ROSCon 2024: Manipulating with MoveIt Like a Pro

Ask questions via Slido: slido.com #4109254

If you haven't already done the getting started steps:

- Install Docker: https://docs.docker.com/engine/install/ubuntu/
  - Follow Docker post-install instructions: https://docs.docker.com/engine/install/linux-postinstall/#manage-docker-as-a-non-root-user
- Exercise 1 setup:
  - Download `docker-image.tar.gz` from the repository releases
  - Clone the repository: `$ git clone https://github.com/moveit/roscon24.git`
  - `cd` into the `roscon24` repository
  - Get the image with the fetch script: `$ docker/fetch ~/Downloads/docker-image.tar.gz`
- Exercise 2 setup:
  - Follow the install instructions on Quick Start | MoveIt Pro
  - You should have received an individual **license key** for this workshop via email
- If you run into technical difficulties, let us know and we will try our best to fix them
- If you don't have a working installation, please group up with someone who does

Manipulating with MoveIt Like a Pro

ROSCon 2024
October 21, 2024

Presenters & Contributors

Sebastian Jahr
Robotics Engineer
PickNik Robotics

Henning Kayser
MoveIt Chief Architect
PickNik Robotics

Stephanie Eng
Senior Autonomy Developer
OTTO Motors

Matthew Hansen
Principal Solutions Architect
PickNik Robotic

Disclaimer

Opinions here are of our own, and are not representative of employer

Agenda

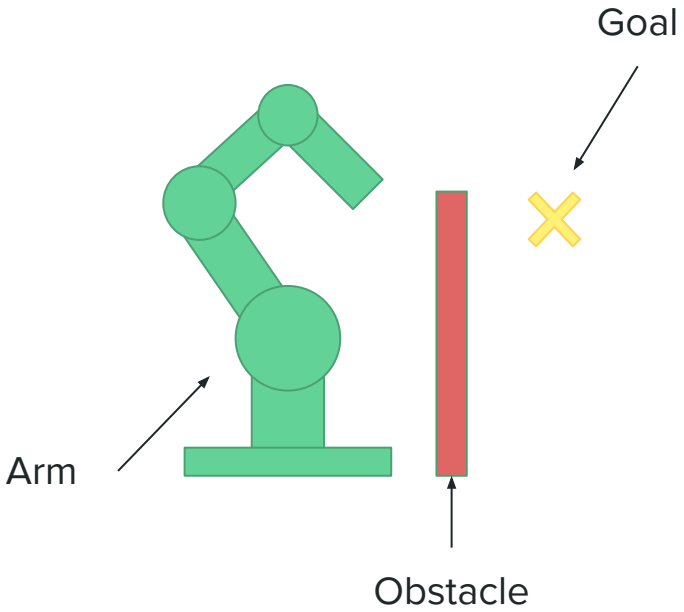
8:00 - 8:30	Introduction to MoveIt
8:30 - 9:30	Exercise 1: Motion Planning and Execution with Move Group
9:30 - 10:00	Manipulation is not just Motion Planning
10:00 - 10:30	☕ Q&A and Break ☕
10:30 - 11:50	Exercise 2: The MoveIt Pro Approach for Robotic Manipulation
11:50 - 12:00	Summary and Q&A

Introduction to MoveIt

Motion Planning (for Arms)

What is Motion Planning?

“Motion planning is the problem of finding a robot motion from a start state to a goal state that avoids obstacles in the environment and satisfies other constraints”
(Lynch, K. M., & Park, F. C. (2017))

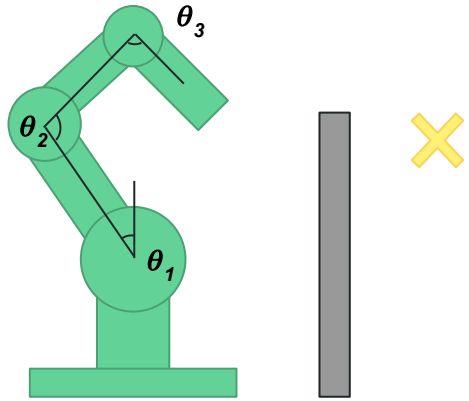


Task & Joint Space

“Motion planning is the problem of finding a robot motion …

➜ We can control the joint angles of the robot arm to change its position

➜ To do a motion, we need to come up with a sequence of joint state setpoints (Robot states) this is called **path**
➜ But we also need to know, when which control is applied or define dynamic properties like joint velocities. A path with timing information is called **trajectory**

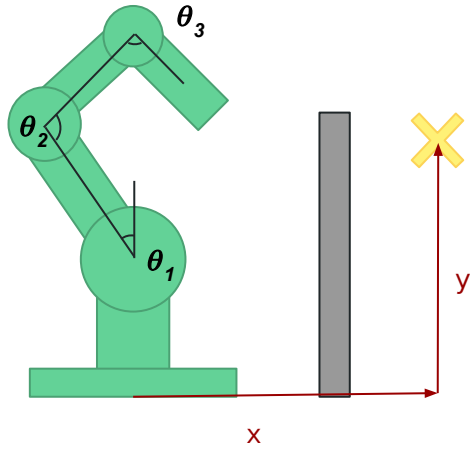


Task & Joint Space

“Motion planning is the problem of finding a robot motion from a start state to a goal state …

➜ Robot state is described by a set of joint angles (θ_1, θ_2, … θ_n). Typical industrial arms have 6-7 joints.

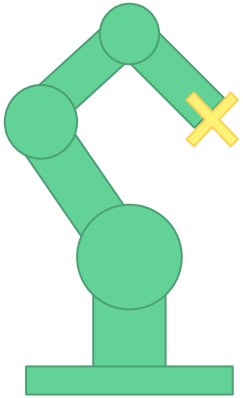
➜ Goal state is often described by operators in Cartesian space (x, y, z, r, p, y)



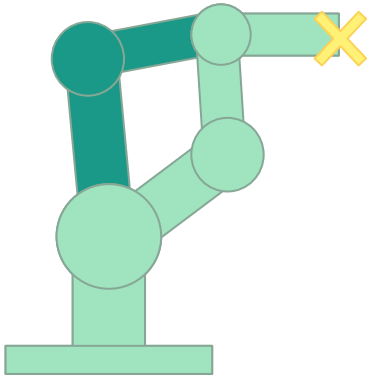
Kinematics

“Motion planning is the problem of finding a robot motion from a start state to a goal state ...

$(\theta_1, \theta_2, \ldots \theta_n) \rightarrow (x, y, z, r, p, y)$ is called Forward Kinematics

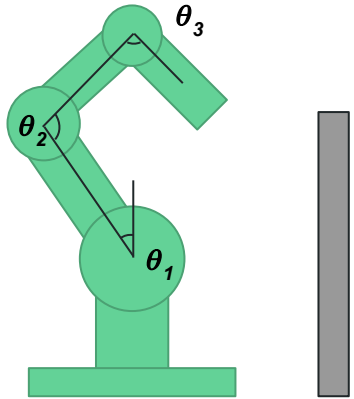


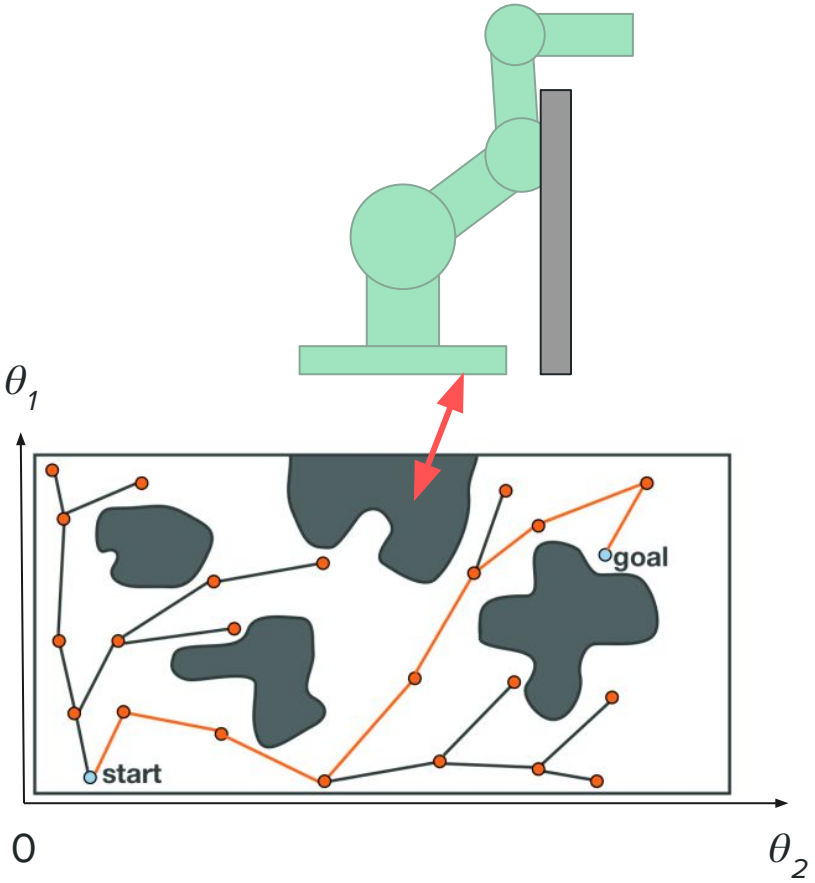
$(x, y, z, r, p, y) \rightarrow (\theta_1, \theta_2, \ldots \theta_n)$ is called Inverse Kinematics



Collision Checking

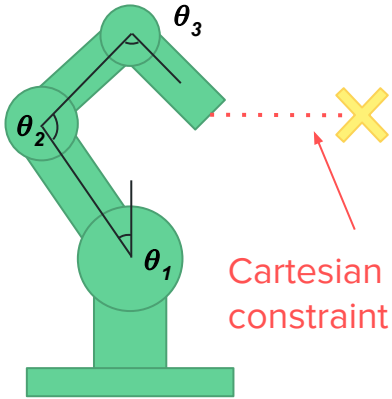
“Motion planning is the problem of finding a robot motion from a start state to a goal state that avoids obstacles …





Constraints

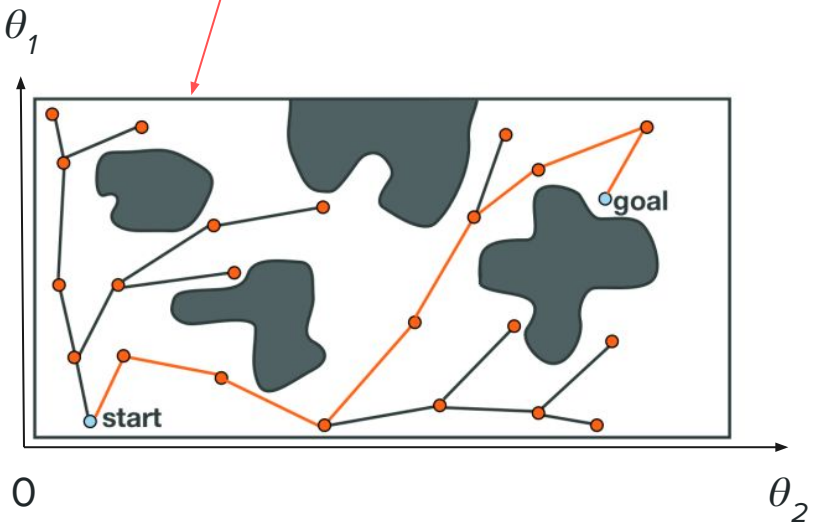
“Motion planning is the problem of finding a robot motion from a start state to a goal state that avoids obstacles and satisfies other constraints”



Cartesian EE constraint

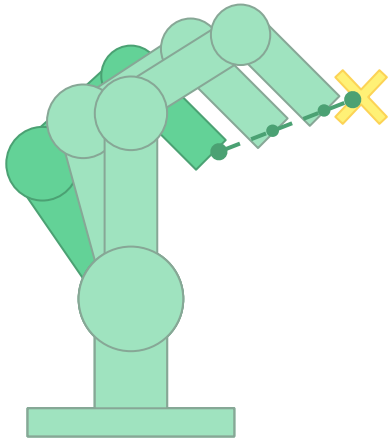
Velocity Limits

Joint Limits

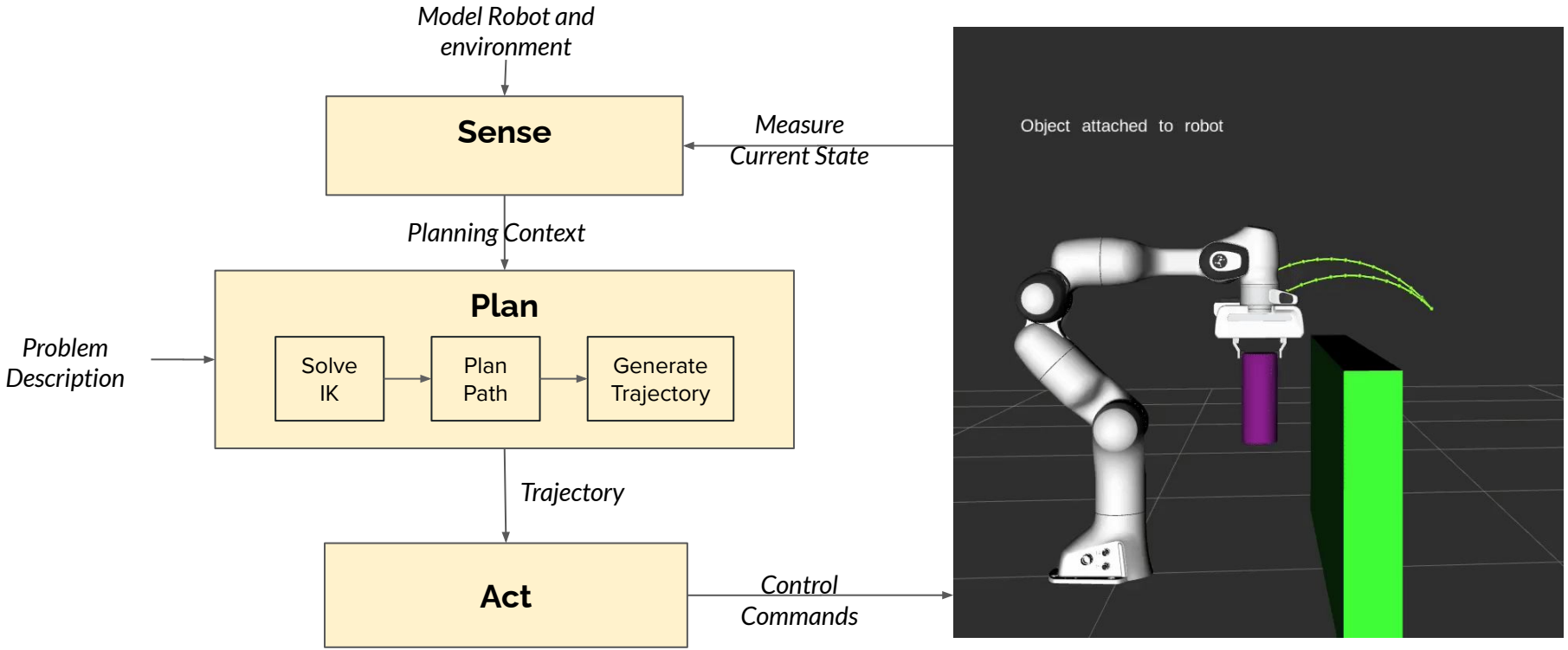


Planning in Tool Space

- Sometimes, you can have a desired end effector trajectory (toolpath, along a line)
  - Welding, cleaning, painting...
- Planning can be done in tool space (Cartesian) or joint space
  - Joint space motion plans may be more performant
    - Able to plan around collisions, potentially faster
  - Tool space allows you to move along a specified path
- Plan path along Cartesian trajectory, solving for joint states along the way
  - Relies heavily on inverse kinematics to solve

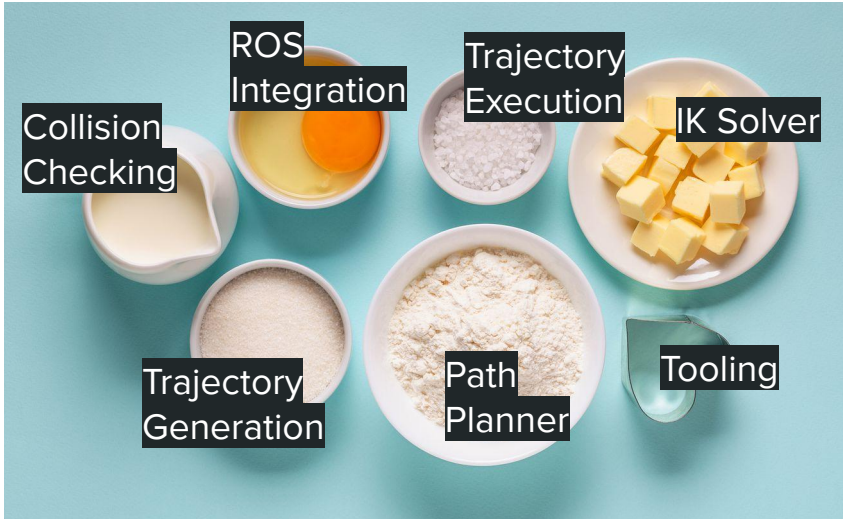


How can you solve a motion planning problem?

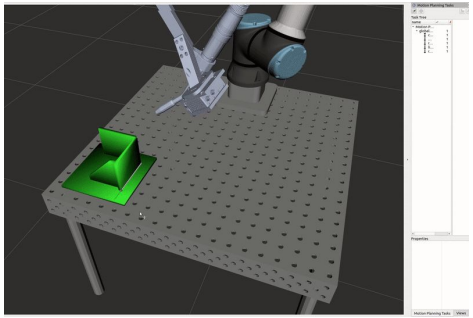


What is MoveIt?

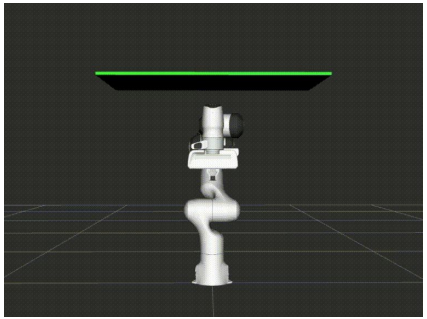
A ROS motion planning framework that integrates algorithms and approaches to plan and execute a robot motion



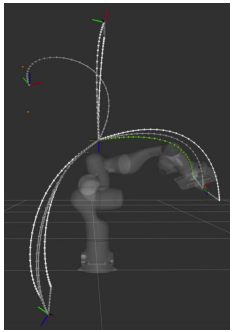
More MoveIt Features



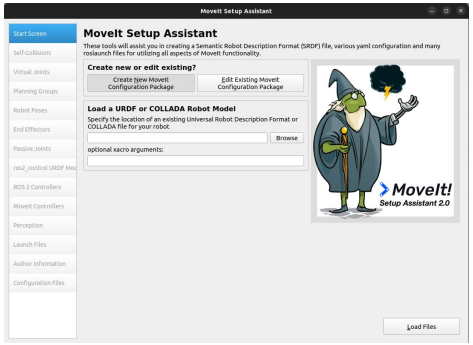
MoveIt Servo



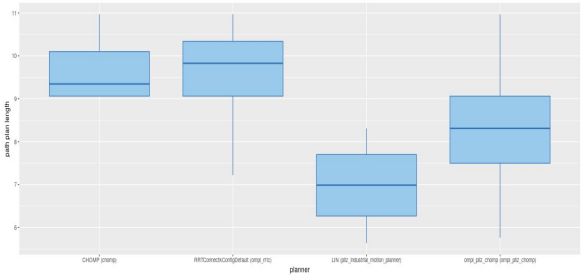
Hybrid Planning



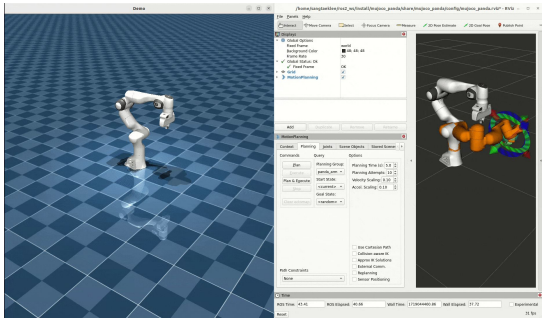
Trajectory Cache NEW



Setup Assistant

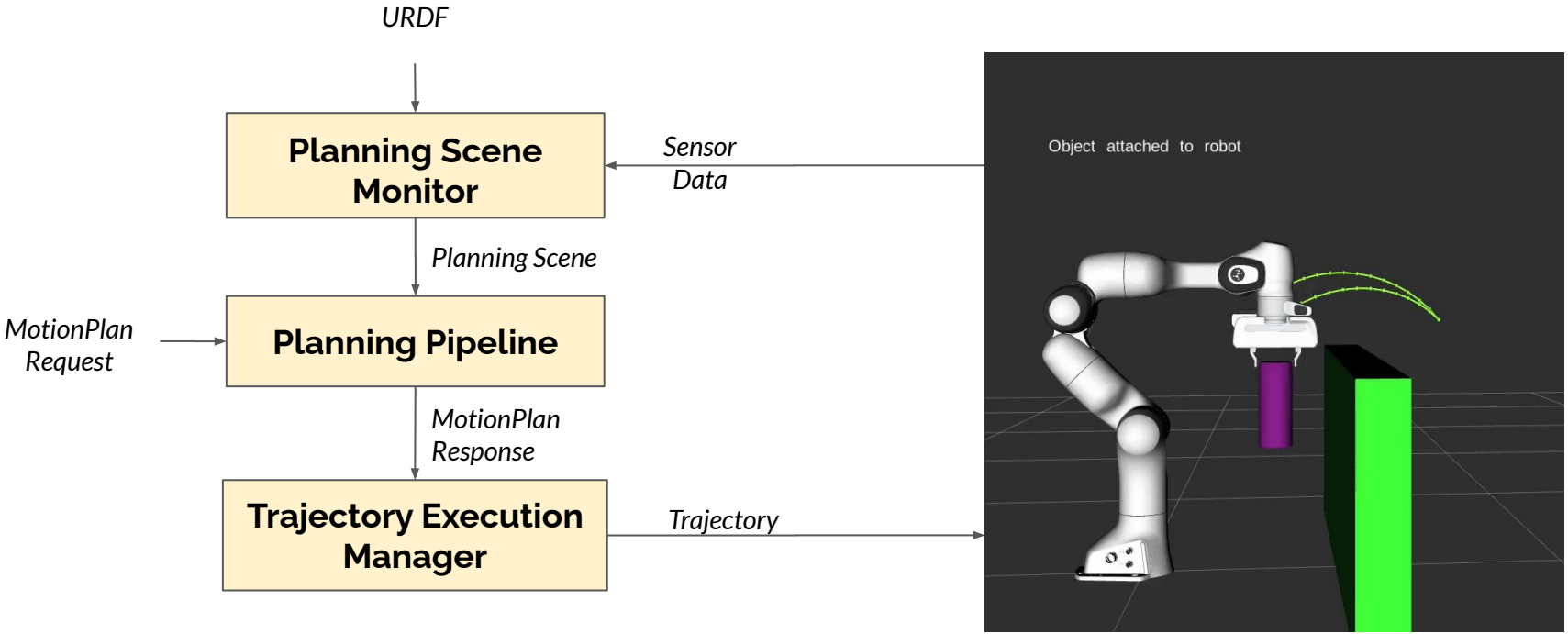


Benchmarking

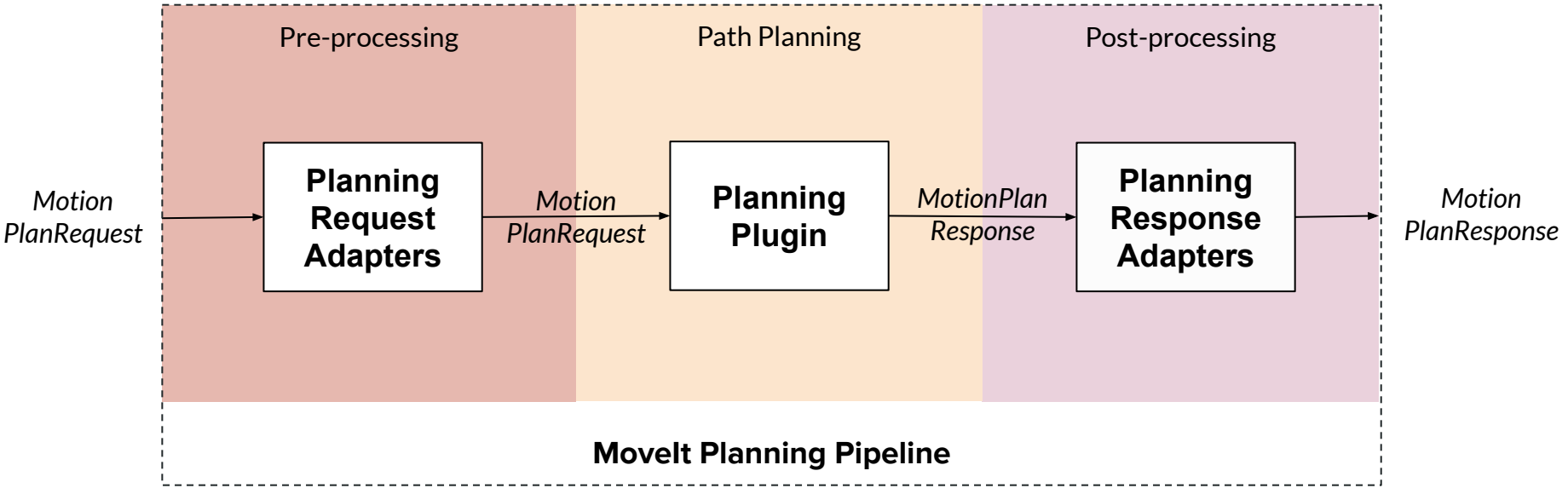


Simulation Integration

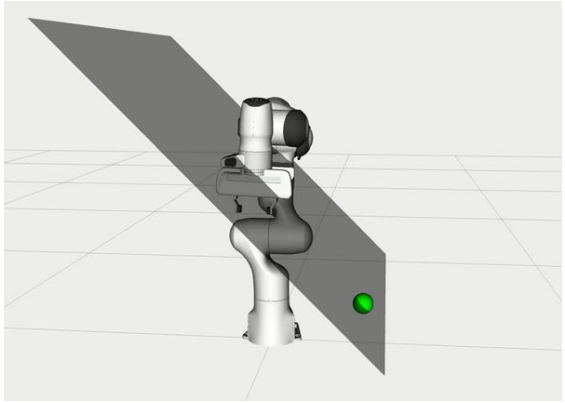
Core components



The Planning Pipeline

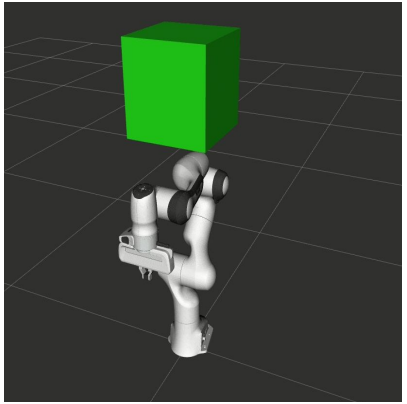


Main Planning Libraries



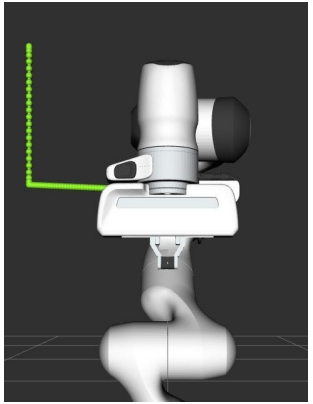
OMPL

- Library of sampling-based planners and frameworks
- Supports constrained planning



