# Mobile Manipulation Tutorial (Expanded)

Jiawei Hou[1*], Yizheng Zhang[*], Andre Rosendo and Sören Schwertfeger[2]

## I. INTRODUCTION

As service robots stepped into the public view in recent years, the development of mobile manipulators has rapidly increased. A mobile manipulator is a robot system composed of a manipulator arm installed on a mobile platform [23]. Through this combination, robots have unlimited workspace and the ability to operate in the environment. Taking advantage of these, mobile manipulators are widely applied in different fields, such as hospital operation, warehouse transportation and laboratory research assistance. Although mobile manipulators are a mature robot system and in high-development demand, so far there is still no systematic tutorial on how to operate mobile manipulators.

Many components are commonly available to accomplish basic functions of a mobile manipulator, and many of these as downloadable software packages. Most of them, such as *MoveIt*, *move_base*, *SMACH*, come with detailed official tutorials or documents. However, most packages require users to configure parameters according to the practical situation of their robots, the official documents are often too broad for beginners, and the example codes cannot be applied to the users' robots directly.

This tutorial provides four demos based on two robots, running on both real and simulation environment, and their corresponding codes. For the simulation part, we also provide a docker with a well-configured environment so that readers can easily run our examples on their own computers. We design the following task for the demos: the robot navigates to location A to pick an object (bottle/can) detected on the table, then brings the object to location B in another room and place it on the table there. Through our tutorial, readers can learn what components are needed to complete such a task, what software packages are recommended for these components, how to configure the packages according to the operational demo, what functions each component undertakes, and how to connect these components. In Table I the packages needed during running the "bring object" task are listed. And Table II lists the packages which are needed offline. In addition to providing all the code on GitHub, we created a webpage[1] to show the videos of the demos and properly guide users through all the modules mentioned in this tutorial.

TABLE I
MODULES RUNNING IN REAL TIME (C FOR CODE, T FOR TUTORIALS)

| Name | Package | Links |
|---|---|---|
| Kinova Bringup | kinova_ros | C |
| RealSense Camera Driver | realsense_ros | C |
| Object Detection & Pose | NOCS | [21] C |
| Object Place Pose | AprilTag_ROS | C T |
| Arm Planning | MoveIt | C T |
| Path Planning | move_base | C T |
| Decision Making | FlexBE | C T |

TABLE II
AUXILIARY MODULES (C FOR CODE, T FOR TUTORIALS)

| Name | Package | Links |
|---|---|---|
| Simulation World Building | Gazebo | T |
| SLAM 2D | Cartographer | C T |
| Camera Calibration | camera_calibration | T |
| Camera Pose Detection | AprilTag_ROS | C T |
| Hand-eye Calibration | easy_handeye | C T |

## II. PREPARATION

This tutorial is based on the assumption that the readers are already familiar with ROS (Robot Operating System). All the examples in this tutorial will be run on ROS Melodic version under Ubuntu 18.04. Before starting the tutorial, one should install ROS under a Linux Operating System.

We build two catkin workspaces, one named "fetch_ws" for the Fetch robot examples, and the other named "kinova_jackal_ws" for the Kinova-Jackal robot examples. We provide all code and configuration files mentioned in this tutorial on GitHub. The repository for the Fetch robot can be found here[2], while the repository for the Kinova-Jackal applications can be found here[3]. Before the start of our tutorials, one should download our source and the dependencies, and compile them:

```
$ git clone <our-source-url>
$ cd fetch_ws # or cd kinova_jackal_ws
$ git submodule update --init --recursive
$ rosdep install --from-paths src --ignore-
    src -y -r
```

*Both authors are first author and denote equal contribution.

[1]Jiawei Hou is with the School of Information Science and Technology, ShanghaiTech University, Shanghai 201210, China, and also with the University of Chinese Academy of Sciences, Beijing 100049, China, and also with the Shanghai Institute of Microsystem and Information Technology, Chinese Academy of Sciences, Shanghai 200050, China houjw@shanghaitech.edu.cn

[2]Yizheng Zhang, Andre Rosendo and Sören Schwertfeger are with the School of Information Science and Technology, ShanghaiTech University, Shanghai 201210, China {zhangyzh1, andre, soerensch}@shanghaitech.edu.cn

[1]https://momantu.github.io/
[2]https://github.com/momantu/momantu_fetch
[3]The code will be available in the final version.

```
$ catkin_make
$ source devel/setup.bash # run this command
    before using the launch file in the
    workspace
```

We also provide another repository[4] for 6D object pose estimation, which need to be installed in another workspace according to its README file.

## III. Simulation

We use *Gazebo* to build the simulation and use *RViz* to visualize and set up the robotics environment. Gazebo is an essential robot simulation tool as it allows users to replace the models of robots, objects and building modules to build the simulation scenes. RViz is a popular ROS tool to setup virtual scenes for debugging robots. The two software can be installed through:

```
$ sudo apt install ros-melodic-rviz
$ sudo apt install ros-melodic-gazebo-*
```

### A. Robot Model

A robot model can be obtained in two ways: download the official model or write URDF/Xacro file on your own. In this section, we describe how to obtain a model of *Fetch* robot and build a *Jackal-Kinova* combined robot model.

*1) Download Official Model:* If you buy a robot that can run with ROS, it is quite common to find its robot model on its official website/source. For example, all the robot models of Fetch, Jackal and Kinova can be downloaded directly.

To install the necessary components of Fetch simulation, we run this command:

```
$ sudo apt install ros-melodic-fetch-gazebo-
    demo
```

Then we can launch our Fetch robot in Gazebo by running

```
$ roslaunch fetch_sim gazebo_simulation.
    launch
```

In the launch file, there is a *world_name* argument specifying the world file to be loaded. If you build another world in Gazebo, you can save it as a ".world" file and change the *world_name* argument to point to this file to use it:

```
<arg name="world_name" value="PATH TO THE
    WORLD FILE"/>
```

*2) Build URDF/Xacro File:* URDF is the abbreviation of Unified Robot Description Format, which is used to describe a robot model, including all the element required to specify the model. Inside the URDF file, a group of joint and link tags are combined to describe the entire robot. Joints have its subtags to describe which two links it is connected to, the type of the joint (continuous, revolute, prismatic, planar, floating, fixed), dynamic feature (damping, friction), motion limit (position limit, speed limit, torque limit). Links also have their subtags, such as appearance, collision, inertial.

Since there are many identical parts in a robot files can be long and have many duplicates when using URDF files to describe it. To overcome such disadvantage, the idea of using Xacro (XML Macros) files, a macro language, comes to mind. Xacro files provide features including macro definition, file inclusion, which allows us to reuse variables and functions, and programmable interface, which enables us to define variable, mathematical calculation and condition statements.

A demo that adds a Velodyne Lidar to the Jackal robot model is shown as follows, and the code can be found in the file "jackal_with_basket.urdf.xacro" of our source.

Firstly, we can use the file inclusion feature provided by the Xacro to import the Jackal model directly.

```
<xacro:include filename="$(find
    jackal_description)/urdf/jackal.urdf.
    xacro"/>
```

Secondly, we add the Velodyne Lidar. Define the link *velodyne_mount* to add the Velodyne.

```
<link name="velodyne_mount">
<visual>
<geometry>
<mesh
filename="package://mani_description/meshes/
    velodyne_mount.stl" scale="0.001 0.001
    0.001"/>
</geometry>
<material name="white"/>
</visual>
<collision>
<geometry>
<mesh
filename="package://mani_description/meshes/
    velodyne_mount.stl" scale="0.001 0.001
    0.001"/>
</geometry>
</collision>
</link>
```

The visual tag specifies what the link looks like. In this demo, we use the geometry tag and material tag. Inside the geometry tag, we can use some basic geometry, such as a box or cylinder, or load an stl file using a mesh tag. In order to enable collision detection, it is needed to define a collision element as well.

Then, we use a joint tag to connect the *velodyne_mount* link with the *front_mount* link in the Jackal robot. A "fixed" type joint is used since the basket is fixed on the Jackal.

```
<joint name="velodyne_mount_front_mount" type
    ="fixed">
<child link="velodyne_mount"/>
<parent link="front_mount"/>
<origin xyz="0 0 0" rpy="0 0 0"/>
</joint>
```

Inside the joint tag it is needed to specify which two links are connected and what is the relative pose between them. The pose contains the position and orientation.

Finally, we import the Velodyne Lidar model and mount it on the link *velodyne_mount*.

```
<xacro:include filename="$(find
    velodyne_description)/urdf/VLP-16.urdf.
    xacro"/>
```

[4] https://github.com/momantu/nocs_ros

```
<VLP-16 parent="velodyne_mount" name="/
    velodyne" topic="/velodyne_points" hz="10
    " samples="440" gpu="false">
<origin xyz="0␣0␣0.044" rpy="0␣0␣0"/>
</VLP-16>
```

When including sensors, like camera or lidar, their parameters need to be configured, such as data frequency, fov, etc.

*3) Work with Gazebo simulation:* To spawn the robot model in Gazebo, it is required to add more tags to describe the model.

An $< inertial >$ element contains the inertia information. It must be properly specified and configured within each $< link >$ element. The inertia information can be provided by modeling programs such as MeshLab. The following example shows how to add an $< inertial >$ element to the link *velodyne_mount*.

```
<link name="velodyne_mount">
...
<inertial>
<origin xyz="0.012␣0.002␣0.067" rpy="${PI/2}␣
    0␣${PI/2}"/>
<mass value="1"/>
<inertia
ixx="1" ixy="0" ixz="0"
iyy="1" iyz="0"
izz="1"/>
</inertial>
</link>
```

*4) Assemble your own mobile manipulator:* There are already many mature mobile manipulators, such as Fetch and Robotnik, on the ROS community. Although it is quite common to find mobile chassis and fixed robotic arms, in here we'd like to introduce a method to combine a chassis with a robotic arm using the Software Development Kit (SDK) provided by the vendor.

