.1

در این سوال به بدلیل استفاده از کتابخانه آماده amcl برای particle فیلتر برای داکیومنتیشنش فقط به توضیح لانچ فایل و عکس ها میپردازیم.
لانچ فایل ما برای localization بدین اجزا است:

اجرای شبیه‌سازی ربات در **Gazebo**

بارگذاری مدل ربات (URDF)

تخمین موقعیت ربات روی نقشه از پیش ساخته‌شده با **AMCL**

تولید داده‌ی اسکن لیزری از PointCloud

تخمین ادومتری بصری با **RTAB-Map**

یکپارچه‌سازی فریم‌ها و زمان شبیه‌سازی

نمایش همه‌چیز در RViz

```python
def generate_launch_description():
    bringup_dir = get_package_share_directory('robot_description')
    map_yaml_file = os.path.join(bringup_dir, 'maps', 'map.yaml')
    amcl_config_file = os.path.join(bringup_dir, 'config', 'amcl.yaml')
    world_file = os.path.join(bringup_dir, 'world', 'depot.sdf')
    urdf_file = os.path.join(bringup_dir, 'src', 'description', 'robot.urdf')
    rviz_config_file = os.path.join(bringup_dir, 'rviz', 'config.rviz')
    gz_bridge_config = os.path.join(bringup_dir, 'config', 'gz_bridge.yaml')
    vo_config_file = os.path.join(bringup_dir, 'config', 'rtabmap_slam.yaml')
```

در ابتدای تابع **`generate_launch_description`** مسیر فایل‌های مهم پروژه مشخص می‌شود:

- فایل نقشه (map.yaml)
- فایل تنظیمات AMCL
- فایل world شبیه‌سازی (depot.sdf)
- فایل مدل ربات (robot.urdf)
- فایل تنظیمات RViz
- تنظیمات bridge بین Gazebo و ROS
- تنظیمات Visual Odometry

سپس محتوای فایل URDF خوانده می‌شود تا به **`robot_state_publisher`** داده شود.

نود use_sim_time برای این است که همه ی نود ها از زمان شبیه سازی گزبو استفاده کنند.

و map هم مسیری است که نود های map_server و amcl برای تبدیل به odometry استفاده میکنند.

و GZ_SIM_RESOURCE_PATH برای این است که گزبو تنظیمات فایل های محیطی را مانند فایل ربات و world را پیدا کند.

ROBOT STATE PUBLISHER

مدل ربات را از URDF منتشر می‌کند.

تبدیل‌های TF بین لینک‌های ربات را تولید می‌کند.

از زمان شبیه‌سازی استفاده می‌کند.

این نود پایه‌ی کل سیستم TF است.

gz_sim

شبیه‌سازی Gazebo با world مشخص‌شده اجرا می‌شود

```
use_sim_time_arg = DeclareLaunchArgument(
    'use_sim_time',
    default_value='true',
    description='Use simulation (Gazebo) clock if true'
)

map_yaml_arg = DeclareLaunchArgument(
    'map',
    default_value=map_yaml_file,
    description='Full path to map yaml file to load'
)

gz_resource_path = SetEnvironmentVariable(
    name='GZ_SIM_RESOURCE_PATH',
    value=':'.join([
        os.path.join(bringup_dir, 'world'),
        str(Path(bringup_dir).parent.resolve())
    ])
)

robot_state_publisher = Node(
    package='robot_state_publisher',
    executable='robot_state_publisher',
    name='robot_state_publisher',
    output='screen',
    parameters=[
        {'use_sim_time': LaunchConfiguration('use_sim_time')},
        {'robot_description': robot_desc}
    ]
)

gz_sim = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(
        [
            os.path.join(
                get_package_share_directory("ros_gz_sim"),
                "launch",
                "gz_sim.launch.py",
            )
        ]
    ),
    launch_arguments={"gz_args": ["-r -v 4 ", world_file]}.items(),
)
```

Bridge بین Gazebo و ROS 2

Bridge Topicها

`ros_gz_bridge parameter_bridge`

این نود:

- داده‌های سنسورها (لیدار، TF و غیره) را بین Gazebo و ROS منتقل می‌کند

انجام شده تا پایدار باشد `/tf_static` برای QoS تنظیم ●

Bridge سرویس کنترل

برای کنترل world در Gazebo از طریق ROS استفاده می‌شود.

