Mobile Manipulation Tutorial

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 robotics arm installed on a mobile robotic platform. Through this combination, robots have an unlimited workspace and the ability to operate in a multitude of environments.

Remote-controlled mobile manipulators are used in certain scenarios, like bomb disposal and planetary exploration. However, autonomous mobile manipulation is far more interesting, as it opens applications in many different areas like service robotics, manufacturing, logistics and construction.

Autonomous mobile manipulation is a highly complicated task, involving many different components. Those include the mechatronics system of the mobile platform, the robotic arm and the end effector, powerful sensors and a complex software system. In this tutorial it is assumed that the hardware problem is already solved, either with a real robot or in simulation. The emphasis is then on the software and algorithms.

In this tutorial we concentrate on one of the most basic applications of mobile manipulation: fetching an object. Obviously, mobile manipulation includes more scenarios than just this pick and place action. For example, the DARPA Robotics Challenge required the use of power tools and the operation of valves and the opening of doors [8].

Given the complex nature of mobile manipulation, the easiest, simplest hardware for the task is assumed here: Using a wheeled base, a commercial robot arm with at least six degrees of freedom (DoF), such that most end-effector poses up to a certain distance are reachable, and a simple parallel gripper. Laser scanners are used for mapping, for their wide field of view, long range and high accuracy. An RGB-D camera is used for object recognition and localization. The robot starts at a known location with a known map and the semantic information for the mapping of rooms to coordinates is provided. The simple task is then to: 1) drive to a given location; 2) there, identify and grasp a specified object; 3) drive to a different location and 4) place the object on a specific QR code.

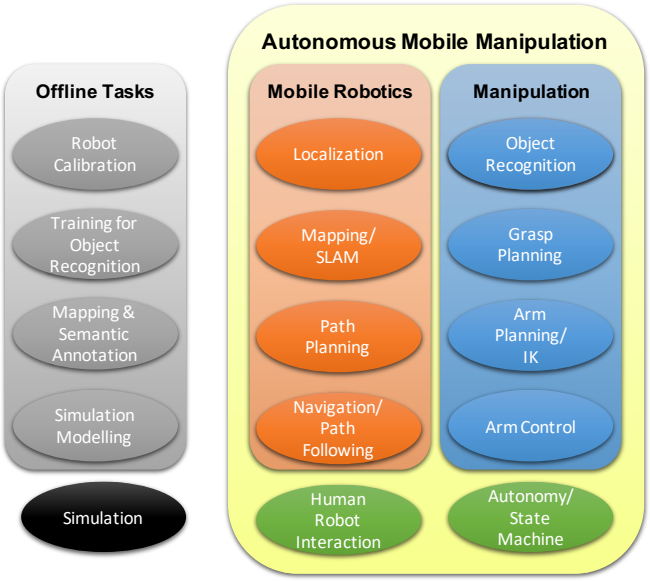


Fig. 1. Overview of the most important problems for a mobile manipulation pick and place task.

Even this simple task already requires all the software components shown in Fig. 1. This assumes that all the more benign robotics software tasks like sensor and actuator drivers, communication, visualization, data recording and error logging are already solved by a middleware. This tutorial is using the Robot Operating System (ROS version 1) for this, as it is widely in robotics research. Thanks to the robotics community many open source software solutions are available for the components shown in Fig. 1. Whenever possible, this tutorial is using well known and established open source ROS packages.

Typically, individual software packages come with their own documentation and tutorials. 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 code cannot be applied to the users' robots directly. Additionally, the system integration of the individual components to a working system is more complex than just the sum of the complexity of the components.

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 func-

*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

tions each component undertakes, and how to connect these components. The tutorial also warns of common pitfalls and provides a good starting point for beginning the research on autonomous mobile manipulation.

The tutorial comes in several parts. This text provides an overview and system overview (Section II) of a typical mobile manipulation software stack. Section III gives information of how to run our simulation demo in docker and how it install is natively. In Section IV some select modules are described. A longer, supplementary text is available online, that goes into the details of all the modules. The Mobile Manipulation Tutorial (MoManTu) GitHub webpage[1] provides videos, recorded sensor data and other supplementary data of the tutorial. It also points to the GitHub repositories of all the software.