STOMP

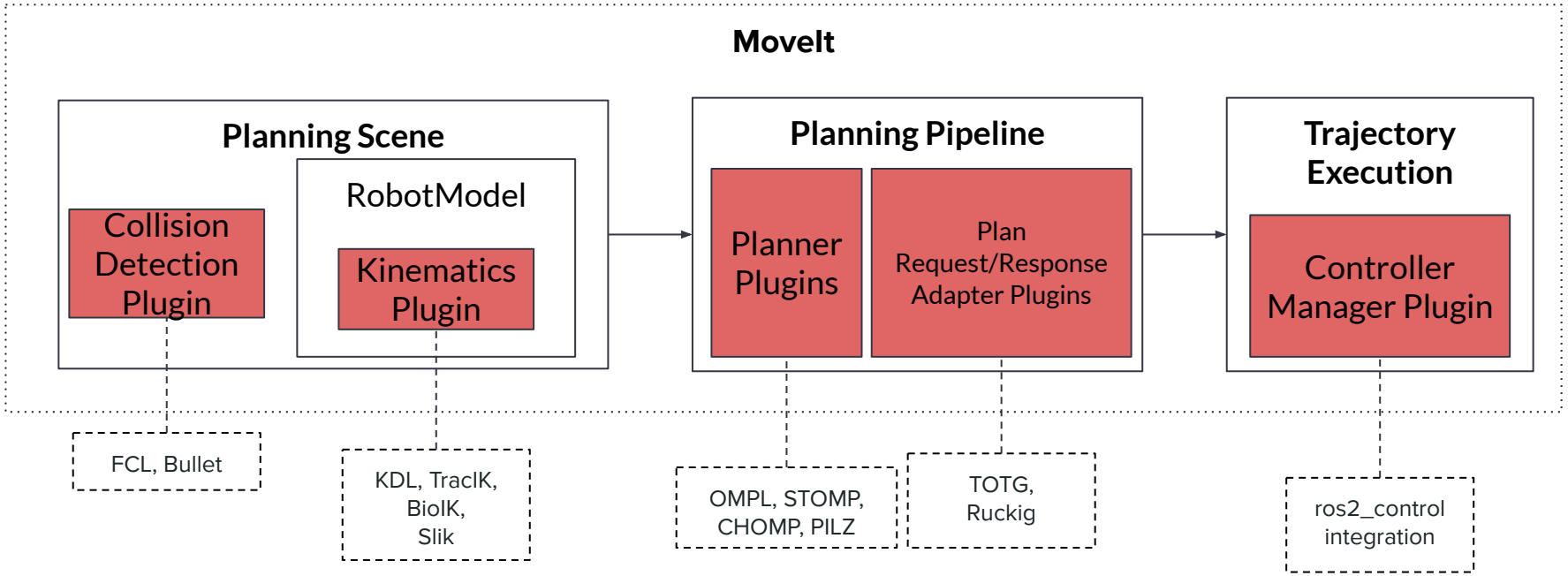
- **S**tochastic **T**rajectory **O**ptimization for **M**otion **P**lanning
- Optimizing planner

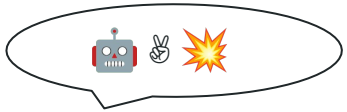


PILZ

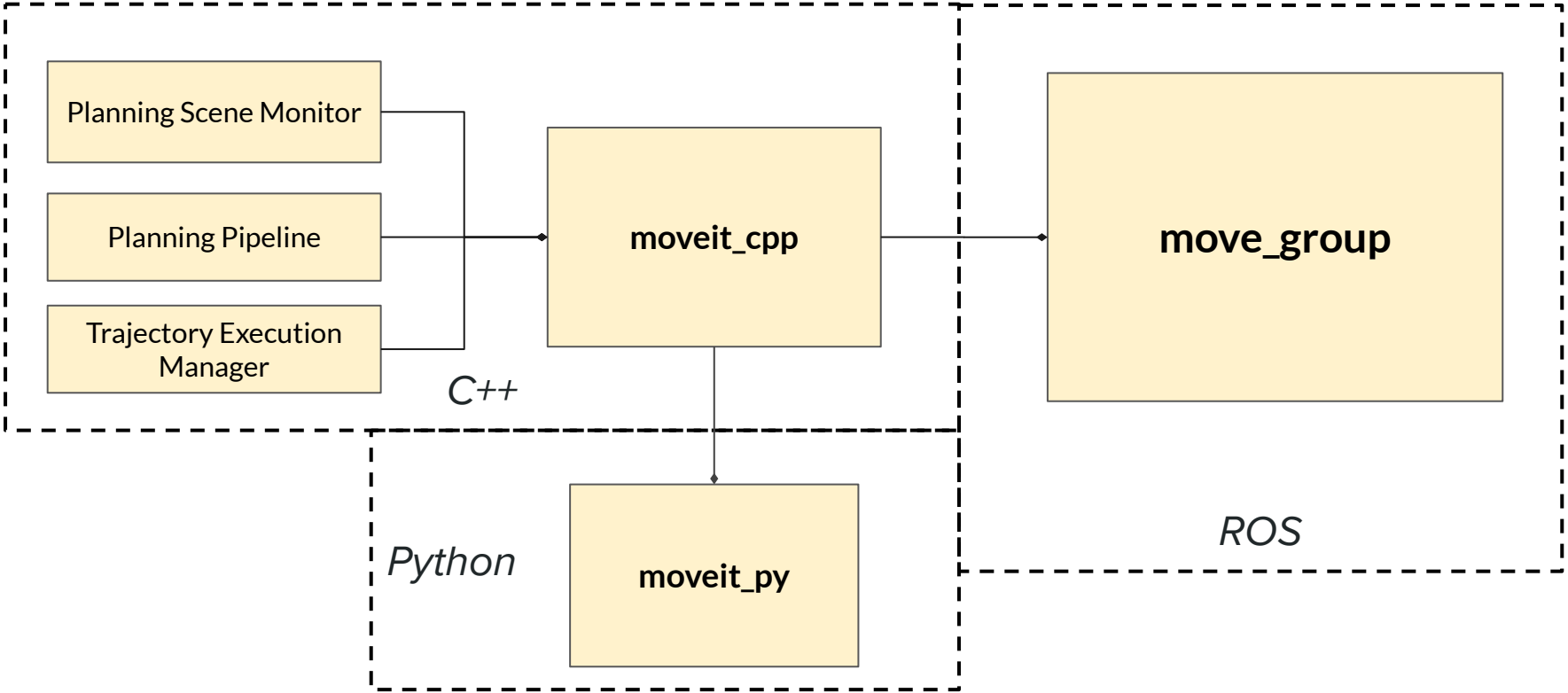
- Industrial Motion Generator (LIN, PTP, CIRC)

Plugins Plugins Plugins





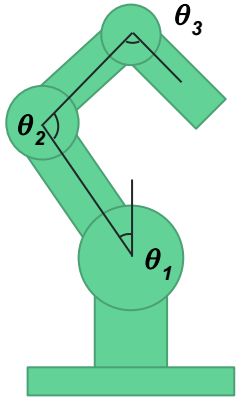
Talking to MoveIt



Exercise 1: Motion Planning and Execution with Move Group

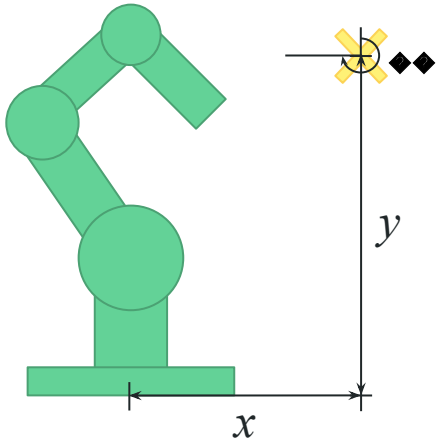
Inverse Kinematics

- Robot state is expressed in joint space (θ_1, θ_2, ... θ_n)



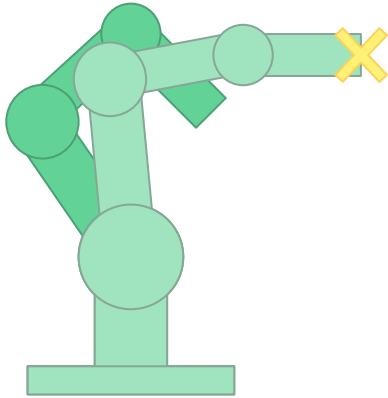
Inverse Kinematics

- Robot state is expressed in joint space (θ_1, θ_2, ... θ_n)
- Goals in Cartesian space (x, y, z, r, p, y)



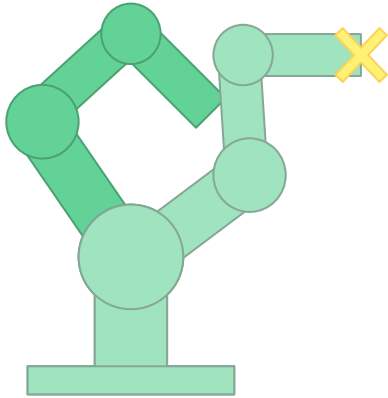
Inverse Kinematics

- Robot state is expressed in joint space (θ_1, θ_2, ... θ_n)
- Goals in Cartesian space (x, y, z, r, p, y)
- Need to know what joint states allow the robot to reach the Cartesian goal
- Solve for this with Inverse Kinematics (IK)



Inverse Kinematics

- Robot state is expressed in joint space (θ_1, θ_2, ... θ_n)
- Goals in Cartesian space (x, y, z, r, p, y)
- Need to know what joint states allow the robot to reach the Cartesian goal
- Solve for this with Inverse Kinematics (IK)

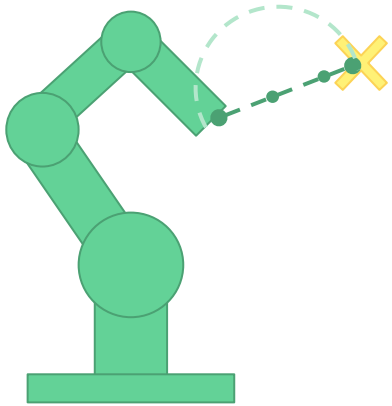


Inverse Kinematics

- IK can be solved numerically or analytically
  - Some IK solutions are more desirable than others (distance, speed, constraints...)
- MoveIt uses a configurable kinematics plugin to solve IK for motion planning
  - KDL, TracIK, BioIK, your own IK plugin ...
- Configurable via the **kinematics.yaml** file (generated by MoveIt Setup Assistant)

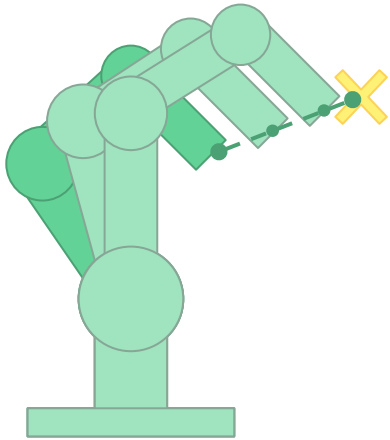
Cartesian Planning

- Cartesian planning is useful when you have a desired end effector trajectory (toolpath, along a line)
  - Welding, cleaning, painting
- Joint space motion plans may be more performant
  - Able to plan around collisions, potentially faster



Cartesian Planning

- Cartesian planning is useful when you have a desired end effector trajectory (toolpath, along a line)
  - Welding, cleaning, painting...
- Joint space motion plans may be more performant
  - Able to plan around collisions, potentially faster
- Plan path along Cartesian trajectory, solving for joint states along the way
  - Relies heavily on IK to solve



Cartesian Planning

- MoveIt has a built-in Cartesian interpolator
  - Interpolates intermediate waypoints along a straight-line path between start and goal
  - Caveats:
    - Cannot plan around obstacles
    - Straight lines only
    - Waypoint-to-waypoint, no blending
- Other options:
  - PILZ Industrial Motion planner
  - OMPL constrained planning
  - Other planning plugins

Hot Dog Pick and Place

If you haven't already done the getting started steps:

- Download `docker-image.tar.gz` from the workshop repository latest releases
- Clone the repository:
  - `$ git clone https://github.com/moveit/roscon24.git`
- `cd` into the `roscon24` repository
- Use the fetch script to get the image:
  - `$ docker/fetch ~/Downloads/docker-image.tar.gz`

Hot Dog Pick and Place

- The `exercise1` directory is a workspace
- Inside the workspace are three packages:
  - `exercise1-1`
    - Move group planning exercise
  - `exercise1-2`
    - Cartesian planning with move group
  - `hot_dog`
    - Meshes used in both exercises
- Start with a warm-up to test your setup and make your first motion plan

If you haven't already done the getting started steps:

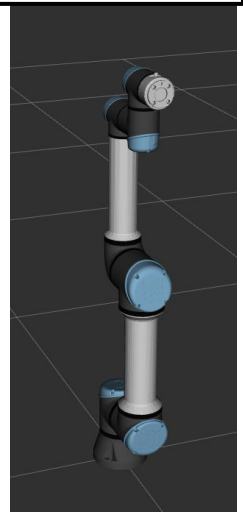
- Download `docker-image.tar.gz` from the workshop repository latest releases
- Clone the repository:
  - `$ git clone https://github.com/moveit/roscon24.git`
- `cd` into the `roscon2024` repository
- Use the fetch script to get the image:
  - `$ docker/fetch ~/Downloads/docker-image.tar.gz`

Warm Up: Your First Motion Plan

- Exercises use a UR robot to assemble a hot dog
- Uses UR assets and driver from upstream UR packages
  - No gripper for simplicity
- Mock hardware
  - Measured joint states will exactly be the commanded joint states

If you haven't already done the getting started steps:

- Download `docker-image.tar.gz` from the workshop repository latest releases
- Clone the repository:
  - `$ git clone https://github.com/moveit/roscon24.git`
- `cd` into the `roscon2024` repository
- Use the fetch script to get the image:
  - `$ docker/fetch ~/Downloads/docker-image.tar.gz`

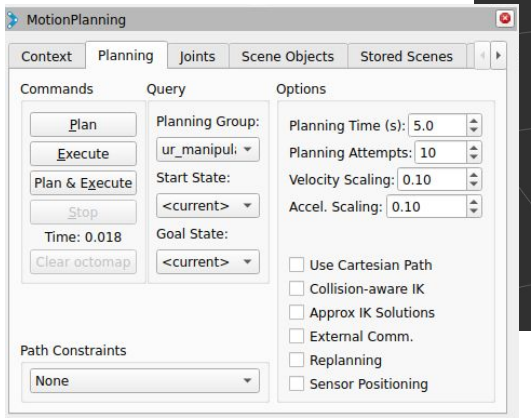


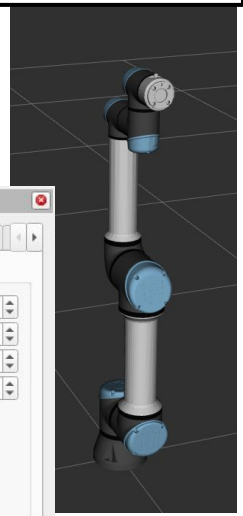
Warm Up: Your First Motion Plan

- Start and exec into the container:
  ```
  $ docker/start
  $ docker/shell
  ```
- Change directories to `exercise1`:
  ```
  $ cd exercise1
  ```
- Build the workspace:
  ```
  $ colcon build
  ```
- Source the workspace:
  ```
  $ source install/setup.sh
  ```
- Start the UR mock with:
  ```
  $ ros2 launch exercise1-1 ur.launch.py
  ```
  - Loads UR robot configuration and driver
  - Starts Move Group
  - Starts RViz
- Motion planning panel should be open on the left

If you haven't already done the getting started steps:

- Download `docker-image.tar.gz` from the workshop repository latest releases
- Clone the repository:
  - `$ git clone https://github.com/moveit/roscon24.git`
- `cd` into the `roscon24` repository
- Use the fetch script to get the image:
  - `$ docker/fetch ~/Downloads/docker-image.tar.gz`

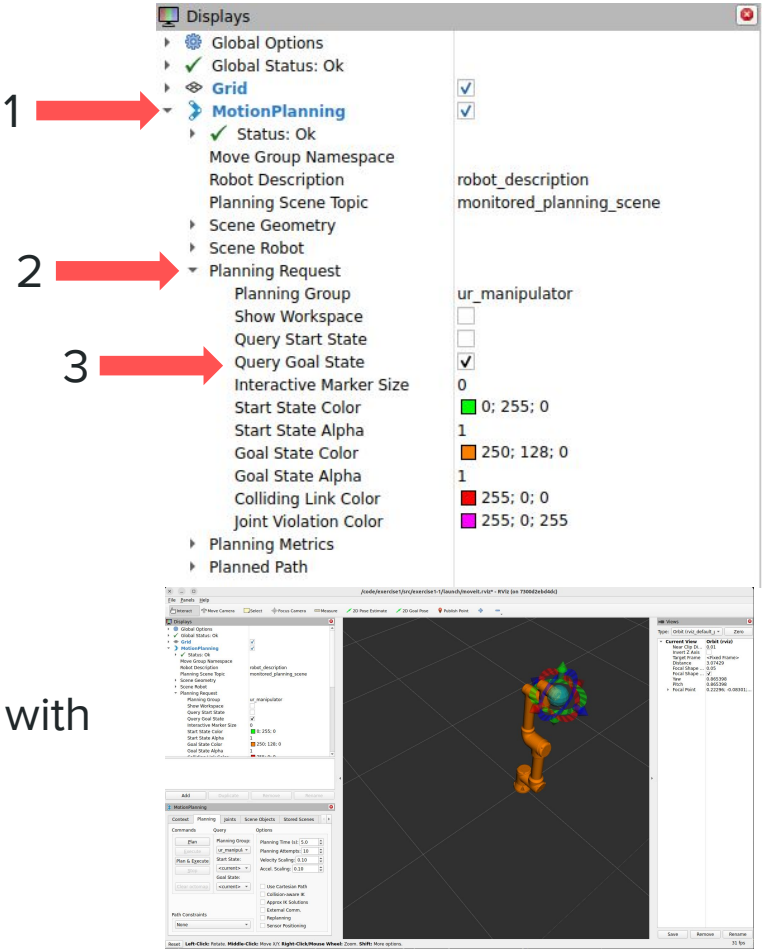




Warm Up: Your First Motion Plan

- Enable visualizing the goal state to start motion planning from RViz

1. Expand the “MotionPlanning” display
2. Expand “Planning Request”
3. Check “Query Goal State”

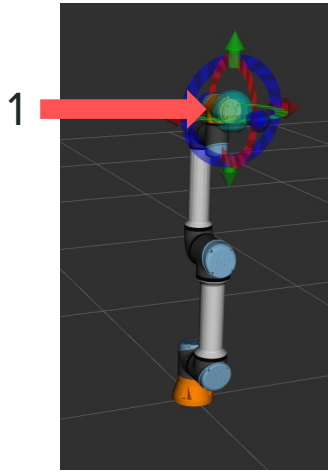


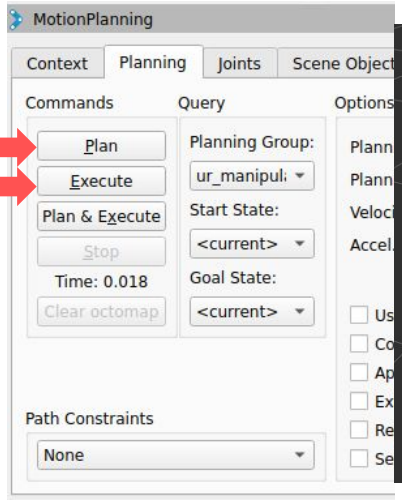
You should see an orange outline of the robot with an interactive marker (right)

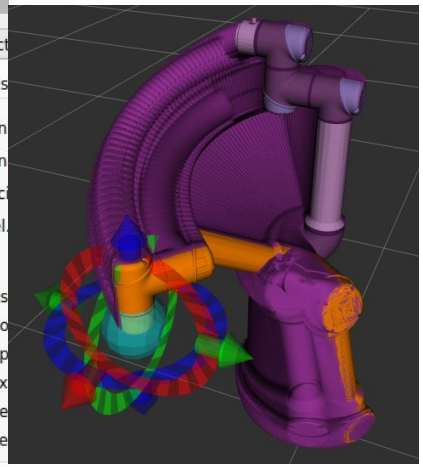
Hot Dog Pick and Place

1. Drag the interactive marker around to set a goal state
2. Use the `Plan` button to plan to that goal state
   - Retry a with a marker position if planning fails
3. Use the `Execute` button to move the robot according to the plan
4. Stop the UR mock (**Ctrl+C** in the terminal you ran it from)

Note: Image on the right has Show Trails enabled for visualization purposes



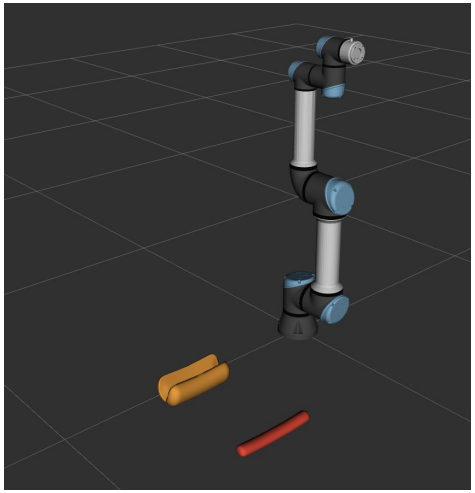




Exercise 1-1: Hot Dog Pick and Place

Run Instructions

- Open a second terminal and run:
  - `$ docker/shell`
  - `$ cd exercise1`
  - `$ source install/setup.sh`
- **Terminal 1**: Restart the UR mock:
  - `$ ros2 launch exercise1-1 ur.launch.py`
- **Terminal 2**: Start exercise 1-1:
  - `$ ros2 launch exercise1-1 exercise1-1.launch.py`
- Should see the robot, a hot dog, and a bun (right)
- Stop all processes with **Ctrl+C**
- Let's make things move!