اسپاون ربات در Gazebo

`spawn_entity`

این نود:

- ربات را با استفاده از `robot_description/` در محیط شبیه‌سازی قرار می‌دهد
- موقعیت اولیه ربات را تنظیم می‌کند

اجرای RViz

`rviz2`

- محیط گرافیکی برای مشاهده:
  - نقشه
  - موقعیت ربات
  - TFها
  - داده لیدار و ادومتری

نودهای کمکی پردازش فریم و سنسورها

تبدیل Frame ID

`frame_id_converter_node`

برای یکسان‌سازی یا اصلاح نام فریم‌ها بین Gazebo و ROS استفاده می‌شود.

EKF برای ادومتری دیفرانسیلی و IMU

`ekf_diff_imu_node`

- داده‌های حرکتی و IMU را ادغام می‌کند
- خروجی ادومتری پایدارتر برای localization فراهم می‌کند

تبدیل PointCloud به LaserScan

`pointcloud_to_laserscan`

از آنجا که AMCL به داده‌ی LaserScan نیاز دارد:

- داده‌های PointCloud لیدار سه‌بعدی Gazebo
- به اسکن دوبعدی تبدیل می‌شوند
- خروجی روی `/scan_pc` منتشر می‌شود.

بدلیل مشکل اینترنت مجبور شدم این پروژه رو کلون کنم و

Map Server (نود چرخه عمر)

`nav2_map_server`

- نقشه‌ی از پیش ساخته‌شده را بارگذاری می‌کند
- به‌صورت LifecycleNode اجرا می‌شود
- نقشه را روی topic مربوطه منتشر می‌کند

AMCL (Adaptive Monte Carlo Localization)

`nav2_amcl`

این نود:

- موقعیت و جهت ربات را روی نقشه تخمین می‌زند
- از داده‌ی اسکن لیزری (`/scan_pc`) استفاده می‌کند
- فریم‌های TF را اصلاح و منتشر می‌کند

Lifecycle Manager

`lifecycle_manager_localization`

وظیفه این نود:

- مدیریت وضعیت نودهای lifecycle

فعال‌سازی خودکار:

`map_server`

`amcl`

بدون این نود، AMCL و Map Server شروع به کار نمی‌کنند.

RTAB-Map با Visual Odometry

`rtabmap_visual_odometry`

این نود:

- ادومتری بصری را با استفاده از تصویر RGB و Depth محاسبه می‌کند
- از دوربین ZED استفاده می‌کند
- TF منتشر نمی‌کند (برای جلوگیری از تداخل)
- خروجی ادومتری روی `/vo/odom` قرار می‌گیرد

اصلاح Transform استاتیک

`static_transform_publisher`

برای رفع ناسازگاری فریم لیدار در Gazebo:

- یک تبدیل استاتیک بین `base_link`
- و فریم سنسور لیدار منتشر می‌شود

```
bridge_topics = Node(
    package='ros_gz_bridge',
    executable='parameter_bridge',
    parameters=[{
        'config_file': gz_bridge_config,
        'qos_overrides./tf_static.publisher.durability': 'transient_local',
    }],
    output='screen'
)

bridge_service_control = Node(
    package='ros_gz_bridge',
    executable='parameter_bridge',
    arguments=[
        '/world/depot/control@ros_gz_interfaces/srv/ControlWorld'
    ],
    output='screen'
)

spawn_entity = Node(
    package="ros_gz_sim",
    executable="create",
    arguments=[
        "-name", "robot",
        "-topic", "/robot_description",
        "-x", "0",
        "-y", "0",
        "-z", "0.9",
    ],
    output="screen",
)

rviz_node = Node(
    package='rviz2',
    executable='rviz2',
    name='rviz2',
    arguments=['-d', rviz_config_file],
    output='screen'
)

frame_id_converter_node = Node(
    package='robot_description',
    executable='frame_id_converter_node',
    name='frame_id_converter_node',
    output='screen',
    parameters=[{'use_sim_time': LaunchConfiguration('use_sim_time')}]
)
```

