints (list of `PoseStamped`). This will execute the chosen `TaskExecutor` plugin at each pose.
followPath(path, controller_id=”, goal_checker_id=”)	Requests the robot to follow a path from a starting to a goal `PoseStamped`, `nav_msgs/Path`.
spin(spin_dist=1.57, time_allowance=10)	Requests the robot to performs an in-place rotation by a given angle.
backup(backup_dist=0.15, backup_speed=0.025, time_allowance=10)	Requests the robot to back up by a given distance.
assistedTeleop(time_allowance=30)	Requests the robot to run the assisted teleop action.
cancelTask()	Cancel an ongoing task, including route tasks.
isTaskComplete(trackingRoute=False)	Checks if task is complete yet, times out at `100ms`. Returns `True` if completed and `False` if still going. If checking a route tracking task, set default argument to `True`.
getFeedback(trackingRoute=False)	Gets feedback from task, returns action server feedback msg. If getting feedback on a tracking task, set default argument to `True`.
getResult()	Gets final result of task, to be called after `isTaskComplete` returns `True`. Returns action server result msg.
getPath(start, goal, planner_id=”, use_start=False)	Gets a path from a starting to a goal `PoseStamped`, `nav_msgs/Path`.
getPathThroughPoses(start, goals, planner_id=”, use_start=False)	Gets a path through a starting to a set of goals, a list of `PoseStamped`, `nav_msgs/Path`.
getRoute(start, goal, use_start=False)	Gets a sparse route and dense path from start to goal, where start and goal may be of type `PoseStamped` or `int` for known NodeIDs.
getandTrackRoute(start, goal, use_start=False)	Gets and tracks a sparse route and dense path from start to goal, where start & goal may be of type `PoseStamped` or `int` for known NodeIDs.
smoothPath(path, smoother_id=”, max_duration=2.0, check_for_collision=False)	Smooths a given path of type `nav_msgs/Path`.
changeMap(map_filepath)	Requests a change from the current map to *map_filepath*’s yaml.
clearAllCostmaps()	Clears both the global and local costmaps.
clearLocalCostmap()	Clears the local costmap.
clearGlobalCostmap()	Clears the global costmap.
getGlobalCostmap()	Returns the global costmap, `nav2_msgs/Costmap`.
getLocalCostmap()	Returns the local costmap, `nav2_msgs/Costmap`.
waitUntilNav2Active(navigator=’bt_navigator’, localizer=’amcl’)	Blocks until Nav2 is completely online and lifecycle nodes are in the active state. To be used in conjunction with autostart or external lifecycle bringup. Custom navigator and localizer nodes can be specified
lifecycleStartup()	Sends a request to all lifecycle management servers to bring them into the active state, to be used if autostart is `False` and you want this program to control Nav2’s lifecycle.
lifecycleShutdown()	Sends a request to all lifecycle management servers to shut them down.
destroyNode()	Releases the resources used by the object.

4.12.3 Costmap API

This is a Python3 API for costmap 2d messages from the stack. It provides the basic conversion, get/set, and handling semantics found in the costmap 2d C++ API.

Costmap Method	Description
getSizeInCellsX()	Get map width in cells.
getSizeInCellsY()	Get map height in cells.
getSizeInMetersX()	Get x axis map size in meters.
getSizeInMetersY()	Get y axis map size in meters.
getOriginX()	Get the origin x axis of the map [m].
getOriginY()	Get the origin y axis of the map [m].
getResolution()	Get map resolution [m/cell].
getGlobalFrameID()	Get global frame_id.
getCostmapTimestamp()	Get costmap timestamp.
getCostXY(mx, my)	Get the cost (`np.uint8`) of a cell in the costmap using mx (`int`) , my (`int`) of Map Coordinate.
getCostIdx(index)	Get the cost (`np.uint8`) of a cell in the costmap using index (`int`)
setCost(mx, my, cost)	Set the cost (`np.uint8`) of a cell in the costmap using mx (`int`) , my (`int`) of Map Coordinate.
mapToWorld(mx, my)	Get the wx (`float`) [m], wy (`float`) [m] of world coordinate XY using mx (`int`), my (`int`) of map coordinate XY
worldToMapValidated(wx, wy)	Get the mx (`int`), my (`int`) of map coordinate XY using wx (`float`) [m], wy (`float`) [m] of world coordinate XY. If wx wy coordinates are invalid, (None,None) is returned.
getIndex(mx, my)	Get the index (`int`) of the cell using mx (`int`), my (`int`) of map coordinate XY

4.12.4 Footprint Collision Checker API

This is a Python3 API for a Footprint Collision Checker. It provides the needed methods to manipulate the coordinates and calculate the cost of a Footprint in a given map.

