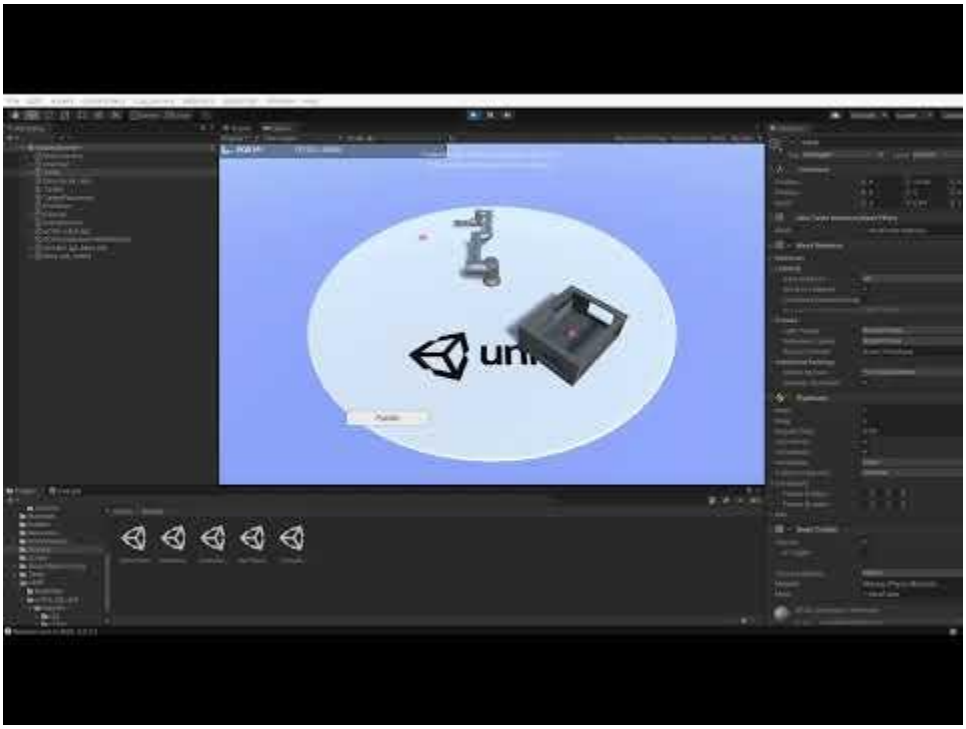
Demo without obstacles

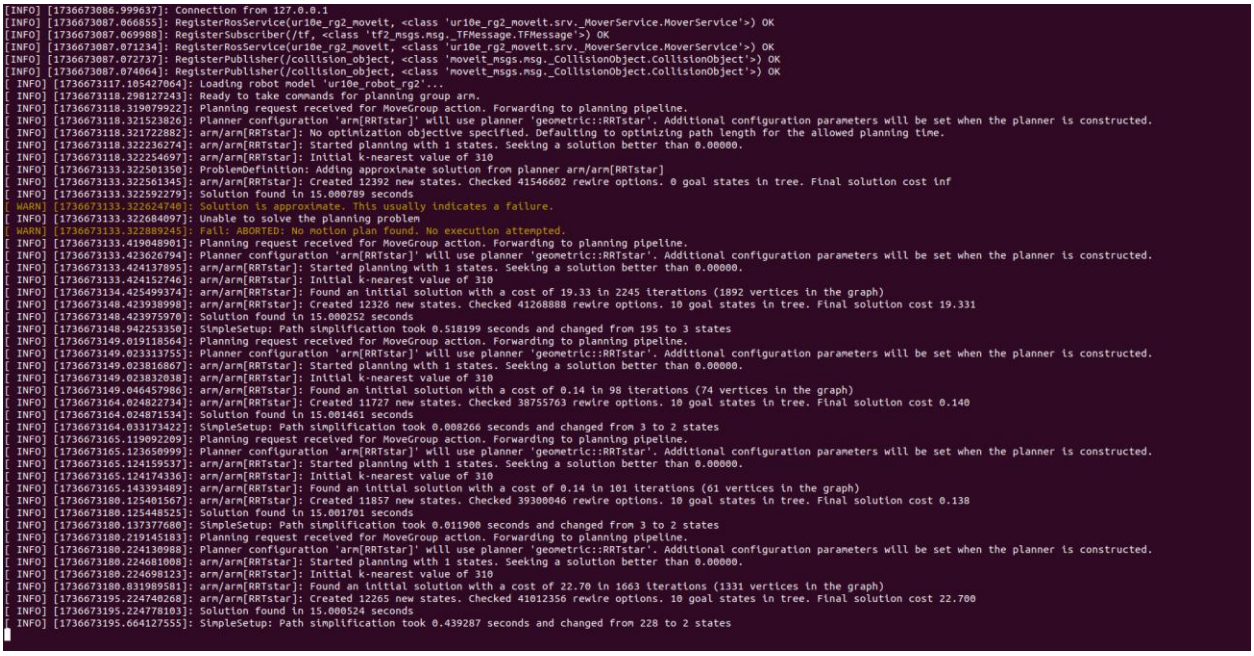

```
[INFO] [1736672501.785613]: Connection from 127.0.0.1
[INFO] [1736672501.814812]: RegisterRosService(ur10e_rg2_moveit, <class 'ur10e_rg2_moveit.srv._MoverService.MoverService'>) OK
[INFO] [1736672501.817156]: RegisterSubscriber(/tf, <class 'tf2_msgs.msg._TFMessage.TFMessage'>) OK
[INFO] [1736672501.818118]: RegisterRosService(ur10e_rg2_moveit, <class 'ur10e_rg2_moveit.srv._MoverService.MoverService'>) OK
[INFO] [1736672501.819556]: RegisterPublisher(/collision_object, <class 'moveit_msgs.msg._CollisionObject.CollisionObject'>) OK
[INFO] [1736672501.820983]: RegisterPublisher(/collision_object, <class 'moveit_msgs.msg._CollisionObject.CollisionObject'>) OK
[ INFO] [1736672513.612483792]: Loading robot model 'ur10e_robot_rg2'...
[ INFO] [1736672514.720419780]: Ready to take commands for planning group arm.
[ INFO] [1736672514.813787975]: Planning request received for MoveGroup action. Forwarding to planning pipeline.
[ INFO] [1736672514.818005136]: Planner configuration 'arm[RRTstar]' will use planner 'geometric::RRTstar'. Additional configuration parameters will be set when the planner is constructed.
[ INFO] [1736672514.818503924]: arm/arm[RRTstar]: Started planning with 1 states. Seeking a solution better than 0.00000.
[ INFO] [1736672514.818517004]: arm/arm[RRTstar]: Initial k-nearest value of 310
[ INFO] [1736672515.858362231]: arm/arm[RRTstar]: Found an initial solution with a cost of 19.09 in 2227 iterations (1865 vertices in the graph)
[ INFO] [1736672529.819002937]: arm/arm[RRTstar]: Created 11536 new states. Checked 37957256 rewire options. 10 goal states in tree. Final solution cost 19.093
[ INFO] [1736672529.819042820]: Solution found in 15.000941 seconds
[ INFO] [1736672530.164315855]: SimpleSetup: Path simplification took 0.345215 seconds and changed from 192 to 2 states
[ INFO] [1736672530.213954820]: Planning request received for MoveGroup action. Forwarding to planning pipeline.
[ INFO] [1736672530.218275848]: Planner configuration 'arm[RRTstar]' will use planner 'geometric::RRTstar'. Additional configuration parameters will be set when the planner is constructed.
[ INFO] [1736672530.218775408]: arm/arm[RRTstar]: Started planning with 1 states. Seeking a solution better than 0.00000.
[ INFO] [1736672530.218789158]: arm/arm[RRTstar]: Initial k-nearest value of 310
