Dokumentasi Tugas Besar Sekolah URO

Dietrich Pepalem Tarigan

10 Januari 2025

1 Mengatur Transformasi

Panduan ini membahas transformasi yang diperlukan oleh Nav2. Transformasi ini memungkinkan Nav2 untuk memahami informasi yang masuk dari berbagai sumber, seperti sensor dan odometri, dengan mengubahnya ke kerangka koordinat yang dapat digunakan. Berikut adalah gambaran pohon transformasi penuh untuk sebuah robot, namun kita akan memulai dengan sesuatu yang lebih sederhana.

Dalam tutorial ini, kita akan:

1. Memberikan pengantar singkat tentang transformasi di ROS.
2. Melakukan demo sederhana dengan menggunakan **TF2 static publisher** melalui command-line.
3. Menjelaskan transformasi yang diperlukan agar Nav2 dapat berfungsi.

2 Pengenalan Transformasi

> **Catatan:** Bagian ini diadaptasi dari tutorial *Setting Up Your Robot using tf* pada dokumentasi navigasi ROS 1.

Banyak paket ROS membutuhkan pohon transformasi robot yang diterbitkan menggunakan paket **TF2 ROS**. Pohon transformasi mendefinisikan hubungan antar sistem koordinat, baik dalam hal translasi, rotasi, maupun gerakan relatif.

Sebagai contoh, bayangkan robot sederhana dengan basis bergerak yang memiliki sensor laser di atasnya. Robot ini memiliki dua kerangka koordinat:

- **base_link**: Titik pusat basis bergerak robot.
- **base_laser**: Titik pusat sensor laser di atas basis.

Misalnya, kita memiliki data dari sensor laser dalam bentuk pengukuran jarak dari titik pusat laser (kerangka koordinat **base_laser**). Untuk membantu robot menghindari rintangan, data tersebut perlu dikonversi ke kerangka koordinat **base_link**. Dengan kata lain, kita perlu mendefinisikan hubungan antara **base_laser** dan **base_link**.

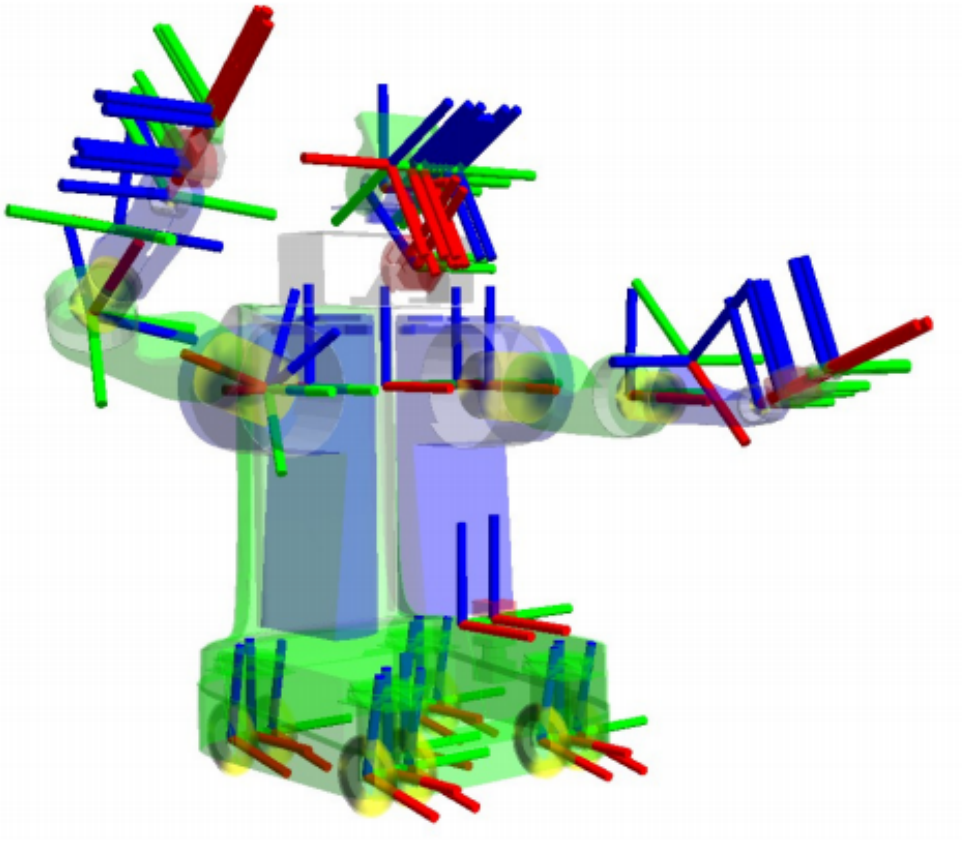

Figure 1: Pohon transformasi robot.

2.1 Mendefinisikan Hubungan Transformasi

Misalkan sensor laser dipasang 10 cm ke depan dan 20 cm di atas pusat basis bergerak. Hubungan translasi ini dapat didefinisikan sebagai:

- Dari **base_link** ke **base_laser**: (x: 0.1m, y: 0.0m, z: 0.2m)
- Dari **base_laser** ke **base_link**: (x: -0.1m, y: 0.0m, z: -0.2m)

Alih-alih mengelola hubungan ini secara manual, kita dapat menggunakan **TF2** untuk mendefinisikan hubungan tersebut sekali saja. TF2 sangat berguna untuk transformasi yang tidak statis, seperti kerangka koordinat yang bergerak relatif satu sama lain.

Untuk menyimpan hubungan ini dalam TF2, kita membuat pohon transformasi. Setiap simpul dalam pohon ini adalah kerangka koordinat, dan setiap tepi adalah transformasi yang menghubungkan simpul-simpul tersebut.

Dalam contoh ini, kita memilih **base_link** sebagai simpul induk dan **base_laser** sebagai simpul anak. Transformasi yang menghubungkan keduanya adalah (x: 0.1m, y: 0.0m, z: 0.2m).

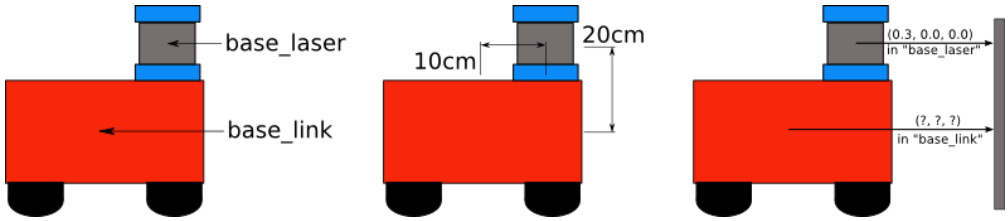

Figure 2: Hubungan transformasi antara base_link dan base_laser.

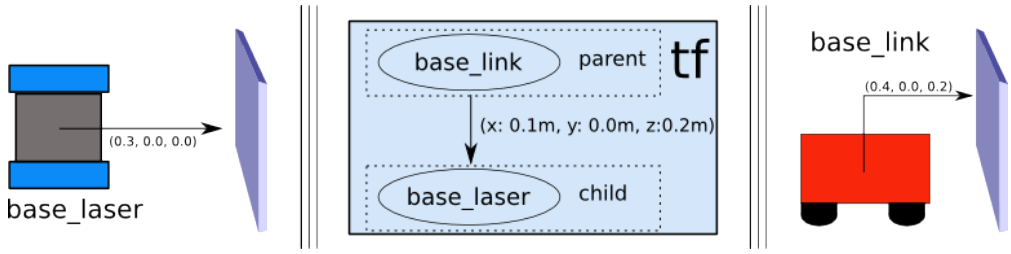

Figure 3: Contoh pohon transformasi dengan TF2.

3 Demo Static Transform Publisher

> **Peringatan:** Jika Anda baru menggunakan ROS 2 atau belum memiliki lingkungan kerja, luangkan waktu untuk mengatur mesin Anda dengan benar menggunakan sumber daya di dokumentasi resmi Instalasi ROS 2.

Sekarang, mari kita coba menerbitkan transformasi sederhana menggunakan alat **static_transform_pu** yang disediakan oleh TF2. Kita akan menerbitkan transformasi dari **base_link** ke **base_laser** dengan translasi (x: 0.1m, y: 0.0m, z: 0.2m). Transformasi ini diambil dari diagram sebelumnya dalam tutorial ini.

Buka command line dan jalankan perintah berikut:

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

Dengan menjalankan perintah tersebut, kita berhasil menerbitkan transformasi dari **base_link** ke **base_laser** di TF2. Sekarang, mari kita periksa apakah transformasi ini bekerja dengan benar menggunakan **tf2_echo**. Buka jendela command line terpisah dan jalankan perintah berikut:

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

Anda seharusnya dapat melihat output berulang yang mirip dengan yang di bawah ini:

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

Itu saja untuk demo singkat ini - kita berhasil menerbitkan transformasi dari **base_link** ke **base_laser** menggunakan perpustakaan TF2. Namun, perlu diingat bahwa kita tidak merekomendasikan menggunakan demo ini untuk menerbitkan transformasi pada proyek

robotika nyata Anda. Untuk sistem robot yang sebenarnya, kita akan membuat file URDF yang menyisipkan informasi ini dan lebih banyak tentang robot Anda untuk penggunaan **robot_state_publisher**, bukan **static_transform_publisher**. Ada cara yang lebih praktis untuk melakukannya yang akan dibahas dalam tutorial *Mengatur URDF*.