The traditional approach is to modify the URDF files and combine these files as a new URDF file which contains the chassis and the manipulator. However, this method can be quite challenging. Usually, robot manufacturers will provide their SDK, which contains the URDF files that describe their robot, to allows consumers to launch their robot. The challenge here is that these URDF files share the same key words, like robot_description, so if you run the simulation code of different robots at the same time these key words will enter conflict.

Here is a demonstration on how to combine a Jackal model with a Kinova model on a Gazebo simulation. Two key points are followed to address the problem:

1. add **namespace** to arguments
2. assemble chassis and robotic arm as one robot.

**Namespace** should be added in the following 5 places:

1. *robot_description* parameter when loading the URDF file to the ROS parameter server. The code is in the file "kinova_jackal_gazebo.launch".

```
<param name="/$(arg␣ns)/robot_description
    "
command="$(find␣jackal_description)/
    scripts/$(arg␣env_runner)
$(find␣jackal_description)/urdf/configs
    /$(arg␣config)
$(find␣xacro)/xacro␣$(find␣
    mani_description)/urdf/
    jackal_with_basket.urdf.xacro
--inorder␣'namespace:=$(arg␣ns)'" />
```

2. *robot_description* parameter when spawn model in Gazebo. The code is in the file "kinova_jackal_gazebo.launch".

```
<node name="urdf_spawner" pkg="gazebo_ros
    " type="spawn_model" ns="$(arg␣ns)"
args="-urdf␣-model␣jackal␣-param␣/$(arg␣
    ns)/robot_description␣-x␣$(arg␣init_x)
    ␣-y␣$(arg␣init_y)␣-z␣$(arg␣
    floor_height)␣-Y␣$(arg␣init_yaw)" />
```

3. *robot_description* parameter loaded by ROS package *robot_state_publisher*. The code is in the file "kinova_jackal_gazebo.launch".

```
<node name="mybase_robot_state_publisher"
     pkg="robot_state_publisher" type="
    robot_state_publisher" ns="$(arg␣ns)">
<param name="robot_description" value="
    /$(arg␣ns)/robot_description"/>
</node>
```

4. Controller in Gazebo. The code is in the file "jackal_control.launch".

```
<node name="controller_spawner" pkg="
    controller_manager" type="spawner" ns=
    "$(arg␣ns)"
args="jackal_joint_publisher␣
    jackal_velocity_controller">
</node>
```

5. $< robotNamespace >$ parameter in Gazebo plugins. The code is in the file "jacka.gazebo".

```
<robotNamespace>/$(arg namespace)</
    robotNamespace>
```

We conclude the first step by adding namespace to these five places and launching all the original launch files. A distinctive difference is that in this way all topics are under a namespace we set.

On the other hand, besides the change of rostopic name, the 5th step will also add the namespace to the $< frame_id >$ tag inside a topic if the topic message is published by a sensor. If we don't need to add the namespace to the $< frame_id >$ filed, we can use the absolute namespace by adding a slash to the *name* parameter of the corresponding sensor in the URDF file. Here is an example of avoiding the use of a namespace to the $< frame_id >$ in the message published by the Velodyne sensor.

```
<xacro:include filename="$(find␣
    velodyne_description)/urdf/VLP-16.urdf.
    xacro"/>
<VLP-16 parent="velodyne_mount" name="/
    velodyne" topic="/velodyne_points" hz="10
    " samples="440" gpu="false">
<origin xyz="0␣0␣0.044" rpy="0␣0␣0"/>
</VLP-16>
```

As a second step we should assemble both chassis and robotic arm as one robot:

Adding a static transform between chassis and manipulator through ROS node *static_transform_publisher*

```
<node pkg="tf2_ros" type="
    static_transform_publisher" name="
    base_arm_link_broadcaster"
args="-0.12␣0.0␣0.184␣0␣0.0␣0␣base_link␣root"
     />
```

Using Gazebo service to attach two robot together. Gazebo itself doesn't provide this service, but it can be done by an additional Gazebo plugin. Gazebo_ros_link_attacher project provides this service. In our world file, add this plugin to enable this service:

```
<sdf version="1.4">
<world name="default">
<!-- A gazebo links attacher -->
<plugin name="ros_link_attacher_plugin"
    filename="libgazebo_ros_link_attacher.so"
    />
<include>
<uri>model://sun</uri>
</include>
<!-- A ground plane -->
<include>
<uri>model://ground_plane</uri>
</include>
</world>
</sdf>
```

To launch the assembled robot model in Gazebo run the file "launch_robot.sh" in our kinova_jackal_ws.

```
$ ./launch_robot.sh
```

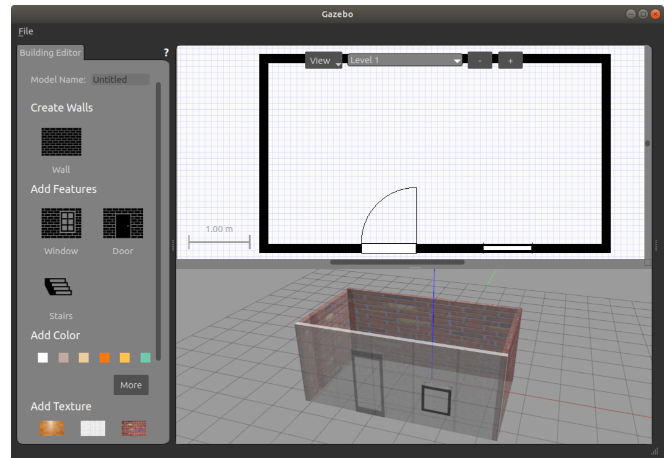



Fig. 1. Using building editor to build a small room.

Fig. 2. Our Gazebo simulation world.

### B. Model in Gazebo

Gazebo provides a building editor and model editor to allow users to make their own scene and models easily through a Graphical User Interface (GUI). Through this building editor, it is easy to build a room, which is shown in Fig. 1. Gazebo comes with a model database already, and Rasouli *et al*. [15] provides a dataset that contains 270+ 3D models to be used on Gazebo simulations.

On the other hand, if the GUI still doesn't meet your requirement to build the model you can write an SDF file to describe the model. The tutorial for writing a custom model can be found on the Gazebo website.

We build a custom Gazebo world for our simulation, which is shown in Fig. 2.

## IV. Robotics Manipulation

### A. Introduction

Manipulation capability is key to many applications. It enables robot to interact with the environment. Generally, the robot manipulator picks something and put it at another place, or just manipulate some machine. Grasping an object requires the robot manipulator to plan a trajectory through cluttered environments and generate a grasp pose to pick the object. While planning a trajectory, the robot needs to use sensors to perceive the environment and see where the manipulator can go through without resulting in a collision. After the trajectory is obtained by the motion planner, the inverse kinematics algorithm is used to calculate joint positions for the given pose. Finally, the low level hardware controllers execute the time-parameterized joint trajectories, which allows the end effector to move to the target pose.

### B. MoveIt

*MoveIt* is an open-source robotics manipulation platform, and its system architecture is shown in Fig. 3. *MoveIt* integrates modules (e.g. motion planning, manipulation, perception, collision detection) and provides a node called *move_group*, which pulls all the individual components together to provide a set of ROS actions and services for users. The framework of *move_group* is shown in Fig. 4. It provides a variety of interfaces, and we will demonstrate how to use *move_group* C++ interface to pick an object and put it on another table with a Fetch mobile manipulator.

Firstly, install *MoveIt* and the ROS package provided by the Fetch robot:

```
$ sudo apt install ros-melodic-moveit
$ sudo apt install ros-melodic-fetch-gazebo-
    demo
$ sudo apt install ros-melodic-moveit-visual-
    tools
```

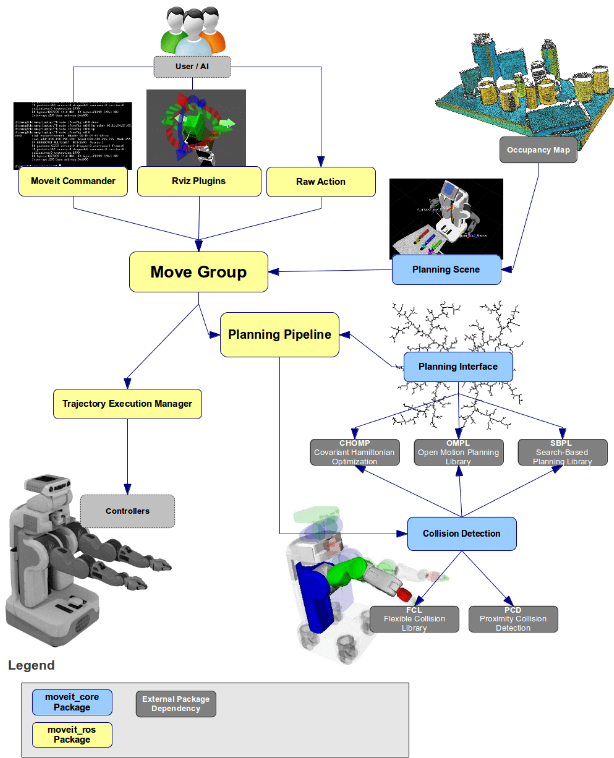



Fig. 3. *MoveIt* pipeline[13].