The MoManTu provides four demos based on two robots, running on both in real and simulation environments, and the corresponding code - see Figure 2. The commercially available Fetch robot features a back-drivable 7 DoF robot arm with a parallel gripper, a torso lift joint, a pitch, yaw head with an RGB-D camera as well as a 2D Laser Scanner. The other robot was build from commercial components at the Mobile Autonomous Robotic Systems Lab (MARS Lab) of ShanghaiTech University. It is a Clearpath Jackal mobile base, that offers higher mobility and speed than the Fetch. The robot uses a Kinova Jaco Gen 2, 6 DoF manipulator with a three finger hand. The sensors consists of a wrist-mounted RGB-D camera (Intel RealSense D435) and a 16-beam 3D LRF (Velodyne Puck). We thus have two very common camera placement covered: head-mounted and arm-mounted.

The simulation is using Gazebo. Other robot simulation software exists, like CoppeliaSim (V-Rep) or Webots, but Gazebo was chosen for its popularity and tight integration of ROS. We provide the MoManTu system for the two simulated robots ready to use in virtual apartment environment. To enable the reader to test and run the software very easily on their own computer, we provide a docker image. After connecting to the docker with a VNC viewer, the demo is ready to run.

In Table I the packages needed during running the "bring object" task are listed, as well as the the packages which are needed offline as a preparation to actually run the system. It can be seen that for several packages two alternatives are offered. This is to keep the tutorial general, educate the reader how to easily switch between the packages, to make the interfaces between the modules clearer and to enable the reader to modify the system and use their own modules. Supporting two different robots serves the same purpose.

II. System Overview

The Mobile Manipulation Tutorial (MoManTu) teaches the reader such a complex system on the example of the simple task of fetching and bringing a bottle from one place to another. From the software side we distinguish the offline tasks of calibration, training for object recognition, mapping the environment and building simulation models for the robot and the environment from the online parts of actually running the robot. Fig. 1 depicts those most important software modules. As a simple solution, MoManTu keeps the path planning problem, the arm planning problem and the task planning of the autonomy separated, while more advanced systems may opt for combined task and motion planning [15]. We can thus separate the problem into a mobile robotics part, that is responsible for driving the robot to a certain location, the manipulation part, that is responsible for finding the object to be picked and the according arm motions, and the more general autonomy part with a simple state machine and the human robot interaction part.

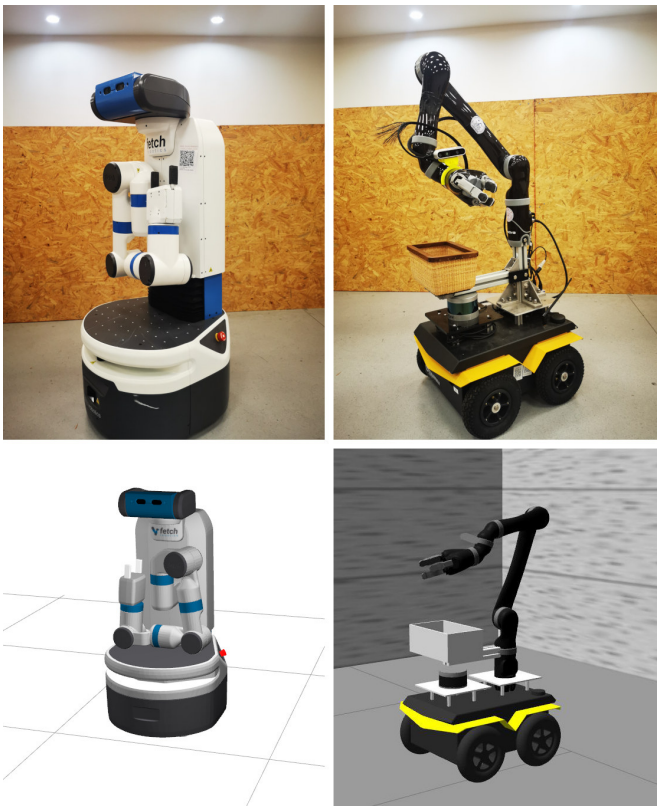


Fig. 2. Mobile Manipulation Systems used in MoManTu: Fetch, real and simulated; Jackal & Kinova, real and simulated.