3.0.1 Lihat Juga

Untuk mempelajari lebih lanjut tentang TF2 dan cara membuat transform publisher Anda sendiri, kunjungi dokumen resmi TF2.

4 Transformasi dalam Navigation2

Ada dua ROS REPs penting yang sangat disarankan untuk Anda periksa. Dokumen ini merinci standar yang ditetapkan oleh komunitas ROS untuk memastikan operasi yang sesuai di berbagai paket. Nav2 juga mematuhi standar dan konvensi ini.

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

Untuk ringkasan cepat tentang REP 105, dokumen ini menetapkan konvensi penamaan dan makna semantik dari kerangka koordinat yang digunakan dalam ROS. Yang menarik untuk tutorial ini adalah kerangka koordinat **base_link**, **odom**, dan **map**.

- **base_link**: Kerangka koordinat yang melekat pada posisi tetap di robot, biasanya di sasis utama dan pusat rotasinya.
- **odom**: Kerangka yang tetap relatif terhadap posisi awal robot dan digunakan untuk representasi jarak yang konsisten secara lokal.
- **map**: Kerangka tetap dunia yang digunakan untuk representasi jarak yang konsisten secara global.

Sementara REP 103 membahas beberapa satuan ukur standar dan konvensi terkait lainnya untuk meminimalkan masalah integrasi antara berbagai paket ROS. Gambaran umumnya adalah kerangka didefinisikan menggunakan aturan tangan kanan, dengan Z mengarah ke atas dan X ke depan, dan unit harus dalam satuan SI standar.

4.1 Transformasi yang Diperlukan untuk Nav2

Nav2 memerlukan transformasi berikut yang harus diterbitkan di ROS:

- `map` =¿ `odom`
- `odom` =¿ `base_link`
- `base_link` =¿ `base_laser` (frame dasar sensor)

> **Catatan:** Kerangka koordinat **base_laser** tidak termasuk dalam standar REP 105. Dalam panduan ini, kita akan menggunakan nama ini untuk merujuk ke kerangka koordinat untuk sensor laser pada platform robot kita. Jika ada beberapa frame dasar sensor (misal: **camera_link**, **base_laser2**, **lidar_link**, dll.), maka transformasi kembali ke **base_link** untuk masing-masing diperlukan.

Transformasi pertama `map` =¿ `odom` biasanya disediakan oleh paket ROS berbeda yang menangani lokalisasi dan pemetaan seperti AMCL. Transformasi ini diperbarui secara langsung sehingga kita tidak menetapkan nilai statis untuk ini dalam pohon TF robot kita. Detail lebih lanjut tentang cara mengatur ini mungkin cukup kompleks, jadi sangat disarankan untuk melihat dokumentasi paket pemetaan atau lokalisasi yang Anda gunakan untuk platform Anda. Semua paket SLAM dan lokalisasi yang sesuai dengan ROS akan menyediakan transformasi ini secara otomatis saat diluncurkan.

Transformasi `odom` =¿ `base_link` biasanya diterbitkan oleh sistem odometri kita menggunakan sensor seperti encoder roda. Hal ini biasanya dihitung melalui penggabungan sensor odometri (IMU, encoder roda, VIO, dll) menggunakan paket **robot_localization**.

Semua transformasi yang didefinisikan secara statis lainnya (misal: **base_link** =¿ **base_laser**, **base_link** =¿ roda, roda =¿ **IMU**, dll) adalah hal yang akan kita bicarakan dalam panduan ini. Pohon transformasi ini digunakan oleh Nav2 untuk secara tepat menghubungkan informasi dari sensor atau frame lain yang menjadi perhatian ke seluruh robot. Transformasi antara dua kerangka koordinat ini biasanya disediakan ke Nav2 melalui **Robot State Publisher** dan **Universal Robot Descriptor File (URDF)**. Jika ada lebih banyak kerangka koordinat sensor pada platform Anda, maka pohon transformasi dari **base_link** ke setiap kerangka koordinat sensor perlu diterbitkan.

4.1.1 Lihat Juga

Untuk diskusi mendalam tentang penggunaan transformasi dan bagaimana transformasi tersebut digunakan untuk memperkirakan keadaan robot Anda yang sekarang, sangat disarankan untuk melihat topik **State Estimation** dalam Konsep Navigasi.

5 Kesimpulan

Dalam tutorial ini, kita telah membahas konsep transformasi dan bagaimana mereka digunakan dalam Nav2.

Di bagian terakhir, kita juga telah menjelajahi penggunaan **static_transform_publisher** dari TF2 untuk menerbitkan transformasi kita. Anda dapat menggunakan ini untuk mengatur transformasi Anda untuk Nav2, tetapi secara umum bukan cara terbaik untuk melakukannya. Dalam kebanyakan proyek robotika, kita menggunakan **Robot State Publisher** karena jauh lebih mudah digunakan dan skala dengan baik seiring dengan semakin kompleksnya robot kita. Kita akan membahas **Robot State Publisher**, URDF, dan cara mengaturnya dalam tutorial berikutnya tentang *Mengatur URDF*.

Terakhir, kita juga telah membahas tiga persyaratan transformasi yang diterbitkan dari Nav2 dan REP (ROS Enhancement Proposals) yang perlu diingat saat membuat pengaturan tersebut.

6 Mengatur URDF

Dalam panduan ini, kita akan membuat file Unified Robot Description Format (URDF) untuk robot penggerak diferensial sederhana guna memberikan pengalaman langsung bekerja dengan URDF. Kita juga akan mengatur **robot state publisher** dan memvisualisasikan model kita di RVIZ. Terakhir, kita akan menambahkan beberapa properti kinematik ke URDF robot kita untuk mempersiapkannya untuk keperluan simulasi. Langkah-langkah ini diperlukan untuk merepresentasikan semua sensor, perangkat keras, dan transformasi robot Anda agar dapat digunakan dalam navigasi.

Lihat Juga

Kode sumber lengkap dalam tutorial ini dapat ditemukan di repositori **navigation2_tutorials** di bawah paket **diditbot_description**. Perhatikan bahwa repositori tersebut berisi kode lengkap setelah menyelesaikan semua tutorial dalam panduan ini.

7 URDF dan Robot State Publisher

Seperti yang dibahas dalam tutorial sebelumnya, salah satu persyaratan untuk Navigation2 adalah transformasi dari `base_link` ke berbagai sensor dan kerangka acuan. Pohon transformasi ini bisa berupa pohon sederhana dengan hanya satu tautan dari `base_link` ke `laser_link` atau pohon yang terdiri dari beberapa sensor yang terletak di berbagai lokasi, masing-masing memiliki kerangka koordinatnya sendiri. Membuat beberapa publisher untuk menangani semua transformasi kerangka koordinat ini mungkin menjadi membosankan. Oleh karena itu, kita akan menggunakan paket **Robot State Publisher** untuk mempublish transformasi kita.

Robot State Publisher adalah paket ROS 2 yang berinteraksi dengan paket tf2 untuk mempublish semua transformasi yang diperlukan yang dapat secara langsung disimpulkan dari geometri dan struktur robot. Kita perlu memberikannya URDF yang benar dan secara otomatis akan menangani penerbitan transformasi. Ini sangat berguna untuk transformasi kompleks, namun tetap direkomendasikan untuk pohon transformasi yang lebih sederhana.

Unified Robot Description Format (URDF) adalah file XML yang merepresentasikan model robot. Dalam tutorial ini, utamanya akan digunakan untuk membangun pohon transformasi yang terkait dengan geometri robot, tetapi juga memiliki kegunaan lain. Salah satu contohnya adalah bagaimana URDF dapat digunakan dalam memvisualisasikan model robot Anda di RVIZ, alat visualisasi 3D untuk ROS, dengan mendefinisikan komponen visual seperti material dan mesh. Contoh lainnya adalah bagaimana URDF dapat digunakan untuk mendefinisikan properti fisik dari robot. Properti ini kemudian digunakan dalam simulator fisika seperti Gazebo untuk mensimulasikan bagaimana robot Anda akan berinteraksi dalam suatu lingkungan.

Fitur utama lainnya dari URDF adalah mendukung **Xacro (XML Macros)** untuk membantu Anda membuat XML yang lebih singkat dan mudah dibaca dalam mendefinisikan robot yang kompleks. Kita dapat menggunakan makro ini untuk menghilangkan kebutuhan

mengulangi blok-blok XML dalam URDF kita. Xacro juga berguna dalam mendefinisikan konstanta konfigurasi yang dapat digunakan kembali dalam URDF.

> **Lihat Juga**: Jika Anda ingin mempelajari lebih lanjut tentang URDF dan Robot State Publisher, kami mendorong Anda untuk melihat Dokumentasi resmi URDF dan Dokumentasi Robot State Publisher.

8 Mengatur Lingkungan

Dalam panduan ini, kita menganggap Anda sudah familiar dengan ROS 2 dan cara mengatur lingkungan pengembangan Anda, jadi kita akan melewati langkah-langkah di bagian ini dengan cepat.

Mari kita mulai dengan menginstal beberapa paket ROS 2 tambahan yang akan kita gunakan selama tutorial ini.

```
sudo apt install ros-humble-joint-state-publisher-gui
sudo apt install ros-humble-xacro
```

Selanjutnya, buat direktori untuk proyek Anda, inisialisasi workspace ROS 2, dan beri nama robot Anda. Untuk proyek ini, kita akan menamakannya 'diditbot'.