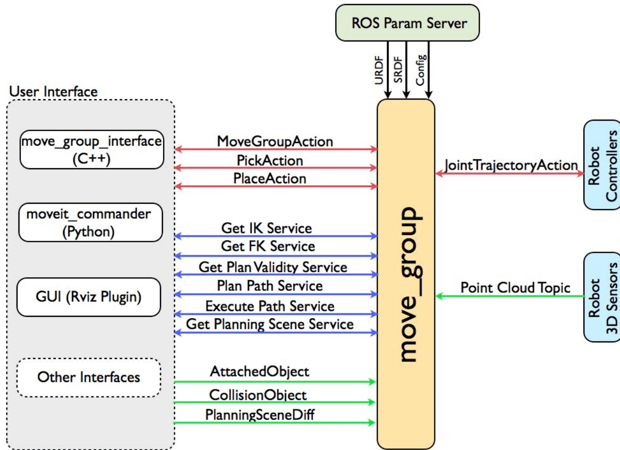



Fig. 4. Move group[13].

```
$ sudo apt install ros-melodic-moveit-
    resources
```

Bring up our robot with *MoveIt* in Gazebo by:

```
$ roslaunch fetch_sim gazebo_simulation.
    launch
$ roslaunch fetch_common moveit.launch
```

As Fig. 3 shows, *move_group* integrates the components, where each component needs parameters to configure itself. This is done by load these parameters from ROS Param Server. However, we don't need to set these parameters one by one, as most of these parameters are already properly set inside the downloaded package. If you

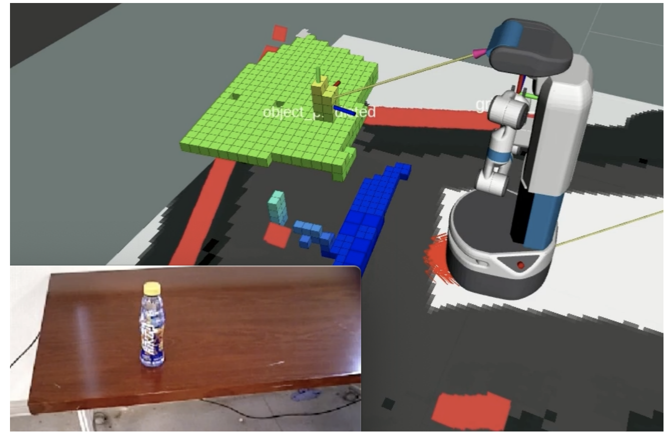


Fig. 5. Integration with *OctoMap*.

build your own robot manipulator, it is recommended using *MoveIt Setup Assistant* to perform the initial set up of your manipulator, since this tool can configure most of the parameters automatically according to your manipulator model. The ROS package *fetch_moveit_config* 's subfolder *config* contains the yaml files which store the *MoveIt* parameters. These parameters can be modified according to our needs.

### *C. Perception pipeline*

To take advantage of the 3D sensor, *MoveIt* allows for seamless integration of it using *OctoMap*. This function is implemented by *MoveIt*'s Occupancy Map Updater, which uses a plugin architecture to process different types of input. Fetch robot has an RGBD camera and gives point cloud, so we use the PointCloud plugin to handle this information. The configuration stores in the file "sensor.yaml" is shown here:

```
sensors:
- sensor_plugin: occupancy_map_monitor/
    PointCloudOctomapUpdater
point_cloud_topic: /head_camera/
    depth_downsample/points
max_range: 1.5
point_subsample: 1
padding_offset: 0.05
padding_scale: 1.0
```

To enable the plugin, the yaml file should be loaded by a launch file.

```
<rosparam command="load" file="$(find␣
    fetch_moveit_config)/config/sensors.yaml"
     />
```

The *OctoMap* needs to be configured by adding the following lines into the launch file:

```
<param name="octomap_frame" type="string"
    value="base_link" />
<param name="octomap_resolution" type="double
    " value="0.05" />
<param name="max_range" type="double" value="
    5.0" />
```

For fetch robot, these parameters are set and loaded in the file "fetch_moveit_sensor_manager.launch.xml" .

### D. Pick and Place

The most common use of robotic arms is to pick and place objects. The prerequisite for this task is knowing the pose of the object, which is done by section VIII. *MoveIt* provides several modules to do pick and place task, including *pick and place*, *MoveIt grasp*, *MoveIt Task Constructor*. The third party packages, like *Grasp Pose Detection (GPD)*[18], uses learning based method to detect 6-DOF grasp poses for a 2-finger robot hand in 3D point clouds.

We demonstrate two methods to do this task, one is using the *move_group* to plan a trajectory directly. The other is using the *MoveIt* pick and place module.

*1) Vanilla* move_group*:* The pipeline of grasp and place an object usually contains these steps:

1. Move the end effector directly over the object.
2. Drop vertically and grab the object.
3. Vertically upward
4. Move to a place directly above the plane to be placed.
5. Drop vertically and place the object.

*MoveIt* operates on sets of joints called "planning groups" and stores them in an object named the *JointModelGroup*. The first step is to specify the planning group with MoveGroupInterface.

```
moveit::planning_interface::
    MoveGroupInterface move_group("
    arm_with_torso");
```

Then, we define 5 poses corresponding to the steps above. A pose contains the position and orientation. We can define these pose according to the target object and the place we want to put the object.

Set the pose through our move_group.

```
geometry_msgs::Pose target_pose1;
target_pose1.orientation.w = 1.0;
target_pose1.position.x = 0.28;
target_pose1.position.y = -0.2;
target_pose1.position.z = 0.5;
move_group.setPoseTarget(target_pose1);
```

Calling the motion planner to compute the trajectory, if the pose you set is out of the manipulator's workspace, the planner will not find the path. So make sure the planner computes the path successfully.

```
moveit::planning_interface::
    MoveGroupInterface::Plan my_plan;
bool success = (move_group.plan(my_plan) ==
    moveit::planning_interface::
    MoveItErrorCode::SUCCESS);
```

The Open Motion Planning Library(OMPL) provides asymptotically-optimal planners like RRT*[9], PRM*[9], LazyPRM* and BFMT, giving you alternative choose of the planner.

Finally, assuming the trajectory is computed successfully, we need to execute it on a real robot using the *move()* method.

```
move_group.move();
```

*2) Pick and Place API:* On the other side, since the vanilla move_group method requires us to define a lot of pose, which can be complicated. *MoveIt* provides *pick* and *place* function to do these steps. For the pick action, it defines a message named *moveit_msgs::Grasp*. This message contains the first three steps mentioned above. According to our pick requirement, define a variable of this type named *grasp_pose* and set the support surface to specify the fact that attached objects are allowed to touch the support surface.

```
move_group.setSupportSurfaceName("table1");
```

After that, we can call the *move_group's pick* function to grasp the target object.

```
move_group.pick("object", grasp_pose);
```

The place action is similar with the pick action. It uses the message named *moveit_msgs::PlaceLocation* to specify the place information. Define a variable of this type named *place_location* according to our place scenario, and set the support surface as pick action.

```
move_group.setSupportSurfaceName("table2");
```

Finally, we can call the *move_group's place* function to place the target object.

```
move_group.place("object", place_location);
```

The MoveIt tutorials of pick and place shows examples of using *moveit_msgs::Grasp* and *moveit_msgs::PlaceLocation*.

## V. HAND-EYE CALIBRATION

### A. Introduction

In order to allow the robotic arm to flexibly operate objects in the environment, a camera is usually mounted on the robot. Because all vision information obtained by the camera is under the camera coordinate system. For the arm, in order to use the visual information, we need to calculate the transformation between the robot coordinate system and that of the camera. This is the main problem solved by hand-eye calibration [20]. There are two camera installations normally: eye-in-hand (Fig. 6(a)) and eye-on-base (Fig. 6(b)) [5]. Eye-in-hand means mounting the camera on the arm. In this way, the camera will move with the movement of the arm. In the second way, eye-on-base, the camera and the robot base are relatively fixed.

Fig. 7(a) shows the eye-in-hand calibration theory. When we move the end effector to two different poses, where the camera can see the marker that is placed relatively fixed to the robot base, the transformation relationship for different coordinate systems in two poses can be represented as:

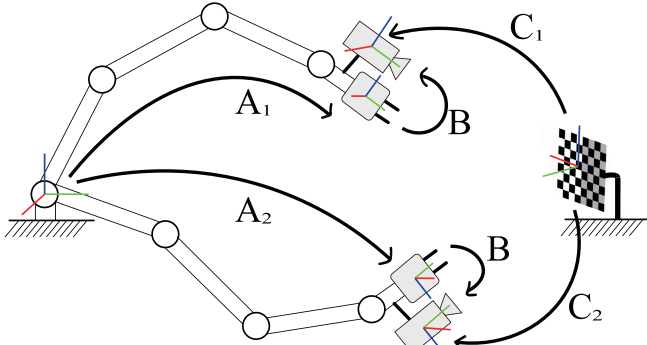
$$A_1 \cdot B \cdot C_1^{-1} = A_2 \cdot B \cdot C_2^{-1}, \tag{1}$$

$$\Rightarrow \quad (A_2^{-1} \cdot A_1) \cdot B = B \cdot (C_2^{-1} \cdot C_1) \tag{2}$$

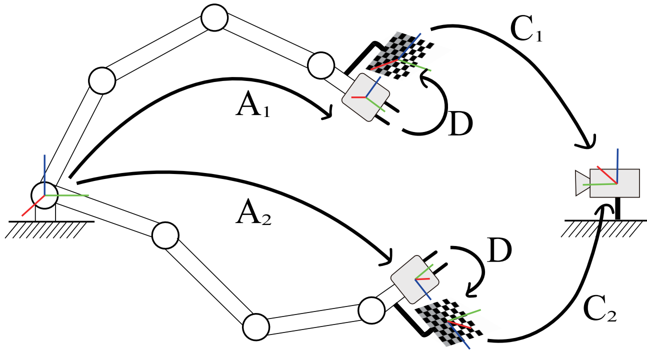
where $A_i$ is the transformation from the end effector to the robot base of the $i$th sample, which is provided by the arm robot; $B$ is the to-solve static transformation from the camera to the end effector; $C_i$ is the camera pose under the marker coordinate system of the $i$th sample. Then it translates into a

(a) An example of eye-in-hand. (b) An example of eye-on-base.