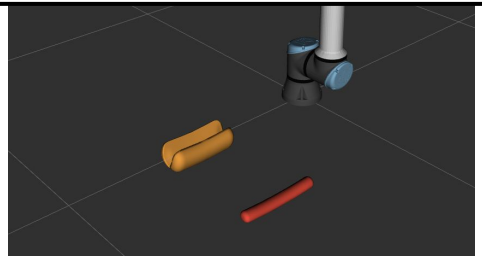


Run Instructions

- Open a second terminal and run:
  - `$ docker/shell`
  - `$ cd exercise1`
  - `$ source install/setup.sh`
- **Terminal 1**: Restart the UR mock:
  - `$ ros2 launch exercise1-1 ur.launch.py`
- **Terminal 2**: Start exercise 1-1:
  - `$ ros2 launch exercise1-1 exercise1-1.launch.py`
- Should see the robot, a hot dog, and a bun (right)
- Stop all processes with **Ctrl+C**
- Let's make things move!

Quick note:

- The workshop Docker container shares files in the repository between the host and container
- The next steps will involve looking at and editing source code
  - Please open the source files in an editor of your choice (on your host, **not** in the container)
- Saved changes to the source file from your editor will also be available in the container



Hot Dog Pick

File: `exercise1/src/exercise1-1/src/main.cpp`

Create a MoveGroupInterface and set the planning group

```
25  // Create the MoveGroupInterface
26  using moveit::planning_interface::MoveGroupInterface;
27  const std::string planning_group{ "ur_manipulator" };
28  MoveGroupInterface move_group_interface(node, planning_group);
29
30  // Get predefined poses for pick and place above the sausage and bun respectively
31  const geometry_msgs::msg::Pose& pick_pose = hot_dog_scenario.getPickPose();
32  const geometry_msgs::msg::Pose& place_pose = hot_dog_scenario.getPlacePose();
33
34  // TODO(Exercise 1-1): Set the goal pose to the pick pose using MoveGroupInterface
35  // Add your code here
36
37  // Create a plan to that target pose
38  MoveGroupInterface::Plan pick_plan;
39  uint16_t pick_planning_attempts{ 0 };
40  bool pick_success{ false };
41  while (!pick_success && pick_planning_attempts < 5)
42  {
43    // TODO(Exercise 1-1): Use MoveGroupInterface to plan to the pick pose and determine planning success
44    // Add your code here
45    ++pick_planning_attempts;
46  }
47  if (pick_success)
48  {
49    // Execute the plan
50    move_group_interface.execute(pick_plan);
```

Hot Dog Pick

File: `exercise1/src/exercise1-1/src/main.cpp`

```
25  // Create the MoveGroupInterface
26  using moveit::planning_interface::MoveGroupInterface;
27  const std::string planning_group{ "ur_manipulator" };
28  MoveGroupInterface move_group_interface(node, planning_group);
29
30  // Get predefined poses for pick and place above the sausage and bun respectively
31  const geometry_msgs::msg::Pose& pick_pose = hot_dog_scenario.getPickPose();
32  const geometry_msgs::msg::Pose& place_pose = hot_dog_scenario.getPlacePose();
33
34  // TODO(Exercise 1-1): Set the goal pose to the pick pose using MoveGroupInterface
35  // Add your code here
36
37  // Create a plan to that target pose
38  MoveGroupInterface::Plan pick_plan;
39  uint16_t pick_planning_attempts{ 0 };
40  bool pick_success{ false };
41  while (!pick_success && pick_planning_attempts < 5)
42  {
43    // TODO(Exercise 1-1): Use MoveGroupInterface to plan to the pick pose and determine planning success
44    // Add your code here
45    ++pick_planning_attempts;
46  }
47  if (pick_success)
48  {
49    // Execute the plan
50    move_group_interface.execute(pick_plan);
```

Create a MoveGroupInterface and set the planning group

Use helper functions to get the pick and place poses

Hot Dog Pick

File: `exercise1/src/exercise1-1/src/main.cpp`

```
25  // Create the MoveGroupInterface
26  using moveit::planning_interface::MoveGroupInterface;
27  const std::string planning_group{ "ur_manipulator" };
28  MoveGroupInterface move_group_interface(node, planning_group);
29
30  // Get predefined poses for pick and place above the sausage and bun respectively
31  const geometry_msgs::msg::Pose& pick_pose = hot_dog_scenario.getPickPose();
32  const geometry_msgs::msg::Pose& place_pose = hot_dog_scenario.getPlacePose();
33
34  // TODO(Exercise 1-1): Set the goal pose to the pick pose using MoveGroupInterface
35  // Add your code here
36
37  // Create a plan to that target pose
38  MoveGroupInterface::Plan pick_plan;
39  uint16_t pick_planning_attempts{ 0 };
40  bool pick_success{ false };
41  while (!pick_success && pick_planning_attempts < 5)
42  {
43    // TODO(Exercise 1-1): Use MoveGroupInterface to plan to the pick pose and determine planning success
44    // Add your code here
45    ++pick_planning_attempts;
46  }
47  if (pick_success)
48  {
49    // Execute the plan
50    move_group_interface.execute(pick_plan);
```

Create a MoveGroupInterface and set the planning group

Use helper functions to get the pick and place poses

We want to plan to the pick pose

Hot Dog Pick

File: `exercise1/src/exercise1-1/src/main.cpp`

```
25  // Create the MoveGroupInterface
26  using moveit::planning_interface::MoveGroupInterface;
27  const std::string planning_group{ "ur_manipulator" };
28  MoveGroupInterface move_group_interface(node, planning_group);
29
30  // Get predefined poses for pick and place above the sausage and bun respectively
31  const geometry_msgs::msg::Pose& pick_pose = hot_dog_scenario.getPickPose();
32  const geometry_msgs::msg::Pose& place_pose = hot_dog_scenario.getPlacePose();
33
34  // TODO(Exercise 1-1): Set the goal pose to the pick pose using MoveGroupInterface
35  // Add your code here
36
37  // Create a plan to that target pose
38  MoveGroupInterface::Plan pick_plan;
39  uint16_t pick_planning_attempts{ 0 };
40  bool pick_success{ false };
41  while (!pick_success && pick_planning_attempts < 5)
42  {
43    // TODO(Exercise 1-1): Use MoveGroupInterface to plan to the pick pose and determine planning success
44    // Add your code here
45    ++pick_planning_attempts;
46  }
47  if (pick_success)
48  {
49    // Execute the plan
50    move_group_interface.execute(pick_plan);
```

Create a MoveGroupInterface and set the planning group

Use helper functions to get the pick and place poses

We want to plan to the pick pose

We want to plan and know if planning was successful

Hot Dog Pick

File: `exercise1/src/exercise1-1/src/main.cpp`

```
25  // Create the MoveGroupInterface
26  using moveit::planning_interface::MoveGroupInterface;
27  const std::string planning_group{ "ur_manipulator" };
28  MoveGroupInterface move_group_interface(node, planning_group);
29
30  // Get predefined poses for pick and place above the sausage and bun respectively
31  const geometry_msgs::msg::Pose& pick_pose = hot_dog_scenario.getPickPose();
32  const geometry_msgs::msg::Pose& place_pose = hot_dog_scenario.getPlacePose();
33
34  // TODO(Exercise 1-1): Set the goal pose to the pick pose using MoveGroupInterface
35  // Add your code here
36
37  // Create a plan to that target pose
38  MoveGroupInterface::Plan pick_plan;
39  uint16_t pick_planning_attempts{ 0 };
40  bool pick_success{ false };
41  while (!pick_success && pick_planning_attempts < 5)
42  {
43    // TODO(Exercise 1-1): Use MoveGroupInterface to plan to the pick pose and determine planning success
44    // Add your code here
45    ++pick_planning_attempts;
46  }
47  if (pick_success)
48  {
49    // Execute the plan
50    move_group_interface.execute(pick_plan);
```

Create a MoveGroupInterface and set the planning group

Use helper functions to get the pick and place poses

We want to plan to the pick pose

We want to plan and know if planning was successful

If we successfully planned, execute the plan

Hot Dog Pick

File: `exercise1/src/exercise1-1/src/main.cpp`

```
25  // Create the MoveGroupInterface
26  using moveit::planning_interface::MoveGroupInterface;
27  const std::string planning_group{ "ur_manipulator" };
28  MoveGroupInterface move_group_interface(node, planning_group);
29
30  // Get predefined poses for pick and place above the sausage and bun respectively
31  const geometry_msgs::msg::Pose& pick_pose = hot_dog_scenario.getPickPose();
32  const geometry_msgs::msg::Pose& place_pose = hot_dog_scenario.getPlacePose();
33
34  // TODO(Exercise 1-1): Set the goal pose to the pick pose using MoveGroupInterface
35  // Add your code here
36
37  // Create a plan to that target pose
38  MoveGroupInterface::Plan pick_plan;
39  uint16_t pick_planning_attempts{ 0 };
40  bool pick_success{ false };
41  while (!pick_success && pick_planning_attempts < 5)
42  {
43    // TODO(Exercise 1-1): Use MoveGroupInterface to plan to the pick pose and determine planning success
44    // Add your code here
45    ++pick_planning_attempts;
46  }
47  if (pick_success)
48  {
49    // Execute the plan
50    move_group_interface.execute(pick_plan);
```

Set the goal to `pick_pose`:

```
move_group_interface.setPoseTarget(pick_pose);
```

Hot Dog Pick

File: `exercise1/src/exercise1-1/src/main.cpp`

```
25  // Create the MoveGroupInterface
26  using moveit::planning_interface::MoveGroupInterface;
27  const std::string planning_group{ "ur_manipulator" };
28  MoveGroupInterface move_group_interface(node, planning_group);
29
30  // Get predefined poses for pick and place above the sausage and bun respectively
31  const geometry_msgs::msg::Pose& pick_pose = hot_dog_scenario.getPickPose();
32  const geometry_msgs::msg::Pose& place_pose = hot_dog_scenario.getPlacePose();
33
34  // TODO(Exercise 1-1): Set the goal pose to the pick pose using MoveGroupInterface
35  // Add your code here
36
37  // Create a plan to that target pose
38  MoveGroupInterface::Plan pick_plan;
39  uint16_t pick_planning_attempts{ 0 };
40  bool pick_success{ false };
41  while (!pick_success && pick_planning_attempts < 5)
42  {
43    // TODO(Exercise 1-1): Use MoveGroupInterface to plan to the pick pose and determine planning success
44    // Add your code here
45    ++pick_planning_attempts;
46  }
47  if (pick_success)
48  {
49    // Execute the plan
50    move_group_interface.execute(pick_plan);
```

Set the goal to `pick_pose`:

```
move_group_interface.setPoseTarget(pick_pose);
```

Plan and determine if planning succeeded:

```
pick_success = static_cast<bool>(move_group_interface.plan(pick_plan));
```

Hot Dog Pick

File: `exercise1/src/exercise1-1/src/main.cpp`

```
25  // Create the MoveGroupInterface
26  using moveit::planning_interface::MoveGroupInterface;
27  const std::string planning_group{ "ur_manipulator" };
28  MoveGroupInterface move_group_interface(node, planning_group);
29
30  // Get predefined poses for pick and place above the sausage and bun respectively
31  const geometry_msgs::msg::Pose& pick_pose = hot_dog_scenario.getPickPose();
32  const geometry_msgs::msg::Pose& place_pose = hot_dog_scenario.getPlacePose();
33
34  // TODO(Exercise 1-1): Set the goal pose to the pick pose using MoveGroupInterface
35  // Add your code here
36
37  // Create a plan to that target pose
38  MoveGroupInterface::Plan pick_plan;
39  uint16_t pick_planning_attempts{ 0 };
40  bool pick_success{ false };
41  while (!pick_success && pick_planning_attempts < 5)
42  {
43    // TODO(Exercise 1-1): Use MoveGroupInterface to plan to the pick pose and determine planning success
44    // Add your code here
45    ++pick_planning_attempts;
46  }
47  if (pick_success)
48  {
49    // Execute the plan
50    move_group_interface.execute(pick_pl
```

Set the goal to `pick_pose`:

```
move_group_interface.setPoseTarget(pick_pose);
```

Plan and determine if planning succeeded:

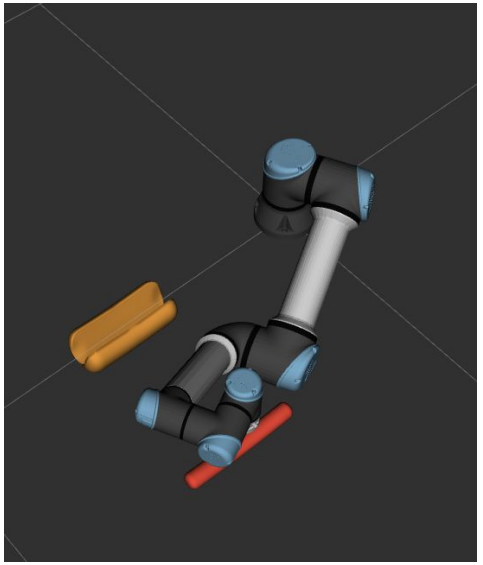
```
pick_success = static_cast<bool>(move_group_interface.plan(pick_plan));
```

Terminal 1: `$ colcon build`

Terminal 1: `$ ros2 launch exercise1-1 ur.launch.py`

Terminal 2: `$ ros2 launch exercise1-1 exercise1-1.launch.py`

Hot Dog Pick



- **Terminal 1:** Rebuild and restart the UR mock
  - `$ colcon build`
  - `$ ros2 launch exercise1-1 ur.launch.py`
- **Terminal 2:** Run the modified exercise 1-1:
  - `$ ros2 launch exercise1-1 exercise1-1.launch.py`
- Watch the robot plan and move to the sausage in RViz
  - It may take a roundabout path due to RRTConnect
  - If you want to watch it again, or if planning fails, stop the UR mock with **Ctrl+C** and restart:
    - **Terminal 1:** `$ ros2 launch exercise1-1 ur.launch.py`
    - **Terminal 2:** `$ ros2 launch exercise1-1 exercise1-1.launch.py`
- Stop the UR mock with **Ctrl+C** when done

Hot Dog Place

File: `exercise1/src/exercise1-1/src/main.cpp`

Helper function to get the sausage collision object

```
52  const moveit_msgs::msg::CollisionObject& sausage = hot_dog_scenario.getSausage();
53
54  // TODO(Exercise 1-1): Attach the sausage to the robot and plan to the place pose
55  // Add your code to attach the sausage here
56  // Add your code to set the goal pose here
57
58  MoveGroupInterface::Plan place_plan;
59  uint16_t place_planning_attempts{ 0 };
60  bool place_success{ false };
61  while (!place_success && place_planning_attempts < 5)
62  {
63    // TODO(Exercise 1-1): Use MoveGroupInterface to plan to the place pose and determine planning success
64    // Add your code here
65    ++place_planning_attempts;
66  }
67
68  // Execute the plan
69  if (place_success)
70  {
71    move_group_interface.execute(place_plan);
72  }
```

Hot Dog Place

File: `exercise1/src/exercise1-1/src/main.cpp`

Helper function to get the sausage collision object

We want to pick up the sausage and plan to the place pose

```
52  const moveit_msgs::msg::CollisionObject& sausage = hot_dog_scenario.getSausage();
53
54  // TODO(Exercise 1-1): Attach the sausage to the robot and plan to the place pose
55  // Add your code to attach the sausage here
56  // Add your code to set the goal pose here
57
58  MoveGroupInterface::Plan place_plan;
59  uint16_t place_planning_attempts{ 0 };
60  bool place_success{ false };
61  while (!place_success && place_planning_attempts < 5)
62  {
63    // TODO(Exercise 1-1): Use MoveGroupInterface to plan to the place pose and determine planning success
64    // Add your code here
65    ++place_planning_attempts;
66  }
67
68  // Execute the plan
69  if (place_success)
70  {
71    move_group_interface.execute(place_plan);
72  }
```

Hot Dog Place

File: `exercise1/src/exercise1-1/src/main.cpp`

```
52  const moveit_msgs::msg::CollisionObject& sausage = hot_dog_scenario.getSausage();
53
54  // TODO(Exercise 1-1): Attach the sausage to the robot and plan to the place pose
55  // Add your code to attach the sausage here
56  // Add your code to set the goal pose here
57
58  MoveGroupInterface::Plan place_plan;
59  uint16_t place_planning_attempts{ 0 };
60  bool place_success{ false };
61  while (!place_success && place_planning_attempts < 5)
62  {
63    // TODO(Exercise 1-1): Use MoveGroupInterface to plan to the place pose and determine planning success
64    // Add your code here
65    ++place_planning_attempts;
66  }
67
68  // Execute the plan
69  if (place_success)
70  {
71    move_group_interface.execute(place_plan);
72  }
```

Helper function to get the sausage collision object

We want to pick up the sausage and plan to the place pose

We want to create a motion plan and know if planning was successful

Hot Dog Place

File: `exercise1/src/exercise1-1/src/main.cpp`

```
52    const moveit_msgs::msg::CollisionObject& sausage = hot_dog_scenario.getSausage();
53
54    // TODO(Exercise 1-1): Attach the sausage to the robot and plan to the place pose
55    // Add your code to attach the sausage here
56    // Add your code to set the goal pose here
57
58    MoveGroupInterface::Plan place_plan;
59    uint16_t place_planning_attempts{ 0 };
60    bool place_success{ false };
61    while (!place_success && place_planning_attempts < 5)
62    {
63      // TODO(Exercise 1-1): Use MoveGroupInterface to plan to the place pose and determine planning success
64      // Add your code here
65      ++place_planning_attempts;
66    }
67
68    // Execute the plan
69    if (place_success)
70    {
71      move_group_interface.execute(place_plan);
72    }
```

Attach the sausage collision object

```
move_group_interface.attachObject(sausage.id);
```

Hot Dog Place

File: `exercise1/src/exercise1-1/src/main.cpp`

```
52  const moveit_msgs::msg::CollisionObject& sausage = hot_dog_scenario.getSausage();
53
54  // TODO(Exercise 1-1): Attach the sausage to the robot and plan to the place pose
55  // Add your code to attach the sausage here
56  // Add your code to set the goal pose here
57
58  MoveGroupInterface::Plan place_plan;
59  uint16_t place_planning_attempts{ 0 };
60  bool place_success{ false };
61  while (!place_success && place_planning_attempts < 5)
62  {
63    // TODO(Exercise 1-1): Use MoveGroupInterface to plan to the place pose and determine planning success
64    // Add your code here
65    ++place_planning_attempts;
66  }
67
68  // Execute the plan
69  if (place_success)
70  {
71    move_group_interface.execute(place_plan);
72  }
```

Attach the sausage collision object

```
move_group_interface.attachObject(sausage.id);
```

Set the planning goal to `place_pose`:

```
move_group_interface.setPoseTarget(place_pose);
```

Hot Dog Place

File: `exercise1/src/exercise1-1/src/main.cpp`

```
52  const moveit_msgs::msg::CollisionObject& sausage = hot_dog_scenario.getSausage();
53
54  // TODO(Exercise 1-1): Attach the sausage to the robot and plan to the place pose
55  // Add your code to attach the sausage here
56  // Add your code to set the goal pose here
57
58  MoveGroupInterface::Plan place_plan;
59  uint16_t place_planning_attempts{ 0 };
60  bool place_success{ false };
61  while (!place_success && place_planning_attempts < 5)
62  {
63    // TODO(Exercise 1-1): Use MoveGroupInterface to plan to the place pose and determine planning success
64    // Add your code here
65    ++place_planning_attempts;
66  }
67
68  // Execute the plan
69  if (place_success)
70  {
71    move_group_interface.execute(place_plan);
72  }
```

Attach the sausage collision object

```
move_group_interface.attachObject(sausage.id);
```

Set the planning goal to `place_pose`:

```
move_group_interface.setPoseTarget(place_pose);
```

Plan and determine if planning succeeded:

```
place_success = static_cast<bool>(move_group_interface.plan(place_plan));
```

Hot Dog Place

File: `exercise1/src/exercise1-1/src/main.cpp`

```
52  const moveit_msgs::msg::CollisionObject& sausage = hot_dog_scenario.getSausage();
53
54  // TODO(Exercise 1-1): Attach the sausage to the robot and plan to the place pose
55  // Add your code to attach the sausage here
56  // Add your code to set the goal pose here
57
58  MoveGroupInterface::Plan place_plan;
59  uint16_t place_planning_attempts{ 0 };
60  bool place_success{ false };
61  while (!place_success && place_planning_attempts < 5)
62  {
63    // TODO(Exercise 1-1): Use MoveGroupInterface to plan to the place pose and determine planning success
64    // Add your code here
65    ++place_planning_attempts;
66  }
67
68  // Execute the plan
69  if (place_success)
70  {
71    move_gro
72  }
```

Attach the sausage collision object

```
move_group_interface.attachObject(sausage.id);
```

Set the planning goal to `place_pose`:

```
move_group_interface.setPoseTarget(place_pose);
```

Plan and determine if planning succeeded:

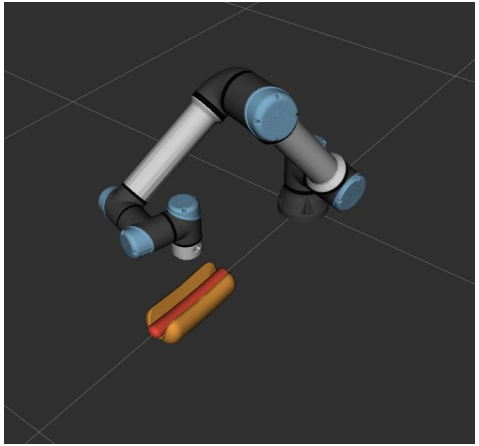
```
place_success = static_cast<bool>(move_group_interface.plan(place_plan));
```

Terminal 1: `$ colcon build`

Terminal 1: `$ ros2 launch exercise1-1 ur.launch.py`

Terminal 2: `$ ros2 launch exercise1-1 exercise1-1.launch.py`

Running the Pick and Place



- **Terminal 1:** Rebuild and restart the UR mock
  - `$ colcon build`
  - `$ ros2 launch exercise1-1 ur.launch.py`
- **Terminal 2:** Run the modified exercise 1-1:
  - `$ ros2 launch exercise1-1 exercise1-1.launch.py`
- Watch the robot plan and move to the sausage, pick it up, plan and move to the bun, and drop the sausage down in RViz
  - It may take a roundabout path due to RRTConnect
  - If you want to watch it again, or if planning fails, stop the UR mock with Ctrl+C and restart:
    - **Terminal 1:** `$ ros2 launch exercise1-1 ur.launch.py`
    - **Terminal 2:** `$ ros2 launch exercise1-1 exercise1-1.launch.py`
- Stop the UR mock with **Ctrl+C** when done

Hot Dog Pick and Place

- Congrats on building a pick and place application!
- General flow:
  - Move to pick location
  - Pick up the object
  - Move to place location
  - Place the object
- How do we actually place the object? Let’s walk through the last lines of code together.