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

Panduan ini memberikan pengalaman langsung dalam membuat URDF, menggunakan paket **Robot State Publisher**, dan mempersiapkan model robot Anda untuk simulasi, yang merupakan langkah penting untuk mengintegrasikan robot Anda dengan Nav2 secara efektif.

9 Menulis URDF

> **Lihat juga**: Selamat datang di pengenalan dasar membuat URDF untuk robot Anda. Jika Anda ingin mempelajari lebih lanjut tentang URDF dan XAcro, lihat Dokumentasi resmi URDF.

9.1 Memulai URDF

Setelah menyiapkan ruang kerja proyek, mari kita mulai menulis URDF. Di bawah ini adalah gambar robot yang akan kita bangun.

Untuk memulai, buat file bernama `diditbot_description.urdf` di bawah `src/description` dan masukkan berikut ini sebagai isi awal file:

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

> **Catatan**: Potongan kode berikut harus ditempatkan dalam tag `<robot>`. Disarankan menambahkannya sesuai urutan yang diperkenalkan dalam tutorial ini.

9.2 Mendefinisikan Konstanta dengan XAcro

Berikutnya, kita mendefinisikan beberapa konstanta menggunakan properti XAcro yang akan digunakan dalam URDF:

```
<!-- Define robot constants -->
<xacro:property name="base_width" value="0.31" />
<xacro:property name="base_length" value="0.42" />
<xacro:property name="base_height" value="0.18" />
<xacro:property name="wheel_radius" value="0.10" />
<xacro:property name="wheel_width" value="0.04" />
<xacro:property name="wheel_ygap" value="0.025" />
<xacro:property name="wheel_zoff" value="0.05" />
<xacro:property name="wheel_xoff" value="0.12" />
<xacro:property name="caster_xoff" value="0.14" />
```

Konstanta ini akan mewakili ukuran sasis utama robot, bentuk roda belakang, dan posisi roda.

9.3 Mendefinisikan `base_link`

Kita akan mendefinisikan `base_link` sebagai elemen sasis utama robot:

```
<!-- Robot Base -->
<link name="base_link">
  <visual>
    <geometry>
      <box size="${base_length} ${base_width} ${base_height}" />
    </geometry>
    <material name="Cyan">
      <color rgba="0 1.0 1.0 1.0" />
    </material>
  </visual>
</link>
```

9.4 Mendefinisikan `base_footprint`

`base_footprint` adalah tautan virtual untuk menentukan pusat robot saat diproyeksikan ke lantai:

```
<!-- 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>
```

9.5 Menambahkan Roda Menggunakan Makro

Kita akan menambahkan dua roda penggerak besar dengan makro untuk menghindari pengulangan kode:

```
<!-- 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" />
    <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" />
```

9.6 Menambahkan Roda Caster

Kita tambahkan roda caster di bagian depan model robot:

```
<!-- 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>
```

Dengan ini, kita telah membangun URDF untuk robot penggerak diferensial sederhana. Di bagian berikutnya, kita akan fokus pada pembuatan Paket ROS yang berisi URDF kita, meluncurkan **robot state publisher**, dan memvisualisasikan robot di RViz.

10 Membangun dan Meluncurkan

10.1 Tambahkan Dependensi

Buka root direktori proyek Anda dan tambahkan baris berikut ke `package.xml` (sebaiknya setelah tag `<buildtool_depend>`):

```
<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>
```

10.2 Buat File Peluncuran

Buat direktori bernama `launch` dan file `display.launch.py` di dalamnya. Gunakan kode berikut:

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

def generate_launch_description():
    pkg_share =
        launch_ros.substitutions.FindPackageShare(package='diditbot_description
    default_model_path = os.path.join(pkg_share,
        'src/description/diditbot_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',
        parameters=[{'robot_description': Command(['xacro ',
            default_model_path])}],
        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
    ])
```

10.3 Konfigurasi RVIZ

Buat direktori bernama `rviz` dan file `urdf_config.rviz` di dalamnya. Tempatkan konfigurasi berikut:

```
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
  - 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: ~
```

10.4 Modifikasi `CMakeLists.txt`

Tambahkan cuplikan berikut ke `CMakeLists.txt`, sebaiknya di atas baris `if(BUILD_TESTING)`:

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

10.5 Membangun dan Meluncurkan Proyek

Bangun proyek dengan `colcon`:

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

Kemudian, luncurkan proyek Anda:

```
ros2 launch diditbot_description display.launch.py
```

ROS 2 akan meluncurkan node penerbit robot dan memulai RVIZ menggunakan URDF Anda. Kita akan melihat robot kita menggunakan RVIZ di bagian selanjutnya.

11 Visualisasi Menggunakan RVIZ

RVIZ adalah alat visualisasi robot yang memungkinkan kita melihat model 3D robot menggunakan URDF. Setelah berhasil meluncurkan menggunakan perintah di bagian sebelumnya, RVIZ seharusnya sudah terlihat di layar Anda. Mungkin Anda perlu memindahkan dan memanipulasi tampilan untuk melihat robot dengan baik.

Kita sudah berhasil membuat dan memvisualisasikan robot penggerak diferensial sederhana di RVIZ. Meskipun tidak selalu diperlukan, visualisasi ini penting untuk memastikan bahwa URDF Anda telah terdefinisi dengan benar. Ini membantu memastikan bahwa **robot state publisher** menerbitkan transformasi yang tepat.

Anda mungkin melihat jendela lain terbuka - ini adalah GUI untuk **joint state publisher**. Ini adalah paket ROS 2 yang menerbitkan status untuk sambungan yang tidak tetap. Anda dapat memanipulasi penerbit ini melalui GUI kecil, dan pose baru dari sambungan akan tercermin di RVIZ. Menggeser bar untuk roda mana pun akan memutar sambungan tersebut. Anda dapat melihat ini beraksi dengan menonton RVIZ saat Anda menggeser slider di GUI Joint State Publisher.

> **Catatan**: Kita tidak akan banyak berinteraksi dengan paket ini untuk Nav2, tetapi jika Anda ingin tahu lebih banyak tentang joint state publisher, silakan lihat Dokumentasi resmi Joint State Publisher.

Pada titik ini, Anda dapat memutuskan untuk menyelesaikan tutorial ini karena kita sudah mencapai tujuan membuat URDF untuk robot penggerak diferensial sederhana. **Robot state publisher** sekarang menerbitkan transformasi yang berasal dari URDF. Transformasi ini dapat digunakan oleh paket lain (seperti Nav2) untuk mendapatkan informasi tentang bentuk dan struktur robot Anda. Namun, untuk menggunakan URDF ini dalam simulasi, kita memerlukan properti fisik agar robot bereaksi terhadap lingkungan fisik seperti robot nyata. Bidang visualisasi hanya untuk visualisasi, bukan untuk tabrakan, sehingga robot Anda akan bergerak menembus rintangan. Kita akan menambahkan properti ini ke dalam URDF pada bagian selanjutnya.

12 Menambahkan Properti Fisik

12.1 Memasukkan Properti Kinematik

Kita akan memodifikasi URDF untuk menyertakan properti kinematik robot, yang digunakan oleh simulator fisika seperti Gazebo.

12.1.1 Definisikan Properti Inersia

Tambahkan makro berikut setelah bagian konstanta di URDF:

```
<!-- Define inertial 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">
  <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>
```

12.1.2 Menambahkan Area Tabrakan ke `base_link`

Tambahkan kode berikut dalam tag `<link name="base_link">`:

```
<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}"/>
```

12.1.3 Menambahkan Properti ke Makro Roda

Tambahkan kode berikut dalam tag `<link name="prefix_link">` untuk makro roda:

```
<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}" />
```

12.1.4 Menambahkan Properti ke Roda Caster

Tambahkan kode berikut dalam tag `<link name="front_caster">` untuk roda caster:

```
<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))}" />
```

> **Catatan**: Kita tidak menambah properti inersia atau tabrakan untuk `base_footprint` karena ini adalah tautan virtual dan non-fisik.

12.2 Membangun dan Meluncurkan

Bangun proyek dan luncurkan RVIZ menggunakan perintah yang sama di bagian sebelumnya:

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

Verifikasi tabrakan dengan mengaktifkan **Collision Enabled** di bawah **RobotModel**. Ini akan menampilkan representasi tabrakan robot.

12.3 Kesimpulan

Dalam tutorial ini, Anda telah berhasil membuat URDF untuk robot penggerak diferensial sederhana. Anda juga telah menyiapkan ROS 2 proyek yang meluncurkan node penerbit robot, yang kemudian menggunakan URDF Anda untuk menerbitkan transformasi robot. Kami telah menggunakan RVIZ untuk memvisualisasikan robot dan memverifikasi apakah

URDF sudah benar. Terakhir, kami menambahkan beberapa properti fisik ke URDF untuk mempersiapkannya untuk simulasi.

Gunakan tutorial ini sebagai template untuk robot Anda sendiri. Tujuan utama Anda adalah menerbitkan transformasi yang tepat dari `base_link` hingga `sensor_frames`. Setelah ini disiapkan, Anda dapat melanjutkan ke panduan pengaturan lainnya.

13 Pengantar Mengatur Odometri

Panduan ini membahas cara mengintegrasikan sistem odometri robot dengan Nav2. Kita akan memulai dengan pengantar singkat tentang odometri dan pesan serta transformasi yang diperlukan agar Nav2 berfungsi dengan benar. Selanjutnya, kita akan menunjukkan dua kasus pengaturan odometri. Pertama, untuk robot dengan encoder roda yang sudah ada. Kedua, simulasi sistem odometri yang berfungsi pada `diditbot` menggunakan Gazebo. Kemudian, kita akan membahas bagaimana sumber odometri yang berbeda dapat digabungkan untuk menghasilkan odometri yang lebih halus menggunakan paket `robot_localization`. Terakhir, kita juga akan menunjukkan cara menerbitkan transformasi `odom => base_link` menggunakan `robot_localization`.

Lihat Juga: Kode sumber lengkap dalam tutorial ini dapat ditemukan di repositori di navigation2_tutorials di bawah paket **tubes_ws**.

14 Pengantar Odometri

Sistem odometri memberikan estimasi posisi dan kecepatan robot berdasarkan gerakannya. Informasi odometri dapat diperoleh dari berbagai sumber seperti IMU, LIDAR, RADAR, VIO, dan encoder roda. Penting untuk dicatat bahwa IMU mengalami drift seiring waktu sementara encoder roda mengalami drift seiring jarak yang ditempuh, oleh karena itu sering digunakan bersama untuk menyeimbangkan karakteristik negatif masing-masing.

Kerangka `odom` dan transformasi yang terkait dengannya menggunakan sistem odometri robot untuk menerbitkan informasi lokalisasi yang kontinu namun menjadi kurang akurat seiring waktu atau jarak. Meski begitu, informasi ini tetap dapat digunakan robot untuk navigasi di sekitar lingkungan terdekatnya (misalnya menghindari tabrakan). Untuk mendapatkan informasi odometri yang akurat secara konsisten, kerangka `map` menyediakan informasi global yang digunakan untuk mengoreksi kerangka `odom`.

Seperti yang dibahas dalam panduan sebelumnya dan di REP 105, kerangka `odom` terhubung ke sistem dan Nav2 melalui transformasi `odom => base_link`. Transformasi ini diterbitkan oleh tf2 broadcaster atau oleh kerangka kerja seperti `robot_localization`, yang juga menyediakan fungsionalitas tambahan. Kita akan membahas lebih lanjut tentang `robot_localization` di bagian berikutnya.

Selain transformasi `odom => base_link` yang diperlukan, Nav2 juga memerlukan penerbitan pesan `nav_msgs/Odometry` karena pesan ini menyediakan informasi kecepatan robot. Secara detail, pesan `nav_msgs/Odometry` berisi informasi berikut:

```
# Ini mewakili estimasi posisi dan kecepatan di ruang bebas.
```

```
# Pose dalam pesan ini harus ditentukan dalam kerangka koordinat
    yang ditentukan oleh header.frame_id
# Twist dalam pesan ini harus ditentukan dalam kerangka koordinat
    yang ditentukan oleh child_frame_id

# Menyertakan ID frame dari induk pose.
std_msgs/Header header

# ID frame yang ditunjukkan pose. Twist ada dalam kerangka
    koordinat ini.
string child_frame_id

# Estimasi pose yang biasanya relatif terhadap kerangka dunia
    tetap.
geometry_msgs/PoseWithCovariance pose

# Estimasi kecepatan linier dan angular relatif terhadap
    child_frame_id.
geometry_msgs/TwistWithCovariance twist
```

Pesan ini memberikan estimasi untuk pose dan kecepatan robot. Pesan `header` menyediakan data yang ditandai waktu dalam kerangka koordinat tertentu. Pesan `pose` memberikan posisi dan orientasi robot relatif terhadap kerangka yang ditentukan dalam `header.frame_id`. Pesan `twist` memberikan kecepatan linier dan angular relatif terhadap kerangka yang didefinisikan dalam `child_frame_id`.

15 Mengatur Odometri pada Robot Anda

Pengaturan sistem odometri untuk Nav2 pada robot fisik sangat bergantung pada sensor odometri yang tersedia. Karena banyaknya konfigurasi, instruksi spesifik di luar cakupan tutorial ini, tetapi kami akan memberikan contoh dasar dan sumber daya yang berguna.

15.1 Contoh: Robot dengan Encoder Roda

Encoder roda sering digunakan meskipun tidak wajib untuk Nav2. Tujuan pengaturan odometri adalah menghitung informasi odometri dan menerbitkan pesan `nav_msgs/Odometry` serta transformasi `odom => base_link` melalui ROS 2.

15.1.1 Menghitung Informasi Odometri

Berikut adalah contoh sederhana untuk menghitung kecepatan linear dan angular:

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

- `right_wheel_est_vel` dan `left_wheel_est_vel`: Kecepatan yang diperkirakan dari roda kanan dan kiri. - `wheel_separation`: Jarak antara roda.

Pemantauan perubahan posisi sambungan roda dari waktu ke waktu dapat memberi nilai kecepatan ini. Informasi ini digunakan untuk menerbitkan persyaratan Nav2.

15.1.2 Menggunakan `ros2_control`

Framework `ros2_control` menyediakan paket untuk kontrol real-time. `diff_drive_controller` dalam paket `ros2_controller` akan:

- Mengambil pesan `geometry_msgs/Twist` dari topik `cmd_vel`.
- Menghitung informasi odometri.
- Menerbitkan pesan `nav_msgs/Odometry` pada topik `odom`.

Lihat Juga

- Dokumentasi ros2_control
- Repositori Github $\text{diff}_{d}rive_{c}ontroller$

Untuk sensor lain seperti IMU, VIO, dll., driver ROS mereka akan memberikan dokumentasi tentang cara menerbitkan informasi odometri. Ingat bahwa Nav2 memerlukan pesan `nav_msgs/Odometry` dan transformasi `odom => base_link`.

16 Mensimulasikan Sistem Odometri menggunakan Gazebo

Dalam bagian ini, kita akan menggunakan Gazebo untuk mensimulasikan sistem odometri dari `diditbot`, robot yang dibangun pada bagian sebelumnya. Anda bisa mengikuti panduan tersebut atau menggunakan kode lengkap yang tersedia.

Catatan: Jika Anda bekerja dengan robot fisik dan telah mengatur sensor odometri, Anda dapat melewati bagian ini dan melanjutkan ke bagian berikutnya tentang penggabungan pesan IMU dan odometri.

16.1 Ikhtisar

1. **Pengaturan Gazebo**: Instal dan konfigurasi Gazebo serta paket-paket yang diperlukan agar berfungsi dengan ROS 2.
2. **Menambahkan Plugin Gazebo**: Simulasikan sensor IMU dan sistem odometri penggerak diferensial untuk menerbitkan pesan `sensor_msgs/Imu` dan `nav_msgs/Odometry`.
3. **Menjalankan `diditbot` di Gazebo**: Verifikasi pesan yang diterbitkan di ROS 2.

16.2 Pengaturan dan Prasyarat

Gazebo adalah simulator 3D yang memungkinkan kita mengamati bagaimana robot virtual akan berfungsi dalam lingkungan simulasi. Untuk memulai penggunaan Gazebo dengan ROS 2, ikuti instruksi di Dokumentasi Instalasi Gazebo.

Instal paket `gazebo_ros_pkgs` untuk mensimulasikan odometri dan mengontrol robot dengan ROS 2 di Gazebo:

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

Anda dapat menguji apakah lingkungan ROS 2 dan Gazebo telah dikonfigurasi dengan benar dengan mengikuti instruksi yang diberikan di sini.

16.3 Menggunakan URDF di Gazebo

Kita menggambarkan `diditbot` menggunakan URDF, namun Gazebo menggunakan Simulation Description Format (SDF) untuk mendeskripsikan robot dalam lingkungan simulasinya. Beruntung, Gazebo secara otomatis menerjemahkan file URDF yang kompatibel menjadi SDF. Syarat utama agar URDF kompatibel dengan Gazebo adalah kehadiran elemen `<inertia>` dalam setiap elemen `<link>`. Persyaratan ini sudah dipenuhi dalam file URDF `diditbot` sehingga siap digunakan di Gazebo.

Lihat Juga: Tutorial: Menggunakan URDF di Gazebo

17 Menambahkan Plugin Gazebo ke URDF

Kita akan menambahkan sensor IMU dan plugin penggerak diferensial Gazebo ke dalam URDF kita. Untuk melihat plugin lainnya, lihat Tutorial: Menggunakan plugin Gazebo dengan ROS.

17.1 Menambahkan Sensor IMU

Gunakan `GazeboRosImuSensor` sebagai `SensorPlugin` yang harus melekat pada `link`. Buat `imu_link` dan atur topik `/demo/imu` untuk penerbitan informasi IMU.

17.1.1 Tambahkan ke URDF

Tambahkan sebelum garis `</robot>` di URDF Anda:

```
<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_referenc
    </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>
```

```
            <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>
```

17.2 Menambahkan Plugin Penggerak Diferensial

Konfigurasikan plugin sehingga pesan `nav_msgs/Odometry` diterbitkan pada topik `/demo/odom`.

17.2.1 Tambahkan ke URDF

Tambahkan setelah tag `</gazebo>` IMU:

```
<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>
    <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>
```

18 Membuat, Menjalankan, dan Memverifikasi

Mari kita jalankan paket kita untuk memeriksa apakah topik `/demo/imu` dan `/demo/odom` aktif dalam sistem.

18.1 Membangun dan Menjalankan

Untuk memulai, navigasikan ke root proyek Anda dan eksekusi perintah berikut:

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

Gazebo seharusnya diluncurkan dan menampilkan model 3D dari `diditbot`.

18.2 Memverifikasi Topik

Buka terminal baru dan jalankan:

```
ros2 topic list
```

Pastikan `/demo/imu` dan `/demo/odom` muncul dalam daftar topik. Untuk melihat lebih banyak detail tentang topik, dapatkan informasi dengan:

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

Anda akan melihat bahwa `/demo/imu` menerbitkan pesan tipe `sensor_msgs/Imu` dan `/demo/odom` menerbitkan pesan tipe `nav_msgs/Odometry`. Informasi yang diterbitkan pada topik-topik ini berasal dari simulasi gazebo dari sensor IMU dan penggerak diferensial. Juga perhatikan bahwa kedua topik saat ini tidak memiliki pelanggan. Dalam bagian berikutnya, kita akan membuat node `robot_localization` yang akan berlangganan dua topik ini dan menggunakan pesan yang diterbitkan untuk memberikan informasi odometri yang lebih halus untuk Nav2.

19 Demo Lokalisasi Robot

Paket `robot_localization` digunakan untuk menyediakan informasi odometri yang lebih halus dan akurat dari beberapa sumber sensor odometri. Informasi ini dapat disediakan melalui pesan `nav_msgs/Odometry`, `sensor_msgs/Imu`, `geometry_msgs/PoseWithCovarianceStamped`, dan `geometry_msgs/TwistWithCovarianceStamped`.

Setelan robot biasanya terdiri dari setidaknya encoder roda dan IMU sebagai sumber sensor. Ketika beberapa sumber disediakan untuk `robot_localization`, ia dapat menggabungkan informasi odometri yang diberikan oleh sensor melalui penggunaan node estimasi keadaan. Node-node ini menggunakan either filter Kalman yang Diperpanjang (`ekf_node`) atau Filter Kalman Tidak Tersebar (`ukf_node`) untuk mengimplementasikan penggabungan ini. Selain itu, paket ini juga mengimplementasikan `navsat_transform_node` yang mengubah koordinat geografis menjadi kerangka dunia robot saat bekerja dengan GPS.

Data sensor yang digabungkan diterbitkan oleh paket `robot_localization` melalui topik `odometry/filtered` dan `accel/filtered`, jika diaktifkan dalam konfigurasinya. Selain itu, ia juga dapat menerbitkan transformasi `odom => base_link` pada topik `/tf`.

Untuk robot yang hanya mampu menyediakan satu sumber odometri, penggunaan `robot_localization` akan memberikan efek minimal selain memperhalus. Dalam kasus ini, pendekatan alternatif adalah menerbitkan transformasi melalui penyiar `tf2` di node sumber odometri tunggal Anda. Namun, Anda masih dapat memilih untuk menggunakan `robot_localization` untuk menerbitkan transformasi dan beberapa sifat perhalusan mungkin masih diamati dalam output.

Untuk informasi lebih tentang cara menulis penyiar `tf2`, Anda dapat memeriksa Menulis penyiar tf2 (C++) atau versi Python.

20 Mengonfigurasi Robot Localization

Sekarang kita akan mengonfigurasi paket `robot_localization` untuk menggunakan Extended Kalman Filter (`ekf_node`) untuk menggabungkan informasi odometri dan menerbitkan transformasi `odom => base_link`.

20.1 Instalasi Paket

Pertama, instal paket `robot_localization` dengan menjalankan perintah berikut:

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

20.2 Konfigurasi dengan File YAML

Selanjutnya, tentukan parameter `ekf_node` menggunakan file YAML. Buat direktori bernama `config` di root proyek Anda dan buat file bernama `ekf.yaml`. Salin baris kode berikut ke dalam file `ekf.yaml`:

```
### ekf config file ###
ekf_filter_node:
  ros__parameters:
    frequency: 30.0
    two_d_mode: false
    publish_acceleration: true
    publish_tf: true
    map_frame: map
    odom_frame: odom
    base_link_frame: base_link
    world_frame: odom
    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]
```

Dalam konfigurasi ini, kita mendefinisikan nilai parameter seperti `frequency`, `two_d_mode`, `publish_acceleration`, `publish_tf`, `map_frame`, `odom_frame`, `base_link_frame`, dan `world_frame`.

Anda bisa menentukan nilai mana dari sensor yang akan digunakan filter menggunakan parameter $_{c}onfig.Urutannilaiparameteriniadalahx, y, z, roll, pitch, yaw, vx, vy, vz, vroll, vpitch, vyaw, ax$

20.3 Menambah Ekf Node ke Berkas Peluncuran

`Buka launch/display.launch.py dan tambahkan baris berikut sebelum garis return launch.LaunchDescription([:`

```
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')}]
```

```
)
```

Tambahkan argumen peluncuran berikut dalam blok return launch.LaunchDescription([:

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

Tambahkan robot_localization_node di atas baris rviz_node:

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

20.4 Menambahkan Ketergantungan di package.xml dan CMakeLists.txt

Buka package.xml dan tambahkan baris berikut di bawah tag <exec_depend> terakhir:

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

Terakhir, buka CMakeLists.txt dan tambahkan direktori config dalam install(DIRECTORY...)

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

21 Build, Run, and Verification

21.1 Membangun dan Menjalankan Paket

Navigasi ke root proyek dan jalankan perintah berikut:

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

Gazebo dan RVIZ seharusnya diluncurkan. Di jendela RVIZ, Anda akan melihat model dan kerangka TF dari diditbot.

21.2 Memverifikasi Topik

Buka terminal baru dan jalankan:

```
ros2 topic list
```

Pastikan odometry/filtered, accel/filtered, dan /tf muncul dalam daftar topik.
Untuk memeriksa jumlah pelanggan dari topik ini, jalankan:

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

Anda akan melihat bahwa /demo/imu dan /demo/odom sekarang masing-masing memiliki 1 pelanggan.

21.3 Verifikasi ekf$_{f}ilter_{n}ode$

Untuk memverifikasi bahwa ekf_filter_node adalah pelanggan topik ini, jalankan:

```
ros2 node info /ekf_filter_node
```

Anda akan melihat bahwa ekf_filter_node berlangganan ke /demo/imu dan /demo/odom, dan menerbitkan data pada topik odometry/filtered, accel/filtered, dan /tf.

21.4 Memeriksa Transformasi odom =¿ base_link

Gunakan utilitas tf2_echo untuk memverifikasi bahwa robot_localization menerbitkan transformasi odom => base_link. Eksekusi perintah berikut di terminal yang berbeda:

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

Anda harus melihat keluaran berkelanjutan yang menunjukkan transformasi posisi dan rotasi.

22 Kesimpulan

Dalam panduan ini, kita telah membahas pesan dan transformasi yang diharapkan oleh Nav2 dari sistem odometri. Kita telah melihat cara mengatur sistem odometri dan cara memverifikasi pesan yang diterbitkan. Kita juga membahas bagaimana beberapa sensor odometri dapat digunakan untuk menyediakan odometri yang difilter dan dihaluskan menggunakan robot_localization. Kita telah memeriksa apakah transformasi odom => base_link diterbitkan dengan benar oleh robot_localization.

23 Pengantar Sensor

Dalam panduan ini, kita akan membahas pentingnya sensor dalam menavigasi robot dengan aman dan cara mengatur sensor dengan Nav2. Pada bagian pertama tutorial ini, kita akan melihat sekilas tentang sensor yang umum digunakan dan pesan sensor

umum dalam Nav2. Selanjutnya, kita akan menambahkan pengaturan sensor dasar pada robot simulasi yang telah kita bangun sebelumnya, diditbot. Terakhir, kita akan memverifikasi pesan sensor simulasi diditbot dengan memvisualisasikannya di RViz.

Setelah sensor terpasang pada robot, pembacaan sensor dapat digunakan dalam pemetaan, pelokalan, dan tugas persepsi. Pada bagian kedua panduan ini, kita akan terlebih dahulu membahas bagaimana pemetaan dan pelokalan menggunakan data sensor. Kemudian, kita juga akan melihat salah satu paket Nav2, nav2_costmap_2d, yang menghasilkan costmap yang akan digunakan dalam perencanaan jalur Nav2. Kita akan mengatur parameter konfigurasi dasar untuk paket ini supaya dapat menerima informasi sensor dari diditbot. Terakhir, kita akan memvisualisasikan costmap yang dihasilkan di RViz untuk memverifikasi data yang diterima.

24 Pengantar Sensor

Robot mobile dilengkapi dengan berbagai sensor yang memungkinkan mereka untuk melihat dan memahami lingkungan mereka. Sensor-sensor ini mendapatkan informasi yang dapat digunakan untuk membangun dan memelihara peta lingkungan, untuk melokalisasi robot pada peta, serta untuk melihat rintangan di lingkungan. Tugas-tugas ini penting agar robot dapat menavigasi melalui lingkungan dinamis dengan aman dan efisien.

Contoh sensor yang umum digunakan adalah lidar, radar, kamera RGB, kamera kedalaman, IMU, dan GPS. Untuk menstandarisasi format pesan dari sensor-sensor ini dan memungkinkan interoperabilitas yang lebih mudah di antara vendor, ROS menyediakan paket sensor_msgs yang mendefinisikan antarmuka sensor umum. Hal ini juga memungkinkan pengguna menggunakan sensor dari vendor mana pun asalkan mengikuti format standar dari sensor_msgs. Pada subbagian berikut, kita akan memperkenalkan beberapa pesan yang umum digunakan dalam navigasi, yaitu sensor_msgs/LaserScan, sensor_msgs/PointCloud2, sensor_msgs/Range, dan sensor_msgs/Image.

Selain paket sensor_msgs, ada juga antarmuka standar radar_msgs dan vision_msgs yang harus diperhatikan. Paket radar_msgs mendefinisikan pesan untuk sensor yang spesifik radar sedangkan paket vision_msgs mendefinisikan pesan yang digunakan dalam visi komputer seperti deteksi objek, segmentasi, dan model pembelajaran mesin lainnya. Pesan yang didukung oleh paket ini adalah vision_msgs/Classification2D, vision_msgs/Classification3D, vision_msgs/Detection2D, dan vision_msgs/Detection3D, untuk menyebutkan beberapa.

Lihat juga

Untuk informasi lebih lanjut, lihat dokumentasi API dari sensor_msgs, radar_msgs, dan vision_msgs.

Sensor robot fisik Anda mungkin memiliki driver ROS yang ditulis untuknya (misalnya, sebuah node ROS yang menghubungkan ke sensor, mengisi data ke dalam pesan, dan menerbitkannya untuk digunakan robot Anda) yang mengikuti antarmuka standar dalam paket sensor_msgs. Paket sensor_msgs memudahkan Anda untuk menggunakan berbagai sensor dari berbagai produsen. Paket perangkat lunak umum seperti Nav2 kemudian dapat membaca pesan standar ini dan melakukan tugas tanpa bergantung pada perangkat

keras sensor. Pada robot simulasi seperti diditbot, Gazebo memiliki plugin sensor yang juga menerbitkan informasi mereka mengikuti paket sensor_msgs.

25 Pesan Sensor Umum

Pada subbagian ini, kita akan membahas beberapa tipe sensor_msgs yang umum Anda temui saat mengatur Nav2. Kita akan memberikan deskripsi singkat untuk setiap sensor, gambar saat disimulasi di Gazebo, dan visualisasi yang sesuai dari pembacaan sensor di RViz.

Catatan

Ada jenis lain dari sensor_msgs selain yang terdaftar di bawah ini. Daftar lengkap pesan dan definisinya dapat ditemukan dalam dokumentasi sensor_msgs.

25.1 sensor_msgs/LaserScan

Pesan ini mewakili satu pemindaian dari pemindai laser planar. Pesan ini digunakan dalam slam_toolbox dan nav2_amcl untuk pelokalan dan pemetaan, atau dalam nav2_costmap_2d untuk persepsi.

25.2 sensor_msgs/PointCloud2

Pesan ini menyimpan kumpulan titik 3D, plus informasi tambahan opsional tentang setiap titik. Ini dapat berasal dari lidar 3D, lidar 2D, kamera kedalaman, atau lebih.

25.3 sensor_msgs/Range

Ini adalah satu pembacaan jarak dari ranger aktif yang memancarkan energi dan melaporkan satu pembacaan jarak yang valid sepanjang busur pada jarak yang diukur. Sonar, sensor IR, atau pemindai jarak 1D adalah contoh sensor yang menggunakan pesan ini.

25.4 sensor_msgs/Image

Pesan ini mewakili pembacaan sensor dari kamera RGB atau kamera kedalaman, yang sesuai dengan nilai RGB atau jarak.

26 Mensimulasikan Sensor menggunakan Gazebo

Untuk memberikan pemahaman yang lebih baik tentang cara mengatur sensor pada robot simulasi, kita akan membangun tutorial sebelumnya dan memasang sensor pada robot simulasi kita diditbot. Mirip dengan tutorial sebelumnya di mana kita menggunakan plugin Gazebo untuk menambahkan sensor odometri ke diditbot, kita akan menggunakan plugin Gazebo untuk mensimulasikan sensor lidar dan kamera kedalaman pada diditbot.

Jika Anda bekerja dengan robot fisik, sebagian besar langkah ini masih diperlukan untuk mengatur kerangka URDF Anda dan juga tidak ada salahnya untuk menambahkan plugin Gazebo untuk penggunaan di kemudian hari.

Untuk mengikuti sisa bagian ini, pastikan Anda telah menginstal Gazebo dengan benar. Anda dapat mengikuti instruksi di bagian Pengaturan dan Prasyarat dari tutorial sebelumnya untuk mengatur Gazebo.

27 Menambahkan Plugin Gazebo ke dalam URDF

Mari kita tambahkan sensor lidar ke diditbot. Buka file URDF, src/description/sam_bot_de dan tempelkan baris-baris berikut sebelum tag </robot>.

```
<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>
  </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>
```

Dalam potongan kode di atas, kita membuat lidar_link yang akan dirujuk oleh plugin gazebo_ros_ray_sensor sebagai lokasi untuk menempelkan sensor kita. Kita juga mengatur nilai pada pemindaian dan properti jarak lidar simulasi. Terakhir, kita menetapkan /scan sebagai topik tempat sensor akan menerbitkan pesan sensor_msgs/Las

Selanjutnya, mari kita tambahkan kamera kedalaman ke diditbot. Tempelkan baris berikut setelah tag </gazebo> sensor lidar.