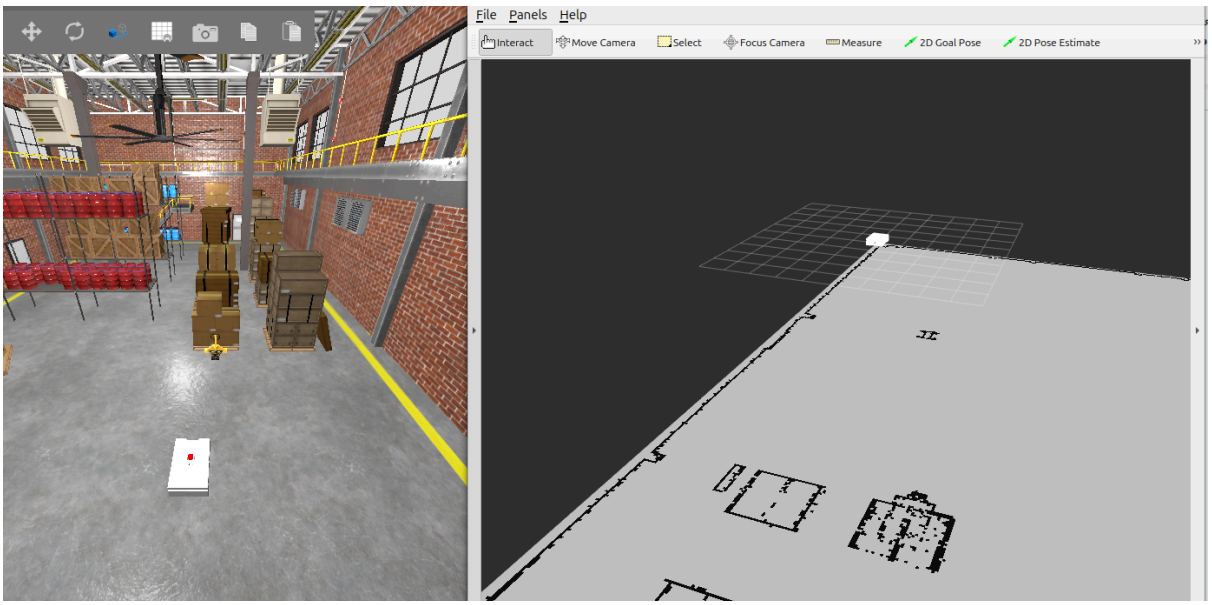
ترتیب اجرای نودها

در انتها، تمام نودها داخل `LaunchDescription` قرار گرفته‌اند تا:

به‌صورت هماهنگ و با زمان شبیه‌سازی و بدون تداخل TF
اجرا شوند.

همچنین کانفیگ amcl

```yaml
amcl:
  ros__parameters:
    use_sim_time: true
    scan_topic: "/scan_pc"

    # Frames
    global_frame_id: "map"
    odom_frame_id: "odom"
    base_frame_id: "base_link"

    # AMCL behavior
    tf_broadcast: true
    transform_tolerance: 0.5

    # Particle filter
    min_particles: 500
    max_particles: 2000
    kld_err: 0.05
    kld_z: 0.99

    # Motion model (tune these)
    alpha1: 0.2
    alpha2: 0.2
    alpha3: 0.2
    alpha4: 0.2
    alpha5: 0.2

    # Laser model
    laser_model_type: "likelihood_field"
    z_hit: 0.5
    z_short: 0.05
    z_max: 0.05
    z_rand: 0.5
    sigma_hit: 0.2
    laser_likelihood_max_dist: 2.0

    # Update thresholds
    update_min_d: 0.25
    update_min_a: 0.2

    # Resampling
    resample_interval: 1
```



.2

در اینجا یک نود در اسکریپت ها به نام `astar_planner` پیاده‌سازی می‌کند که هدف آن **پیدا کردن مسیر بهینه و امن (بدون برخورد)** بین موقعیت فعلی ربات و یک هدف مشخص روی نقشه‌ی دوبعدی است.