Final Touches

File: `exercise1/src/exercise1-1/src/main.cpp`

```
78      // Detach the hot dog
79      move_group_interface.detachObject(sausage.id);
80      if (place_success)
81      {
82        // Drop the hot dog down into the bun
83        hot_dog_scenario.placeSausage();
84      }
85    }
```

Detach the sausage collision object from the end effector

Final Touches

File: `exercise1/src/exercise1-1/src/main.cpp`

```
78      // Detach the hot dog
79      move_group_interface.detachObject(sausage.id);
80      if (place_success)
81      {
82        // Drop the hot dog down into the bun
83        hot_dog_scenario.placeSausage();
84      }
85    }
```

Detach the sausage collision object from the end effector

Don’t drop the hot dog into the bun if we failed to plan the place motion

Final Touches

File: `exercise1/src/exercise1-1/src/main.cpp`

```
78      // Detach the hot dog
79      move_group_interface.detachObject(sausage.id);
80      if (place_success)
81      {
82        // Drop the hot dog down into the bun
83        hot_dog_scenario.placeSausage();
84      }
85    }
```

Detach the sausage collision object from the end effector

Don’t drop the hot dog into the bun if we failed to plan the place motion

Helper function that lets us drop the hot dog into the bun

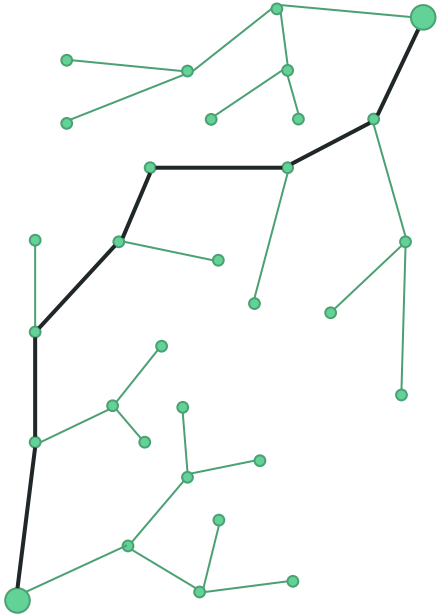
What’s missing?

- This example is somewhat simplified from real life, as it doesn’t contain a gripper. This allows us to simplify some of the nuances of grasping
- More general flow for a typical pick and place application:
  - Move to a pre-pick location
  - Allow collisions with the object to pick
  - Cartesian move from pre-pick to grasp
  - Close gripper
  - Pick up the object
  - Cartesian retreat
  - Move to pre-place location
  - Cartesian move from pre-place to place
  - Place the object
  - Cartesian retreat
  - Forbid collisions with the object
- Exercise 1-2 will cover Cartesian planning

Exercise 1-2: Hot Dog Cartesian Planning

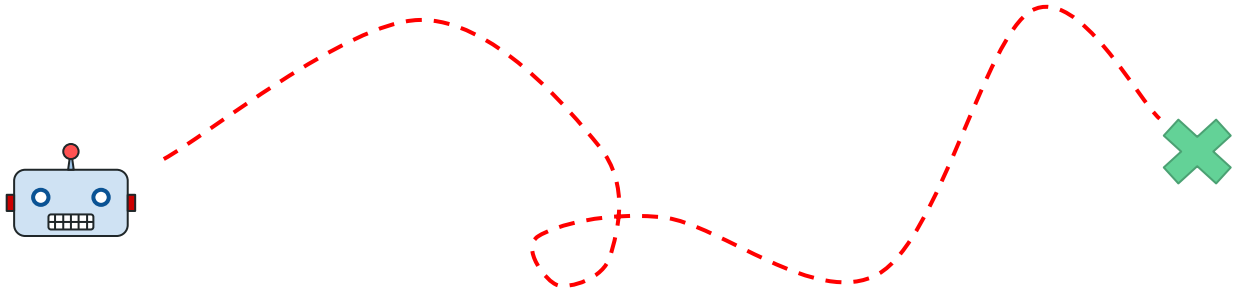
RRTConnect

- Did anyone see the robot go an unexpected direction (backward, twisting unnecessarily, under the hotdog, etc.)?
- Default planner is OMPL RRTConnect
  - Connects two trees that randomly expand
  - Fast, good at finding a solution, non-deterministic, does not return shortest path
- Can configure your planner with:
  - `move_group_interface.setPlannerId(planner_id)`
  - Planners are configured in ***_planning.yaml** (e.g. **ompl_planning.yaml**)
- Try different planners and see how they perform!



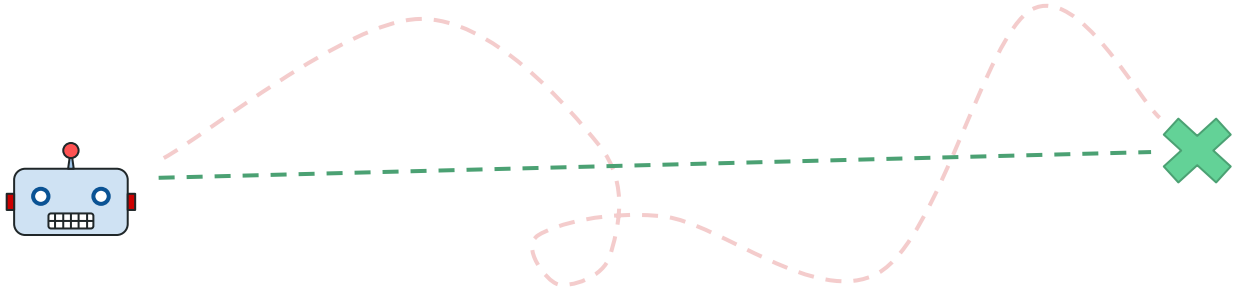
Hot Dog Cartesian Planning

- In this example, we'd like to apply mustard to our hot dog
- Wild motions from are not a particularly good fit for applying mustard
  - What a mess!
  - Want a controlled path that moves in predictable, straight lines
- Let's use Cartesian planning!



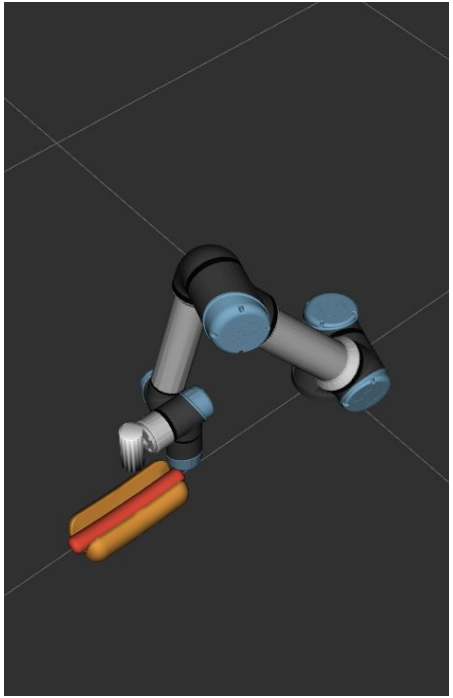
Hot Dog Cartesian Planning

- In this example, we'd like to apply mustard to our hot dog
- Wild motions from are not a particularly good fit for applying mustard
  - What a mess!
  - Want a controlled path that moves in predictable, straight lines
- Let's use Cartesian planning!



Hot Dog Cartesian Planning

- **Terminal 1**: Restart the UR mock (same as in `exercise1-1`):
  - `$ ros2 launch exercise1-2 ur.launch.py`
- **Terminal 2**: Start exercise 1-2:
  - `$ ros2 launch exercise1-2 exercise1-2.launch.py`
- Should see:
  - The assembled hot dog from exercise 1-1
  - The robot holding a mustard bottle
  - The robot should plan and move to just above the hot dog to apply mustard
- Stop all processes with **Ctrl+C**
- Let’s make Cartesian moves!



Hot Dog Cartesian Planning

File: `exercise1/src/exercise1-2/src/main.cpp`

```
26  // Create the MoveGroupInterface
27  using moveit::planning_interface::MoveGroupInterface;
28  const std::string planning_group{ "ur_manipulator" };
29  MoveGroupInterface move_group_interface(node, planning_group);
30
31  // A pose just above the bun
32  const auto& start_pose = hot_dog_scenario.getStartPose();
33  move_group_interface.setPoseTarget(start_pose);
34
35  MoveGroupInterface::Plan start_plan;
36  bool success{ false };
37  uint16_t attempts{ 0 };
38  while (!success && attempts < 5)
39  {
40    success = static_cast<bool>(move_group_interface.plan(start_plan));
41    ++attempts;
42  }
43
44  if (success)
45  {
46    move_group_interface.execute(start_plan);
```

Create a MoveGroupInterface and set the planning group

Hot Dog Cartesian Planning

File: `exercise1/src/exercise1-2/src/main.cpp`

```
26  // Create the MoveGroupInterface
27  using moveit::planning_interface::MoveGroupInterface;
28  const std::string planning_group{ "ur_manipulator" };
29  MoveGroupInterface move_group_interface(node, planning_group);
30
31  // A pose just above the bun
32  const auto& start_pose = hot_dog_scenario.getStartPose();
33  move_group_interface.setPoseTarget(start_pose);
34
35  MoveGroupInterface::Plan start_plan;
36  bool success{ false };
37  uint16_t attempts{ 0 };
38  while (!success && attempts < 5)
39  {
40    success = static_cast<bool>(move_group_interface.plan(start_plan));
41    ++attempts;
42  }
43
44  if (success)
45  {
46    move_group_interface.execute(start_plan);
```

Create a MoveGroupInterface and set the planning group

Use helper method to get the pose just above the hot dog

Hot Dog Cartesian Planning

File: `exercise1/src/exercise1-2/src/main.cpp`

```
26  // Create the MoveGroupInterface
27  using moveit::planning_interface::MoveGroupInterface;
28  const std::string planning_group{ "ur_manipulator" };
29  MoveGroupInterface move_group_interface(node, planning_group);
30
31  // A pose just above the bun
32  const auto& start_pose = hot_dog_scenario.getStartPose();
33  move_group_interface.setPoseTarget(start_pose);
34
35  MoveGroupInterface::Plan start_plan;
36  bool success{ false };
37  uint16_t attempts{ 0 };
38  while (!success && attempts < 5)
39  {
40    success = static_cast<bool>(move_group_interface.plan(start_plan));
41    ++attempts;
42  }
43
44  if (success)
45  {
46    move_group_interface.execute(start_plan);
```

Create a MoveGroupInterface and set the planning group

Use helper method to get the pose just above the hot dog

Set that pose as the planning goal

Hot Dog Cartesian Planning

File: `exercise1/src/exercise1-2/src/main.cpp`

```
26  // Create the MoveGroupInterface
27  using moveit::planning_interface::MoveGroupInterface;
28  const std::string planning_group{ "ur_manipulator" };
29  MoveGroupInterface move_group_interface(node, planning_group);
30
31  // A pose just above the bun
32  const auto& start_pose = hot_dog_scenario.getStartPose();
33  move_group_interface.setPoseTarget(start_pose);
34
35  MoveGroupInterface::Plan start_plan;
36  bool success{ false };
37  uint16_t attempts{ 0 };
38  while (!success && attempts < 5)
39  {
40    success = static_cast<bool>(move_group_interface.plan(start_plan));
41    ++attempts;
42  }
43
44  if (success)
45  {
46    move_group_interface.execute(start_plan);
```

Create a MoveGroupInterface and set the planning group

Use helper method to get the pose just above the hot dog

Set that pose as the planning goal

Solve for a motion plan (similarly to exercise 1-1) and check for success

Hot Dog Cartesian Planning

File: `exercise1/src/exercise1-2/src/main.cpp`

```
26  // Create the MoveGroupInterface
27  using moveit::planning_interface::MoveGroupInterface;
28  const std::string planning_group{ "ur_manipulator" };
29  MoveGroupInterface move_group_interface(node, planning_group);
30
31  // A pose just above the bun
32  const auto& start_pose = hot_dog_scenario.getStartPose();
33  move_group_interface.setPoseTarget(start_pose);
34
35  MoveGroupInterface::Plan start_plan;
36  bool success{ false };
37  uint16_t attempts{ 0 };
38  while (!success && attempts < 5)
39  {
40    success = static_cast<bool>(move_group_interface.plan(start_plan));
41    ++attempts;
42  }
43
44  if (success)
45  {
46    move_group_interface.execute(start_plan);
```

Create a MoveGroupInterface and set the planning group

Use helper method to get the pose just above the hot dog

Set that pose as the planning goal

Solve for a motion plan (similarly to exercise 1-1) and check for success

Execute if successful and move just above the hot dog, ready to apply mustard

Hot Dog Cartesian Planning

File: `exercise1/src/exercise1-2/src/main.cpp`

Helper method to get a vector of poses in a zig-zag to apply our mustard

```
48    const std::vector<geometry_msgs::msg::Pose> waypoints = hot_dog_scenario.getMustardWaypoints(start_pose);
49
50    moveit_msgs::msg::RobotTrajectory trajectory;
51
52    // TODO(Exercise 1-2): Plan a Cartesian path
53    double fraction = 0;
54
55    if (fraction > 0)
56    {
57      move_group_interface.execute(trajectory);
58      hot_dog_scenario.applyMustard();
59    }
```

Hot Dog Cartesian Planning

File: `exercise1/src/exercise1-2/src/main.cpp`

```
48    const std::vector<geometry_msgs::msg::Pose> waypoints = hot_dog_scenario.getMustardWaypoints(start_pose);
49
50    moveit_msgs::msg::RobotTrajectory trajectory;
51
52    // TODO(Exercise 1-2): Plan a Cartesian path
53    double fraction = 0;
54
55    if (fraction > 0)
56    {
57      move_group_interface.execute(trajectory);
58      hot_dog_scenario.applyMustard();
59    }
```

Helper method to get a vector of poses in a zig-zag to apply our mustard

We want to create a Cartesian plan through our list of poses and we want to know what fraction of the path we were successful in planning

Hot Dog Cartesian Planning

File: `exercise1/src/exercise1-2/src/main.cpp`

```
48  const std::vector<geometry_msgs::msg::Pose> waypoints = hot_dog_scenario.getMustardWaypoints(start_pose);
49
50  moveit_msgs::msg::RobotTrajectory trajectory;
51
52  // TODO(Exercise 1-2): Plan a Cartesian path
53  double fraction = 0;
54
55  if (fraction > 0)
56  {
57    move_group_interface.execute(trajectory);
58    hot_dog_scenario.applyMustard();
59  }
```

Helper method to get a vector of poses in a zig-zag to apply our mustard

We want to create a Cartesian plan through our list of poses and we want to know what fraction of the path we were successful in planning

Execute the trajectory to apply mustard!

Hot Dog Cartesian Planning

File: `exercise1/src/exercise1-2/src/main.cpp`

```
48  const std::vector<geometry_msgs::msg::Pose> waypoints = hot_dog_scenario.getMustardWaypoints(start_pose);
49
50  moveit_msgs::msg::RobotTrajectory trajectory;
51
52  // TODO(Exercise 1-2): Plan a Cartesian path
53  double fraction = 0;
54
55  if (fraction > 0)
56  {
57    move_group_interface.execute(trajectory);
58    hot_dog_scenario.applyMustard();
59  }
```

Helper method to get a vector of poses in a zig-zag to apply our mustard

We want to create a Cartesian plan through our list of poses and we want to know what fraction of the path we were successful in planning

Execute the trajectory to apply mustard!

Helper method to show mustard visualization

Hot Dog Cartesian Planning

File: `exercise1/src/exercise1-2/src/main.cpp`

```
48  const std::vector<geometry_msgs::msg::Pose> waypoints = hot_dog_scenario.getMustardWaypoints(start_pose);
49
50  moveit_msgs::msg::RobotTrajectory trajectory;
51
52  // TODO(Exercise 1-2): Plan a Cartesian path
53  double fraction = 0;
54
55  if (fraction > 0)
56  {
57    move_group_interface.execute(trajectory);
58    hot_dog_scenario.applyMustard();
59  }
```

Remove the 0 and replace it with:

```
move_group_interface.computeCartesianPath(waypoints,
0.002, 0.0, trajectory);
```

Hot Dog Cartesian Planning

File: `exercise1/src/exercise1-2/src/main.cpp`

```
48  const std::vector<geometry_msgs::msg::Pose> waypoints = hot_dog_scenario.getMustardWaypoints(start_pose);
49
50  moveit_msgs::msg::RobotTrajectory trajectory;
51
52  // TODO(Exercise 1-2): Plan a Cartesian path
53  double fraction = 0;
54
55  if (fraction > 0)
56  {
57    move_group_interface.execute(trajectory);
58    hot_dog_scenario.applyMustard();
59  }
```

Remove the 0 and replace it with:

```
move_group_interface.computeCartesianPath(waypoints,
0.002, 0.0, trajectory);
```

Terminal 1: `$ colcon build`

Terminal 1: `$ ros2 launch exercise1-2 ur.launch.py`

Terminal 2: `$ ros2 launch exercise1-2 exercise1-2.launch.py`

Hot Dog Cartesian Planning

File: `exercise1/src/exercise1-2/src/main.cpp`

```
48  const std::vector<geometry_msgs::msg::Pose> waypoints = hot_dog_scenario.getMustardWaypoints(start_pose);
49
50  moveit_msgs::msg::RobotTrajectory trajectory;
51
52  // TODO(Exercise 1-2): Plan a Cartesian path
53  double fraction = 0;
54
55  if (fraction > 0)
56  {
57    move_group_interface.execute(trajectory);
58    hot_dog_scenario.applyMustard();
59  }
```

Remove the 0 and replace it with:

```
move_group_interface.computeCartesianPath(waypoints, 0.002, 0.0, trajectory);
```

`eef_step` defines the distance in meters between end effector configurations of consecutive points in the result trajectory

Terminal 1: `$ colcon build`

Terminal 1: `$ ros2 launch exercise1-2 ur.launch.py`

Terminal 2: `$ ros2 launch exercise1-2 exercise1-2.launch.py`

Hot Dog Cartesian Planning

File: `exercise1/src/exercise1-2/src/main.cpp`

```
48  const std::vector<geometry_msgs::msg::Pose> waypoints = hot_dog_scenario.getMustardWaypoints(start_pose);
49
50  moveit_msgs::msg::RobotTrajectory trajectory;
51
52  // TODO(Exercise 1-2): Plan a Cartesian path
53  double fraction = 0;
54
55  if (fraction > 0)
56  {
57    move_group_interface.execute(trajectory);
58    hot_dog_scenario.applyMustard();
59  }
```

Remove the 0 and replace it with:

```
move_group_interface.computeCartesianPath(waypoints, 0.002, 1.5, trajectory);
```

`eef_step` defines the distance in meters between end effector configurations of consecutive points in the result trajectory

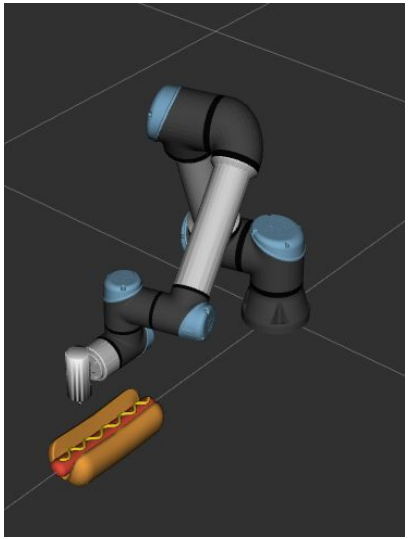
`jump_threshold` limits joint space jumps, which can be large even if the Cartesian distance between the points is small

Terminal 1: `$ colcon build`

Terminal 1: `$ ros2 launch exercise1-2 ur.launch.py`

Terminal 2: `$ ros2 launch exercise1-2 exercise1-2.launch.py`

Running the Cartesian Planning Example

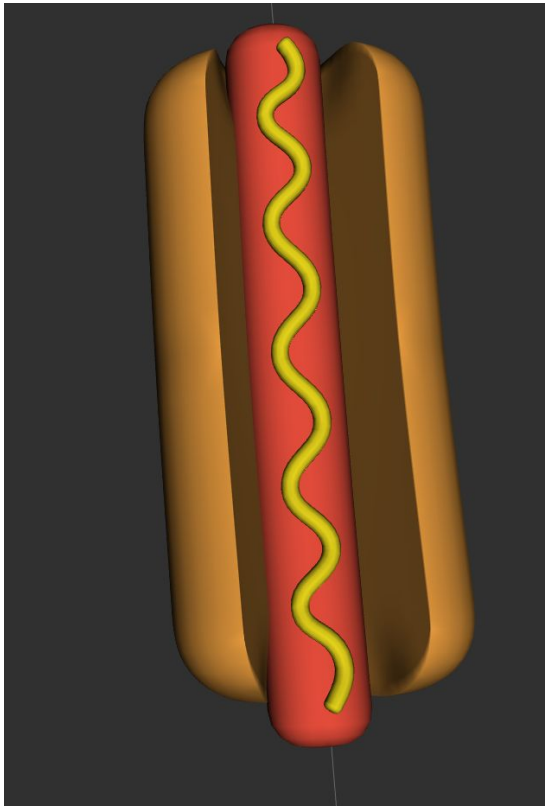


- Rebuild:
  - `$ colcon build`
- Restart the UR mock:
  - `$ ros2 launch exercise1-2 ur.launch.py`
- Run the modified exercise 1-2:
  - `$ ros2 launch exercise1-2 exercise1-2.launch.py`
- Robot should plan and move to the hot dog using RRTConnect, then move through the mustard zig-zag with Cartesian motions
  - If you want to watch it again, or if planning fails, stop the UR mock with Ctrl+C and restart:
    - **Terminal 1:** `$ ros2 launch exercise1-2 ur.launch.py`
    - **Terminal 2:** `$ ros2 launch exercise1-2 exercise1-2.launch.py`
- Stop the UR mock with **Ctrl+C** when done

Exercise 1 Summary

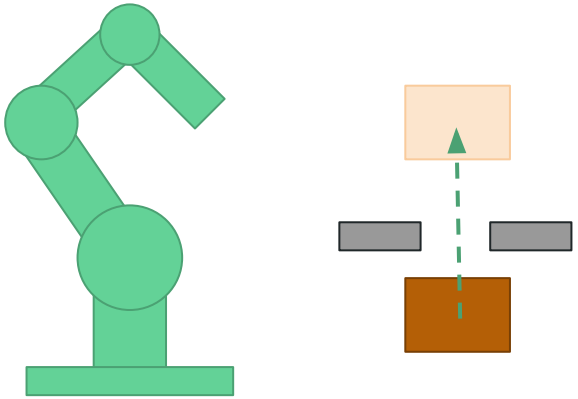
What did we learn?

- We used MoveIt’s MoveGroupInterface to make motion plans
  - Set the goal pose
  - Plan, check for success, and execute
  - Attaching collision objects
  - Detaching collision objects
- We also use MoveGroupInterface to make Cartesian motion plans
  - Pipeline planner isn’t always appropriate
    - Pipeline planner can be configured
  - Cartesian interpolator can be used for straight-line motions



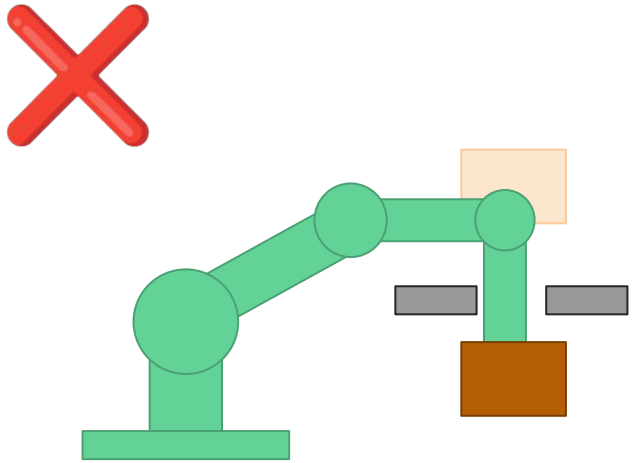
What else should we know?

- MoveIt's Cartesian interpolator has limitations
  - Subject to collisions
    - Cannot avoid collisions, will fail
  - Inverse kinematics
- Most tasks consist of a series of motion plans
  - Plan, if successful, plan again, if successful, plan again...
  - What happens if we fail to plan part way through?
  - What happens if we plan to a joint state that makes the next motion difficult or impossible?



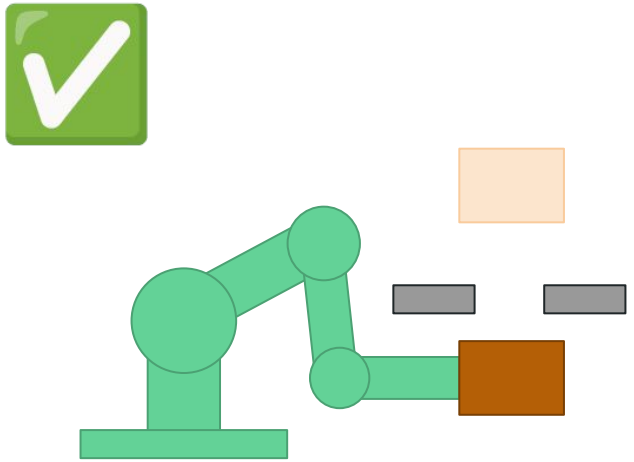
What else should we know?