Fig. 6. Two camera installation for arm robot [19].




(a) Calibration of eye-in-hand case.




(b) Calibration of eye-on-base case.

Fig. 7. Transformation relationship for two camera-installation styles.

problem of solving $B$ with known $A_i$ and $C_i$ by moving the hand effector to different poses to collect groups of samples.

For the eye-on-base case, whose calibration is explained by Fig. 7(b). We move the robot's end effector to two different poses, and ensure that in both poses the camera can see the marker fixed at end effector of the arm. Because the relative transformation between robot base and the camera is fixed, we have this equation:

$$A_1 \cdot D \cdot C_1 = A_2 \cdot D \cdot C_2, \tag{3}$$

$$\Rightarrow \quad (A_2^{-1} \cdot A_1) \cdot D = D \cdot (C_2 \cdot C_1^{-1}) \tag{4}$$

where $D$ is the to-solve static transformation from the marker to the end effector.

### B. Necessary Precursor Stages

In our demo, we calibrate a Kinova Gen2 robot with a Intel Realsense D435 camera mounted at its end effector, in eye-in-hand installation. The hand-eye calibration only needs to be done once after the camera is mounted fixedly on the robot. As the equations derived above, the hand-eye calibrations for both camera mounting cases are the problem of solving a static transformation matrix from an equation set. For this purpose, we need to record enough groups of $A_i$ and $C_i$ to obtain enough equations.

*1) End Effector Transformation from Robot Base:* Transformation $A$ is the pose of the robot arm end effector in the robot base coordinate system, which is the positive kinematics solution in robotics. It usually can be provided by the robot driver. For the robot with ROS support, this robot joint transform information is published in the tf message.

To support the Kinova arm in ROS, the repository *kinova-ros* needs to be downloaded and compiled first. Then the tf messages for the Kinova (our Kinova's robot type is j2n6s300) are published when the robot is started:

```
$ roslaunch kinova_bringup kinova_robot.
    launch kinova_robotType:=j2n6s300
```

Then you can plug in the joystick and use it to move the arm. Alternatively, the arm can also be moved with the Motion Planning Plugin in RViz. For our Kinova, start *MoveIt* and the RViz with Motion Planning Plugin with:

```
$ roslaunch j2n6s300_moveit_config
    j2n6s300_demo.launch
```

In RViz, the Motion Planning Plugin can be added in this way: 1) Press "Add" in the RViz *Displays* tab; 2) choose "MotionPlanning" from the *moveit_ros_visualization* folder and press "OK". With this plugin, you can drag the arm to change its position and orientation, then select the "Planning Group" in the *Planning* tab and click "Plan" then "Execute" buttons to move the arm to the given goal state.

*2) Camera in World Coordinate System:* As for matrix $C$, the external parameters for camera, it is the transformation from the marker coordinate system to the camera's.

To capture images with camera, first you need to start the camera driver. If you use other cameras instead of Realsense, you need to install their drivers according to their manual. To launch Realsense camera, first install *librealsense-dkms* and *librealsense2-utils* as dependencies by:

```
$ sudo apt-key adv --keyserver keys.gnupg.net
     --recv-key
    F6E65AC044F831AC80A06380C8B3A55A6F3EFCDE
    || sudo apt-key adv --keyserver hkp://
    keyserver.ubuntu.com:80 --recv-key
    F6E65AC044F831AC80A06380C8B3A55A6F3EFCDE
$ sudo add-apt-repository "deb http://
    realsense-hw-public.s3.amazonaws.com/
    Debian/apt-repo bionic main" -u
$ sudo apt install librealsense2-dkms
$ sudo apt install librealsense2-utils
```

Intel Realsense can be started and publish images as ROS topic with *realsense-ros*, which is installed and run with:

```
$ sudo apt install ros-melodic-realsense2-
    camera  #install
$ roslaunch realsense2_camera rs_rgbd.launch
     #run
```

Note that USB3.0 is needed to connect this camera. Then you can see the two aligned topics: *"/camera/color/image_raw"* and *"/camera/aligned_depth_to_color/image_raw"*. Other RGBD cameras should also have similar topics.

In the camera coordinate system, a 3D point in space and its corresponding 2D point on the image satisfy the following relationship:

$$\lambda \begin{bmatrix} x_c \\ y_c \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} \tag{5}$$

where the transform matrix in middle is the camera internal parameters, which can be obtained by camera calibration. But you don't have to calibrate the camera, because most products, e.g., the Realsense cameras, are well-calibrated and their */camera_info* topics already contain their intrinsic matrices. An easy-use camera calibration package is *camera-calibration*, which can be installed with:

```
$ sudo apt install ros-melodic-camera-
    calibration
```

Some arguments should be set when calibrating the camera with the node *camera_calibration*. In this example, an 8*6 chessboard with 2.25cm side length is used, while the arguments *image* and *camera* should also be set according to your topic. Thus, run this command to start the calibration:

```
$ rosrun camera_calibration cameracalibrator.
    py --size 8x8 --square 0.033 image:=/
    camera/color/image_raw camera:=/camera/
    color --no-service-check
```

Then move the chessboard and take enough images including the chessboard in different positions and orientations until the "save" button is available. Because the Realsense driver doesn't implement service */camera/color/set_camera_info*, we use the option *"–no-service-check"*. Without this option, the camera calibration results, the internal parameters, can be written into */camera_info* by this node directly. Otherwise, you need to save the calibration information to the driver manually. For rectifying the image, use the ROS package *image_proc*.

The intrinsic matrix can project the 3D points in the camera coordinate system into 2D. However, the camera pose in the world coordinate system is unknown, while the 3D points are needed to be described in the world coordinate system. If the correspondence between 2D points and the 3D points described in the world coordinate can be obtained, the transformation from the world system to the camera system can be derived. Combining the intrinsic matrix and extrinsic matrix, we got the complete model:

$$p = K_{Intrinstic} \times K_{Extrinsic} \times P_W \tag{6}$$

$$= \begin{bmatrix} f_x & 0 & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R_{3\times3} & t_{3\times1} \\ 0_{1\times3} & 1 \end{bmatrix} \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{bmatrix} \tag{7}$$

where $p$ is a 2D point in image and $P_W$ is its corresponding 3D point in the world coordinate system.

*apriltag_ros* [12], [3], [22] is a ROS package that can provide relative transformation between the camera frame and the given detected tag's, which can be seen as the world coordinate system with the tag as coordinate origin. The package can be installed with:

```
$ sudo apt install ros-melodic-apriltag-ros
```

Here we use tag "TAG36H11 - 7". First we modify "setting.yaml" to tell which tag family to use and if transformation should be published to the tf message.

```
tag_family:         'tag36h11'
publish_tf:         true
```

Then modify "tag.yaml" to tell which tag you will use. The parameter "size" requests the length of the inside black rectangle in meters.

```
standalone_tags:
[
{id: 7, size: 0.1485},
]
```

In the launch file "continuous_detection.launch", which is copied from the apriltag_ros package, fill the camera frame and published topic in the arguments. Then we can start the AprilTags detection with:

```
$ roslaunch apriltag_cfg continuous_detection
    .launch
```

If the tag is detected, in the image published by the topic *"/tag_detections_image"*, you can see the tag is marked with a rectangle and labeled with its id. Besides, a frame "tag_7" appears in the tf tree.

### C. Hand-eye Calibration Stages

After a series of steps above, we have obtained the end effector pose in the robot base frame and the camera pose in the tag's frame. Everything is ready for the hand-eye calibration. We recommend a easy-using ROS package for hand-eye calibration *easy_handeye* as a solution. Its required input is the transformations between the frames of robot base, end effector, camera and the marker, which should already be published correctly if you have finished the steps described in last sub-section.

Since it includes the routine from the *ViSP library*, the *visp_hand2eye_calibration* package should be installed before building the easy_handeye package.

```
$ sudo apt install ros-melodic-visp-hand2eye-
    calibration
```

The package easy_handeye is included as a git submodule in our source. Otherwise, you can get it from the GitHub repository *IFL-CAMP/easy_handeye*.

Then fill the required tf frame names to the arguments of the launch file "kinova_realsense_tag7_calibration.launch", created referring to official Github README file, to launch the calibration GUI. Run this command to start the calibration:

```
$ roslaunch handeye_cfg
    kinova_realsense_tag7_calibration.launch
```

Move the arm and make sure in each pose the camera can detect the given tag and press the "take sample" button. When you take enough samples, press "compute", and press "save" when you satisfy with the calibration results.

The package includes a file "publish.launch" that can publish the transformation into the tf message. Here we also create a launch file "kinova_camera_tf_publisher.launch" to run "publish.launch" and fill some required parameters. Henceforth, you can run this launch file to publish the transformation when you want to use the visual information to cooperate with the arm:

```
$ roslaunch easy_handeye
    kinova_camera_tf_publisher.launch
```

### D. Fetch Calibration

The Fetch Robot is already calibrated by the factory, whose transform between the camera and the robot base is included in its URDF file. Nevertheless, the Fetch Robotics encourages users to re-calibrate the robot and provide a convinent tool *calibrate_robot* in the package *fetch_calibration*.