Footprint Collision Checker Method	Description
footprintCost(footprint)	Checks the footprint (`Polygon`) for collision at its implicit provided coordinate pose.
lineCost(x0, x1, y0, y1, step_size=0.5)	Iterate over all the points along a line and check for collision. The line is defined by x0, y0, x1, y1, step_size (`int`) or (`float`).
worldToMapValidated(wx, wy)	Get the mx (`int`), my (`int`) of map coordinate XY using wx (`float`) [m], wy (`float`) [m] of world coordinate XY. If wx wy coordinates are invalid, (None,None) is returned. Returns None if costmap is not defined yet through (`setCostmap(costmap)`).
pointCost(x, y)	Get the cost of a point in the costmap using map coordinates XY. (`int`)
setCostmap(costmap)	Specify which costmap to use with the footprint collision checker. (`PyCostmap2D`)
footprintCostAtPose(x, y, theta, footprint)	Get the cost of a footprint at a specific Pose in map coordinates. x, y, theta (`float`) footprint (`Polygon`).

4.12.5 Examples and Demos

All of these can be found in the package.

The `nav2_simple_commander` has a few examples to highlight the API functions available to you as a user:

- `example_nav_to_pose.py` - Demonstrates the navigate to pose capabilities of the navigator, as well as a number of auxiliary methods.
- `example_nav_through_poses.py` - Demonstrates the navigate through poses capabilities of the navigator, as well as a number of auxiliary methods.
- `example_waypoint_follower.py` - Demonstrates the waypoint following capabilities of the navigator, as well as a number of auxiliary methods.
- `example_follow_path.py` - Demonstrates the path following capabilities of the navigator, as well as a number of auxiliary methods like path smoothing.
- `example_assisted_teleop.py` - Demonstrates the assisted teleop capabilities of the navigator.
- `example_route.py` - Demonstrates the Route server capabilities of the navigator.

The `nav2_simple_commander` has a few demonstrations to highlight a couple of simple autonomy applications you can build using the API:

- `demo_security.py` - A simple security robot application, showing how to have a robot follow a security route using Navigate Through Poses to do a patrol route, indefinitely.
- `demo_picking.py` - A simple item picking application, showing how to have a robot drive to a specific shelf in a warehouse to either pick an item or have a person place an item into a basket and deliver it to a destination for shipping using Navigate To Pose.
- `demo_inspection.py` - A simple shelf inspection application, showing how to use the Waypoint Follower and task executors to take pictures, RFID scans, etc of shelves to analyze the current shelf statuses and locate items in the warehouse.

4.13 Roadmaps

This is the list of major issues and features the Nav2 maintainers are committing for completion for various ROS 2 releases. This is *not* an exhaustive list of planned features or what changes may be found in the new distribution. It represents only the items of direct commitment to give insight into commitments for REP-2005 repositories in the ROS 2 Roadmap. For a full list of important completed changes in the project, see the Migration Guides *Migration Guides*.

4.13.1 Jazzy Roadmap

Smac Planner Improvements	Large (in progress)
Get CI Green Again	Small (DONE)
Various MPPI Improvements	Medium (DONE)
Fuse Migration	Medium
Ignition Migration	Medium (in progress)
Route Graph Planner	Large (in progress)
Provide Advanced Capabilities Tutorials	Medium
ROS Time Respect Across Stack	Medium (in progress)

4.13.2 Iron Roadmap

Plugin Name	Size
Pluginize Navigators	Medium (DONE)
MPPI Controller	Very Large (DONE)
Route Graph Planner	Large (incomplete)
90% unit test coverage	Small (DONE)
Velocity Smoother	Medium (DONE)
ROS Time Respect Across Stack	Medium (incomplete)

4.13.3 Humble Roadmap

Plugin Name	Size
Nav2 1 Node Per Server	Medium (DONE)
Smac Lattice Planner	Large (DONE)
Safety Collision Nodes	Medium (DONE)
Fix Min Range Bug	Small (DONE)
Move Development from Master to Rolling	Small (DONE)
Push Test Coverage to 88%	Medium (DONE)
Complete First Time Guide	Medium (DONE)
Rotation Shim Controller	Small (DONE)
Dynamic Composition	Medium (DONE)

4.14 About and Contact

4.14.1 Related Projects

This is a community maintained list of related repositories and projects to Navigation2.