- MoveIt's Cartesian interpolator has limitations
  - Subject to collisions
    - Cannot avoid collisions, will fail
  - Inverse kinematics
- Most tasks consist of a series of motion plans
  - Plan, if successful, plan again, if successful, plan again...
  - What happens if we fail to plan part way through?
  - What happens if we plan to a joint state that makes the next motion difficult or impossible?



What else should we know?

- MoveIt's Cartesian interpolator has limitations
  - Subject to collisions
    - Cannot avoid collisions, will fail
  - Inverse kinematics
- Most tasks consist of a series of motion plans
  - Plan, if successful, plan again, if successful, plan again...
  - What happens if we fail to plan part way through?
  - What happens if we plan to a joint state that makes the next motion difficult or impossible?



Motion Planning and Manipulation

- MoveIt is a **motion planning** framework
  - Enables you to create motion plans to move your robot from one point to another
- Manipulation tasks generally require a series of connected motions and motion plans
  - Pick and place, material handling, packaging, assembly...
- Motion planning can and will fail
  - Reliable systems require logic to handle failures
- MoveIt is one piece of the manipulation puzzle

Manipulation is not just Motion Planning

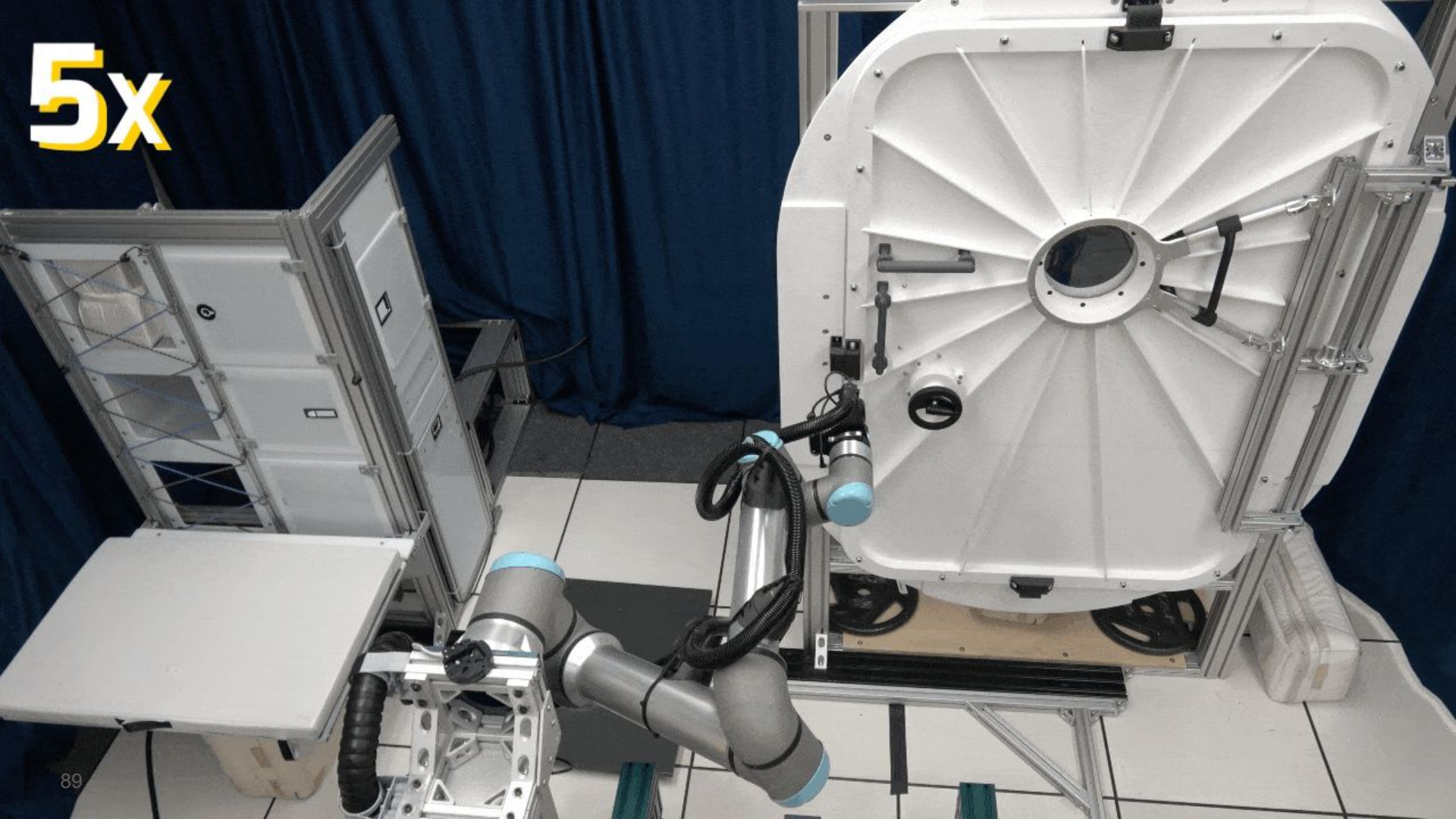
Motion Planning != Manipulation

What is Motion Planning?

“Motion planning is the problem of finding a robot motion from a start state to a goal state that avoids obstacles in the environment and satisfies other constraints” (Lynch, K. M., & Park, F. C. (2017))

Robotic manipulation refers to technical approaches that enable robots to interact with objects in the real world. In addition to motion planning, manipulation may include **sensor processing**, **visual perception**, **higher level reasoning**, **motion controls**, **modeling of process dynamics**, and **handling of errors** or uncertainties.

PICKNIK

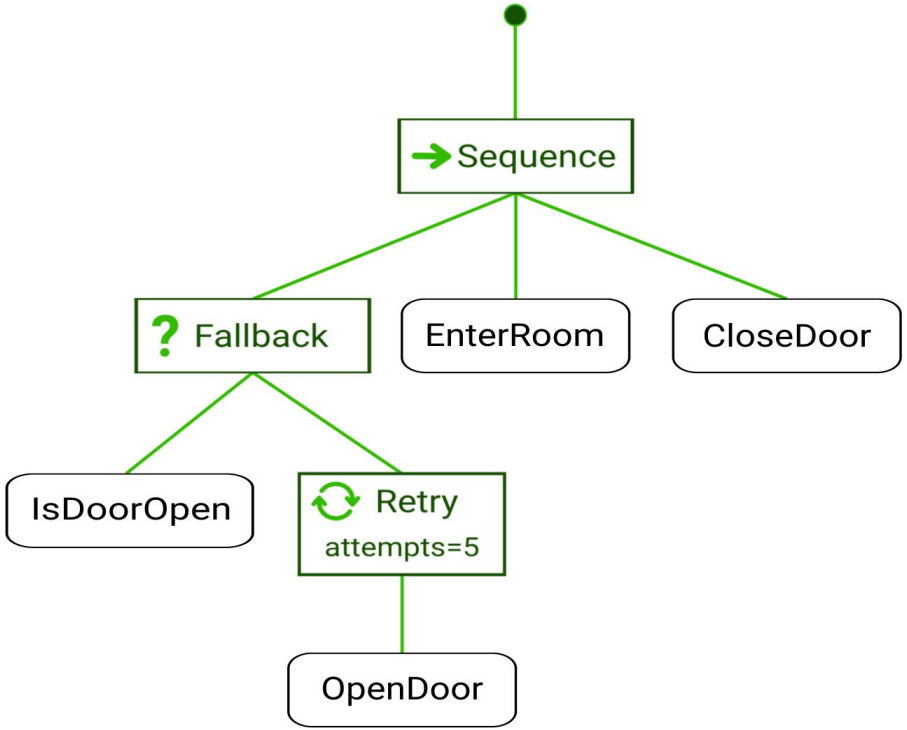

5x

Manipulation Challenges (Example)

- **High-level runtime logic**
  - integrates all these components and enables error handling
- **Force sensing & Perception**
  - detecting objects, door, valve, door handle
  - noticing when valve is open or closed
- **Controller Switching**
  - some trajectories need to tolerate external and unpredictable forces using admittance/impedance control
    - opening and closing of door and lid, turning of valve
  - some paths can be controlled in Cartesian space, others in joint space
- **Multi-step Motion Planning**
  - rearranging the arm while opening and closing the door

PICKNIK

Runtime Logic - Behavior Trees



PICKNIK

Runtime Logic - Behavior Trees

ControlNode

- controls ticking of multiple child nodes

DecoratorNode

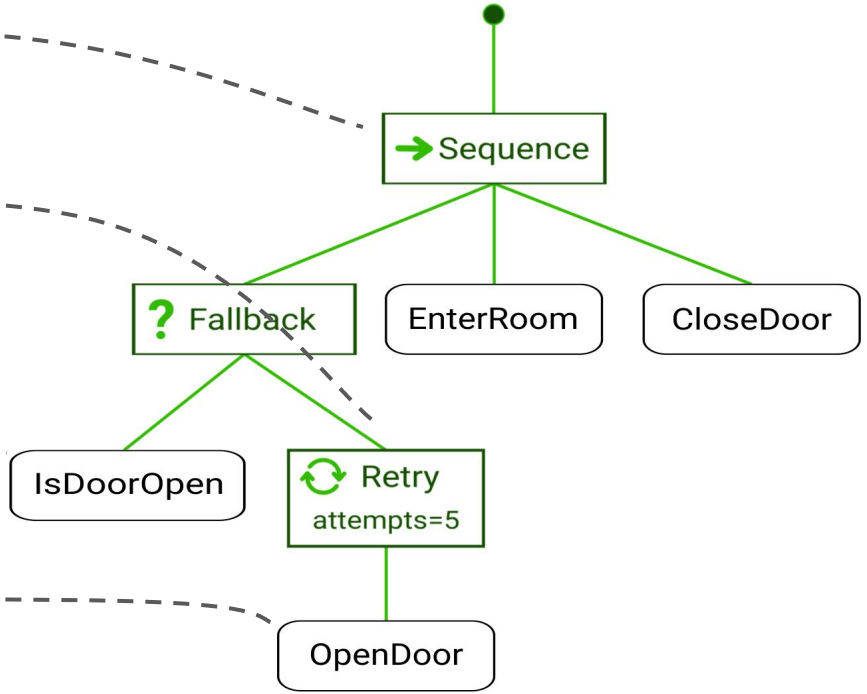
- ticks and processes result of single child node

ConditionNode

- Evaluates a condition, no child nodes

ActionNode

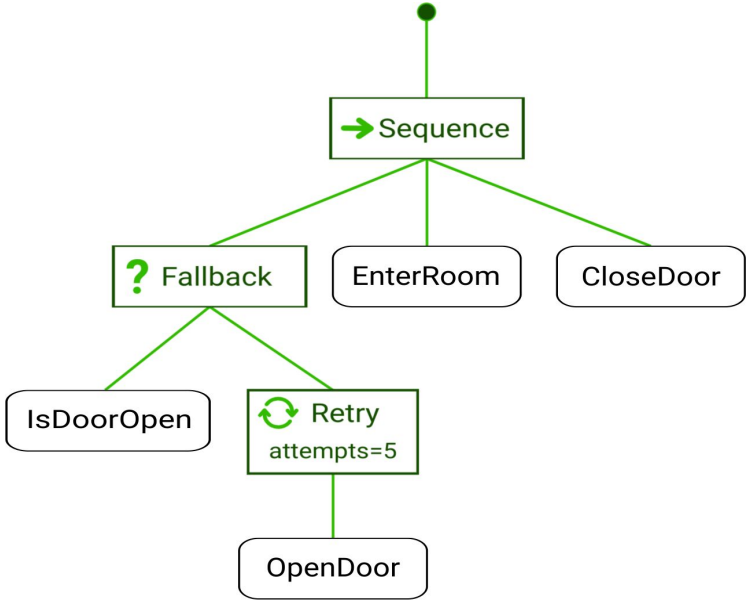
- “does something”, no child nodes



Runtime Logic - Behavior Trees

Behavior trees

- model hierarchical behaviors using composable tree structures
- are built of runtime actions (tree nodes) which can be executed asynchronously
- actions have defined SUCCESS, RUNNING, FAILURE results
- support shared data access (blackboard) for actions through in- and output ports
- MoveIt Pro uses **BehaviorTree.CPP**
  - great ROS support
  - simple XML file structure

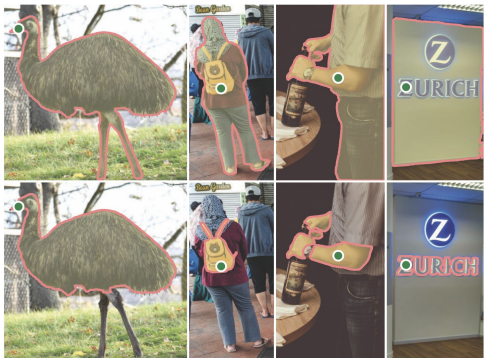


PICKNIK

Integrating sensor and vision feedback

ROS message interfaces already provide a great solution for this!

- isolate sensor/perception in separate ROS node
- implement topic/service/action client inside *ActionNode*
- share processed data with the remaining logic via *blackboard*



This also solves a very common design problem:

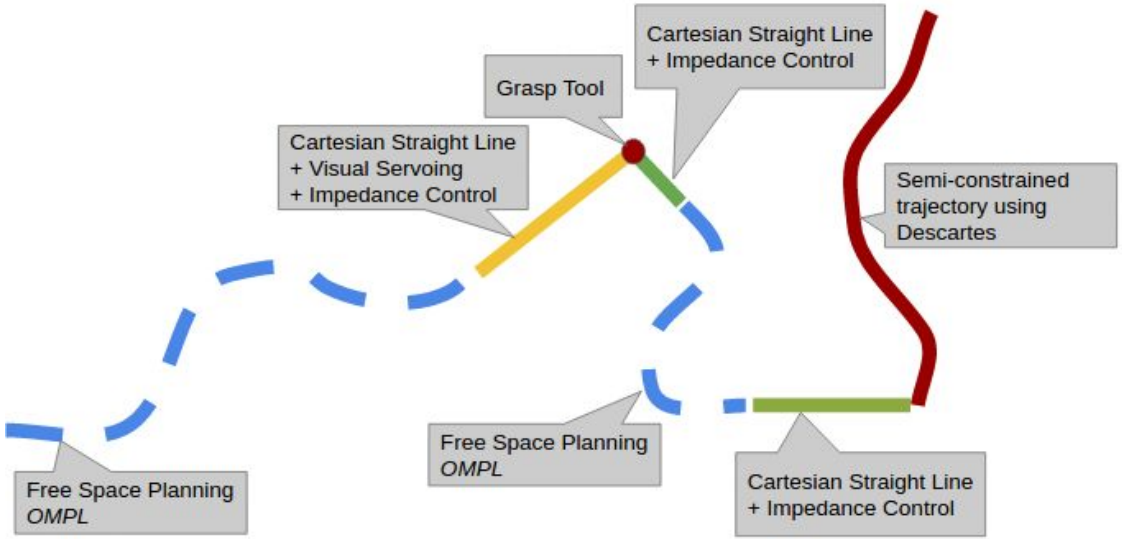
- tightly coupled application logic and ROS callbacks, making refactoring and testing harder

PICKNIK

Controller Switching - ros2_control

ros2_control

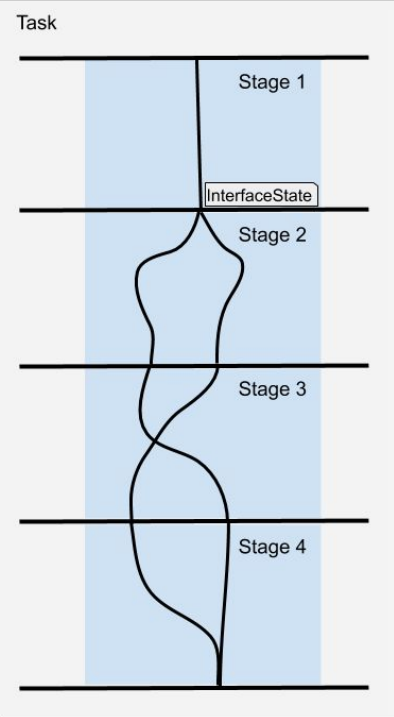
- framework for managing hardware resources and controllers
- multiple controllers can be configured and registered for the same hardware
- conflict-free (de)activation or “switching” of controllers is realized using a ROS service



Multi-step Motion Planning - Moveit Task Constructor

MoveIt Task Constructor (MTC)

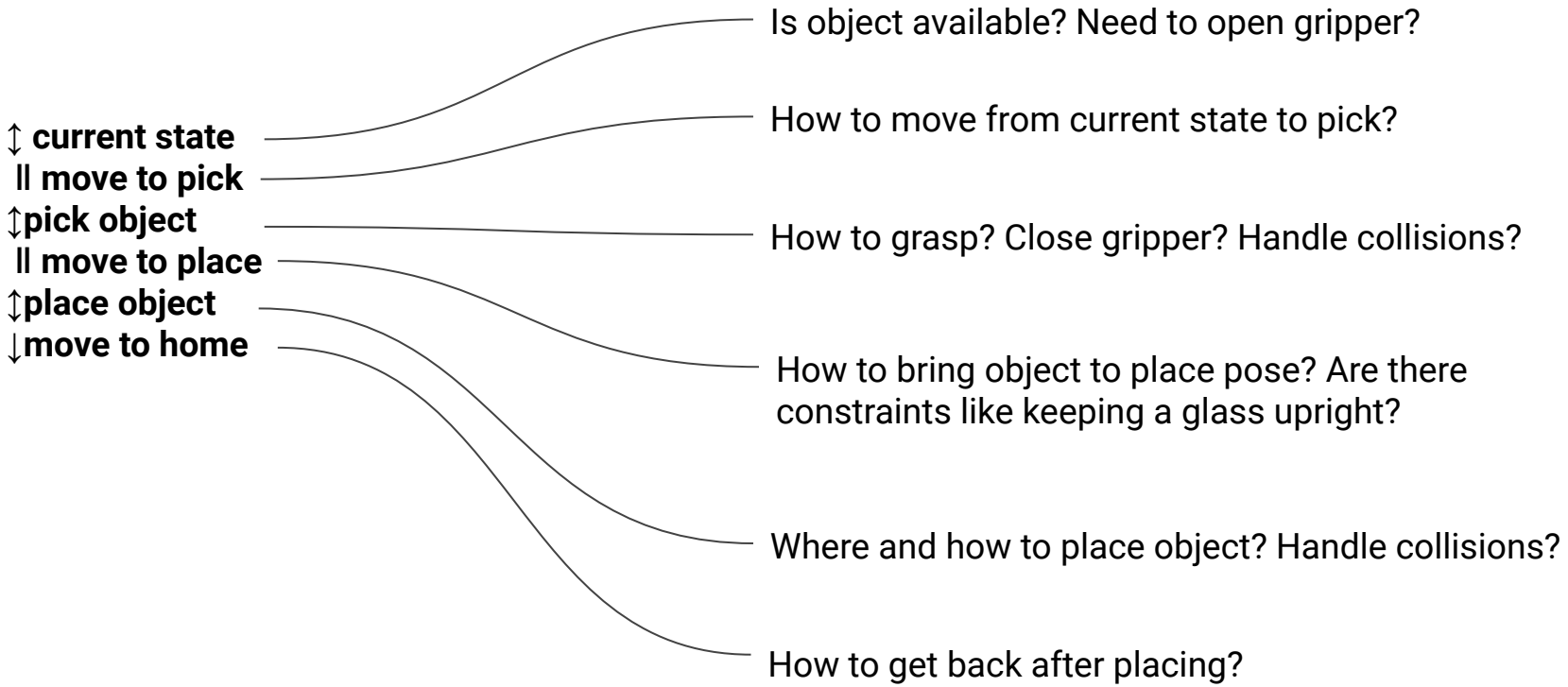
- defines a single motion plan from many subproblems
- solves each subproblem individually
- computes connected start-to-end trajectories, optimized with the lowest cost over all subproblems
- does not support “reacting” or “fixing” of computed solutions during execution



https://moveit.github.io/moveit_task_constructor/

PICKNIK

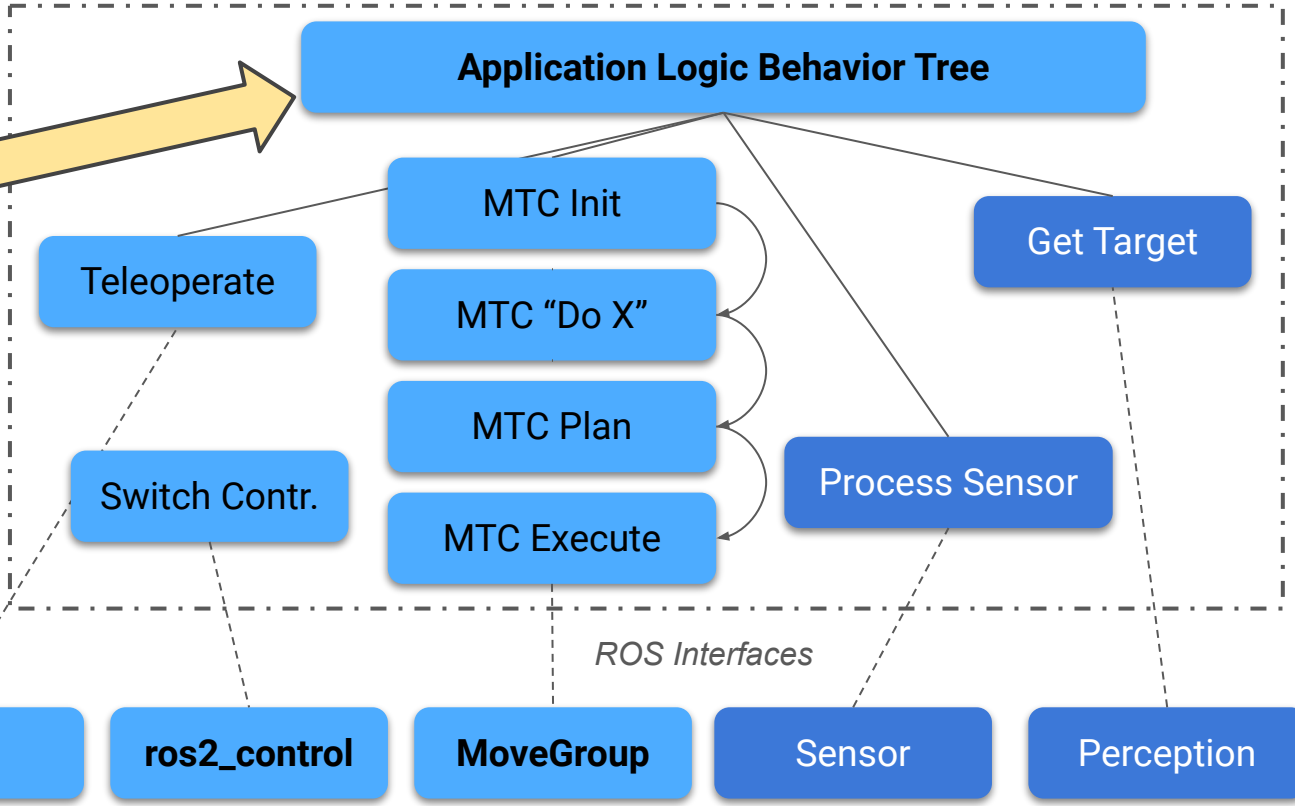
Multi-step Motion Planning - MTC example



https://moveit.github.io/moveit_task_constructor/

PICKNIK

Architecture: Spaghetti vs Separation of Concerns



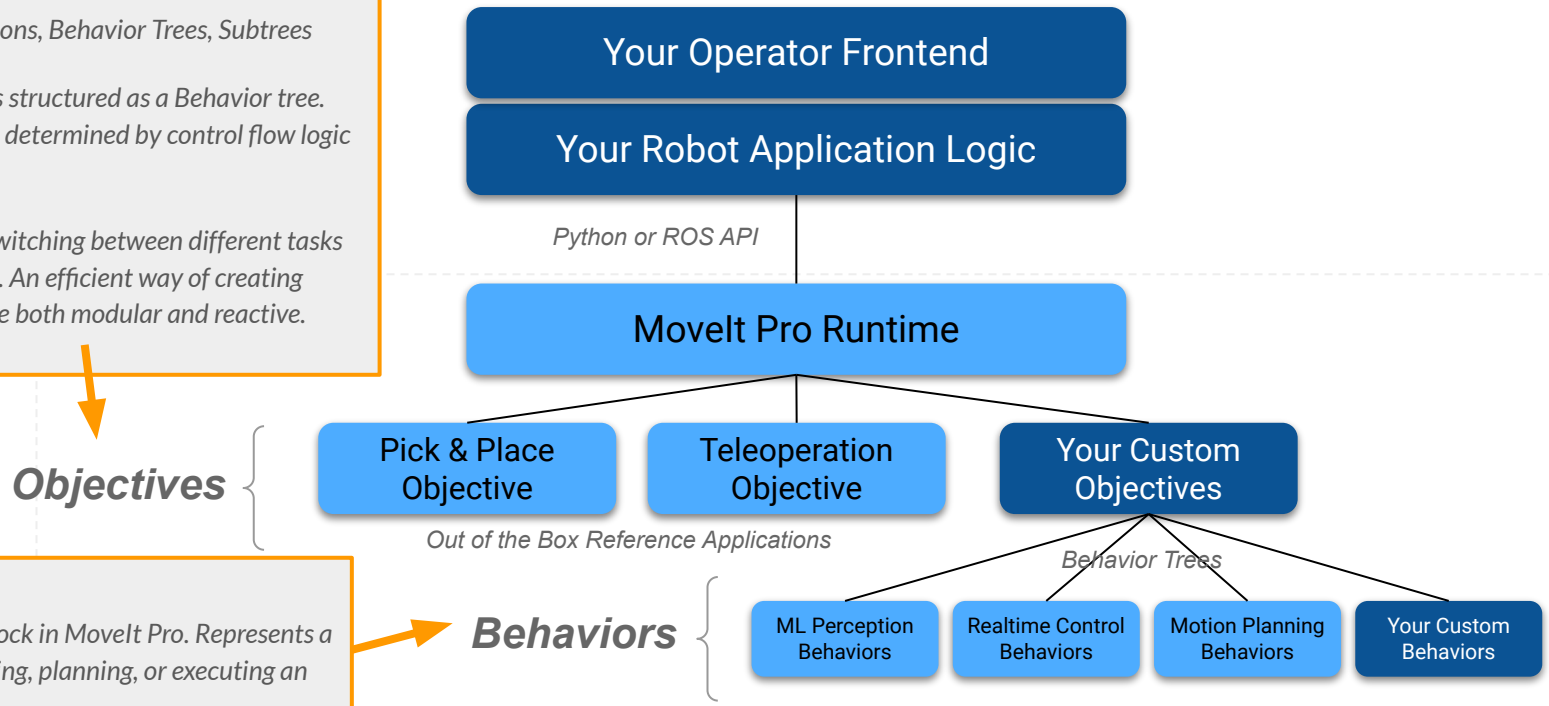
The MoveIt Pro Approach

Objectives
Also known as: Applications, Behavior Trees, Subtrees

A collection of Behaviors structured as a Behavior tree. The order of execution is determined by control flow logic within the tree.

