Chapter 1 - Creating a Simple ROS Package

1. Overview:

The my_ros_package is a ROS package created as a practice exercise from Chapter 1 of "Mastering ROS." It is designed to introduce the core underlying concepts of ROS, including package structure, simple node creation, and basic communication.

2. Package Structure:

The package includes several components, each integral to the ROS package architecture:
CMakeLists.txt: The build configuration file, which specifies how to build the code, the necessary dependencies, and the targets (executables and libraries).
package.xml: Contains metadata about the package, such as the name, version, maintainers, and dependencies.
src/: Directory containing the C++ source files for the node.
launch/: Holds the launch files for starting nodes and setting parameters.

3. Node:

my_node.cpp: A simple ROS node that publishes "Hello ROS World" messages to the my_topic. This node demonstrates the basic structure of a ROS node, initialization of the node, creation of a publisher, and the publishing loop.

4. CMakeLists.txt and package.xml:

CMakeLists.txt: Defines the build process for the package. It includes instructions for building the source files and linking against ROS libraries. It's where the executable for my_node is defined.
package.xml: Provides a standardized format for maintaining package information and managing dependencies. It ensures the ROS build system recognizes and correctly builds the package.

5. Launch File:

my_launch_file.launch: A launch file used to start the my_node. Launch files in ROS are used to start one or more nodes with specific parameters and configurations, simplifying the process of launching complex systems.

6. Building and Running:

The package is built using the catkin build system, which is the standard for ROS 1. After building, the node can be run individually, or the launch file can be used to start the node with any specified parameters or configurations.

7. Practical Implications:

Through this package, users gain a practical understanding of:

ROS Package Structure: Learning how a standard ROS package is structured and the purpose of each component.

Node Basics: Understanding how to write a simple node, including initializing the ROS system, creating publishers, and sending messages.
Building and Running: Gaining skills in compiling a ROS package and running nodes, either individually or through launch files.

8. Conclusion:

The my_ros_package serves as a foundational learning tool for understanding the basics of ROS. It introduces users to the standard package structure, node creation, and the build system. This knowledge forms the base for more complex and integrated ROS projects.

Chapter 2 - Mastering ROS Programming

1. Overview:

The mastering_ros_demo_pkg is a ROS package created as a practice exercise from Chapter 2 of "Mastering ROS." It demonstrates fundamental ROS concepts including creating packages, nodes, custom messages, services, and using launch files.

2. Package Structure:

The package consists of several directories and files, each serving a specific purpose in the ROS ecosystem:

CMakeLists.txt: The build configuration file for the package. It specifies the package name, dependencies, build options, and the targets (executables and libraries) to build.
package.xml: Contains metadata about the package such as name, version, maintainers, and dependencies.
src/: The source directory contains the C++ source files for the nodes.
msg/: This directory holds custom message type definitions used by the nodes for communication.
srv/: Contains custom service type definitions for providing synchronous remote procedure calls.
launch/: Launch files for starting multiple nodes and setting parameters are stored here.

3. Nodes:

demo_topic_publisher.cpp: A simple publisher node that continuously broadcasts a string message on the topic "chatter." It demonstrates how a node can publish data to a topic.
demo_topic_subscriber.cpp: A subscriber node that listens to the messages on the "chatter" topic and prints the data to the console. It showcases how nodes can receive and process data from topics.

4. Custom Message and Service:

demo_msg.msg: A custom message type that includes a string and an integer. It demonstrates how to define custom data structures for communication in ROS.
demo_srv.srv: A custom service definition comprising a request and response part. This service is used to demonstrate synchronous communication between nodes.

5. CMakeLists.txt and package.xml:

CMakeLists.txt: This file is crucial for building the ROS package. It defines how to build the code and generate messages. It also links the necessary ROS libraries and includes directories.
package.xml: Provides package information and manages dependencies. It ensures that all necessary components are present for building and running the package.

6. Launch File:

demo_topic.launch: A launch file to start both the publisher and subscriber nodes simultaneously. It simplifies the process of running multiple nodes and can be customized to set parameters and reconfigure node settings.

7. Building and Running:

The package is built using the catkin build system. Once built, the nodes can be run individually or together using the provided launch file. This demonstrates the practical aspects of compiling and running ROS nodes.