این نود از **نقشه‌ی اشغال (Occupancy Grid)** که توسط `map_server` منتشر می‌شود استفاده می‌کند و با الگوریتم Astar مسیر را محاسبه می‌کند.

ورودی‌ها و خروجی‌های نود

ورودی‌ها (Subscriptions)

1. **/map** از نوع `nav_msgs/OccupancyGrid`
   این پیام شامل:

- رزولوشن (اندازه هر سلول به متر)
- ابعاد نقشه (width, height)
- مختصات مبدا نقشه (origin)

- آرایه data که مقدار اشغال هر سلول را نگه می‌دارد

2. **amcl_pose/** از نوع `geometry_msgs/PoseWithCovarianceStamped`
   این پیام موقعیت تخمینی ربات را در فریم `map` ارائه می‌دهد.

اگر AMCL هنوز فعال نشده باشد، نود از TF برای گرفتن تبدیل `map -> base_link` استفاده می‌کند.

ورودی سرویس (Service Request)

سرویس `plan_path` از نوع `robot_description/srv/PlanPath` یک هدف از نوع `geometry_msgs/PoseStamped` دریافت می‌کند.

یعنی کلاینت می‌گوید:هدف ربات در فریم map اینجاست.

خروجی‌ها

1. **پاسخ سرویس:** `nav_msgs/Path`
   مسیر نهایی به صورت لیستی از Poseها برگردانده می‌شود.

2. **publish /global_path** و از نوع `nav_msgs/Path`
   همان مسیر برای نمایش در RViz منتشر می‌شود.

تبدیل مختصات (World ,Grid)

نقشه OccupancyGrid یک شبکه‌ی سلولی است، ولی ربات و هدف در مختصات متری (world) هستند.

تبدیل world به grid

برای تبدیل نقطه متری `(wx, wy)` به خانه `(gx, gy)`:

- ابتدا نسبت به مبدا نقشه offset می‌گیریم:
  `(wx - origin_x , wy - origin_y)`

- بعد تقسیم بر رزولوشن:

```
gx = floor((wx - origin_x) / resolution
gy = floor((wy - origin_y) / resolution
```

اگر خانه خارج از محدوده نقشه باشد، برنامه‌ریزی مسیر انجام نمی‌شود.

تبدیل grid به world

برای خروجی مسیر و نمایش، مرکز هر سلول را به مختصات متری برمی‌گردانیم:

```
wx = origin_x + (gx + 0.5) * resolution
wy = origin_y + (gy + 0.5) * resolution
```

تشخیص برخورد و ایمنی مسیر (Collision-Free)

هر سلول در داده‌ی نقشه می‌تواند:

0 آزاد

100 اشغال

-1 ناشناخته

در این نود از پارامتر occupied_threshold استفاده می‌شود (مثلاً 50):

- اگر مقدار سلول کمتر مساوی 50 باشد یعنی اشغال است

رفتار با سلول‌های ناشناخته

پارامتر allow_unknown:

- اگر True باشد، سلول‌های ناشناخته (-1) آزاد در نظر گرفته می‌شوند
- اگر False باشد، ناشناخته‌ها اشغال در نظر گرفته می‌شوند

افزایش ایمنی با **Inflation** (بافر اطراف مانع‌ها)

اگر فقط سلول‌های اشغال را حذف کنیم، مسیر ممکن است خیلی نزدیک دیوار/مانع عبور کند و در واقعیت برخورد کند.