[ INFO] [1736672530.258137794]: arm/arm[RRTstar]: Found an initial solution with a cost of 0.13 in 116 iterations (115 vertices in the graph)
[ INFO] [1736672545.218475599]: arm/arm[RRTstar]: Created 11022 new states. Checked 35815474 rewire options. 10 goal states in tree. Final solution cost 0.129
[ INFO] [1736672545.218523864]: Solution found in 15.000154 seconds
[ INFO] [1736672545.226414991]: SimpleSetup: Path simplification took 0.007846 seconds and changed from 3 to 2 states
[ INFO] [1736672546.313881223]: Planning request received for MoveGroup action. Forwarding to planning pipeline.
[ INFO] [1736672546.317604368]: Planner configuration 'arm[RRTstar]' will use planner 'geometric::RRTstar'. Additional configuration parameters will be set when the planner is constructed.
[ INFO] [1736672546.318086833]: arm/arm[RRTstar]: Started planning with 1 states. Seeking a solution better than 0.00000.
[ INFO] [1736672546.318100150]: arm/arm[RRTstar]: Initial k-nearest value of 310
[ INFO] [1736672546.331416566]: arm/arm[RRTstar]: Found an initial solution with a cost of 0.13 in 52 iterations (46 vertices in the graph)
[ INFO] [1736672561.318038895]: arm/arm[RRTstar]: Created 12185 new states. Checked 40676124 rewire options. 10 goal states in tree. Final solution cost 0.128
[ INFO] [1736672561.318074224]: Solution found in 15.000382 seconds
[ INFO] [1736672561.325674311]: SimpleSetup: Path simplification took 0.007559 seconds and changed from 3 to 2 states
[ INFO] [1736672561.413885814]: Planning request received for MoveGroup action. Forwarding to planning pipeline.
[ INFO] [1736672561.418337537]: Planner configuration 'arm[RRTstar]' will use planner 'geometric::RRTstar'. Additional configuration parameters will be set when the planner is constructed.
[ INFO] [1736672561.418826236]: arm/arm[RRTstar]: Started planning with 1 states. Seeking a solution better than 0.00000.
[ INFO] [1736672561.418841316]: arm/arm[RRTstar]: Initial k-nearest value of 310
[ INFO] [1736672561.926996083]: arm/arm[RRTstar]: Found an initial solution with a cost of 22.49 in 1185 iterations (1080 vertices in the graph)
[ INFO] [1736672576.418687047]: arm/arm[RRTstar]: Created 11897 new states. Checked 39467646 rewire options. 10 goal states in tree. Final solution cost 22.481
[ INFO] [1736672576.418719909]: Solution found in 15.000295 seconds
[ INFO] [1736672576.730886640]: SimpleSetup: Path simplification took 0.312117 seconds and changed from 226 to 2 states
```