## VI. Mapping

If we want the robot to arrive at some specific place to accomplish some tasks, a map is necessary. The goal of mapping is creating a map of the environment so that the robot can navigate with the recorded environmental information in the future. The main technique behind is simultaneous localization and mapping (SLAM) [4], [1]. SLAM is a process that building a map while figuring out where the agent is located in the environment. Building a 2D map saves computation resource and sensor costs, comparing to building a 3D map, and is normally sufficient for ground navigation. But if we consider the arm status in the space while the mobile platform is moving, a 3D map helps. From the point of this view, both 2D mapping and 3D mapping are included in this tutorial. *Cartographer* provides both 2D and 3D SLAM in its ROS package, which will be used in our following 2D and 3D mapping.

### A. 2D Mapping Introduction

A basic input requirement of mapping is a range measurement device. For the 2D case, a 2D laser scanner meets the requirement, which is cheap and widely used. Besides, the data from the Inertial Measurement Unit (IMU) or odometry helps the localization. 2D mapping is a fairly mature research field, for which there are some popular open-source packages. Their different back-end algorithms determine their different application scenarios. *Gmapping* [6], [7] is a 2D laser SLAM based on particle filter framework, combining odometry and laser information. The algorithm needs to calculate the posterior probability of each particle for robot localization. Thus this method is suitable for small scenes mapping needing high accuracy. *Hetor_SLAM* [10] matches laser data between frames with optimized method without requiring odometry information. But its performance is affected by radar frequency. *Karto* [14], [11] is a graph optimization SLAM based on matrix sparseness, including loop closure, which has advantages in large-scale mapping. *Cartographer* [8] is a very complete real-time laser SLAM system, including rich interfaces. It contains a robust front end and an pose graph back end based on submap and node constraints. We will introduce how to configure and use its ROS package for 2D mapping in the following.

### B. 2D Mapping Stages

*1) Input Data:* First of all, the sensor to measure the environment should be decided. For example, *Hokuyo Laser Scanners* have well ROS interface *"hokuyo_node"*, which publishes 2D laser data to topic "/scan". This topic message is in the type of *sensor_msgs::LaserScan*, which can be the input of the 2D mapping ROS package directly.

In our mapping and navigation examples, a 3D surrounding sensor, *Velodyne's VLP-16 Lidar sensor*, mounted on Jackal is used. The data from Velody VLP-16 sensor is published with *velodyne_pointcloud* package, which provides point cloud conversions for Velodyne's 3D Lidar sensors. It can be installed and run with:

```
$ sudo apt install ros-melodic-velodyne-
    pointcloud
$ roslaunch velodyne_pointcloud VLP16_points.
    launch
```

Make sure the correct transformation from "base_link" to "velodyne" is included in the tf message so that the mapping algorithm can obtain correct height information of the point cloud. If the transformation is not declared in the URDF file, it can be published as *static transform* with *tf2_ros* in the format of:

```
$ rosrun static_transform_publisher 0 0 0.09
    0 0 0 front_mount velodyne
```

where "front_mount" is a frame of Jackal, whose transformation to "base_link" already exists in the tf message. The Velodyne sensor center is 9 cm higher than the frame "front_mount". The parameters above are set in this order:

```
$ static_transform_publisher x y z yaw pitch
    roll parent_frame_id child_frame_id
```

To use 3D laser data as input for 2D mapping, normally we convert its topic from message sensor_msgs::PointCloud2 to *sensor_msgs::LaserScan* before transferring the data to the mapping node, which can be done by a ROS tool *pointcloud_to_laserscan*. For example,

```
$ rosrun pointcloud_to_laserscan
    pointcloud_to_laserscan_node cloud_in:=/
    velodyne_points _min_height:=-0.3
    _max_height:=1.05
```

In the above command, the input point cloud topic is "/velodyne_points" from the Velodyne sensor. Besides, the input point cloud height range can be limited with the parameter *_min_height* and *_max_height*, which is decided by your robot height and how high is the Velodyne from the ground. Above operations are wrapped into a launch file "velodyne_pc2scan.launch" in our source.

*2) Cartographer_ROS Configuration:* The ROS package of Cartographer can be installed with:

```
$ sudo apt install ros-melodic-cartographer-
    ros
```

First fill some sensor information in a lua file, which can be copied from the templates given in the folder "configuration_files". For example, "revo_lds.lua" is a template for low-cost 2D lidar. The most important thing in this file is to set *tracking_frame* and *published_frame*. The *tracking_frame* is set according to which frame will be tracked by the SLAM algorithm, sensor frame is a common choice. Make sure the tf message including the transformation between the tracking frame and the robot base frame. The *published_frame* is the frame that is already published and will be the child frame of the Cartographer-providing frame. In our configuration, since the Cartographer algorithm provides "odom", the *published_frame* is set to "base_link".

For the explanation of other parameters, please refer to their official tutorial. Then write a launch file to start *cartographer_node* to start mapping. Remember to point to your own lua file and then remap the "scan" topic to the one your laser scan publishing. For example, Fetch's laser publishes "/base_scan", for which the remapping is needed:

```
<remap from="scan" to="base_scan" />
```

For the Jackal robot in our source, the Cartographer mapping can be started with:

```
$ roslaunch carto_setting cartographer_2d.
    launch
```

*3) Save Map:* When you change the *Fixed Frame* to "map" and add a map topic "/map" in RViz, you will see a 2D map is generating. When you think the map building has been finished, the map can be saved with the following command:

```
$ rosrun map_server map_saver -f 2d_map
```

Where "2d_map" is the name of the map file name. Then the files "2d_map.yaml" and "2d_map.pgm" are created in the current directory. The *pgm* file saves the 2D map as an image. The *yaml* file saves the map information, including the map origin, resolution, occupied/free threshold, and so on.

### *C. 3D Mapping Introduction*

For laser SLAM in 3D case, there are also some ROS package. Here we introduce some famous ones. LOAM is a system for 3D Lidar odometry and mapping in real-time [25], based on features matching for localization. It is currently closed source, but there are a lot of SLAM open source packages developed on it, such as *A-LOAM*, *LeGO-LOAM* [17] and *Lio-mapping* [24]. *BLAM* is an easy-used and open-source package only requiring laser data as input, having fast loop closure. Cartographer is also a famous ROS package for 3D mapping with laser point cloud as input while allowing combination with IMU data and odometry information. In our 3D mapping example, the *Cartographer_ROS* package is used, with the Velodyne's point cloud and the IMU data as inputs.

### *D. 3D Mapping Stages*

*1) Input Data:* We run a Jackal carrying a 3D laser sensor, Velodyne's VLP-16 Lidar Sensor, in a laboratory of about two hundred square meters to create a 3D indoor map as an example. Firstly, start the Velodyne sensor to publish 3D point cloud and publish the transformation between the robot and the sensor if it is not included:

```
$ roslaunch velodyne_pointcloud VLP16_points.
    launch
$ rosrun static_transform_publisher 0 0 0.09
    0 0 0 front_mount velodyne
```

These two commands are wrapped into "velodyne_tf.launch" in our source.

*2) Cartographer_ROS Configuration:* Same as the first step in 2D configuration, the sensor information should be filled into a lua file. The file "taurob_tracker.lua" in the *cartographer_ros* package is a good template for 3D laser configuration. We refer to it and create the configuration file "velodyne_3d.lua" for our example. Set the parameters "tracking_frame" and "published_frame" as we do for 2D case. That is, we still require the Cartographer algorithm to provides "odom" frame. Another highly-influential parameter is *"TRAJECTORY_BUILDER_3D.num_accumulated_range_data"*, which decides the number of range data to accumulate in a single map. The higher the number is set, the better the map would be, however, the slower the generation becomes. It is tuned to 360 according to the area of our laboratory, 200 square meters. Note that, a large "TRAJECTORY_BUILDER_3D.num_accumulated_range_data" leads to high computational load. You can refer to this page to tune the parameters for lower latency. Then a launch file is needed to start the cartographer mapping node and point to our configuration file, which is "cartographer_3d" in the package "carto_setting". The launch file is started with:

```
$ roslaunch carto_setting cartographer_3d.
    launch
```

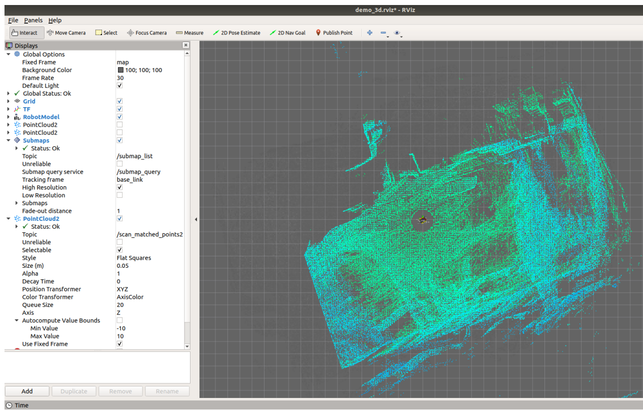
In RViz, the mapping point cloud is shown (Figure 8(a)).