Table I shows which open source software we use for the different modules. The mobile robotics part is making full use of the ROS Navigation stack. The robot is provided with a map of the environment, so it does not have to do exploration. AMCL is used to localize the robot in this map. In order to drive to a specified 2D coordinate in the map, ROS Navigation is building costmaps (global and local) that combine the provided map with current scan data. A global path planner is then planning a path to the coordinates, taking the kinematic constraints of the robot (e.g. differential drive for the systems in this paper and the robot size) into account. A local planner is then actively avoiding obstacles while

[1] https://momantu.github.io/

[2] Not all items/ code are finished at the time of the initial submission of the paper. We are continuing working on the system and paper.

TABLE I

MODULES USED IN THE MOMANTU SYSTEM

Role	Package
Localization	AMCL
Local Costmap	ROS Navigation
Path Planning	ROS Navigation
Path Following	ROS Navigation
Arm Control 1	kinova_ros
Arm Control 2	fetch_ros
Category Detection & Pose	NOCS
Object Detection & Pose	(todo[2])
Object Place Pose	AprilTag_ROS
Grasp Planning	/
Arm Planning 1 & IK	MoveIt Pick and Place
Arm Planning 2 & IK	MoveIt directly (todo)
Human Robot Interaction 1	RViz & FlexBE App
Human Robot Interaction 2	Speech (todo)
Decision Making	FlexBE
Offline Packages	
Simulation World Building	Gazebo
SLAM 2D	Cartographer
SLAM 3D	Cartographer (todo)
Camera Calibration	camera_calibration
Camera Pose Detection	AprilTag_ROS
Hand-eye Calibration	easy_handeye

following the global path.

In MoManTu, the goal coordinates come from the state machine that is coordinating and controlling all actions of the robot. Fig. 3 is showing the simple machine, depicting the steps needed to pickup and place the bottle. In our demo the coordinates for robot navigation are saved as attributes in the state machine, while more sophisticated systems may store those in a semantic map, for example with coordinates for "kitchen" or "office". We are using FlexBE, which allows to create complex robot behaviors as a state machine without the need for manually coding them. It also serves as a Graphical User Interface (GUI) to start and control the robot, thus also being a simple form of Human Robot Interaction. An additional interface to the robot, that closely interacts with FlexBE, is the speech control (ToDo).

Once the robot has driven to the goal position, it is using the RGB-D camera to find the object to be picked up. The color image is used to recognize the object, while the depth data is essential to find the 6D pose of the object. Through deep learning, computer vision has seen tremendous developments for object recognition in the recent years. We are providing three approaches to this problem, which can be used interchangeably. In the simplest case AprilTags are used to specify an object's pose. The second option we implemented in MoManTu is to use the Normalized Object Coordinate Space for Category-Level (NOCS) 6D object pose and size estimation [14]. We wrapped this into a ROS service. This software can distinguish different categories (e.g. bottle and bowl), but not different objects (e.g. coke vs. water bottle). As a third option we implement a Deep Learning object detection solution (ToDo).

Once an object's pose is detected, complex manipulation systems typically do grasp planning, i.e. they find the optimal ways of how to grasp the object with the given end effector.

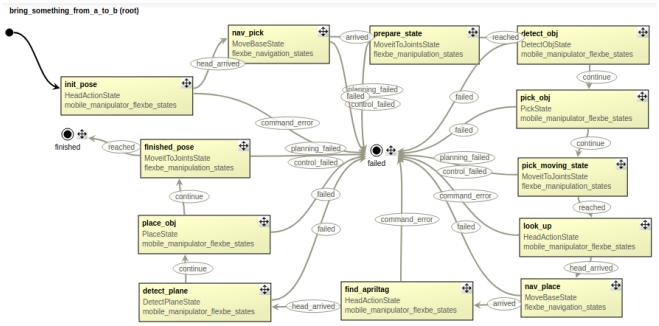


Fig. 3. States of the FlexBE State Machine.