- In this scene we use the RRT* planner which is a single query-planner in Geometric Planner of OMPL
- Despite of adjust many things like reduce the velocity, the speed of the arm as much as possible or increase the friction of the target and the gripper, the arm just cannot hold to target and put it into the target placement normally but we have calculate the path to control the arm to move from the start position to the target position and grab it to move to the target placement and drop.

- Because using RRT* planner the algorithm minimizes the cost of the path, which can be defined by distance, energy, or other metrics. Although, the planning time of this is increase significantly compared to other planner so this affect the planning time as you can see in the demo.
- Look in to the above log, at the beginning the planner calculate the path that cost a lot of time to execute and iterations but after recalculate many time it reduce to execution time and iterations that mean it can find the most optimal solution but it takes lots of time to execute.

Demo with obstacles (printer)





- Same as the above the demo this demo we use the same RRT* planner but this time we add the printer into the scene and put the target inside the printer.
- Despite of adjust many physical factors, the arm just cannot hold to target and put it into the target placement normally same as above demo but we have calculate the path to control the arm to move from the start position to the target position in the printer and grab it to move to the target placement and drop.
- As you can see the log, the planner has do it things same as above the find the most optimal solution but still cost a lot of planning time.

Explain about the algorithm

OMPL, the Open Motion Planning Library, consists of many state-of-the-art sampling-based motion planning algorithms. OMPL itself does not contain any code related to, e.g., collision checking or visualization. This is a deliberate design choice, so that OMPL is not tied to a particular collision checker or visualization front end. The library is designed so it can be easily integrated into systems that provide the additional needed components.

Overview

- **Purpose**: OMPL focuses on sampling-based motion planning algorithms. It is used to determine feasible paths for a robot or object from a start state to a goal state while avoiding obstacles.
- **Language**: Written in C++, with Python bindings available.
- **Platform**: Cross-platform, supported on Linux, macOS, and Windows.
- **Integration**: Can integrate with other tools such as ROS (Robot Operating System) for enhanced functionality.

Core Features

1. **Algorithms**:
    - Includes a variety of motion planning algorithms like RRT, RRT*, PRM, PRM*, and more.
    - Designed to be extensible, allowing users to add custom algorithms.
2. **Flexible State Space**:
    - Provides abstract definitions of state spaces, making it adaptable to different robotic systems.
    - Supports custom state spaces to model unique problem domains.
3. **Collision Checking**:
    - OMPL separates planning from collision checking, allowing users to pair it with custom or third-party collision detection libraries.
4. **Visualization**:
    - Can visualize planning problems and solutions using tools like RViz in ROS.
5. **Benchmarking**:
    - Built-in benchmarking tools for comparing the performance of different planners.
6. **Documentation and Examples**:
    - Comes with comprehensive documentation and tutorials to help new users get started.