*3) Save 3D Map as PCD File:* Then you can drive the Jackal robot around to scan the environment point cloud. When all the areas that need to be built are scanned, the 3D map can be saved as a pcd file with:

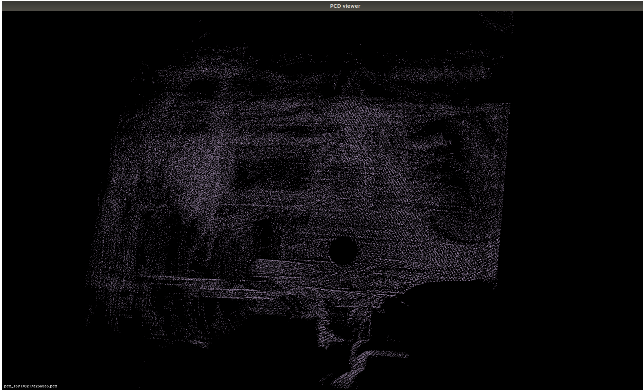
```
$ rosrun pcl_ros pointcloud_to_pcd input:=/
    scan_matched_points2 _prefix:=pcd_
```

Running the command before starting mapping can save the point clouds under construction. Then a list of pcd files are created in the folder. These pcd files can be viewed with *pcl_viewer* (Fig. 8(b)), which can be installed from "pcl-tools" and run with:

```
$ sudo apt install pcl-tools  # installation
$ pcl_viewer -multiview 1
    pcd_1591271240335105.pcd  # view the
    point cloud saved as "
    pcd_1591271240335105.pcd"
```



(a) Mapping point cloud in RViz.



(b) Saved point cloud viewed with pcl viewer.

Fig. 8. Environment point cloud.

## VII. Navigation

### A. Introduction

Navigation is the technique that supports the robot to move from place to another specific place. Normally, a map, at least a local map, is required to navigate to the goal. A local map can be provided directly by a range sensor mounted on the robot. However, without prior environmental knowledge, the robot can only move step by step carefully, without a global plan. With a map recording known environmental information, the goal can be described with an accurate coordinate and the robot can plan a global path before navigation. A map can be created in ahead with SLAM, which is introduced in the mapping section (Section VI). Due to the mobile chassis only need to navigate on 2D plane, we use the widely-used Navigation Stack, including *move_base* as a main component. The move_base package contains a local planner to avoid obstacles and a global planner to plan a path for the robot to reach the given goal. A well-configured version of move_base for Jackal is *Jackal_navigation*, installed with:

```
$ sudo apt install ros-melodic-jackal-
    navigation
```

### B. Navigation without Map

The input requirements of move_base without map include sensor data, sensor transformations, odometry and a goal. The introduction of how to publish data from our 3D laser sensor, Velodyne Lidar, and its transformation to the robot are in Section VI-B.1. The odometry data is provided by the robot. For Jackal the topic "/odometry/filtered" is used, which fuses the IMU data and the encoder data through extended Kalman filter (EKF). Then move_base publishes a controller topic "cmd_vel", which provides a stream of velocity commands.

A template to start move_base without map is "odom_navigation_demo.launch" under package *jackal_navigation*. We create package *mb_setting* for the real Jackal robot and package *sim_nav* for simulation to save the configuration files and launch files related to navigation. The default subscribing topics of laser data and odometry are "/scan" and "/odom" respectively. If other topics are specified to subscribed instead, remap the topics when including the "move_base.launch". If your robot subscribes to another controller topic rather than "/cmd_vel", e.g., a topic with namespace, a remapping for it is needed before include the "move_base.launch". As mentioned in Section III-A.4, we add namespace for Jackal's topic in simulation to avoid name conflict. Thus, for the Jackal in our simulation, the move_base node is included as:

```
<remap from="/cmd_vel" to="/$(arg ns)/cmd_vel
    "/>
<include file="$(find mb_setting)/launch/
    include/move_base.launch" >
<remap from="odom" to="/$(arg ns)/odometry/
    filtered"/>
</include>
```

The parameter configurations are loaded as yaml files. Users can check these files to explore the effect of various parameters. The detailed parameters tuning will be discuss in Section VII-E.

### C. Navigation given Map

The environmental map can be loaded and published by *map_sever*. In launch file the node is included as:

```
<node name="map_server" pkg="map_server" type
    ="map_server" args="$(arg map_file)" />
```

where "map_file" points to the yaml file of a saved map. Then launch the move_base node as introduced in last sub-section. With a given map, we localize the robot in the environment with the package *amcl*, which is a probabilistic 2D localization system. Besides the map, amcl requires odometry data, laser sensor data and an initial pose as inputs. The launch file to run the move_base and amcl nodes is modified from the template "amcl_demo.launch" under jackal_navigation package and saved in our package "mb_setting" for the real Jackal robot, which can be run with:

```
$ roslaunch mb_setting amcl_demo.launch
```

Then start RViz, set the *Fixed Frame* under the *Global Options* folder as "/map", and a map can be shown by

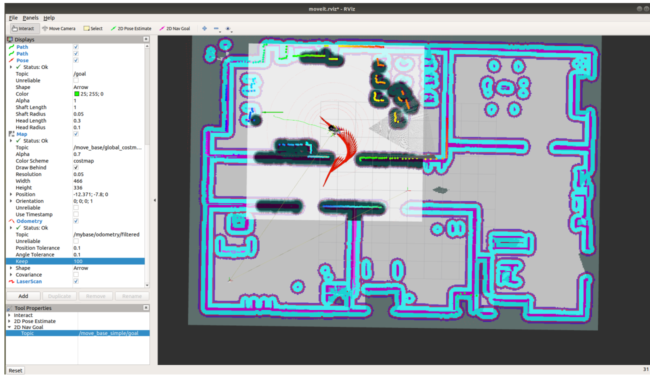


Fig. 9. Navigation given a map. The global map is padding and shown as costmap. The colorful points are the obstacle points scanned by the Lidar. Based on it, the padded obstacles are shown in the local costmap. The red arrow series are the robot poses provided by the odometry. The green arrow is the given goal.

adding the map topic "/map". Press "2D Pose Estimate" in the GUI to set a rough initial robot position in the world for AMCL algorithm if the robot is not at the same initial pose given in the launch file. Then we can send a goal with the "2D Nav Goal" button. A path to the goal will be planned by move_base, which can be shown in RViz by adding the topic "/plan". After moving several steps, AMCL will publish better and better estimation of robot poses in the world to the tf message. Fig. 9 shows the RViz interface during navigation, in which the sensor scan points overlap with the map pixels and the robot is moving toward the given goal.

### D. Fetch Navigation

Of course the mapping and navigation tutorials above are suitable for Fetch Robot as well. While *fetch_navigation* package wraps the ROS 2D navigation stack so that it fits fetch better. A detailed tutorial for Fetch mapping and navigation is also provided here.

### E. Parameter Tuning

Although both Fetch and Jackal provide configuration files fitting their robots in their navigation packages, the parameters need to be tuned according to the practical robot situation. [26] is a detailed navigation tuning guide, which helps readers to tune the navigation parameters efficiently.

Here we introduce some experience during our tuning. First, we need to input an initial pose of the robot to the AMCL algorithm. For the simulation case, the robot is usually located at the same place when the simulation starts. Thus, the initial pose can be set in the launch file fixedly. While a real robot usually starts at different locations. In this case, press the "2D Pose Estimate" button and drag an arrow in the RViz to set the initial pose each time the robot started. The initial pose doesn't have to be set accurately. As the robot moves, the AMCL algorithm updates the pose estimation. The poses will converge correctly soon if the AMCL parameters are set appropriately.

There are two AMCL parameters mainly affect the estimation converge speed, *update_min_a* and *update_min_d*, which decide the intervening movements needed before performing the filter update. The default setting of *update_min_a* in Fetch official configurations is 0.25. We decrease it to 0.05, which makes the estimation converge much faster to a correct pose.

During the Fetch navigation, sometimes there are some ghost obstacles that are not actually exist in the local costmap. It is because the wrong setting of *global_frame* under *local_costmap* namespace, which should be frame "map" but not "odom".

Goal tolerance parameters (*yaw_goal_tolerance* and *xy_goal_tolerance*) need to be set carefully. Too-low tolerance makes it difficult for robots to reach the given goal, in which case the robot will start to rotate constantly when it approaches the goal.

The rotating at the goal is necessary to reach the given goal. This value of *min_in_place_vel_theta* is the minimum in-place rotation angular velocity at the goal. The default setting is 0.3 for Fetch, which lead to a slow moving, as a result, the robot may move beyond the goal tolerance and rotate in the normal given angular velocity and then miss the given angle.

The footprint normally is set slightly larger than the real contour of the robot base. However, in the mobile manipulator application, the part of arm beyond the robot base need to be taken into account.

### F. Running ROS across Multiple Machines

It is dangerous and inconvenient to control the robot with screen and keyboard directly while it is moving around. A solution of this problem is communicating the robot with another computer so that we can receive messages from the robot and send command remotely to the robot.

To communicate the two machines, first we need to set a machine as master, because only one roscore is needed. Here we set the robot as master. On the master machine, open the ".bashrc" file under home directory and add these two lines:

```
export ROS_MASTER_URI=http://R.R.R.R:11311
export ROS_IP=R.R.R.R
```

where "R.R.R.R" is the IP address of the robot.

And add these lines to ".bashrc" file of the PC:

```
export ROS_MASTER_URI=http://R.R.R.R:11311
export ROS_IP=M.M.M.M
```

where "M.M.M.M" is the IP address of the PC.