```
<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>
```

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

Mirip dengan sensor lidar, kita membuat camera_link yang akan dirujuk oleh plugin gazebo_ros_camera sebagai lokasi tempat sensor dipasang. Kita juga membuat camera_depth_frame yang terpasang ke camera_link dan akan ditetapkan sebagai <frame_name> dari plugin kamera kedalaman. Kita juga mengonfigurasi plugin sehingga akan menerbitkan pesan sensor_msgs/Image dan sensor_msgs/PointCloud2 ke topik /depth_camera/image_raw dan /depth_camera/points masing-masing. Terakhir, kita juga menetapkan properti konfigurasi dasar lainnya untuk kamera kedalaman kita.

28 File Peluncuran dan Pengaturan

Untuk memverifikasi bahwa sensor telah diatur dengan benar dan dapat melihat objek di lingkungan kita, mari kita luncurkan diditbot di dunia Gazebo dengan objek. Mari kita buat dunia Gazebo dengan satu kubus dan satu bola yang berada dalam jangkauan sensor diditbot sehingga kita dapat memverifikasi apakah ia dapat melihat objek dengan benar.

Untuk membuat dunia, buat direktori bernama world di root proyek Anda dan buat file bernama my_world.sdf di dalam folder world. Kemudian salin isi dari world/my_world dan tempelkan ke dalam my_world.sdf.

Sekarang, mari kita edit file peluncuran kita, launch/display.launch.py, untuk meluncurkan Gazebo dengan dunia yang baru saja kita buat. Pertama, tambahkan path dari my_world.sdf dengan menambahkan baris berikut di dalam generate_launch_descript

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

Terakhir, tambahkan path dunia di dalam baris launch.actions.ExecuteProcess(cmd=['gaz seperti ditunjukkan di bawah ini.

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

Kita juga harus menambahkan direktori dunia ke dalam file CMakeLists.txt kita. Buka CMakeLists.txt dan tambahkan direktori dunia di dalam install(DIRECTORY...), seperti ditunjukkan pada potongan di bawah ini.

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

29 Membangun, Menjalankan, dan Verifikasi

Sekarang kita dapat membangun dan menjalankan proyek kita. Navigasikan ke root proyek dan eksekusi baris berikut:

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

RViz dan Gazebo kemudian akan diluncurkan dengan diditbot hadir di keduanya. Di jendela Gazebo, dunia yang kita buat harus diluncurkan dan diditbot harus muncul di dunia itu. Anda seharusnya sekarang dapat mengamati diditbot dengan sensor lidar 360 derajat dan kamera kedalaman.

Di jendela RViz, kita dapat memverifikasi apakah kita telah memodelkan sensor kita dengan benar dan apakah transformasi dari sensor yang baru ditambahkan sudah benar.

Terakhir, kita juga dapat memvisualisasikan pembacaan sensor di RViz. Untuk memvisualisasikan pesan sensor_msgs/LaserScan yang diterbitkan pada topik /scan, klik tombol tambah di bagian bawah jendela RViz. Kemudian pergi ke tab By topic dan pilih opsi LaserScan di bawah /scan.

Selanjutnya, atur Kebijakan Keandalan di RViz ke Best Effort dan atur ukuran ke 0.1 untuk melihat titik-titik dengan lebih jelas. Anda akan melihat visualisasi deteksi LaserScan seperti yang ditunjukkan di bawah ini. Ini sesuai dengan kubus dan bola yang terdeteksi yang kita tambahkan ke dunia Gazebo.

Untuk memvisualisasikan sensor_msgs/Image dan sensor_msgs/PointCloud2, lakukan hal yang sama untuk topik /depth_camera/image_raw dan /depth_camera/points masing-masing.

Setelah menambahkan topik /depth_camera/image_raw di RViz, atur Kebijakan Keandalan di RViz ke Best Effort. Maka Anda akan melihat kubus di jendela gambar di sisi kiri bawah jendela RViz.

Anda juga harus melihat sensor_msgs/PointCloud2, seperti yang ditunjukkan di bawah ini.

30 Pemetaan dan Pelokalan

Sekarang kita memiliki robot dengan sensor yang terpasang, kita dapat menggunakan informasi sensor yang diperoleh untuk membangun peta lingkungan dan untuk melokalisasi robot pada peta. Paket slam_toolbox adalah sekumpulan alat dan kemampuan untuk 2D Simultaneous Localization and Mapping (SLAM) di peta besar dengan ROS2. Ini juga merupakan salah satu pustaka SLAM yang didukung secara resmi dalam Nav2, dan kami merekomendasikan menggunakan paket ini dalam situasi di mana Anda perlu menggunakan SLAM pada pengaturan robot Anda. Selain itu, pelokalan juga dapat diimplementasikan melalui paket nav2_amcl. Paket ini menerapkan Adaptive Monte Carlo Localization (AMCL) yang memperkirakan posisi dan orientasi robot di peta. Teknik lainnya juga mungkin tersedia, silakan periksa dokumentasi Nav2 untuk informasi lebih lanjut.

Baik slam_toolbox maupun nav2_amcl menggunakan informasi dari sensor pemindaian laser untuk dapat memahami lingkungan robot. Oleh karena itu, untuk memverifikasi bahwa mereka dapat mengakses pembacaan sensor pemindaian laser, kita harus memastikan bahwa mereka terdaftar pada topik yang benar yang menerbitkan pesan sensor_msgs/LaserSca
Ini dapat dikonfigurasi dengan mengatur parameter scan_topic mereka ke topik yang menerbitkan pesan tersebut. Biasanya pesan sensor_msgs/LaserScan diterbitkan pada topik /scan. Oleh karena itu, secara default, parameter scan_topic diset ke /scan. Ingat bahwa ketika kita menambahkan sensor lidar ke diditbot di bagian sebelumnya, kita telah menetapkan topik di mana sensor lidar akan menerbitkan pesan sensor_msgs/LaserScan sebagai /scan.

Diskusi mendalam tentang parameter konfigurasi lengkap tidak akan termasuk dalam ruang lingkup tutorial kami karena bisa sangat kompleks. Sebagai gantinya, kami merekomendasikan Anda untuk melihat dokumentasi resmi mereka di tautan di bawah.

Lihat juga

- Untuk daftar lengkap parameter konfigurasi dari slam_toolbox, lihat repositori Github dari slam_toolbox.
- Untuk daftar lengkap parameter konfigurasi dan contoh konfigurasi dari nav2_amcl, lihat Panduan Konfigurasi AMCL.
- Anda juga dapat merujuk ke panduan (SLAM) Menavigasi Sambil Memetakan untuk tutorial tentang cara menggunakan Nav2 dengan SLAM. Anda dapat memverifikasi bahwa slam_toolbox dan nav2_amcl telah diatur dengan benar dengan memvisualisasikan peta dan pose robot di RViz, mirip dengan apa yang ditunjukkan di bagian sebelumnya.

31 Costmap 2D

Paket costmap 2D memanfaatkan informasi sensor untuk menyediakan representasi lingkungan robot dalam bentuk grid okupansi. Sel-sel dalam grid okupansi menyimpan nilai biaya antara 0-254 yang menunjukkan biaya untuk melalui zona tersebut. Biaya 0 berarti sel tersebut bebas, sedangkan biaya 254 berarti sel tersebut terisi secara mematikan. Nilai di antara kedua ekstrem ini digunakan oleh algoritma navigasi untuk mengarahkan robot Anda menjauh dari rintangan sebagai bidang potensial. Costmap dalam Nav2 diimplementasikan melalui paket nav2_costmap_2d.

Implementasi costmap terdiri dari beberapa lapisan, masing-masing memiliki fungsi tertentu yang berkontribusi pada biaya keseluruhan sel. Paket ini terdiri dari lapisan-lapisan berikut, tetapi berbasis plugin untuk memungkinkan kustomisasi dan penggunaan lapisan baru: lapisan statis, lapisan inflasi, lapisan rentang, lapisan rintangan, dan lapisan voxel. Lapisan statis mewakili bagian peta dari costmap, yang diperoleh dari pesan yang diterbitkan ke topik /map seperti yang dihasilkan oleh SLAM. Lapisan rintangan mencakup objek yang terdeteksi oleh sensor yang menerbitkan pesan baik LaserScan maupun PointCloud2. Lapisan voxel mirip dengan lapisan rintangan dalam artian dapat menggunakan baik LaserScan maupun PointCloud2 sebagai informasi sensor tetapi menangani data 3D sebagai gantinya. Lapisan rentang memungkinkan inklusi informasi yang diberikan oleh sensor sonar dan inframerah. Terakhir, lapisan inflasi mewakili nilai biaya tambahan di sekitar rintangan mematikan sehingga robot kita menghindari navigasi ke dalam rintangan karena geometri robot. Pada subbagian berikut dari tutorial ini, kita akan membahas konfigurasi dasar dari berbagai lapisan dalam nav2_costmap_2d.

Lapisan-lapisan ini diintegrasikan ke dalam costmap melalui antarmuka plugin dan kemudian diinflasi menggunakan radius inflasi yang ditentukan oleh pengguna, jika lapisan inflasi diaktifkan. Untuk diskusi lebih dalam tentang konsep costmap, Anda dapat melihat dokumentasi costmap_2D ROS1. Perlu dicatat bahwa paket nav2_costmap_2 sebagian besar adalah versi port langsung ROS2 dari versi stack navigasi ROS1 dengan perubahan kecil yang diperlukan untuk mendukung ROS2 dan beberapa plugin lapisan baru.

32 Mengonfigurasi nav2_costmap_2d

Di subbagian ini, kita akan menunjukkan contoh konfigurasi nav2_costmap_2d sehingga menggunakan informasi yang diberikan oleh sensor lidar dari diditbot. Kita akan menunjukkan contoh konfigurasi yang menggunakan lapisan statis, lapisan rintangan, lapisan voxel, dan lapisan inflasi. Kita mengatur baik lapisan rintangan maupun lapisan voxel untuk menggunakan pesan LaserScan yang diterbitkan ke topik /scan oleh sensor lidar.

```
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
```

```
      width: 3
      height: 3
      resolution: 0.05
      robot_radius: 0.22
      plugins: ["voxel_layer", "inflation_layer"]

      voxel_layer:
        plugin: "nav2_costmap_2d::VoxelLayer"
        enabled: True
        publish_voxel_map: True
        origin_z: 0.0
        z_resolution: 0.05
        z_voxels: 16
        max_obstacle_height: 2.0
        mark_threshold: 0
        observation_sources: scan
        scan:
          topic: /scan
          max_obstacle_height: 2.0
          clearing: True
          marking: True
          data_type: "LaserScan"

      inflation_layer:
        plugin: "nav2_costmap_2d::InflationLayer"
        cost_scaling_factor: 3.0
        inflation_radius: 0.55
      always_send_full_costmap: True
```

Dalam konfigurasi di atas, perhatikan bahwa kita mengatur parameter untuk dua costmap yang berbeda: global_costmap dan local_costmap. Kita menyiapkan dua costmap karena global_costmap terutama digunakan untuk perencanaan jangka panjang di seluruh peta, sementara local_costmap digunakan untuk perencanaan jangka pendek dan penghindaran tabrakan.

Lapisan-lapisan yang kita gunakan untuk konfigurasi kita didefinisikan dalam parameter plugins, seperti yang ditunjukkan pada baris 13 untuk global_costmap dan baris 50 untuk local_costmap. Nilai ini diset sebagai daftar nama lapisan yang dipetakan yang juga berfungsi sebagai ruang nama untuk parameter lapisan yang kita atur yang dimulai pada baris 14 dan baris 51. Perhatikan bahwa setiap lapisan/ruang nama dalam daftar ini harus memiliki parameter plugin (seperti yang ditunjukkan pada baris 15, 18, 32, 52, dan 68) yang mendefinisikan tipe plugin yang akan dimuat untuk lapisan tertentu tersebut.

Untuk lapisan statis (baris 14-16), kita mengatur parameter map_subscribe_transient_lo ke True. Ini mengatur pengaturan QoS untuk topik peta. Parameter penting lainnya untuk lapisan statis adalah map_topic yang mendefinisikan topik peta yang akan

di-subscribe. Ini default ke topik /map ketika tidak didefinisikan.

Untuk lapisan rintangan (baris 17-30), kita mendefinisikan sumber sensornya di bawah parameter observation_sources (baris 20) sebagai scan yang parameternya diatur pada baris 22-30. Kita mengatur parameter topiknya sebagai topik yang menerbitkan sumber sensor yang ditentukan dan kita mengatur data_type sesuai dengan sumber sensor yang akan digunakan. Dalam konfigurasi kita, lapisan rintangan akan menggunakan LaserScan yang diterbitkan oleh sensor lidar untuk /scan.

Perhatikan bahwa lapisan rintangan dan lapisan voxel dapat menggunakan baik atau keduanya LaserScan dan PointCloud2 sebagai data_type mereka tetapi secara default diatur ke LaserScan. Potongan kode di bawah ini menunjukkan contoh penggunaan baik LaserScan maupun PointCloud2 sebagai sumber sensor. Ini mungkin sangat berguna saat mengatur robot fisik Anda sendiri.