This tutorial is skipping this step by simply assigning a grasp pose with zero roll, pitch and yaw w.r.t. the manipulator base and the bottle's center as position. This approach works for this simple task, but may fail for more complex objects.

ROS comes with a sophisticated arm planning and control system called MoveIt!, which is also used in this tutorial. Once it is properly setup with the arm parameters, robot collision information, calibration data and access to live depth information, it can solve the Inverse Kinematics (IK) of the arm, plan trajectories and control the execution of those trajectories all while avoiding collisions with itself, the known robot base and dynamic objects it knows from the depth camera. We provide two approaches of how to use MoveIt. The first implementation uses the basic MoveIt commands to plan trajectories, execute them and use the gripper (ToDo). The second approach is using Pick and Place, an extension to MoveIt to simplify such pick and place tasks.

The video attached to the tutorial, which can also be found on the MoManTu website, shows how the system is solving our simple mobile manipulation task.

A. *Offline Modules*

There are several tasks that need to be done before the robot can be run. The first of those offline tasks is the extrinsic calibration of the camera that is used to find the object with the robot arm. The tutorial is providing a detailed discussion on how to solve this hand-eye calibration problem.

The object recognition with deep learning relies on training images with ground truth labels. In the final version of this tutorial we will provide one such module and also describe how to collect the training data and do the training. (ToDo).

In a typical mobile manipulation scenario a map of the environment is already provided, possibly with semantic information about the names of rooms, etc. More advanced systems might also update the map while driving around - but this is not part of MoManTu. But we provide detailed instructions on how to create such a map using 2D and 3D laser scanning data. Cartographer [6] is used for that. It is also explained how to store and then use this map in the live system.

For many robot researchers the use of a simulator is important to do experiments and tests without the additional

Fig. 4. Our Gazebo simulation world.

complexity that real robot systems pose. MoManTu is providing models of the two different models and also two different environments - Fig. 4 is showing one of them. It is also outlined how to model new robots and environments and how to use the Gazebo simulator.

III. Running MoManTu

A. MoManTu Simulation from Docker

The easiest way to get started with playing with the tutorial is to use the provided Docker image.

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
```

Launch the simulator and the MoManTu software by running this command in the terminal:

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

Our pose estimation module based on NOCS, which runs 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 ~/nocs_ws/devel/setup.bash
$ roslaunch nocs_srv pose_estimation_server.
    launch
```

Finally, launch the *flexbe_app* and *robot_server* node to connect other modules and start the FSM. The demo video can be found on our web page.

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

B. MoManTu from Source

Besides using the provided Docker container, users are encouraged to work with the system natively. Thus, this section introduces how to fetch, compile and run the code by hand. 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 "momantu_fetch_ws" for the Fetch robot examples, and the other named "momantu_kinova_ws" for the Kinova-Jackal robot examples. We provide all code and configuration files mentioned in this tutorial on GitHub[3]. The main dependencies should be installed through apt by using the command provided in the readme of the momantu_fetch repository. The two repositories above both include the simulation and real-robot parts. Let's first download the sources and their further dependencies, and compile them:

```
$ cd
$ source /opt/ros/melodic/setup.bash
$ mkdir -p momantu_fetch_ws/src
$ cd momantu_fetch_ws/src
$ git clone https://github.com/momantu/
    momantu_fetch.git
$ cd momantu_fetch
$ git submodule update --init --recursive
$ cd ../..
$ rosdep install --from-paths src --ignore-
    src -y -r
$ catkin_make
$ source ~/momantu_fetch_ws/devel/setup.
    bash
```

In order to get the Jackal-Kinova version simply replace "fetch" with "kinova".