برای همین یک پارامتر inflation_radius (متر) داریم:

- این شعاع به تعداد سلول تبدیل می‌شود:

```
(inflation_cells = ceil(inflation_radius / resolution
```

وقتی می‌خواهیم یک سلول را آزاد حساب کنیم، بررسی می‌کنیم که آیا همه سلول‌های اطراف آن در این محدوده هم آزاد هستند؟
به این شکل مسیر فاصله‌ی امن از موانع دارد.

ساخت گراف و همسایه‌ها (Connected Grid-8)

ما گراف جداگانه نمی‌سازیم؛ خود نقشه گراف است:

- هر سلول آزاد یک نود است
- همسایه‌ها در 8 جهت بررسی می‌شوند (بالا، پایین، چپ، راست، و قطرها)

جلوگیری از Corner Cutting

در حرکت قطری، اگر دو سلول کناری بسته باشند، حرکت قطری ممنوع می‌شود تا از رد شدن غیرواقعی بین گوشه‌ی دو مانع جلوگیری شود.

الگوریتم A* چگونه اجرا می‌شود؟

A* برای هر سلول سه مقدار اصلی دارد:

1) g(n)

هزینه طی شده از شروع تا این سلول

هزینه حرکت:

- حرکت مستقیم = 1
- حرکت قطری = √2

2) h(n)

تخمین فاصله تا هدف (Heuristic)

در این پروژه از فاصله اقلیدسی استفاده می‌شود:

h = hypot(dx, dy)

3) f(n)

```
f(n) = g(n) + h(n)
```

و این معیار انتخاب بهترین نود برای گسترش است.

ساختار داده‌ها

Open Set: صف اولویت‌دار بر اساس کمترین f.

Closed Set: نودهایی که بررسی و گسترش داده شده‌اند.

g_score: بهترین هزینه رسیدن به هر سلول.
came_from: برای اینکه بعداً مسیر را از هدف به شروع بازسازی کنیم .

بازسازی مسیر (Path Reconstruction)

وقتی به هدف رسیدیم:

- از هدف شروع می‌کنیم و با came_from به عقب برمی‌گردیم تا به start برسیم.
- لیست را reverse می‌کنیم تا مسیر از start به goal باشد.
- هر سلول به PoseStamped تبدیل می‌شود و در Path .

در آخر یک service در پوشه srv برای این درست کردیم تا سرویس تشکیل شده و goal مشخص شود و سپس پیدا کردن path در شبیه سازی شروع میشود.

برای لانچ کردن نیز تنها کانفیگ زیر را با توجه به ابعاد و پارامتر هایی که وجود دارد اضافه کردم:

Bonus Question

PID

هدف

نود pid_path_follower وظیفه دارد ربات را طوری کنترل کند که مسیر منتشر شده روی /global_path (از نوع nav_msgs/Path) را دنبال کند و با تولید پیام geometry_msgs/Twist روی /cmd_vel حرکت ربات را تنظیم کند.

ورودی‌ها و خروجی‌ها
ورودی (Subscribe)

/global_path از نوع nav_msgs/Path
شامل لیستی از waypointها (PoseStamped) در فریم map است.

TF: تبدیل map -> base_link
برای به‌دست آوردن موقعیت و زاویه فعلی ربات (بدون نیاز مستقیم به odom).

خروجی (Publish)

/cmd_vel از نوع geometry_msgs/Twist
شامل سرعت خطی linear.x و سرعت زاویه‌ای angular.z.

ایده کلی کنترل PID در این نود

این کنترل‌کننده از یک روش رایج در دنبال‌کردن مسیر استفاده می‌کند:

1. ربات ابتدا نزدیک‌ترین نقطه مسیر به خودش را پیدا می‌کند.
2. سپس یک نقطه‌ی هدف جلوتر از ربات (Lookahead) روی مسیر انتخاب می‌شود.
3. اختلاف زاویه بین جهت حرکت ربات و جهت نقطه‌ی هدف محاسبه می‌شود.
4. یک PID روی **خطای زاویه‌ای** اعمال می‌شود تا سرعت زاویه‌ای **ω** ساخته شود.
5. سرعت خطی **v** با توجه به میزان خطا کم یا زیاد می‌شود (Speed scheduling) تا در پیچ‌ها ربات آرام‌تر حرکت کند و نوسان کمتر شود.