Name	Maintainer	Description
Navigation2	Steve Macenski	Main ROS 2 Navigation Stack
navigation.ros.org	Steve Macenski	Official documentation website source code
navigation2_tutorials	Steve Macenski	Source code and examples used in the documentation tutorials
navigation2_dynamic	Steve Macenski	Tools, algorithms, and capabilities for working with dynamic obstacles in ROS 2
robot_localization	Tom Moore	State estimation software for ROS 2 used for odometry fusion and GPS transformations
slam_toolbox	Steve Macenski	Default 2D SLAM library

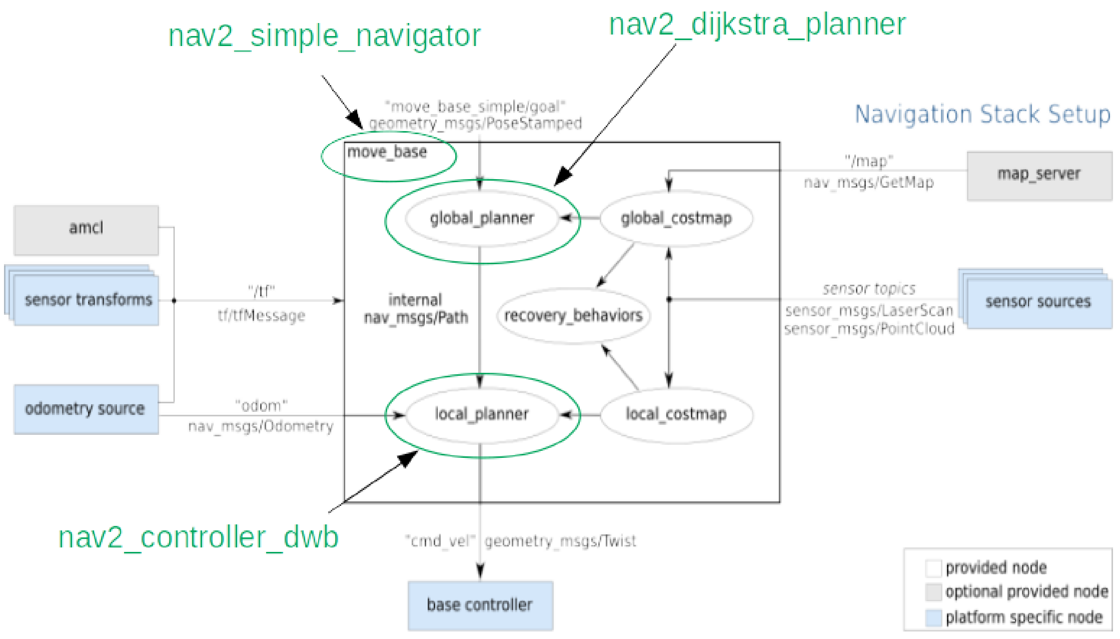
4.14.2 ROS to ROS 2 Navigation

`move_base` has been split into multiple components. Rather than a single monolithic state machine, navigation 2 makes use of action servers and ROS 2's low-latency, reliable communication to separate ideas. A behavior tree is used to orchestrate these tasks. This allows Navigation2 to have highly configurable navigation behavior without programming by rearranging tasks in a behavior tree xml file.

The *nav2_bt_navigator* replaces `move_base` at the top level, with an *Action* interface to complete a navigation task with a tree-based action model. It uses *Behavior Trees* to make it possible to have more complex state machines and to add in behaviors as additional *Action Servers*. These behavior trees are configurable XMLs and we provide several starting examples.

The planning, behaviors, smoother, and controller servers are also action servers that the BT navigator can call to compute. All 4 servers can host many plugins of many algorithms each and individually called from the navigation behavior tree for specific behaviors. These servers are called from the BT navigator through their action servers to compute a result or complete a task. The state is maintained by the BT navigator behavior tree.

All these changes make it possible to replace any of these nodes at launch/run time with any other algorithm that implements that same interface. See each package README.md for more details.



Note: `nav2_simple_navigator` no longer exists, it has been replaced by `nav2_bt_navigator`.