The last line of the listing is sourcing the ROS environment of the momantu workspace to the current terminal session. Any new terminal instance that is created needs to run that line if any program or ROS message from the workspace is to be used.

We also provide the NOCS wrapper repository[4] for 6D object pose estimation, which needs to be installed in another workspace according to its README file. The startup procedure for the whole system is then the same as in docker.

IV. Modules

Some more details for a few modules are presented in this section, while in-depth tutorials for all modules are provided in the extended online version of the mobile manipulation tutorial.

A. MoveIt

MoveIt is an open-source robotics manipulation platform. *MoveIt*[9] integrates modules like motion planning, manipulation, perception, collision detection and provides a node called *move_group*, which pulls all the individual components

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

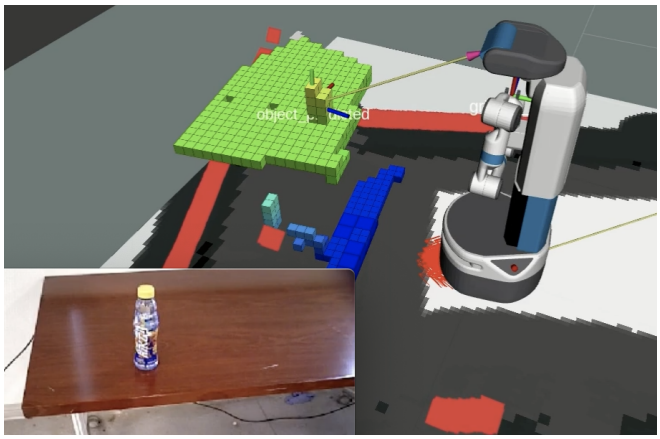


Fig. 5. Integrate with *OctoMap*.

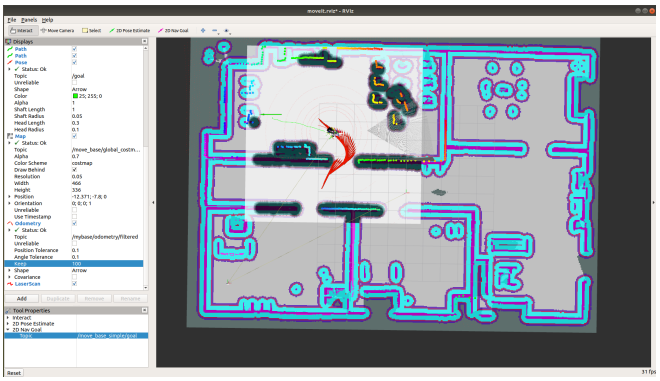


Fig. 6. Navigation given a map. The global map is padding and shown as costmap. The colorful points are the obstacle points scanned by the Lidar. Based on it, the padded obstacles are shown in the local costmap. The red arrow series are the robot poses provided by the odometry. The green arrow is the given goal.

together to provide a set of ROS actions and services for users to use. We demonstrate how to use *move_group* C++ interface to pick an object and put it on another table by Fetch mobile manipulator.

After installing *MoveIt*, you can bring up our robot with in Gazebo by:

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

move_group integrates different components, each components needs parameters, which are loaded from a ROS Param Server, to configure itself. The parameters for our robots have been setup already. A *MoveIt Setup Assistant* exists to help configuring custom robot arms.

1) 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 RGB-D camera and gives point cloud, so we use thePointCloud plugin to handle this information.

2) Pick and Place: The most common usage 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 the object recognition step. *MoveIt* provides several modules to do the pick and place task, including *Pick and Place*, *MoveIt grasp* or the *MoveIt Task Constructor*. Third party packages, like the *Grasp Pose Detection (GPD)*[12], use learning-based methods to detect 6-DOF grasp poses for a 2-finger robot hand in 3D point clouds.

MoManTu is demonstrating two ways of how to use MoveIt: 1) using the API directly to directly control each step and 2) use the *Pick and Place* package. More details on both can be found in the online version of this tutorial.