انتخاب نقطه هدف (Lookahead Point)

پارامتر `lookahead_distance` مشخص می‌کند نقطه هدف باید چقدر جلوتر از ربات باشد.
هدف این است که ربات به جای دنبال‌کردن دقیق نقطه‌ی نزدیک، به سمت یک نقطه جلوتر حرکت کند تا حرکت نرم‌تر و پایدارتر شود.

محاسبه خطای زاویه‌ای (Heading Error)

ابتدا زاویه ربات از کواترنیون استخراج می‌شود robot_yaw

سپس زاویه هدف نسبت به موقعیت ربات:

```
target_yaw = atan2(dy, dx)
```

خطای زاویه‌ای:

```
angular_error = target_yaw - robot_yaw
```

این خطا در بازه **[-π, +π]** نرمال می‌شود تا پرش زاویه‌ای ایجاد نشود.

قانون PID برای کنترل زاویه

کنترل PID روی خطای زاویه‌ای انجام می‌شود که فرمول آن بدین صورت است:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}$$

تنظیم سرعت خطی (Speed Scheduling)

برای کم کردن نوسان در پیچ‌ها، سرعت خطی ثابت نیست.
اگر خطای زاویه‌ای زیاد باشد، سرعت خطی کاهش می‌یابد:

- وقتی |خطا| زیاد است ← سرعت کمتر
- وقتی |خطا| کم است ← سرعت بیشتر

این باعث می‌شود ربات در مسیرهای منحنی کنترل‌پذیرتر حرکت کند.

پارامترهای مهم PID

- ang_kp, ang_ki, ang_kd: ضرایب PID

- `lookahead_distance`: فاصله نقطه هدف
- `goal_tolerance`: شرط توقف
- `max_linear_vel, max_angular_vel`: محدودیت سرعت‌ها
- `v_nominal, v_min`: تنظیم سرعت خطی
- `control_frequency`: نرخ کنترل

MPC

هدف

نود `mpc_path_follower` نیز مانند PID مسیر `/global_path` را دنبال می‌کند، اما به جای کنترل لحظه‌ای با PID، از **کنترل پیش‌بین مدل (MPC)** استفاده می‌کند تا عملکرد بهتر در پیچ‌ها و حرکت نرم‌تر داشته باشد.

در این پروژه از نسخه ساده و قابل‌اجرا در پروژه‌های دانشجویی استفاده می‌شود:
Sampling MPC (بدون نیاز به QP Solver)

ورودی‌ها و خروجی‌ها

ورودی (Subscribe)

`/global_path` از نوع `nav_msgs/Path`
TF برای `map -> base_link`
خروجی (Publish)

`/cmd_vel` از نوع `geometry_msgs/Twist`

ایده اصلی MPC چیست؟

در MPC، کنترل‌کننده فقط به خطای لحظه‌ای نگاه نمی‌کند.
بلکه برای چند قدم آینده پیش‌بینی می‌کند:

1. مجموعه‌ای از ورودی‌های ممکن (v, ω) را بررسی می‌کند.
2. برای هر ورودی، حرکت ربات را برای یک افق زمانی (Horizon) شبیه‌سازی می‌کند.
3. برای هر شبیه‌سازی یک "هزینه" (Cost) حساب می‌کند.

4. ورودی‌ای را انتخاب می‌کند که کمترین هزینه را داشته باشد.
5. فقط همان ورودی مرحله اول اجرا می‌شود و در سیکل بعد دوباره محاسبه می‌شود.

این بخش برای MPC کامل نشده و در launch file هم دیفالت بر روی همان PID است.

عکس ها و کد ها داخل گیت هاب قرار داده شده اند.

https://github.com/ampardra/robotic_finale/tree/main