In Summary:

Ported packages:

- amcl: Ported to nav2_amcl
- map_server: Ported to nav2_map_server
- nav2_planner: Replaces global_planner, hosts `N` planner plugins
- nav2_controller: Replaces local_planner, hosts `N` controller plugins
- Navfn: Ported to nav2_navfn_planner
- DWB: Replaces DWA and ported to ROS 2 under nav2_dwb_controller metapackage
- nav_core: Ported as nav2_core with updates to interfaces
- costmap_2d: Ported as nav2_costmap_2d

New packages:

- nav2_bt_navigator: replaces `move_base` state machine
- nav2_lifecycle_manager: Handles the server program lifecycles
- nav2_waypoint_follower: Can take in many waypoints to execute a complex task through

- nav2_system_tests: A set of integration tests for CI and basic tutorials in simulation
- nav2_rviz_plugins: An rviz plugin to control the Navigation2 servers, command, cancel, and navigation with
- navigation2_behavior_trees: wrappers for the behavior tree library to call ROS action servers

among many others.

4.14.3 About

Navigation is a community effort to lower the barrier of entry of mobile robotics technology for all. This project is one of the flagship projects along with MoveIt that acts as an applications entry point and frameworks for ROS. Navigation in ROS 2 builds on the wild success of the original Navigation Stack (Nav Stack) in ROS (1). This project has been used by researchers, educators, and industry for over 10 years. There are very few projects that have lasted as long or were as successful as ROS (1) Navigation. We would like to thank David Lu!! and Mike Ferguson for their tireless over the span of a decade to keep the Nav Stack alive and well. Without their hard work, this project would not have been able to happen. If you're interested in a comparison between ROS 1 Navigation and ROS 2 Navigation, see *ROS to ROS 2 Navigation*. For a list of robots using Navigation2, see *Robots Using*.

Aslas, as time moves on, so must we. ROS (1) had a great run and those of us that build projects using it will remember it fondly(ish). With change comes possibilities. Navigation2 builds on the success but redesigns the framework to be more flexible and the feedback gathered over 10 years.

We strive to create an open community and encourage new ROS users and experts alike to collaborate. However, that can't happen without your issues, pull requests, and support. Navigation, like all open-source projects, is kept going by a dedicated group of developers, maintainers, users, and collaborators. We would like to thank here our current and past contributors and maintainers.

If you or your organization are interested in sponsoring Navigation or any work around it, please reach out to the current project lead.

Our current leadership team includes:

Name	Organization	GitHub ID	Current Role
Steve Macenski	Samsung Research	SteveMacenski	Project Lead
Ruffin White	UC San Diego	ruffsl	CI Wizard
Joshua Wallace	Locus Robotics	JoshuaWallace	Main Contributor
Alexey Merzlyakov	Samsung Research	AlexeyMerzlyakov	Maintainer

Our former leadership team includes:

Name	Organization	GitHub ID	Role
Matt Hansen	Intel Corporation	mkhansen	Former Project Lead
Brian Wilcox	Intel Corporation	bpwilcox	Former Maintainer
Carl Delsey	Intel Corporation	crdelsey	Maintainer

4.14.4 Contact

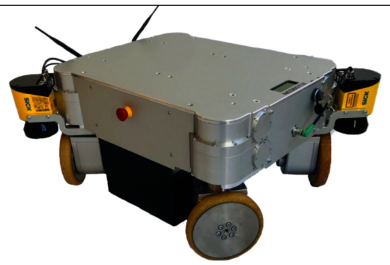
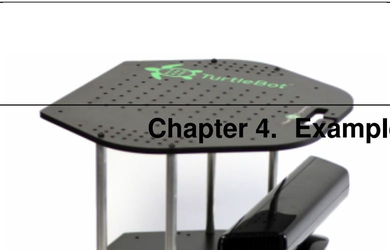
If you are interested in contacting someone about Navigation, ROS 2, or related projects, please email the project leader. We intentionally make our emails easy to find. If your inquiry relates to bugs or open-source feature requests, consider posting a ticket on our GitHub project. If your inquiry relates to configuration support or private feature development, reach out and we may be able to connect you with independent consultants or contractors that know this project well.

Check out the Navigation 2 Slack community.

4.15 Robots Using

It's always helpful (and fun!) to have a list of robots using or ship with our work. Below is a very early list of robots we have encountered using our software as examples.

Click on the images below for a link to the drivers or navigation configurations.

		Seasony
angsa		
TAILOS		birds eye
PIXEL	brisa	TOYOTA RESEARCH INSTITUTE
		SICK
Unbounded Robotics		 TurtleBot