Add the two machines' IP and hostname to their hosts file, e.g., "/etc/hosts", as follows:

```
R.R.R.R    robot-hostname
M.M.M.M    PC-hostname
```

where the hostname of computer can be found with this commandline:

```
$ hostname
```

Then you can check if the PC can receive messages from the robot with:

```
$ rostopic list    # To show received topics
$ rostopic echo /tf # To check data received
```

## VIII. Object Pose Estimation

### A. Introduction

Before grip a specific object, first we need to discover the target object and estimate its 6D pose to tell the gripper where to move. Although there is more work on instance-level 6D pose estimation, in order to allow users to easily use our examples in their actual work without having to train the model, we will introduce a method based on category-level object detection. Normally, there include two basic steps to estimate 6D poses of objects: object detection from the scene and pose estimation of the masked object. A classical way to achieve this goal is training a network to detect an object and obtain object masks, e.g., Mask R-CNN, then applying an instance-level 6D pose estimation. The work[21] is a state-of-the-art method to estimate the 6D pose of unknown specific-category objects. First, they train a region-based network to predict object masks and class labels and to directly correspond the observed pixels with their proposed representation, Normalized Object Coordinate Space (NOCS), for object instances. Then they combine the prediction and depth information to estimate the 6D pose of multiple objects within given categories. An advantage of their work is that it includes both the detection and estimation steps so that we can skip the combination work. Another advantage is that the category-level detection the algorithm can detect unseen objects without requiring the object meshes. If the target object is included in the trained categories, the users can apply the pre-trained model directly without training.

### B. Use Data from RGBD camera

Because the computer on robot is usually not good enough to run neural networks, a solution is transporting the camera data to a PC with GPU and return the estimated poses of detected objects to the robot. However, if we subscribe the raw image topic from the robot directly, the message can be lost frequently because it takes a lot of bandwidth to transport images. When we transport compressed images instead, the image transmission could be faster. Here we use the ROS plugin *image_transport* to send color images and depth images from robot to a PC.

Before transporting data between machines, make sure the machines can communicate with each other. Multiple machines communication is introduced in Section VII-F. The image transport ROS plugin can be installed with:

```
$ sudo apt install ros-melodic-image-
    transport-plugins
```

Run the launch file “pub_images.launch” on the robot to publish compressed color image topics and depth image topics, in which:

```
<launch>
<node pkg="image_transport" type="republish"
    name="rgb_compress" args="␣raw␣in:=/
    head_camera/rgb/image_rect_color␣
    compressed␣out:=/rgb_republish"/>
<node pkg="image_transport" type="republish"
    name="dep_compress" args="raw␣in:=/
    head_camera/depth_registered/image␣
    compressedDepth␣out:=/depth_republish"/>
</launch>
```

The launch file launches the node *image_transport* to republish compressed color and depth messages. The parameter setting in “args” means that the input is raw data. The input topic of color image is “/head_camera/rgb/image_rect_color”, which is the images de-distorted with the camera’s intrinsics. The depth input topic is “/head_camera/depth_registered/image”, which publishes the depth images that is aligned to the color images. Here we choose the depth data encoding with “16UC1” because the detection model accepts 16-byte depth images as input. The output data type is compressed for color messages and compressedDepth for depth messages, and the output topics are “/rgb_republish” and “/depth_republish”.

Subscribe and decompress these two topics on another machine with the launch file “sub_image.launch” correspondingly:

```
<launch>
<node pkg="image_transport" type="republish"
    name="rgb_decompress" args="␣compressed␣
    in:=/rgb_republish␣raw␣out:=/
    rgb_trans_raw" >
<param name="compressed/mode" value="color"/>
</node>
<node pkg="image_transport" type="republish"
    name="depth_decompress" args="␣
    compressedDepth␣in:=/depth_republish␣raw␣
    out:=/depth_trans_raw" >
<param name="compressed/mode" value="depth"/>
</node>
</launch>
```

The launch file decompresses the subscribed topics and republishes as “/rgb_trans_raw” and “/depth_trans_raw”.

### C. Call Pose Estimation in ROS

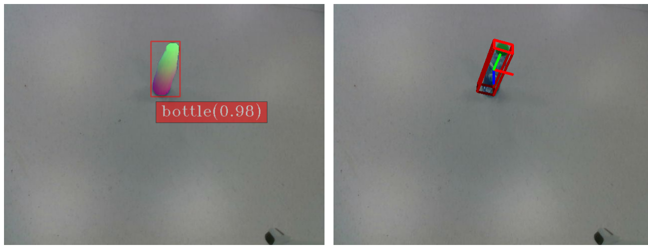
The 6D object pose estimation work [21] is open-source on Github. However, there is no ROS implementation of the work. To apply pose estimation during our task, we wrap it to a ROS package and the estimation can be called as a ROS service.

The environment and dependence installation is introduced in the README file. After the installation, the topics and the intrinsic matrix of the RGBD camera need to be set in the launch file “pose_estimation_server.launch”. The parameter *camera_optical_frame* is the frame of your rgb image topic. Then the object pose estimation service can be started with:

```
$ roslaunch nocs_srv pose_estimation_server.
    launch
```

Make sure the specific image topic are published. Then you can call the estimation service with:

```
$ rosservice call /estimate_pose_nocs True
```




(a) Predicted mask of the detected object. (b) Estimated pose of detected object with NOCS representation.

Fig. 10. NOCS model predicted results.

The detection and estimation results are saved under the "output" folder. Fig. 10(a) shows the predicted mask with the label of the detected object, while Fig. 10(b) shows its predicted pose and its NOCS representation, a bounding box.

When the estimation is finished, the transform between object and camera is published to tf message so that the robot know the object's estimated pose in the world coordinate system.

# IX. Decision Making

## A. Introduction

After having all the modules work, a framework like Behavior Trees(BT) and Finite State Machine(FSM) is needed to combines all of them and creates complex behavior, which allows the robot to finish different tasks. These methods have existed for a long time and played a important role in the game. Among them, we choose to use FSM, which is a mature, widely used framework.

There are two ROS packages implement FSM, *SMACH*[2] and *FlexBE*[16]. SMACH is a task-level architecture for rapidly creating complex robot behaviors. At its core, SMACH is a ROS-independent Python library to build hierarchical state machines. FlexBE's core is based on SMACH and provides features like drag&drop behavior creation, automated code generation, which allows us to create a behavior without coding.

## B. FlexBE

The FlexBE can be installed with:

```
$ sudo apt install ros-melodic-flexbe-
  behavior-engine
```

Then, we create ROS packages through ROS node *flexbe_widget* to stores our states and behaviors.

```
$ rosrun flexbe_widget create_repo
  mobile_manipulator
```

This command will creates two ROS packages, *mobile_manipulator_flexbe_states* and *mobile_manipulator_flexbe_behaviors*.

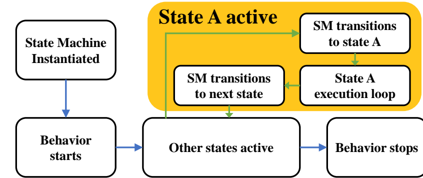



Fig. 11. State Machine state lifecycle.

*1) Custom FlexBE state:* The state machine consists of a series of states. The state in FlexBE is defined by Python class *EventState*. This class contains 6 functions that will called sequential when the state is triggered. To define a custom state, it is needed to inherit this class and then implement these 6 functions. These functions and their roles are listed in Table III. To connect with ROS, we can define some ROS publisher or subscriber in the *__init__* function and enable them in the *on_enter* function. After a state is finished, it needs to transfer to another state, so the state needs to define its *outcomes* and using *execute* function to check conditions to trigger *outcomes*. The FlexBE also provides *userdata*, which allows data to flow between states and change at runtime. The *userdata* can be treated as the *input_keys* and *output_keys*. Here is an example of declare *outcomes*, and *userdata*, which is at the file "example_state.py"

```
class ExampleState(EventState):
def __init__(self):
# Declare outcomes, input_keys, and
   output_keys by calling the super
   constructor with the corresponding
   arguments.
super(ExampleState, self).__init__(outcomes =
    ['continue', 'failed'], input_keys = ['
   input_data'], output_keys = ['output_data
   '])
```

TABLE III
FlexBE state lifecycle

| Name | Role |
|---|---|
| __init__ | state instantiated |
| on_start | (execute once) initialize resources |
| on_enter | start state actions, initialize variables |
| execute | (execute periodic) check conditions to trigger outcomes |
| on_exit | to stop running processes |
| on_stop | to clean up |

We have to develop some states according to our needs. In our case, we regard different modules as servers that provide different services. Each state is a client that sends request to one server and enable it. We write a ROS node called *robot_server* in our ROS package *fetch_common* that provides different services to interact with FlexBE.

On the other hand, for standard ROS functionality, there already exists some collections of states, like generic_flexbe_states, vigir_behaviors, youbot_behaviors and flexbe_strands which can be used directly or for reference.

*2) Custom Behavior:* The states can be combined into a behavior. The workflow of the state machine and the activity

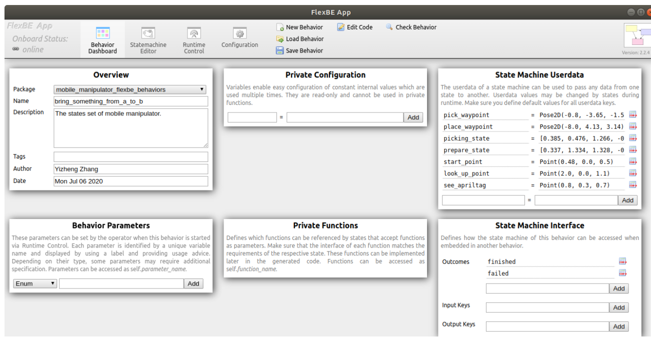


Fig. 12. FlexBE App

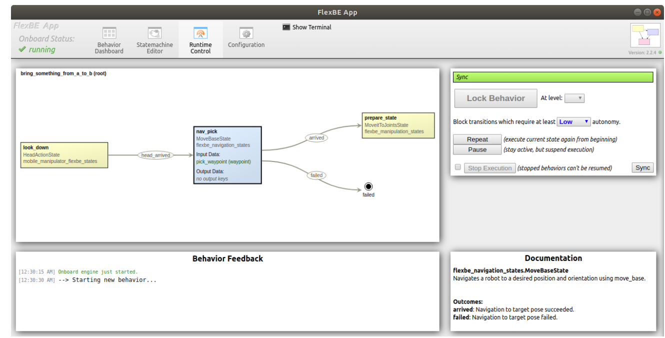


Fig. 14. FlexBE FSM Running Control

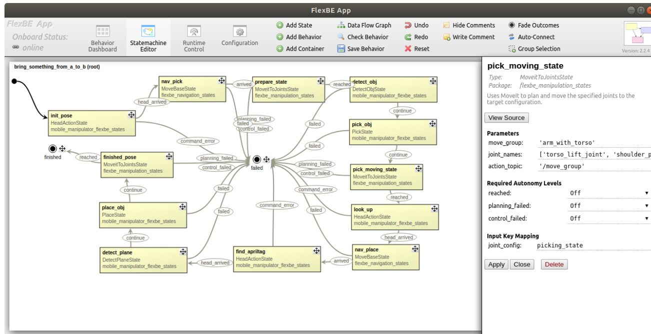


Fig. 13. FlexBE States

inside a state is shown in Fig. 11.