```
obstacle_layer:
  plugin: "nav2_costmap_2d::ObstacleLayer"
  enabled: True
  observation_sources: scan pointcloud
  scan:
    topic: /scan
    data_type: "LaserScan"
  pointcloud:
    topic: /depth_camera/points
    data_type: "PointCloud2"
```

Untuk parameter lainnya dari lapisan rintangan, parameter max_obstacle_height mengatur tinggi maksimum dari pembacaan sensor untuk dikembalikan ke grid okupansi. Tinggi minimum dari pembacaan sensor juga dapat diatur menggunakan parameter min_obstacl yang default ke 0 karena kita tidak mengaturnya dalam konfigurasi. Parameter clearing digunakan untuk mengatur apakah rintangan akan dihapus dari costmap atau tidak. Operasi pembersihan dilakukan dengan raytracing melalui grid. Jarak maksimum dan minimum untuk raytrace menghapus objek dari costmap diatur menggunakan raytrace_max_r dan raytrace_min_range masing-masing. Parameter marking digunakan untuk mengatur apakah rintangan yang dimasukkan akan ditandai ke dalam costmap atau tidak. Kita juga mengatur jarak maksimum dan minimum untuk menandai rintangan dalam costmap melalui obstacle_max_range dan obstacle_min_range masing-masing.

Untuk lapisan inflasi (baris 31-34 dan 67-70), kita mengatur faktor peluruhan eksponensial melalui radius inflasi menggunakan parameter cost_scaling_factor. Nilai radius untuk menginflasi di sekitar rintangan mematikan didefinisikan menggunakan inflation_radius.

Untuk lapisan voxel (baris 51-66), kita mengatur parameter publish_voxel_map ke True untuk mengaktifkan penerbitan grid voxel 3D. Resolusi voxel dalam tinggi didefinisikan menggunakan parameter z_resolution, sementara jumlah voxel dalam setiap kolom didefinisikan menggunakan parameter z_voxels. Parameter mark_threshold mengatur jumlah minimum voxel dalam kolom untuk ditandai sebagai terisi dalam grid okupansi. Kita mengatur parameter observation_sources dari lapisan voxel ke scan, dan kita mengatur parameter scan (di baris 61-66) mirip dengan parameter

yang telah kita diskusikan untuk lapisan rintangan. Seperti yang didefinisikan dalam parameter topik dan data_typenya, lapisan voxel akan menggunakan LaserScan yang diterbitkan pada topik /scan oleh pemindai lidar.

Perlu dicatat bahwa kita tidak menggunakan lapisan rentang untuk konfigurasi kita tetapi mungkin berguna untuk pengaturan robot Anda sendiri. Untuk lapisan rentang, parameter dasarnya adalah topik, input_sensor_type, dan parameter clear_on_max_r Topik rentang untuk di-subscribe didefinisikan dalam parameter topik. input_sensor_type diatur ke ALL, VARIABLE, atau FIXED. Parameter clear_on_max_reading adalah parameter boolean yang mengatur apakah akan menghapus pembacaan sensor pada jarak maksimum. Periksalah panduan konfigurasi di tautan di bawah ini jika Anda perlu mengatur ini.

Lihat juga

Untuk informasi lebih lanjut tentang nav2_costmap_2d dan daftar lengkap parameter plugin lapisan, lihat Panduan Konfigurasi Costmap 2D.

33 Membangun, Menjalankan, dan Verifikasi

Kami akan pertama meluncurkan display.launch.py yang meluncurkan penerbit status robot yang menyediakan transformasi base_link => sensors dalam URDF kita. Ini juga meluncurkan Gazebo yang bertindak sebagai simulator fisika kita dan juga menyediakan transformasi odom => base_link dari plugin penggerak diferensial, yang kita tambahkan ke diditbot dalam panduan sebelumnya, Simulating an Odometry System Using Gazebo. Ini juga meluncurkan RViz yang dapat kita gunakan untuk memvisualisasikan robot dan informasi sensor.

Kemudian kita akan meluncurkan slam_toolbox untuk menerbitkan ke topik /map dan menyediakan transformasi map => odom. Ingat bahwa transformasi map => odom adalah salah satu persyaratan utama dari sistem Nav2. Pesan yang diterbitkan pada topik /map kemudian akan digunakan oleh lapisan statis dari global_costmap.

Setelah kita mengatur deskripsi robot, sensor odometri, dan transformasi yang diperlukan dengan benar, kita akhirnya akan meluncurkan sistem Nav2 itu sendiri. Untuk saat ini, kita hanya akan menjelajahi sistem pembangkitan costmap dari Nav2. Setelah meluncurkan Nav2, kita akan memvisualisasikan costmap di RViz untuk mengkonfirma keluaran kita.

34 Meluncurkan Node Deskripsi, RViz, dan Gazebo

Mari kita sekarang meluncurkan Node Deskripsi Robot, RViz, dan Gazebo melalui file peluncuran display.launch.py. Buka terminal baru dan eksekusi baris-baris di bawah ini.

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

RViz dan Gazebo sekarang harus diluncurkan dengan diditbot hadir di keduanya. Ingat bahwa transformasi base_link => sensors sekarang diterbitkan oleh robot_state_publi dan transformasi odom => base_link oleh plugin Gazebo kita. Kedua transformasi ini sekarang harus ditampilkan tanpa kesalahan di RViz.

35 Meluncurkan slam_toolbox

Untuk dapat meluncurkan slam_toolbox, pastikan bahwa Anda telah menginstal paket slam_toolbox dengan menjalankan perintah berikut:

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

Kami akan meluncurkan async_slam_toolbox_node dari slam_toolbox menggunakan file peluncuran bawaan paket tersebut. Buka terminal baru dan eksekusi baris-baris di bawah ini:

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

slam_toolbox sekarang harus mempublikasikan ke topik /map dan menyediakan transformasi map => odom.

Kita bisa memverifikasi di RViz bahwa topik /map sedang diterbitkan. Di jendela RViz, klik tombol tambah di bagian bawah kiri kemudian pergi ke tab By topic lalu pilih Map di bawah topik /map. Anda seharusnya dapat memvisualisasikan pesan yang diterima di /map seperti yang ditunjukkan di bawah ini.

Kita juga bisa memeriksa bahwa transformasi tersebut benar dengan menjalankan baris berikut di terminal baru:

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

Catatan : Untuk Galactic dan yang lebih baru, seharusnya view_frames dan bukan view_frames.py. Baris di atas akan membuat file frames.pdf yang menunjukkan pohon transformasi saat ini. Pohon transformasi Anda harus mirip dengan yang ditunjukkan di bawah ini.

36 Meluncurkan Nav2

Pertama, pastikan Anda telah menginstal paket Nav2 dengan menjalankan perintah berikut:

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

Sekarang kita akan meluncurkan Nav2 menggunakan file peluncuran yang dibangun dalam nav2_bringup, navigation_launch.py. Buka terminal baru dan eksekusi baris berikut:

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

Perhatikan bahwa parameter nav2_costmap_2d yang kita diskusikan di subbagian sebelumnya termasuk dalam parameter default dari navigation_launch.py. Selain parameter dari nav2_costmap_2d, ini juga berisi parameter untuk node lain yang termasuk dalam implementasi Nav2.

Setelah kita mengatur dan meluncurkan Nav2 dengan benar, topik /global_costmap dan /local_costmap sekarang harus aktif.

Catatan

Untuk membuat costmap muncul, jalankan 3 perintah ini dalam urutan ini:

- Meluncurkan Node Deskripsi, RViz, dan Gazebo - dalam log tunggu untuk "Terkoneksi ke gazebo master"
- Meluncurkan slam_toolbox - dalam log tunggu untuk "Mendaftarkan sensor"
- Meluncurkan Nav2 - dalam log tunggu untuk "Membuat pengikat timer"

37 Memvisualisasikan Costmap di RViz

global_costmap, local_costmap, dan representasi voxel dari rintangan yang terdeteksi dapat divisualisasikan di RViz.

Untuk memvisualisasikan global_costmap di RViz, klik tombol tambah di bagian bawah kiri jendela RViz. Pergi ke tab By topic kemudian pilih Map di bawah topik /global_costmap/costmap. global_costmap harus muncul di jendela RViz, seperti yang ditunjukkan di bawah ini. global_costmap menunjukkan area yang harus dihindari (hitam) oleh robot kita saat menavigasi dunia simulasi kita di Gazebo.

Untuk memvisualisasikan local_costmap di RViz, pilih Map di bawah topik /local_costmap Atur skema warna di RViz ke costmap untuk membuatnya muncul mirip dengan gambar di bawah ini.

Untuk memvisualisasikan representasi voxel dari objek yang terdeteksi, buka terminal baru dan eksekusi baris berikut:

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

Baris di atas mengatur topik di mana marker akan diterbitkan ke /my$_{m}arker.Untukmeliha$

Kemudian atur kerangka tetap di RViz ke odom dan Anda seharusnya sekarang melihat voxel di RViz, yang mewakili kubus dan bola yang kita miliki di dunia Gazebo.

38 Kesimpulan

Dalam bagian panduan pengaturan robot kita ini, kita telah membahas pentingnya informasi sensor untuk berbagai tugas yang terkait dengan Nav2. Lebih spesifik lagi, tugas-tugas seperti pemetaan (SLAM), pelokalan (AMCL), dan tugas persepsi (costmap).

Kita juga membahas tipe-tipe umum pesan sensor dalam Nav2 yang menstandarkan format pesan untuk berbagai vendor sensor. Kita juga mendiskusikan cara menambahkan sensor pada robot simulasi menggunakan Gazebo dan bagaimana memverifikasi bahwa sensor berfungsi dengan benar melalui RViz.

Terakhir, kita mengatur konfigurasi dasar untuk paket nav2_costmap_2d menggunakan berbagai lapisan untuk memproduksi costmap global dan lokal. Kita kemudian memverifikas pekerjaan kita dengan memvisualisasikan costmap ini di RViz.