B. Navigation

Navigation is the technique that supports the robot to move from one place to another place. Since our the mobile robot base only needs to navigate in the 2D plane, we employ the widely-used ROS 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
```

1) Navigation given Map: A map is required to navigate to a specified goal. In MoManTu, the environmental map created beforehand is loaded and published by a *map_sever*.

The *move_base* node is then responsible for the navigation. The input requirements of *move_base* include sensor data, sensor transformations, odometry and a goal.

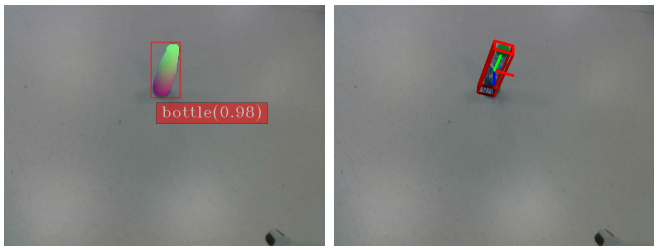
The odometry data is provided by the robot. The default subscribing topics of laser data and odometry to *move_base* are "/scan" and "/odom", respectively. If other topics are specified to subscribed instead, remapping of topics will be needed. If the robot subscribes to another controller topic rather than "/cmd_vel", e.g., a topic with namespace, a remapping for it is needed, too. The parameter configurations for the navigation are loaded as yaml files. Users can check these files to explore the effect of various parameters. [16] gives a detailed navigation tuning guide, which helps readers to tune the navigation parameters efficiently.

With a given map, we localize the robot in the environment with the *amcl* package, which is a probabilistic 2D localization system. Besides the map, *amcl* requires odometry data, laser sensor data and an initial pose as inputs.

Fig. 6 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.

C. Object Pose Estimation

Before gripping 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



(a) Predicted mask of the detected object.

(b) Estimated pose of detected object with NOCS representation.

Fig. 7. NOCS model predicted results.

model, we will introduce a method based on category-level object detection. The work [14] 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 category-level detection is that 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.

1) Call Pose Estimation in ROS: The 6D object pose estimation work [14] 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 RGB-D 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.

The detection and estimation results are saved under the "output" folder. Fig. 7(a) shows the predicted mask with the label of the detected object, while Fig. 7(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 a tf message, such that the robot knows the object's estimated pose in the world coordinate system.

D. Decision Making

A state machine is needed to combine and coordinate the execute of the different modules to generate complex behaviors. This then allows the robot to finish different tasks. There are two popular ROS packages that implement state machines, *SMACH*[2] and *FlexBE*[11]. 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. MoManTu is implemented with FlexBE.

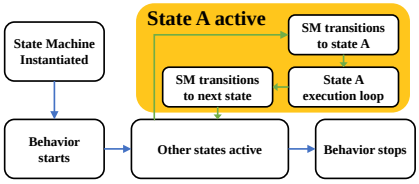


Fig. 8. State Machine state lifecycle.

A state in FlexBE is defined by a Python class *EventState*. This class contains six functions that are called sequentially when the state is triggered. To define a custom state, it is needed to inherit this class and then implement these six functions. These functions and their roles are listed in Table II. 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*.

TABLE II
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

The states can be combined into a behavior. The workflow of the state machine and the activity inside a state is shown in Fig. 8.

FlexBE provides the *flexbe_app*, an extensive user interface allows us to use a graphical interface to directly design state machines. The states defined for MoManTu can be found in Fig. 3.

E. Hand-Eye Calibration

In order to allow the robotic arm to flexibly operate objects in the environment, a camera is usually mounted on the robot. All vision information obtained by the camera is in 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 [13] and can be done offline, before running the MoManTu system. Two camera installations are common: eye-in-hand and eye-on-base [4]. Eye-in-hand means mounting the camera on the arm. 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.

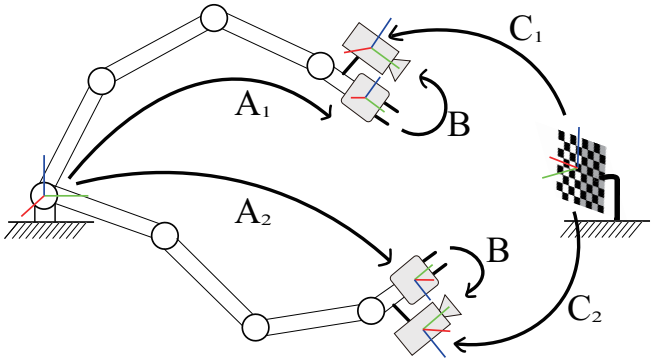


Fig. 9. Eye-in-hand calibration.