A way to structure the switching between different tasks in an autonomous agent. An efficient way of creating complex systems that are both modular and reactive.

Behaviors
The smallest building block in MoveIt Pro. Represents a discrete action like sensing, planning, or executing an action.



PICKNIK

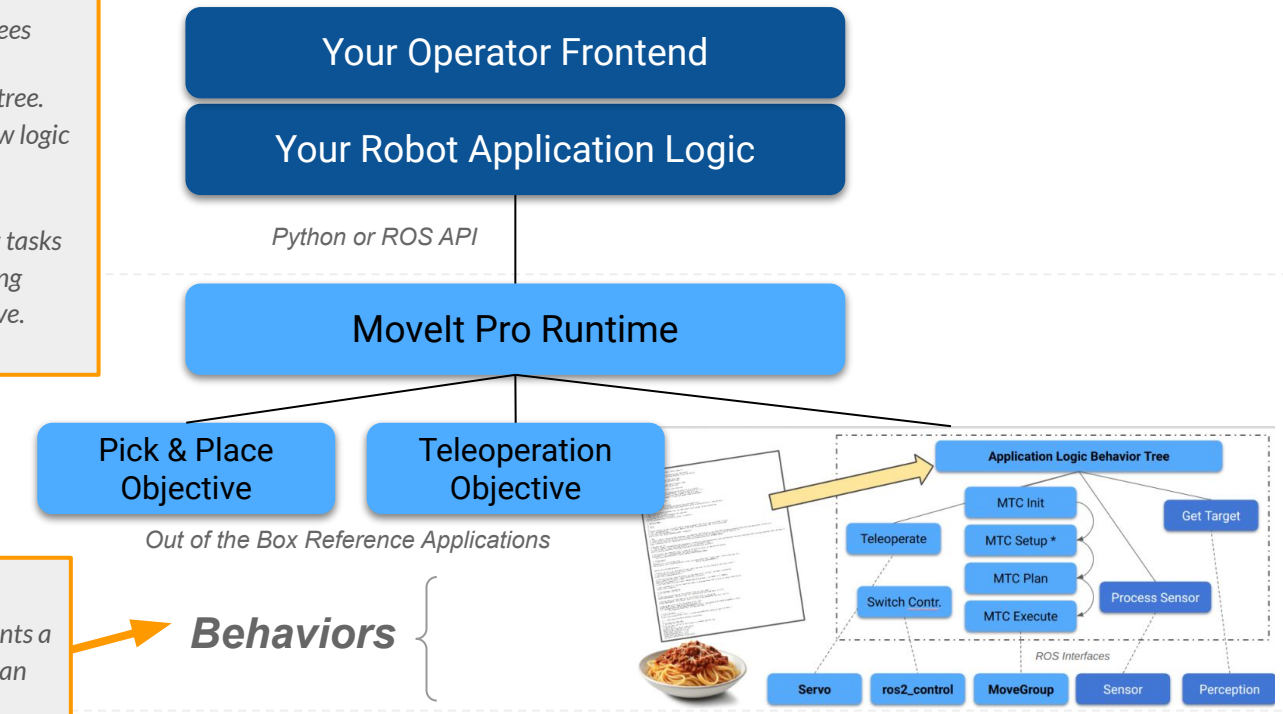
The MoveIt Pro Approach

Objectives
Also known as: Applications, Behavior Trees, Subtrees

A collection of Behaviors structured as a Behavior tree. The order of execution is determined by control flow logic within the tree.

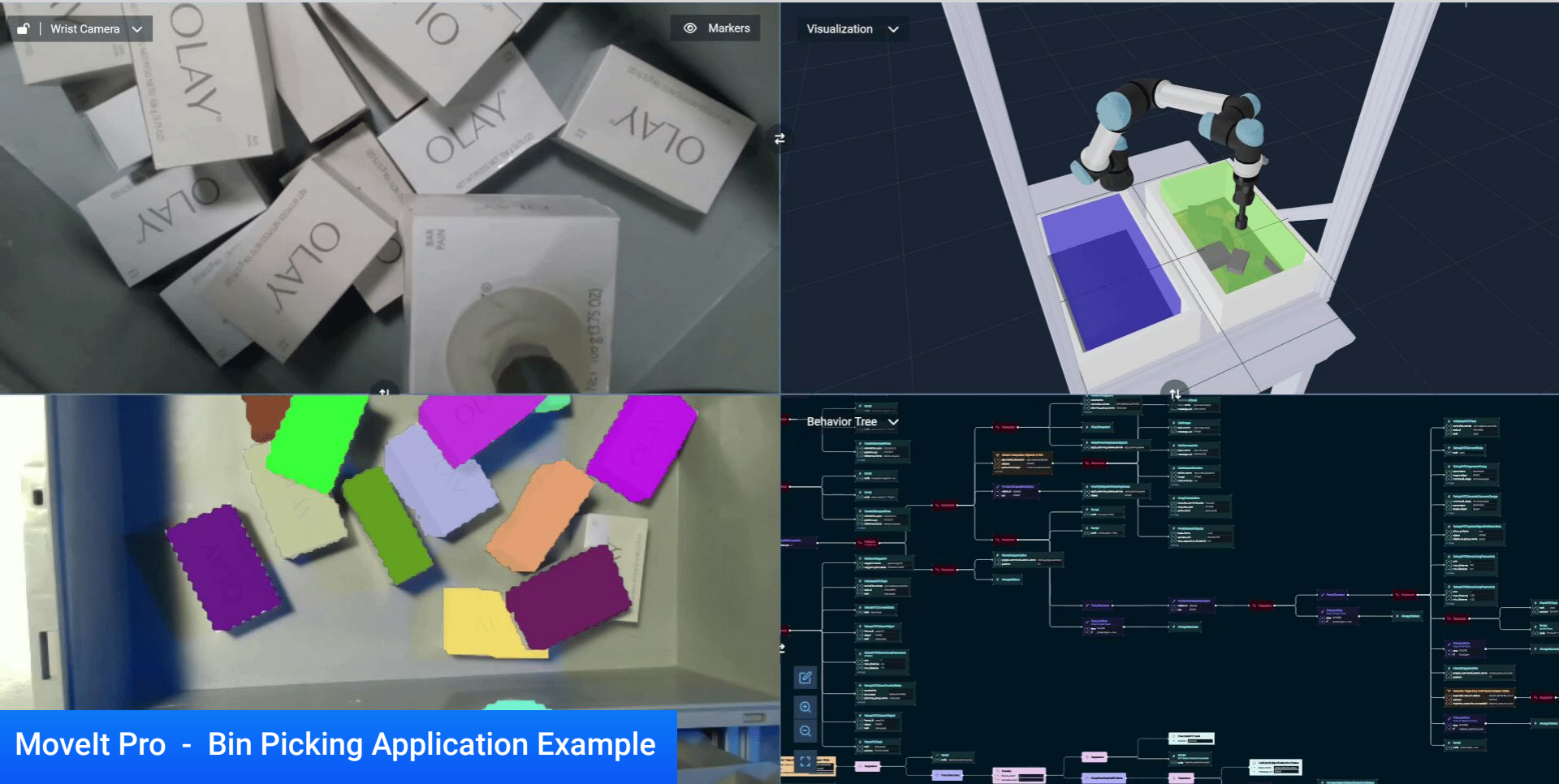
A way to structure the switching between different tasks in an autonomous agent. An efficient way of creating complex systems that are both modular and reactive.

Behaviors
The smallest building block in MoveIt Pro. Represents a discrete action like sensing, planning, or executing an action.



PICKNIK

Pick and Place example



MoveIt Pro - Bin Picking Application Example

Reminder: Install MoveIt Pro now if you haven't already

- **Follow the install instructions on [Quick Start | MoveIt Pro]**
- You should have received an individual **license key** for this workshop via email
- Execute **moveit_pro run** to configure the workspace and build the required images
- If you run into technical difficulties, let us know and we will try our best to fix them
- If you don't have a working installation, please group up with someone who does

PICKNIK

Coffee Time!

PICKNIK

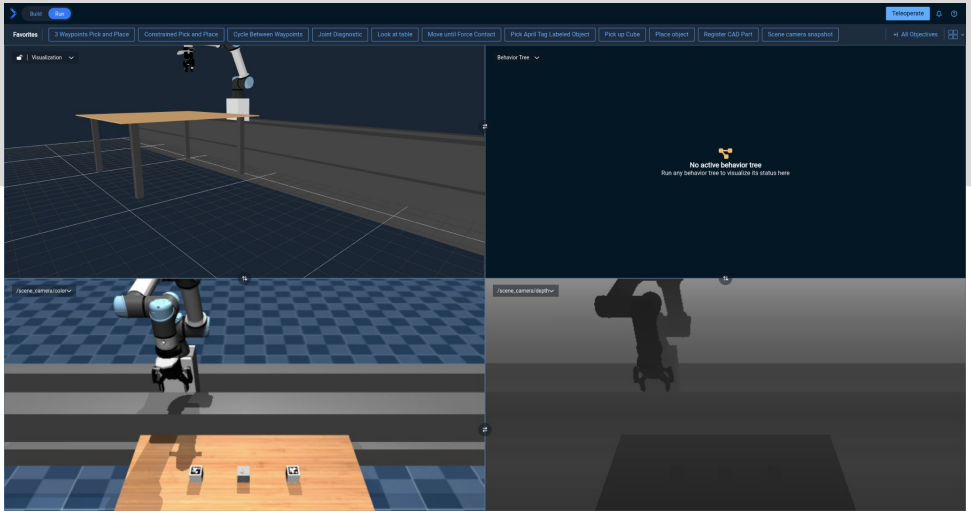
Exercise 2: External Framework Integration with Pick and Place

Quick Start

An overview of the out of the box functionality

PICKNIK

Starting MoveIt Pro



Open a terminal and run:

```
moveit_pro run
```

The first time you run this command, it enters into a “Quick Start” setup mode that will prompt you with a series of setup questions. Choose all the recommended answers:

```
Do you want to download the default workspace now? [Y/n]
Do you want to rebuild the user workspace (recommended)? [Y/n]
Enter the name of the configuration package: arm_on_rail_sim
```

After loading you should see the screen on the right. Ask for help if you run into issues here.

In the future, you can simply launch the same robot config directly via:

```
moveit_pro run
```

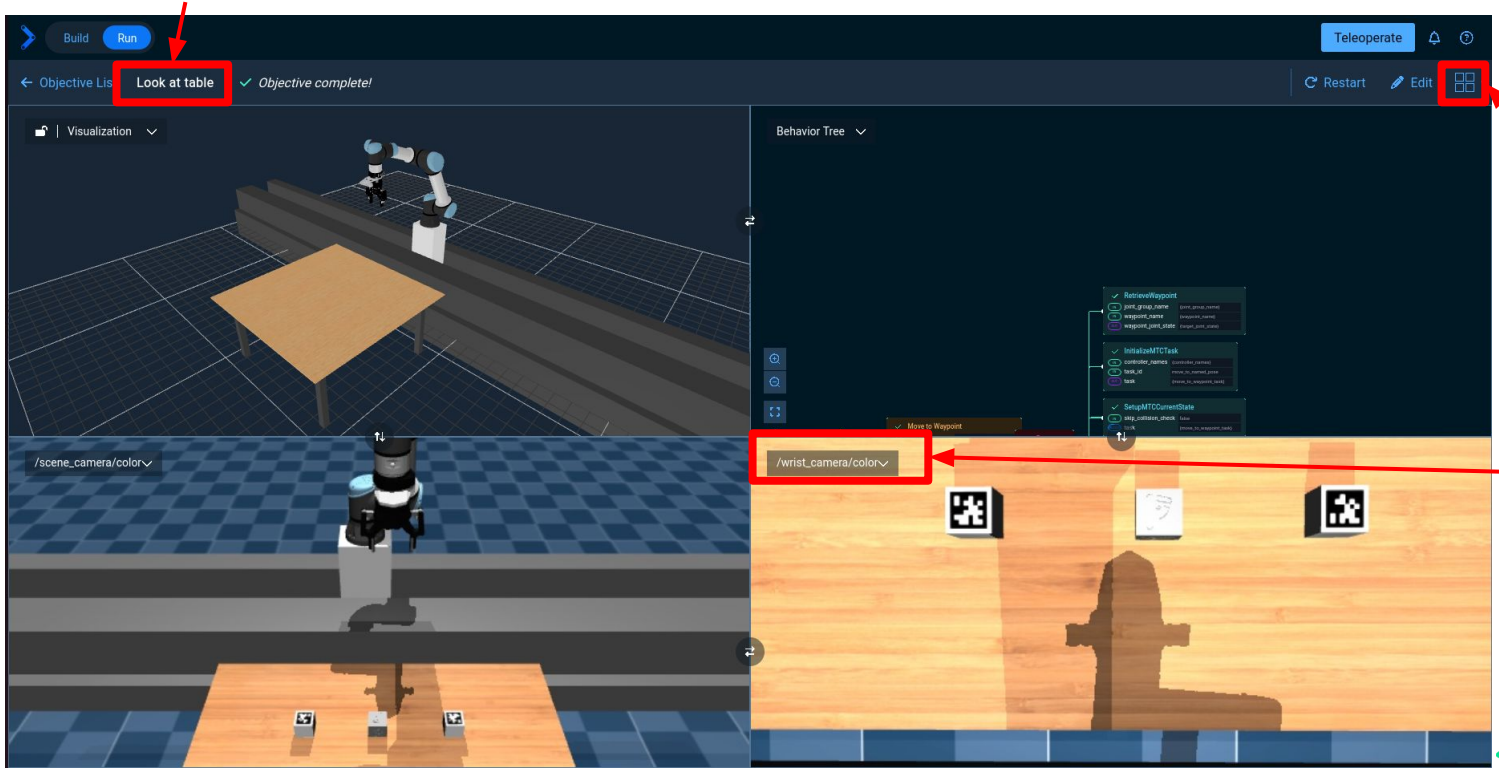
Or more explicitly (in case you have changed robot configs):

```
moveit_pro run -c arm_on_rail_sim
```


PICKNIK

Run your first objective

Run “Look at table” Objective



Try some different layouts

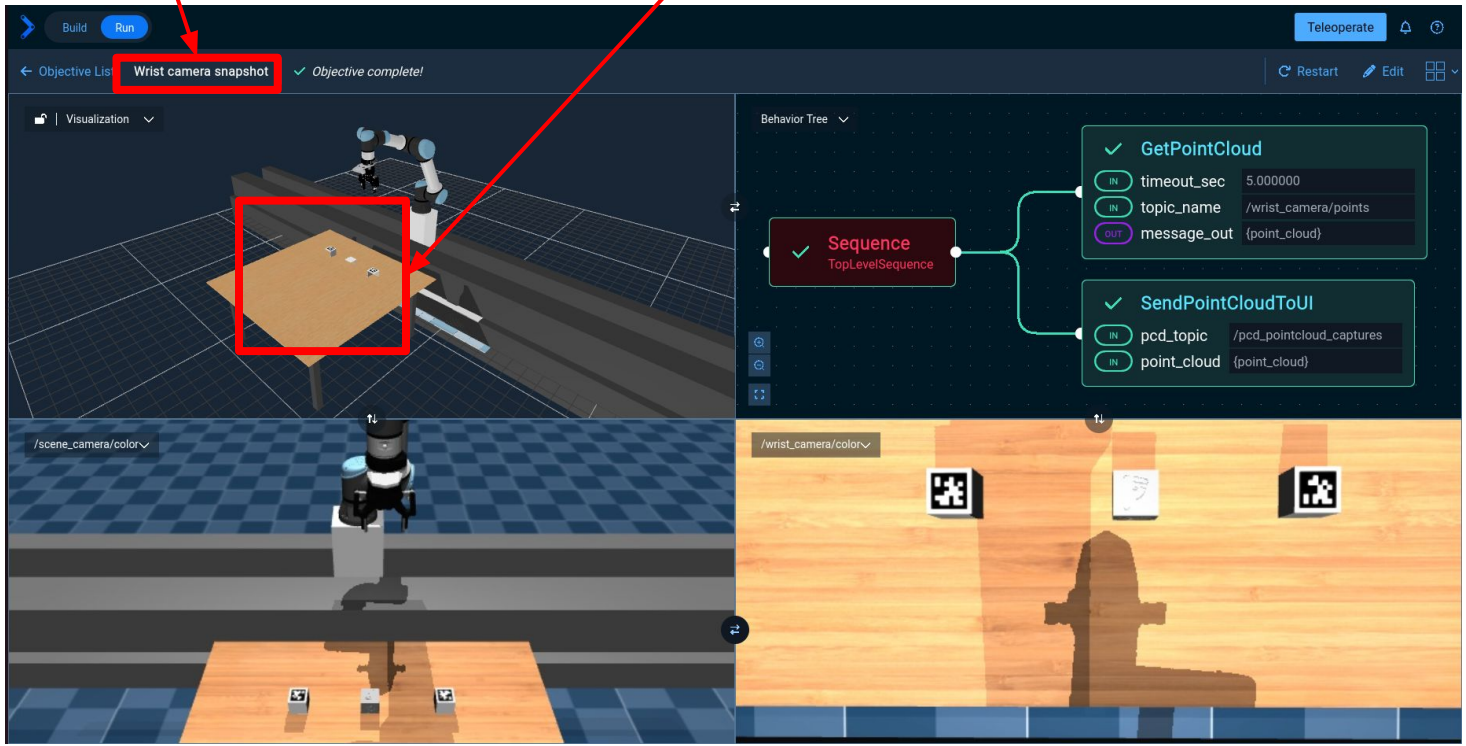
Select your cameras

PICKNIK

Use a camera to take a snapshot

Run “Wrist camera snapshot” Objective

The cubes should appear in the visualization



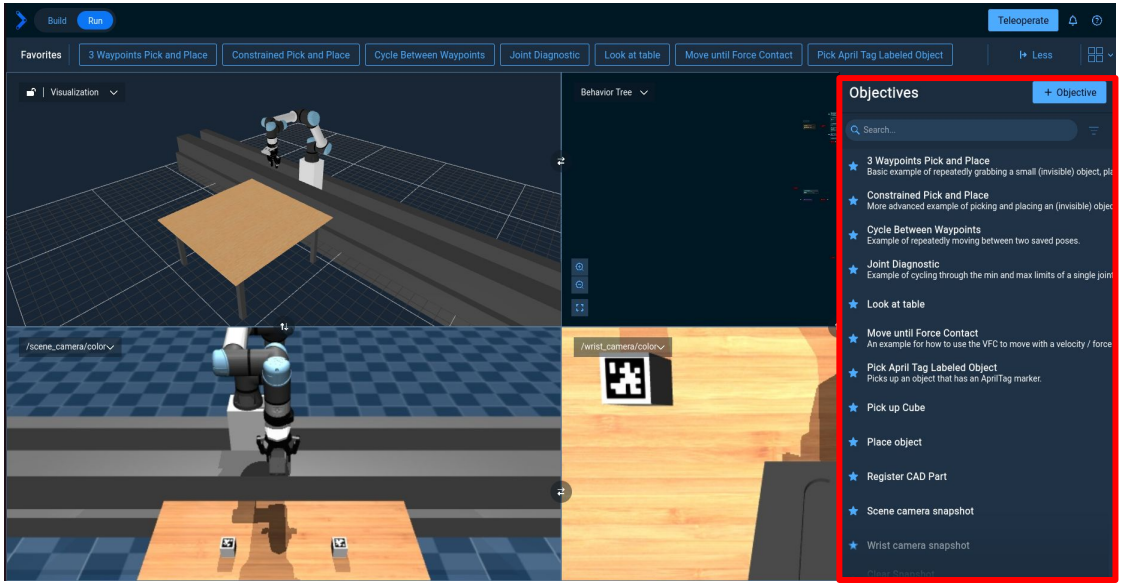
PICKNIK

Try out more example Objectives

Before diving in deeper, let's see some more Objectives in action:

Try the following:

- Pick cube
- Place cube
- Open / Close Gripper
- Scene camera snapshot
- Clear snapshot
- 3 Waypoints Pick and Place
    - Use "Stop Motion" button



Recommend point in training: 1hr 30min

PICKNIK

Cancel/Stop your Objective

- What can we do to get the robot back to a known pose?