FlexBE provides the *flexbe_app*, an extensive user interface, shown in Fig. 12, allows us to use a graphical interface to directly design state machine. The following command launchs the *flexbe_app*

```
$ roslaunch flexbe_app flexbe_full.launch
```

In the "Behavior Dashboard" page, the description of the behavior can be written in the "Overview" and the userdata should be declared in the "State Machine Userdata".

In the menu bar, the "Statemachine Editor" page provides functions like "add state", "add behavior" and "add container". State can be added by "add state" button. A state usually contains serveral outputs according to the execution result of the state. To construct the FSM, drag the arrow to connect states. An example FSM is shown in Fig. 13.

Finally, open the "Runtime Control" page and click the "Start Execution" button to launch the FSM. This page will visualize the current active state and report the state's status. Fig. 14 shows an example of this page, in which the "nav_pick" state is active.

## X. Demo

All the modules that are needed for our mobile manipulator application have been discussed above. We are going to demonstrate how to use FlexBE to finish a task that a mobile robot navigates to a point A, recognizes a can on the table, pick it up, and navigates to another place B, and place it on another table.

We have four demos for this task: 1) run a real Fetch robot to pick a bottle in a room and place it in another room; 2) run a Fetch robot in the simulation environment to pick a coke can from a room to another; 3) run the real Kinova-Jackal robot to pick a bottle from a room and place it in another room; 4) run a Kinova-Jackal robot in the same Gazebo world as Fetch's to do the same task as the Fetch robot does in the simulation[5].

Here we introduce the demo running on simulation for Fetch robot and on real Fetch robot, respectively. Firstly, mapping will be used to create a map for navigation. In our case, the real environment map file is stored as "fetch_real/maps/sistD2.pgm". The simulation map is stored as "fetch_sim/maps/big_indoor.pgm".

For the simulation demo, firstly, we launch simulation, localization, navigation, *MoveIt* by

```
$ roslaunch fetch_sim demo.launch
```

While for the real robot demo, after bringing up the robot, we start the localization, navigation and *MoveIt* with

```
$ roslaunch fetch_real demo.launch
```

Our pose estimation module based on NOCS, is also run under a conda environment of Python3.5. We activate the environment and launch the object pose estimation module with:

```
$ conda activate NOCS #activate environment
$ source devel/setup.bash
$ roslaunch nocs_srv pose_estimation_server.
  launch
```

Then launch the *flexbe_app* and *robot_server* node to connect other modules.

```
$ roslaunch fetch_sim robot_service.launch
```

Load our "bring_something_from_a_to_b" behavior by clicking the "Load Behavior" button in "Behavior Dashboard" page, which is shown in Fig. 12.

We add 12 states to this behavior, these states will called sequential and their role are listed in Table IV. If any module executes failed, the behavior will be ended.

Finally, run the behavior by clicking the "Start Execution" button in "Runtime Control" page. The result is shown in Fig. 14.

[5]The last two demos for the Kinova-Jackal robot will be introduced in the final version, because they have not be finished yet.

TABLE IV
FLEXBE STATE LIFECYCLE

| State Name | Role |
|---|---|
| __init__ pose | Reset the robot pose |
| nav_pick | navigate to point A |
| prepare_state | set the manipulator to a prepare pose to pick the can |
| detect_obj | estimate the can's pose |
| pick_obj | using *MoveIt* pick API to pick the can |
| pick_moving_state | set the manipulator to a pose that can move from the pick pose |
| look_up | move the camera to look in front of the robot |
| nav_place | navigate to point B |
| find_apriltag | find the apriltag |
| detect_plane | using the apriltag to detect the place plane |
| place_obj | using *MoveIt* place API to place the can |
| finished_pose | set the robot to its original pose from the place pose |

### A. Run Simulation in Docker

In order to run our demo without being troubled by the configuration environment, we provide a Docker image for our reader.

1. Pull our docker image and run the demo by:

```
$ docker run --name momantu -i -t -p
    6900:5900 -e RESOLUTION=1920x1080
    yz16/momantu
```

2. Open a VNC client like "VNC viewer", and connect to the "127.0.0.1:6900". It will open a GUI and we can interact with it like our local computer. Our two workspace are put in the "root" folder and ready to use.
3. Other operation like stop/ restart from a stopped container/close the docker container by:

```
$ docker stop momantu
$ docker start momantu
$ docker kill momantu
```

## REFERENCES


[1] Tim Bailey and Hugh Durrant-Whyte. Simultaneous localization and mapping (slam): Part ii. *IEEE robotics & automation magazine*, 13(3):108–117, 2006.

[2] J. Bohren and S. Cousins. The smach high-level executive [ros news]. *IEEE Robotics Automation Magazine*, 17(4):18–20, 2010.

[3] Christian Brommer, Danylo Malyuta, Daniel Hentzen, and Roland Brockers. Long-duration autonomy for small rotorcraft UAS including recharging. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, page arXiv:1810.05683. IEEE, oct 2018.

[4] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part i. *IEEE robotics & automation magazine*, 13(2):99–110, 2006.

[5] Marco Esposito. Ifl-camp/easy_handeye.

[6] G. Grisetti, C. Stachniss, and W. Burgard. Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE Transactions on Robotics*, 23(1):34–46, 2007.

[7] G. Grisettiyz, C. Stachniss, and W. Burgard. Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 2432–2437, 2005.

[8] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. Real-time loop closure in 2d lidar slam. pages 1271–1278, 05 2016.

[9] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[10] S. Kohlbrecher, O. von Stryk, J. Meyer, and U. Klingauf. A flexible and scalable slam system with full 3d motion estimation. In *2011 IEEE International Symposium on Safety, Security, and Rescue Robotics*, pages 155–160, 2011.

[11] K. Konolige, G. Grisetti, R. Kmmerle, W. Burgard, B. Limketkai, and R. Vincent. Efficient sparse pose adjustment for 2d mapping. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 22–29, 2010.

[12] Danylo Malyuta, Christian Brommer, Daniel Hentzen, Thomas Stastny, Roland Siegwart, and Roland Brockers. Long-duration fully autonomous operation of rotorcraft unmanned aerial systems for remote-sensing data acquisition. *Journal of Field Robotics*, page arXiv:1908.06381, August 2019.

[13] MoveIt. Concept of moveit. `https://moveit.ros.org/documentation/concepts/`, 2020.

[14] E. B. Olson. Real-time correlative scan matching. In *2009 IEEE International Conference on Robotics and Automation*, pages 4387–4393, 2009.

[15] Amir Rasouli and John K Tsotsos. The effect of color space selection on detectability and discriminability of colored objects. *arXiv preprint arXiv:1702.05421*, 2017.

[16] Philipp Schillinger, Stefan Kohlbrecher, and Oskar von Stryk. Human-Robot Collaborative High-Level Control with an Application to Rescue Robotics. In *IEEE International Conference on Robotics and Automation*, Stockholm, Sweden, May 2016.

[17] Tixiao Shan and Brendan Englot. Lego-loam: Lightweight and ground-optimized lidar odometry and mapping on variable terrain. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4758–4765. IEEE, 2018.

[18] Andreas ten Pas, Marcus Gualtieri, Kate Saenko, and Robert Platt. Grasp pose detection in point clouds. *The International Journal of Robotics Research*, 36(13-14):1455–1473, 2017.

[19] R. Y. Tsai and R. K. Lenz. A new technique for fully autonomous and efficient 3d robotics hand/eye calibration. *IEEE Transactions on Robotics and Automation*, 5(3):345–358, 1989.

[20] Roger Y Tsai, Reimar K Lenz, et al. A new technique for fully autonomous and efficient 3 d robotics hand/eye calibration. *IEEE Transactions on robotics and automation*, 5(3):345–358, 1989.

[21] He Wang, Srinath Sridhar, Jingwei Huang, Julien Valentin, Shuran Song, and Leonidas J. Guibas. Normalized object coordinate space for category-level 6d object pose and size estimation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[22] John Wang and Edwin Olson. AprilTag 2: Efficient and robust fiducial detection. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4193–4198. IEEE, oct 2016.

[23] Wikipedia contributors. Mobile manipulator — Wikipedia, the free encyclopedia, 2020. [Online; accessed 30-August-2020].

[24] Haoyang Ye, Yuying Chen, and Ming Liu. Tightly coupled 3d lidar inertial odometry and mapping. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 3144–3150. IEEE, 2019.

[25] Ji Zhang and Sanjiv Singh. Loam: Lidar odometry and mapping in real-time. In *Robotics: Science and Systems*, volume 2, 2014.

[26] Kaiyu Zheng. Ros navigation tuning guide. *arXiv preprint arXiv:1706.09068*, 2017.