Applications

- Robotics: Pathfinding for robotic arms, mobile robots, and drones.
- Autonomous Vehicles: Planning safe paths in dynamic environments.
- Animation: Generating smooth motion paths for animated characters.
- Games: Implementing pathfinding and AI navigation systems.

Library Contents

- OMPL contains implementations of many sampling-based algorithms such as PRM, RRT, EST, SBL, KPIECE, SyCLOP, and several variants of these planners.
- All these planners operate on very abstractly defined state spaces. Many commonly used state spaces are already implemented (e.g., SE(2), SE(3), R^n, etc.).
- For any state space, different state samplers can be used (e.g., uniform, Gaussian, obstacle-based, etc.).

Available Planners

All implementations listed below are considered fully functional. Within OMPL planners are divided into three categories:

- Geometric planners
- Control-based planners
- Multilevel-based planners

Geometric planners

Planners in this category only accounts for the geometric and kinematic constraints of the system. It is assumed that any feasible path can be turned into a dynamically feasible trajectory. Any of these planners can be used to plan with geometric constraints. Planners in this category can be divided into several overlapping subcategories:

- Multi-query planners
- Single-query planners
- Multi-level planners
- Optimizing planners

Control-based planners

If the system under consideration is subject to differential constraints, then a control-based planner is used. These planners rely on state propagation rather than simple interpolation to generate motions. These planners do not require a steering function, but all of them (except

KPIECE) will use it if the user implements it. The first two planners below are kinodynamic adaptations of the corresponding geometric planners above.

Multilevel-based planners

To solve problems involving high-dimensional state spaces, we often can use multilevel abstractions to simplify the state spaces, thereby allowing dedicated planner to quicker find solutions. The planner in this class support sequences of state spaces and can be utilized both for state spaces with geometric and dynamic constraints.

In this project, we use the RRT* and RRT Connect planner which is a single-query planner in Geometric Planners. We use these 2 planners alternately to compare which one is more optimal so we are going to through briefly about these 2 planners.

RRT* in OMPL

The **Rapidly-exploring Random Tree Star (RRT*)** is an optimized variant of the RRT algorithm. It aims to find a path and improve it over time to converge towards the optimal solution, considering path cost. OMPL provides a robust implementation of RRT*, allowing users to integrate it easily into their motion planning tasks.

Key Features of RRT* in OMPL

1. **Optimality**: Unlike RRT, RRT* improves the quality of the solution as the planning time increases.
2. **Path Cost Minimization**: The algorithm minimizes the cost of the path, which can be defined by distance, energy, or other metrics.
3. **Asymptotic Convergence**: Given enough time, RRT* converges to the optimal solution.
4. **Custom Cost Functions**: OMPL allows defining cost functions to suit specific planning problems.

How RRT* Works

RRT* builds upon the original RRT algorithm with additional steps to improve the quality of the solution. Here's a clearer breakdown:

---

1. **Random Sampling:**

- The algorithm generates random states (nodes) within the configuration space. These are potential points that the planner might connect to the tree.

2. **Nearest Neighbor:**
    - For each sampled random state, the algorithm identifies the nearest node in the existing tree.
3. **Steering:**
    - A new node is created by "steering" from the nearest node towards the sampled state. This ensures the new node is within a specified step size or distance limit.
4. **Collision Check:**
    - The new node and the path connecting it to the nearest neighbor are checked for validity (e.g., ensuring they don't collide with obstacles).
5. **Tree Expansion:**
    - If the new node is valid, it is added to the tree.
6. **Rewiring (Key Feature of RRT*):**
    - After adding the new node, the algorithm checks its neighboring nodes (within a specified radius) to see if connecting to the new node reduces their cost to the root (start node).
    - If a better path is found, the neighboring nodes are "rewired" to connect through the new node. This step ensures the tree optimizes path cost over time.
7. **Goal Bias:**
    - To ensure the algorithm progresses towards the goal, some samples are biased to be closer to the goal. This helps accelerate convergence.
8. **Path Refinement:**
    - Over multiple iterations, the tree grows and rewires itself, gradually improving the quality of the path to minimize the total cost (e.g., distance, time, or energy).
9. **Termination:**
    - The algorithm runs until a specified time limit, a certain number of iterations, or an acceptable path cost is achieved.

Key Enhancements in RRT* Compared to RRT

- **Rewiring:** This allows the tree to "fix" suboptimal connections and continually improve the path.

- **Optimality**: RRT* is an asymptotically optimal planner, meaning the solution approaches the best possible path as time goes on.
- **Path Cost Evaluation**: Uses cost functions to prioritize better solutions.