PICKNIK

Navigating the User Interface

View and execute **Objectives**
(behavior trees that perform a task)

Create and modify behavior trees (**Objectives)** using a library of **Behaviors** (behavior tree nodes)

You can also write your own custom Behaviors in C++

Control logging levels

Help and info

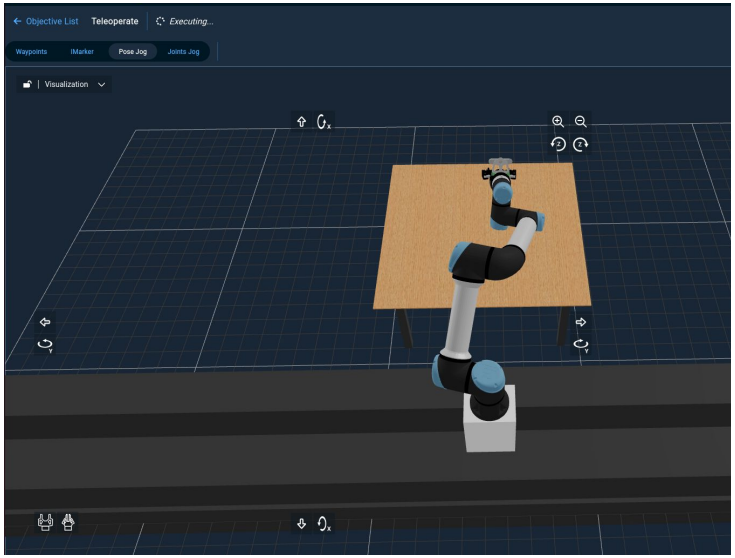
Switch to various manual control modalities, including:

- Cartesian jogging
- Joint jogging
- IMarker control
- Waypoint execution
- Waypoint editing

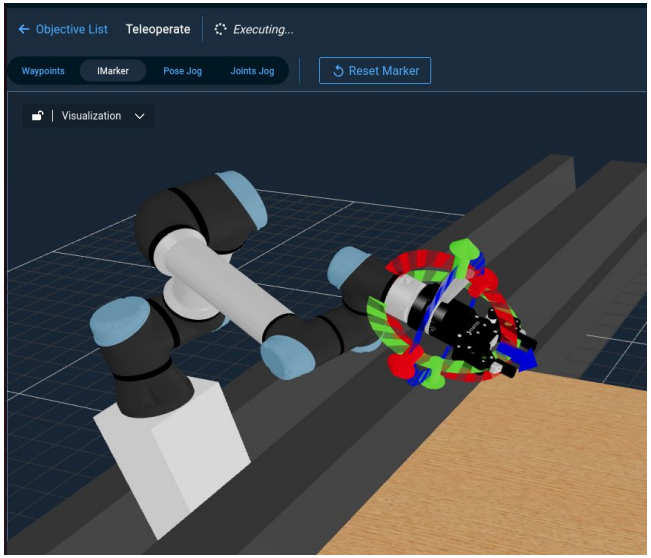
PICKNIK

Teleoperate Screen

MoveIt Pro provides several forms of manual control for when a robot needs to be setup, diagnosed, or recovered



Cartesian “Pose Jog”



Interactive Marker

Joint Jog

Try it hands on now!

PICKNIK

Saving Waypoints

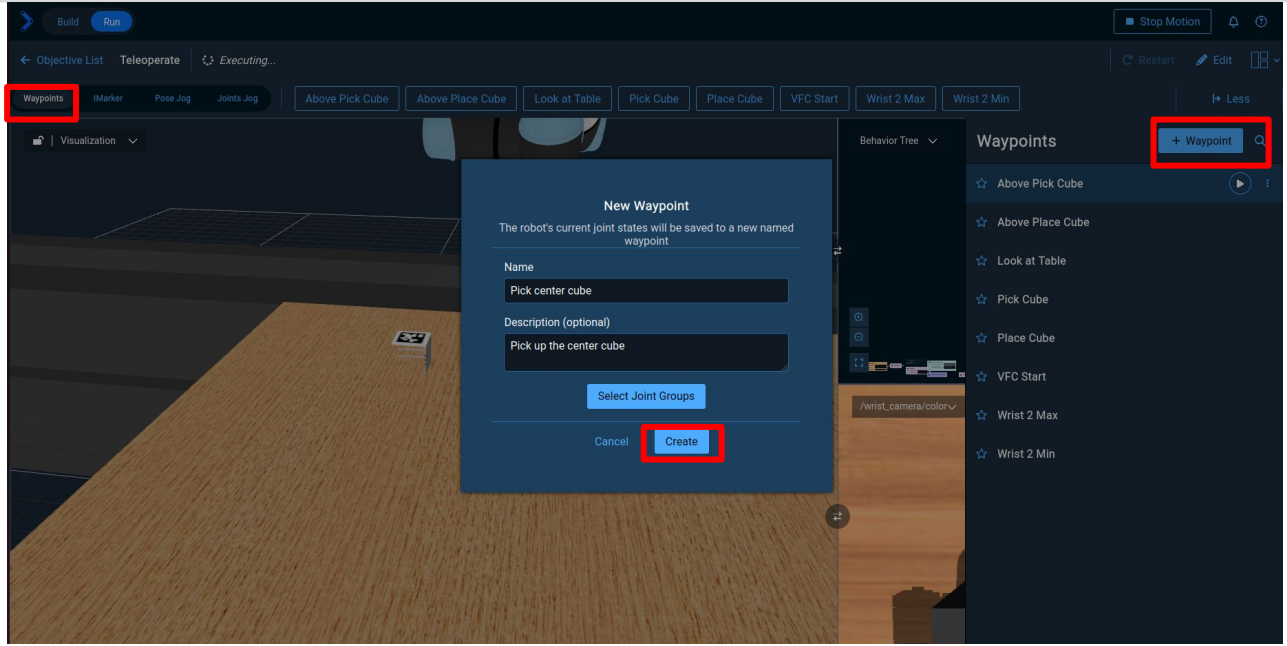
Click **Teleoperate** to control your robot

Select **Waypoints** mode

Click the **All Waypoints** button

Click "**+ Waypoint**" button to save the current pose

We will use these as target joint states for our Objective



PICKNIK

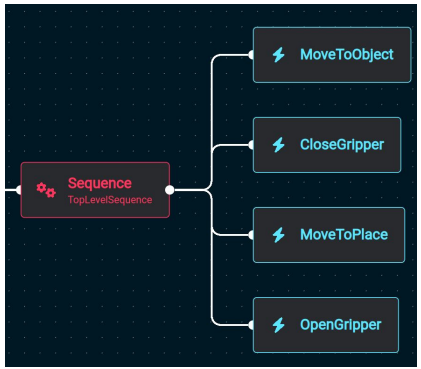
Behavior tree crash course

What is a behavior tree?

PICKNIK

Behavior Trees in MoveIt Pro

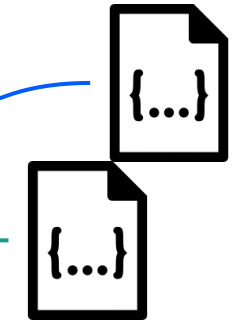
Objectives in MoveIt Pro are implemented as **behavior trees**:



Graphical conceptual model

```
<root BTCPP_format="4">
  <BehaviorTree ID="MainTree">
    <Sequence name="root_sequence">
      <Action ID="MoveToObject"/>
      <Action ID="CloseGripper"/>
      <Action ID="MoveToPlace"/>
      <Action ID="OpenGripper"/>
    </Sequence>
  </BehaviorTree>
</root>
```

XML file



ROS node implementations

Behavior trees repeatedly **tick** each node, left to right then top to bottom

PICKNIK

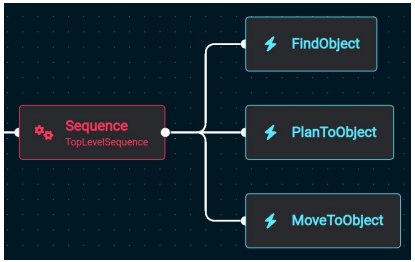
Behavior Tree Crash Course

NOTE: There also exist **decorators** to modify the status of a node. Examples:

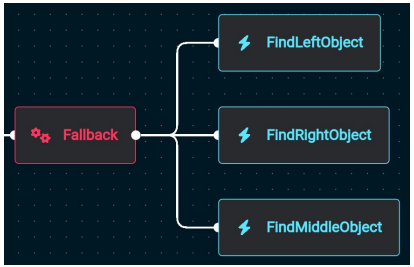
- Invert (NOT operator)
- Retries / loops
- Delays
- …

All Behaviors (individual behavior tree nodes) must either:

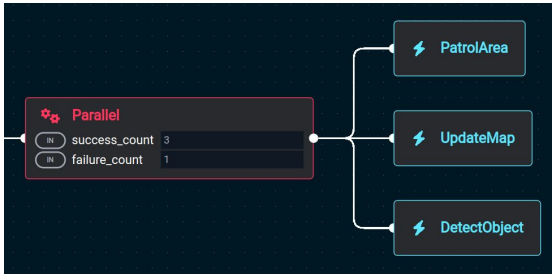
- Control execution of one or more child nodes
- Contain code that executes at every time step, or **tick**
- Return a **status**: SUCCESS, FAILURE, or RUNNING



Sequence
All nodes must run in order and return SUCCESS



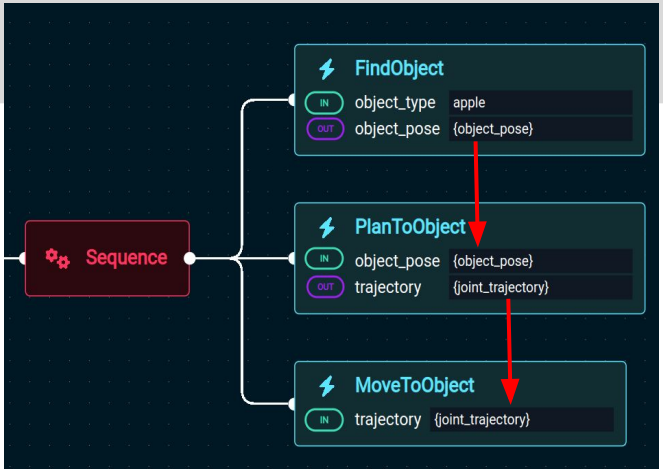
Fallback
Try alternatives in order until one of them returns SUCCESS



Parallel
Execute all nodes each tick until a terminal state is reached

PICKNIK

Data Sharing in Behavior Trees



Behaviors communicate with each other via a **Blackboard** (key/value storage).

To manage access to data, Behaviors can assign **ports.** There are 3 types of ports:

- Input
- Output
- Bidirectional

The blackboard can be used for:

- Passing data as parameters between behaviors
- Updating state (e.g, counters, queues, lists)

KEY	TYPE	VALUE
object_type	string	`apple`
object_pose	Pose	`{object_pose}`
trajectory	Trajectory	`{joint_trajectory}`

Note: {brackets} = value set by a Behavior

PICKNIK

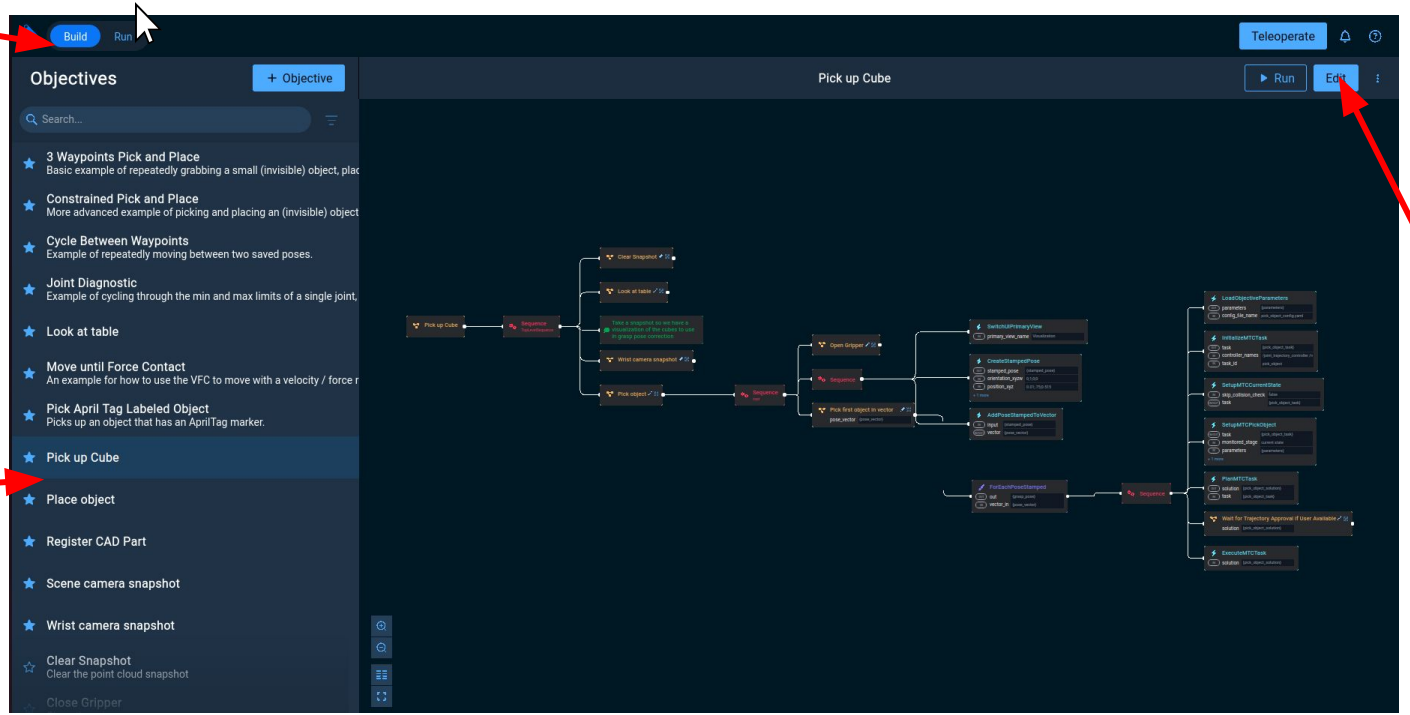
Modify your first Objective

How to customize an existing application

PICKNIK

Edit the Pick up Cube Objective

1. Select Build

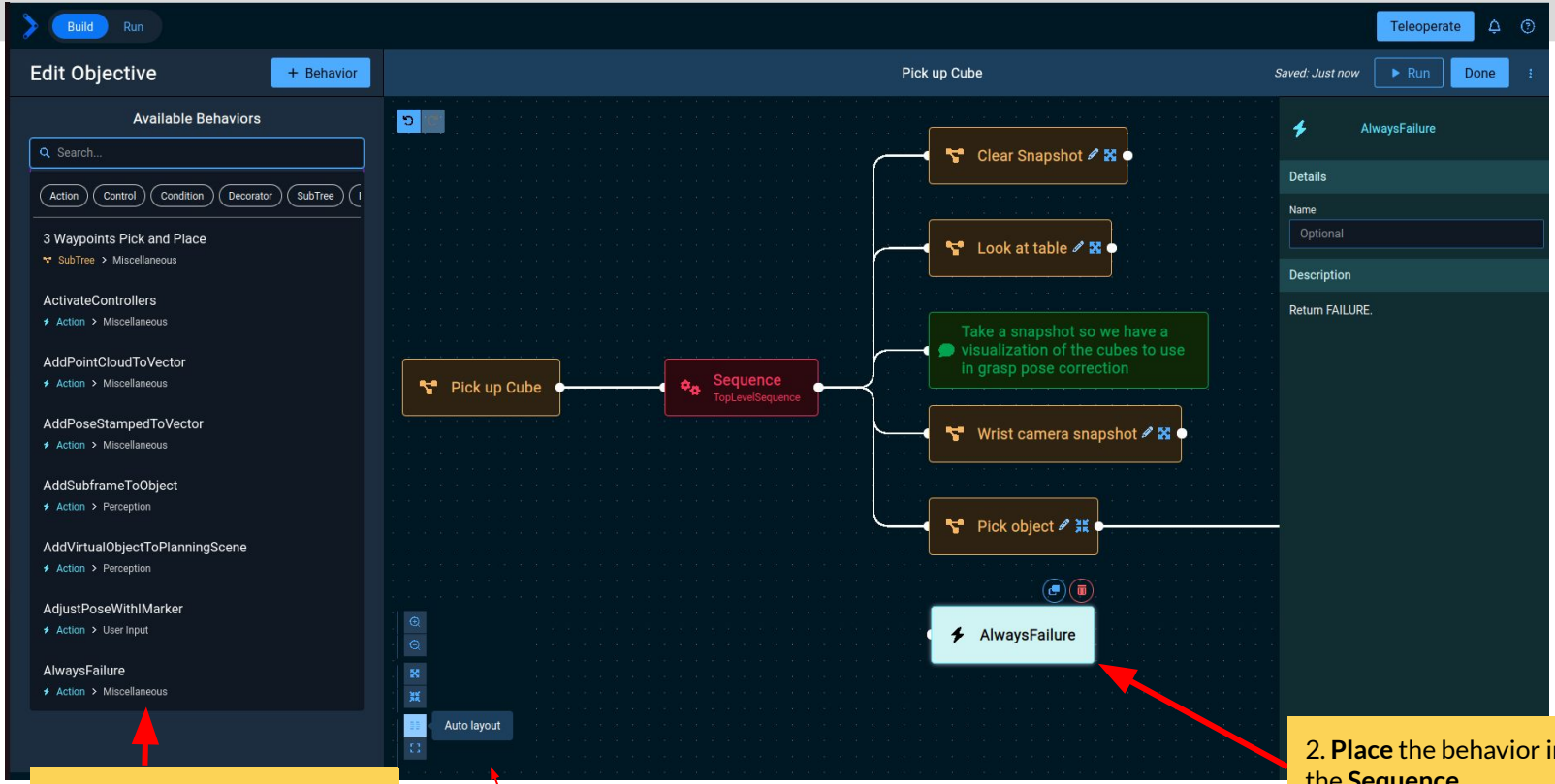


2. Choose Pick up Cube

3. **Edit** the objective to start customizing it

PICKNIK

Create a Failure Scenario



1. **Drag** the **AlwaysFailure** behavior to the canvas

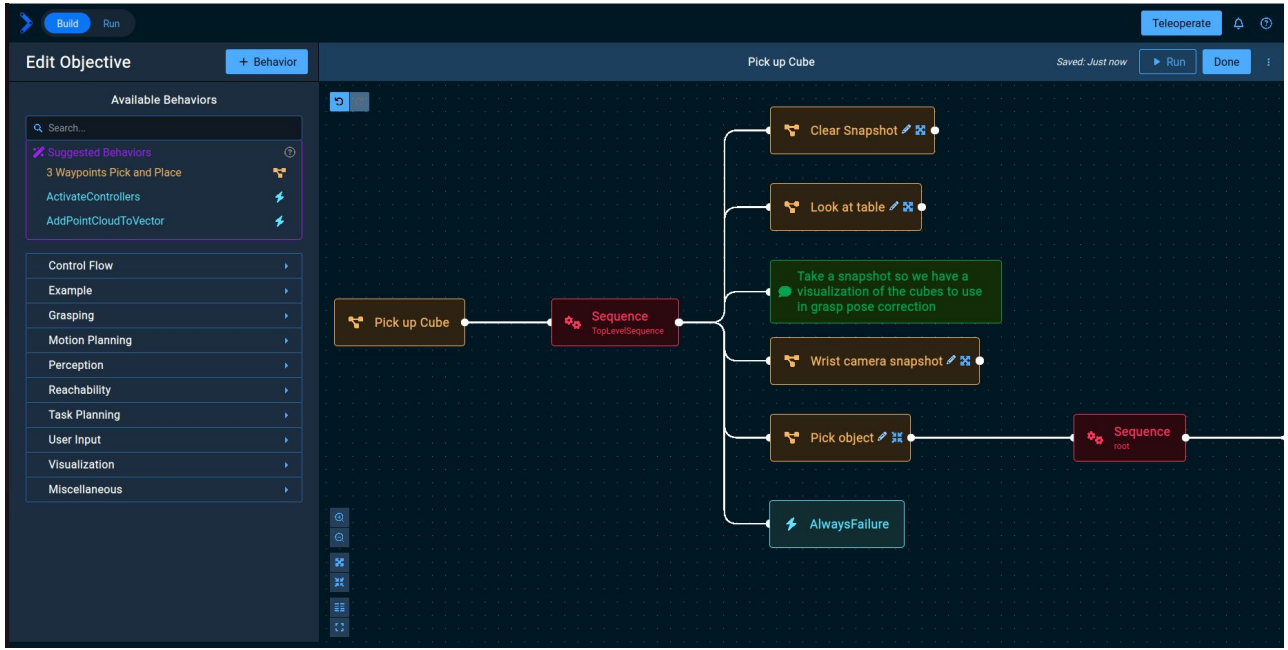
Pro Tip: *use the Auto Layout button to cleanup your BTree layout.*

2. **Place** the behavior in the **Sequence**

PICKNIK

Discussion

- What happens when we run the objective?
- What can we do to recover from failures?



PICKNIK

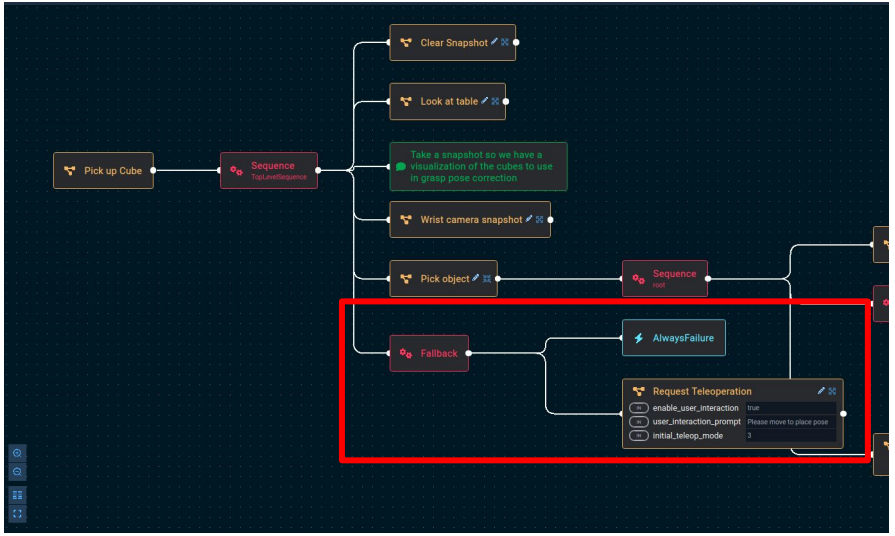
Add Error Recovery to an Objective

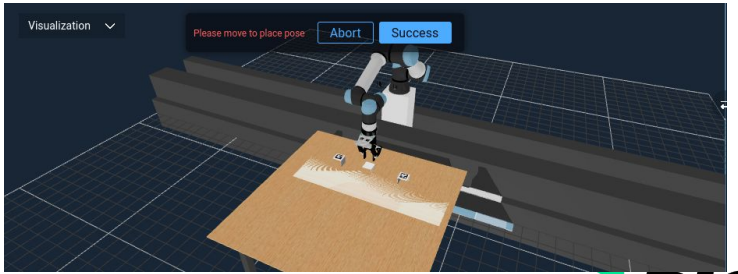
How to build failure recovery into your behavior tree

PICKNIK

Add a fallback node

1. Edit “Pick up Cube”
2. Click the “AlwaysFailure” behavior input connection in the BTree
3. Highlight the input connection then [Delete] to disconnect
4. Drag in a “Fallback” control node
   a. *From the list of available Behaviors on the left panel*
   b. *Controls -> Miscellaneous -> Fallback*
5. Use the white connection points (dots) to connect nodes
6. Add a recovery behavior
   a. *Use **Request Teleoperation** to recover*
   b. *Set enable_user_interaction to* **true**
   c. *Set user_interaction_prompt to “**Please move to place pose**”*





What happens when you run now?