Manipulators usually have a big operating space, which typically larger than the field of view of the camera. A simple solution to this problem is to mount the camera near the end-effector of the arm, which corresponds to the eye-in-hand calibration problem. The Jackal-Kinova robot used in this tutorial has such a setup. Mounting the camera on the robot with a pan, tilt mechanism is more complicated, but has the advantage of a typically better view on the whole workspace. The Fetch robot uses this setup and is thus calibrated using the eye-on-base approach.

Here we only introduce hand-in-eye calibration, because the calibration of the hand-on-base case is quite similar and because the Fetch robot comes already calibrated.

Fig. 9 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 ith 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 ith sample. Then it translates into a problem of solving B with known A_i and C_i by moving the hand effector to different poses to collect groups of samples.

The steps for the calibration, which are covered in more detail in the supplementary material, are: 1) Intrinsic calibration of the camera (focal length, image sensor format, principal point, lens distortion). 2) Collecting synchronized motion samples from of the end effector (w.r.t. the robot base, through forward kinematics) and the camera (w.r.t. a global frame, using an AprilTag). 3) Solving the hand-eye-calibration by calculating the transform B between the end effector and the camera, through optimization. 4) Publishing B in the live system.

F. Mapping

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) [3], [1]. SLAM is a process that builds the map while figuring out where the agent is located in the environment. Building a 2D map saves computation resource and sensor costs, compared to building a 3D map, and is usually sufficient for ground navigation. 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* [5] is a 2D laser SLAM based on a particle filter framework, combining odometry and laser information. *Hetor_SLAM* [7] matches laser data between frames with an optimization method without requiring odometry information. *Karto* [10] is a graph optimization SLAM based on matrix sparseness, including loop closure, which has advantages in large-scale mapping. In the tutorial we are using *Cartographer* [6], which is a complete real-time laser SLAM system. It contains a robust front end and a pose graph back end based on submaps and node constraints.

When using 3D laser scanners such as Velodyne's VLP-16 Lidar sensor, its 3D sensor data can be converted to a simulated 2D scan, and it is thus also suitable for 2D mapping. In MoManTu cartographer is used to crate and save a map in a separate mapping run, before the actual demo system is run. More details can be found in the online tutorial.

V. Conclusions

This paper provides demo code for a simple mobile manipulation task. Open source modules are used and detailed descriptions and steps how to use them are provided in the extended online version of the tutorial. The system was demonstrated on two different real mobile manipulation platforms, but is also working in the Gazebo simulator. To lower the entry barrier for trying out the software, a docker image is provided that can run the demo out of the box. The MoManTu website provides additional content like videos, ros bagfiles from the real robot runs and links to all the github repositories used. The repositories additionally provide documentation in the readme files and provide issue trackers to answer questions about the system. The website is also the place where future updates to the tutorial will be announced.

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

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

[5] 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.

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

[7] 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.

[8] Eric Krotkov, Douglas Hackett, Larry Jackel, Michael Perschbacher, James Pippine, Jesse Strauss, Gill Pratt, and Christopher Orlowski. The darpa robotics challenge finals: Results and perspectives. *Journal of Field Robotics*, 34(2):229–240, 2017.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Jason Andrew Wolfe, Bhaskara Marthi, and Stuart J Russell. Combined task and motion planning for mobile manipulation. In *ICAPS*, pages 254–258, 2010.

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