Illustrative Analogy

Imagine exploring a forest (the configuration space):

1. Start at the edge of the forest (the root node).
2. Randomly pick a direction (sampling) and walk a few steps (steering).
3. If there's a clear path (validity check), mark that point on your map (add to the tree).
4. Periodically check if you can take a shorter path back to the start (rewiring).
5. Gradually work your way towards a specific destination (goal bias).
6. Over time, refine your map to find the shortest, clearest trail to the destination.

This iterative process ensures both exploration and optimization, leading to a better solution with time.

Advantages of RRT* in OMPL

- **Scalability**: Works well for high-dimensional problems.
- **Customizability**: Supports cost functions and constraints.
- **Integration**: Seamlessly integrates with OMPL's benchmarking and visualization tools.

RRT-Connect in OMPL

The **Rapidly-exploring Random Tree Connect (RRT-Connect)** algorithm is a variant of RRT designed to improve planning efficiency by using two trees: one grows from the start state and the other from the goal state. These trees attempt to connect with each other, significantly speeding up the process of finding a feasible path in large or high-dimensional spaces.

Key Features of RRT-Connect

1. **Bidirectional Planning**:
    - Grows two trees simultaneously: one from the start state and another from the goal state.
    - Trees "meet" in the configuration space to establish a connection.

2. **Efficient Tree Expansion**:
    - Each tree grows toward the sampled state until it cannot proceed further, making the exploration faster.
3. **Collision Checking**:
    - Ensures that both trees only grow along valid paths, avoiding obstacles.
4. **Goal Connection**:
    - Once the trees connect, the solution path is immediately found by combining the paths from both trees.
5. **Faster Convergence**:
    - Compared to standard RRT, the bidirectional approach makes RRT-Connect faster for finding a solution in large configuration spaces.

How RRT-Connect Works

RRT-Connect is a bidirectional motion planning algorithm that grows two trees—one from the start state and the other from the goal state—and attempts to connect them. Here's a clear breakdown of its working process:

---

1. **Tree Initialization:**
    - Two trees are created:
        - Start Tree: Starts from the initial position.
        - Goal Tree: Starts from the target position.
2. **Random Sampling:**
    - A random state is sampled from the configuration space. This state serves as a guide for the trees to expand toward unexplored areas.
3. **Tree Growth:**
    - Step 1: Grow Start Tree:
        - Identify the nearest node in the start tree to the sampled state.
        - "Steer" the tree by extending a branch toward the sampled state until:
            - It reaches the sampled state, or
            - It encounters an obstacle.
        - Add the new node to the start tree.

- Step 2: Attempt to Connect Goal Tree:
    - From the new node in the start tree, the goal tree attempts to grow directly toward it using the same steering process.
    - This step continues until:
        - The goal tree successfully reaches the new node in the start tree, or
        - An obstacle prevents further growth.

4. **Connection Check:**
    - If the two trees meet (i.e., a new node in the start tree connects directly to a node in the goal tree), a solution path is found by combining the paths of both trees.
5. **Repeat Until Success:**
    - If the trees don't connect, the process repeats by sampling a new random state, alternating the roles of the trees:
        - Grow the goal tree first and attempt to connect the start tree.
    - This ensures comprehensive exploration of the configuration space.
6. **Termination:**
    - The algorithm continues iterating until:
        - A solution path is found, or
        - A predefined time limit or iteration count is reached.

Illustrative Analogy

Imagine two explorers:

1. One starts at a mountain base (start tree), and the other starts at the peak (goal tree).
2. They both walk toward a randomly chosen point in the terrain (random sampling).
3. When one explorer gets close to this point, the other tries to meet them there (connection attempt).
4. They continue this process until they meet, forming a complete path from the base to the peak.

Advantages of RRT-Connect

1. **Speed**: Faster than standard RRT and RRT* due to its bidirectional nature.
2. **Efficiency**: Focuses on connecting the trees, which reduces the overall search space.
3. **Simplicity**: Straightforward to implement and integrate into existing systems.

Performance Tips

1. **State Validity Checker**:
    - Optimize the collision-checking function for faster performance.
2. **Goal Bias**:
    - Tune goal bias to balance exploration and exploitation.
3. **Tree Growth Parameters**:
    - Adjust the step size for tree growth to ensure efficient exploration while maintaining collision-free paths.

Use Cases

- Pathfinding in high-dimensional spaces.
- Motion planning for robotic arms, mobile robots, and drones.
- Applications where finding a feasible path quickly is more important than finding the optimal path.