PICKNIK

Create an Objective from Scratch

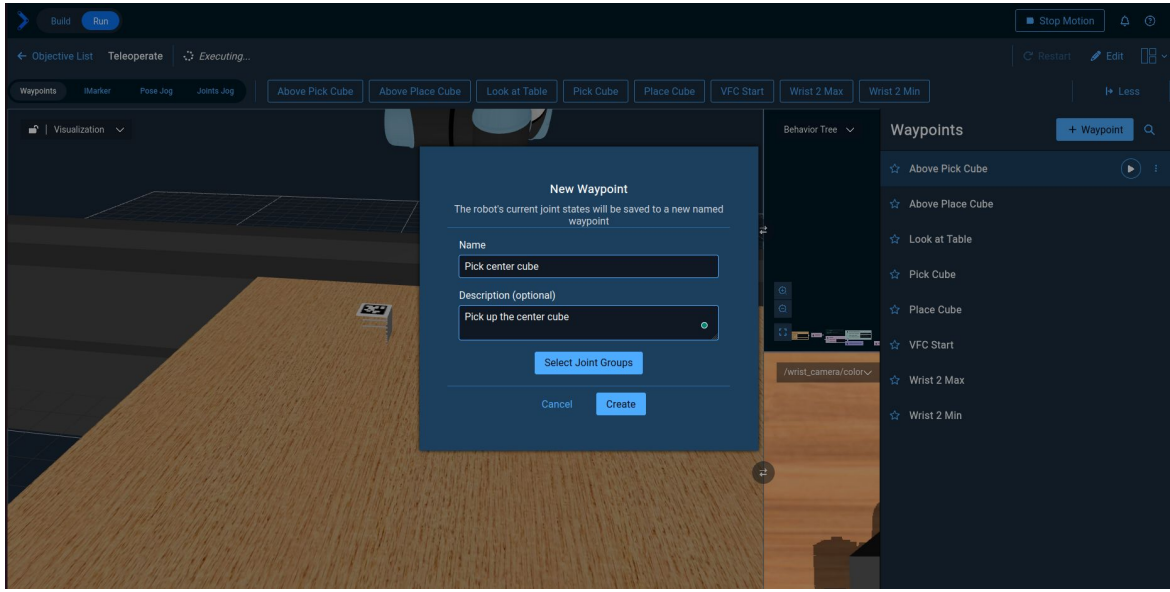
Without modifying an existing Objective

Build a Waypoint-Based Pick and Place Application

Example:

1. Look at Table
2. Take snapshot
3. Open gripper
4. Move to pick pose
5. Close gripper
6. Move away from the object
7. Move to a placement pose
8. Open gripper

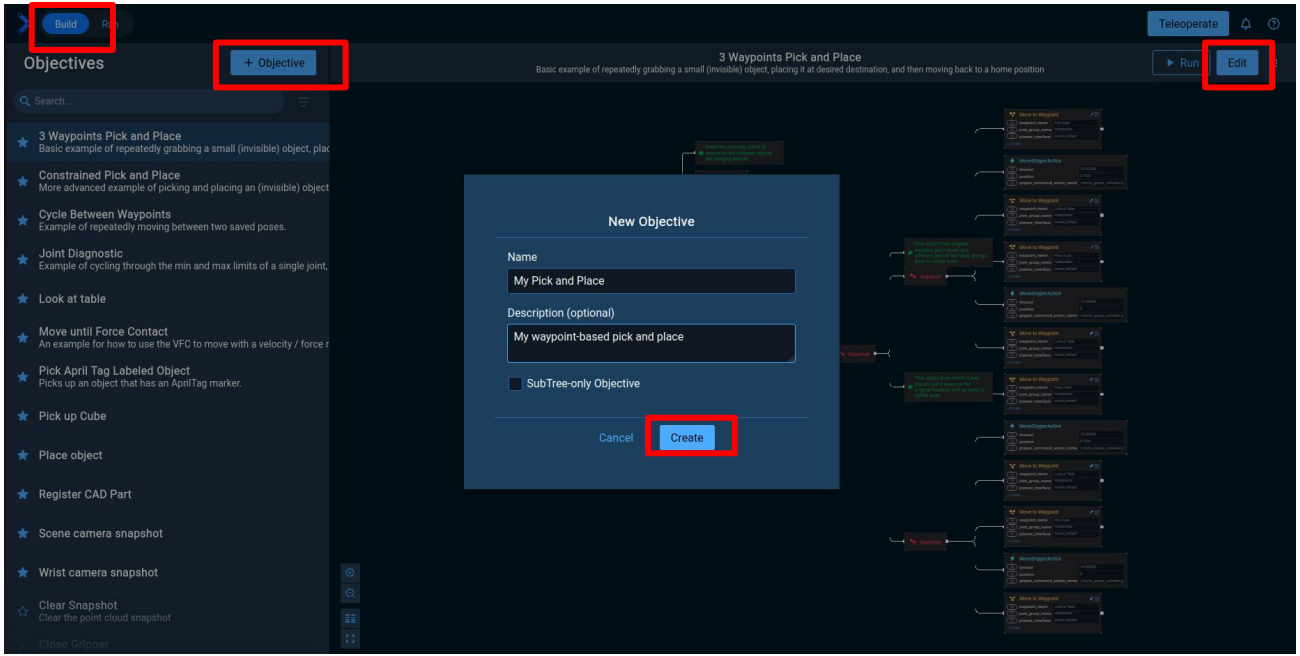
Can we use **Waypoints** and built-in **Behaviors** to make an **Objective**?



PICKNIK

Creating a New Objective

1. Use the **Build** tab to create a new Objective.
2. Name it: **`My Pick and Place`**
3. Select **Edit** and start adding Behaviors from the list on the left side.
4. Build and test incrementally!



PICKNIK

Using the Behavior Tree Blackboard

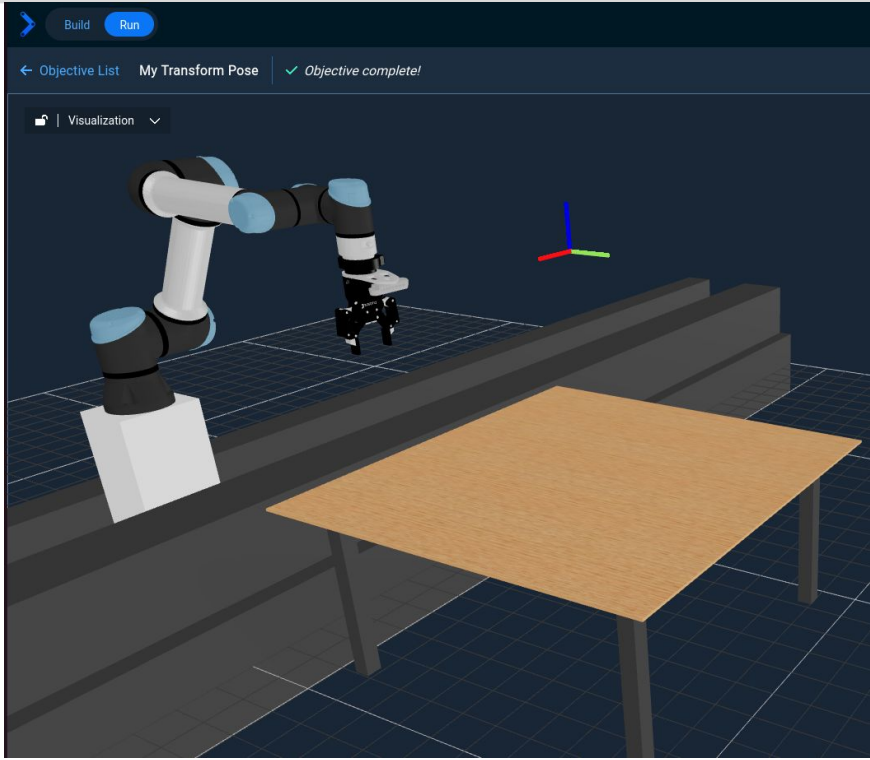
Creating pose goals programmatically

PICKNIK

New Objective - My Transform Pose

Overall flow:

1. Create a pose
2. Visualize the new pose
3. Transform (move) the pose



PICKNIK

My Transform Pose
Create a pose

CreateStampedPose

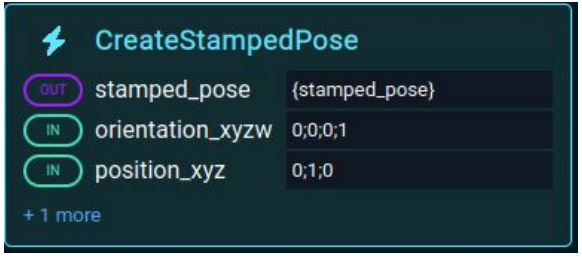
- Places a PoseStamped on the Blackboard

Inputs

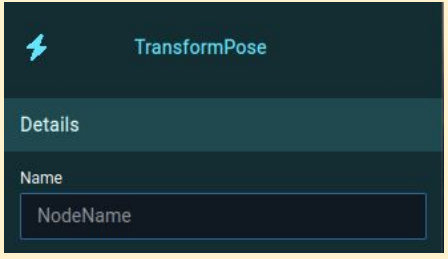
- A quaternion to define orientation within frame
  - `orientation_xyzw: 0;0;0;1`
- Define the position within frame
  - `position_xyz: 0;1;0`
- Set the reference frame
  - `reference_frame: world`

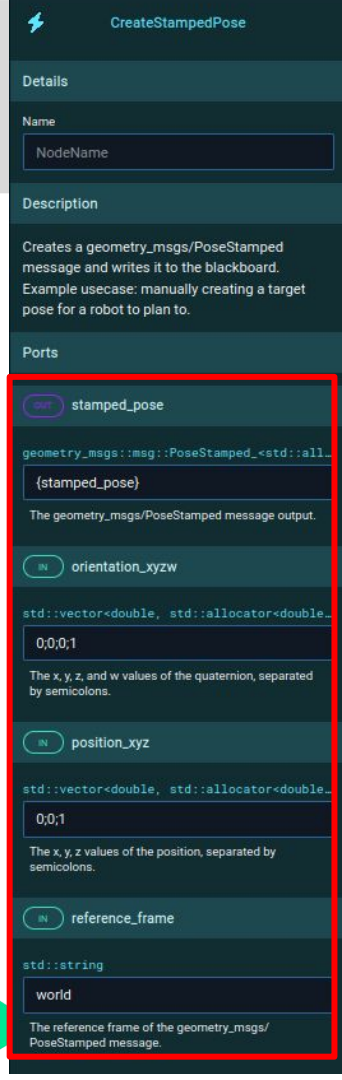
Output

- `stamped_pose: {stamped_pose}`



***Pro Tip:** Are NodeNames required?*
No, you do not need to fill in the name of the Behavior Node, it is optional but can be used for provide a description/comment of what it does





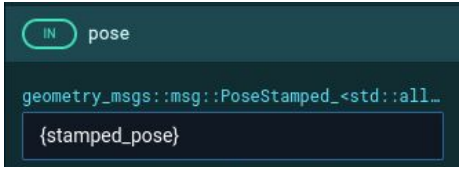
My Transform Pose
Visualize the pose

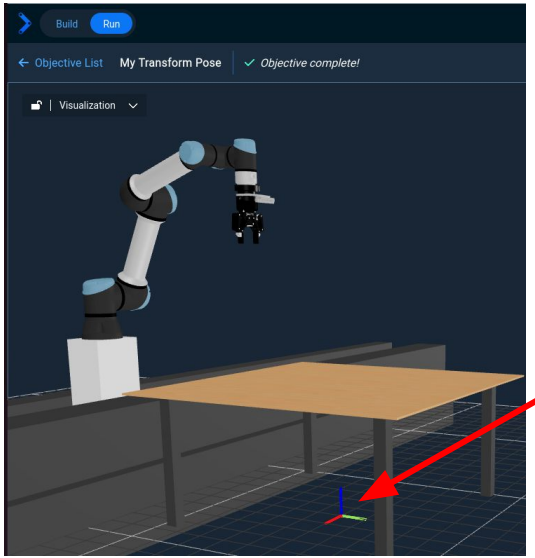
VisualizePose

- Accepts a PoseStamped as input from the Blackboard
- Adds it to the Visualization pane

Inputs

- **`marker_name: pose`**
- **`pose: {stamped_pose}`**





Your pose is on the floor here

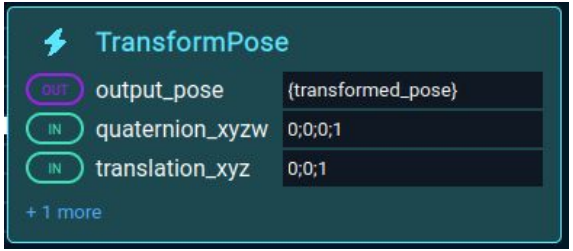
You just used the blackboard to pass a pose from one behavior to the next!

PICKNIK

My Transform Pose
Transform the pose

TransformPose

- Transform a pose that is on the Blackboard
- If output_pose is the same as input_pose, it will *modify* the pose on the blackboard
- If they are different, it will add a pose to the blackboard



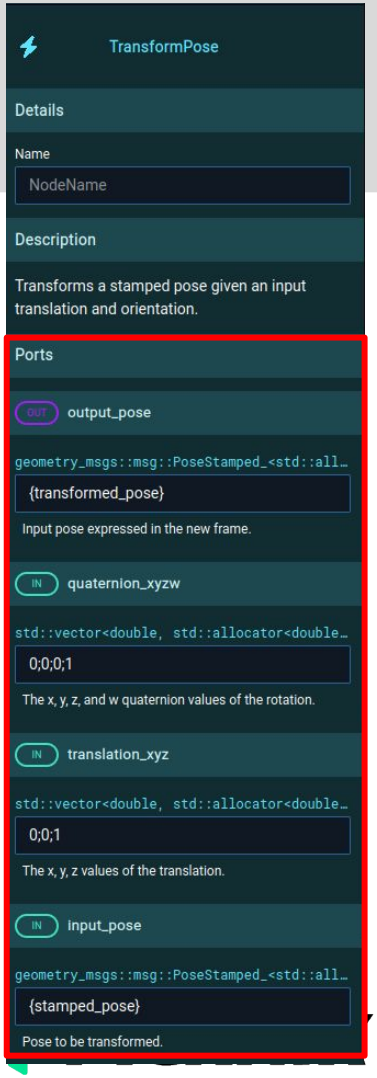
Inputs

- Use a quaternion to define orientation
  - **`quaternion_xyzw: 0;0;0;1`**
- Move the position 1 meter in **z direction**
  - **`translation_xyz: 0;0;z`**

Output

- **`output_pose: {output_pose}`**

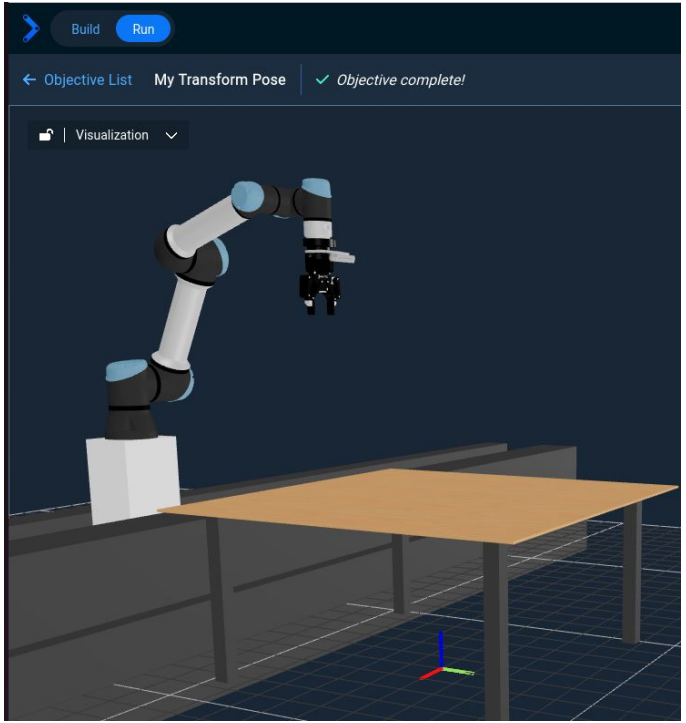
What happens when we run now?



My Transform Pose

Overall flow:

1. Create a pose
2. Visualize the new pose
3. Transform (move) the pose



Is this what we want?

PICKNIK

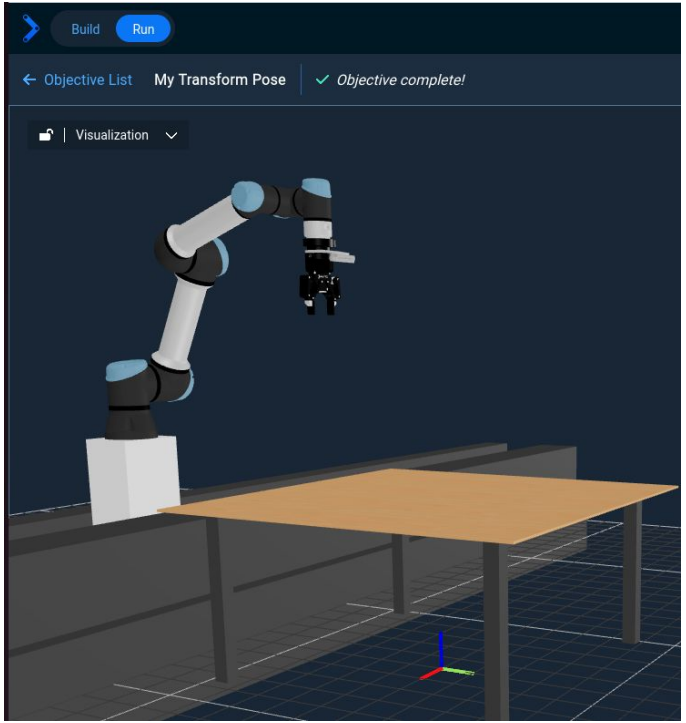
My Transform Pose

Overall flow:

1. Create a pose
2. Visualize the new pose
3. Transform (move) the pose

We need to visualize *after* we transform!

1. Create a pose
2. Transform the pose
3. Visualize the pose

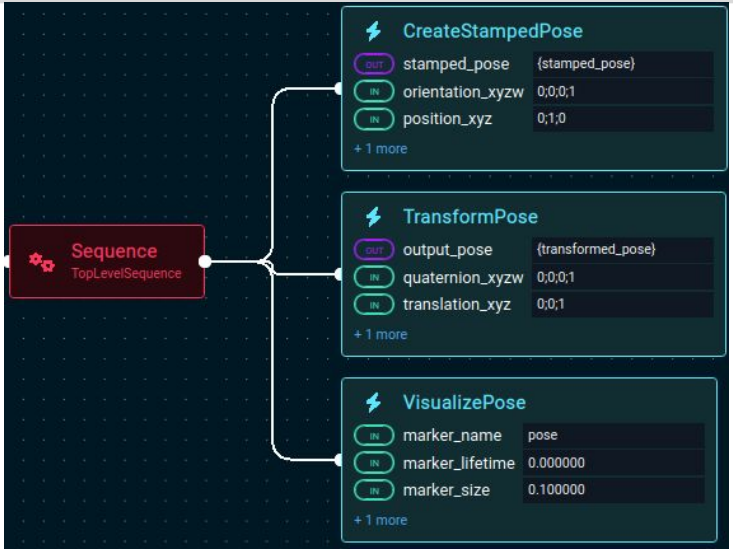


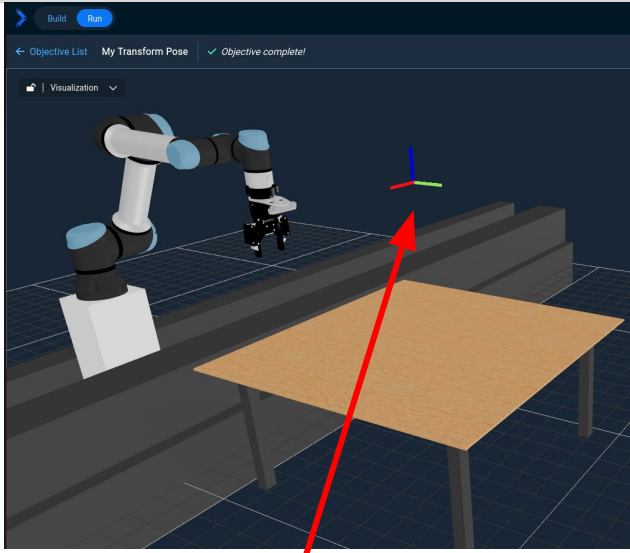
PICKNIK

My Transform Pose

New Flow:

1. Create
2. Transform
3. Visualize





Looks good!

But can we see both before and after?

The new pose is above the table

PICKNIK

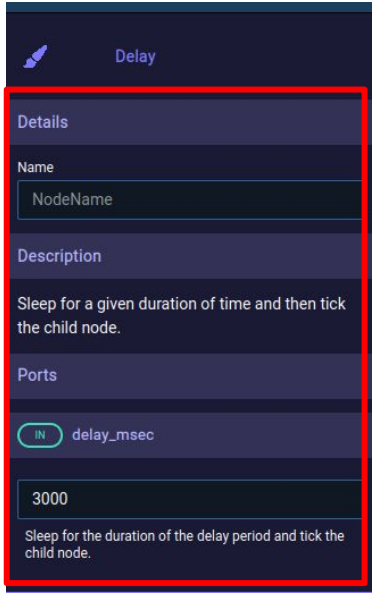
My Transform Pose

Delay

- Decorator node
- Adds a delay before executing another behavior

Inputs

- Delay time in milliseconds
  - **`delay_msec: 3000`**



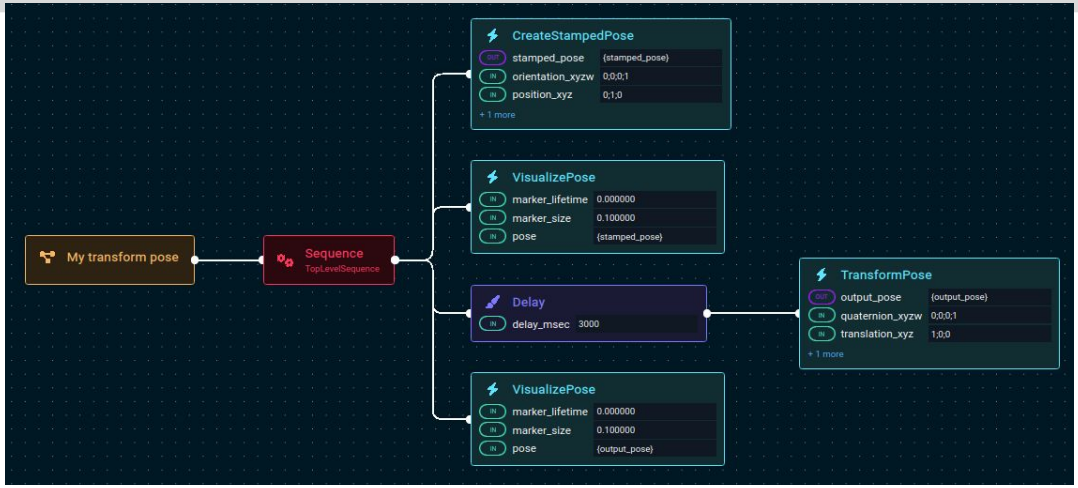
Need to delay between the first and second pose

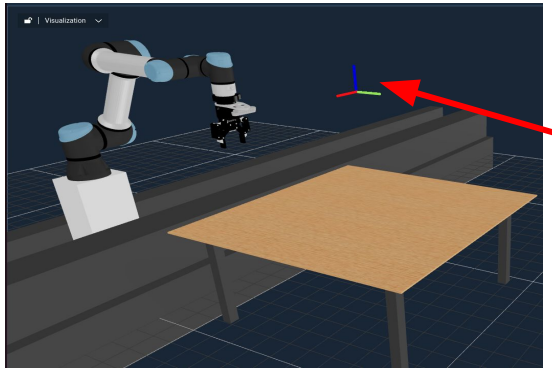
PICKNIK

My Transform Pose solution

Final flow:

1. Create Pose
2. Visualize Pose
3. Delay 3s
4. Transform Pose
5. Visualize new Pose





You should see the pose move from the floor to above the table

PICKNIK

Summary and Conclusion

What have we discussed?

- What are motion planning and MoveIt, and what you can do with them
- What are some planners available in MoveIt and what goes into motion planning?
- MoveIt concepts and configurations
- How to use MoveGroup to generate motion plans
- What is manipulation, and how that is different from motion planning
- What you need to solve a manipulation problem in production
- How to use frameworks, tools, and libraries on top of MoveIt to structure your manipulation tasks

Takeaways

- MoveIt is a motion planning framework
  - Generate motion plans using MoveGroup or MoveItCpp
- Manipulation tasks involve more than just motion planning
- Tools built on top of MoveIt's motion planning capabilities can help solve manipulation tasks reliably
  - MoveIt Task Constructor is an open-source tool to build a manipulation task that consists of a series of connected motion plans
  - BehaviorTree.CPP is an open-source behaviour tree implementation that can be used to control execution of a manipulation task (perception failures, planning failures, driver errors, etc.)
- MoveIt Pro builds upon MoveIt motion planning, MoveIt Task Constructor, behavior trees, perception, pre-built implementations of common manipulation steps, and more.

Resources

- **MoveIt 2**
  - Website: https://moveit.ai
  - Repository: https://github.com/moveit/moveit2
- **MoveIt Task Constructor**: https://github.com/moveit/moveit_task_constructor
  - MoveIt Maintainer Team, Robert Haschke and Michael Görner in particular
- **Behavior Tree.CPP**: https://www.behaviortree.dev/
  - Davide Faconti (he's here at ROSCon 2024, feel free to ask him about behavior trees)
- **MoveIt Pro**: https://picknik.ai/pro/
  - Docs: https://docs.picknik.ai/
  - Contact: https://picknik.ai/connect/

Join our Happy Hour for drinks and chats!

When? Today, October 21st at 5PM
Where? Anarkist Beer & Food Lab, Albanigade 20 (10 min walk)

sponsored by

PICKNIK

Q&A

Raise your hand or ask on Slido:

slido.com #4109254