8. Learning Outcomes:

Through this package, the user learns how to set up a basic ROS package, write publisher and subscriber nodes, define custom message and service types, and manage multiple nodes using a launch file. It serves as a foundational practice for understanding how ROS nodes communicate and perform tasks in a robotic system.

Chapter 3 - Working with ROS for 3D Modeling

1. Overview:

Chapter 3 of "Mastering ROS" introduces the concepts and practices for 3D modeling in ROS, focusing on designing and visualizing two complex robotic structures: a seven-DOF manipulator and a differential drive robot. The process involves using Unified Robot Description Format (URDF), XML Macros (Xacro), and visualization tools like RViz.

2. Seven-DOF Manipulator:

- Purpose & Design:
    - Objective: The seven-DOF manipulator is designed to simulate a robotic arm with complex movements and reachability. Each DOF represents a joint, allowing for intricate positioning and orientation.
    - Design Considerations: The manipulator's design considers link lengths, joint types (mostly revolute), and the range of motion. The end effector's design is crucial for the intended task (e.g., gripping, welding).
- Modeling with URDF/Xacro:
    - URDF: Initially, URDF files are used to define the physical and visual aspects of the robot, including links (arm segments) and joints (connections between links).
    - Xacro Enhancement: To manage complexity, Xacro is used to create reusable components and parameterize the design, making the model more manageable and adaptable.
- Components & Structure:
    - Links: Defined with visual (appearance), collision (physical space), and inertial (mass and inertia) properties. Each link represents a part of the manipulator, like the base, upper arm, forearm, wrist, etc.
    - Joints: Specify the relationships and movements between links. They define the type of motion (e.g., revolute, continuous) and the axes of rotation or translation.
- Visualization with RViz:
    - Setup: The view_seven_dof_arm.launch file includes the necessary nodes for visualization, such as robot_state_publisher and rviz.
    - Process: Upon launching, RViz provides a 3D visual representation of the manipulator. Users can interact with the model, examine joint movements, and understand the robot's physical capabilities.

3. Differential Drive Robot:

- Purpose & Design:
    - Objective: The differential drive robot is a mobile robot with two main driving wheels on either side and one or more caster wheels for stability. It's designed for navigation and is commonly used in various robotics applications.
    - Design Considerations: Wheel size, spacing, and the robot's body dimensions are critical. The design also includes provisions for sensors and computing hardware.
- Modeling with URDF/Xacro:

    - URDF: Defines the robot's base, wheels, and caster wheels, including their dimensions, colors, and physical properties.
    - Xacro Enhancement: Simplifies the URDF by allowing parameters and macros, making adjustments and maintenance more straightforward.
- Components & Structure:
    - Base: The central part of the robot, usually housing the motors, battery, and computing resources.
    - Wheels: Two main wheels are attached to the base with continuous joints, allowing rotation. Their motion is synchronized to achieve forward, backward, and turning movements.
    - Caster Wheels: One or more caster wheels provide stability and support. They usually have passive joints allowing free movement in any direction.
- Visualization with RViz:
    - Setup: The view_diff_wheeled_robot.launch file prepares the environment for visualizing the differential drive robot.
    - Process: Users can observe the robot in a 3D space, understand its structure, and simulate basic movements in RViz.

4. Building and Running:

- Building Process:
    - Both robot models are built using the ROS build system (catkin). The URDF/Xacro files are processed, and the necessary dependencies are resolved.
    - The build process ensures that all files are correctly formatted and that the model is structurally sound.
- Running and Interaction:
    - Launch files are used to start the visualization process. They set the necessary parameters and start the nodes required for RViz and state publishing.
    - Users can interact with the models in RViz, adjusting parameters, simulating movements, and gaining insights into the robot's design and functionality.

Chapter 4 - Simulating Robots Using ROS and Gazebo

Simulation Setup:
1. Gazebo Integration with ROS: Gazebo is integrated seamlessly with ROS, providing a dynamic environment for simulating robots with realistic physics. ROS nodes interact with Gazebo using plugins and controllers to simulate sensor data, joint movements, and environmental interactions.

2. Robotic Arm Simulation:
   - Model Preparation: The robotic arm's URDF is augmented with Gazebo-specific tags to simulate physical properties, visual aspects, and controls.
   - Transmission and Control: Transmission elements and the `gazebo_ros_control` plugin are used to link ROS controllers to the simulated joints.
   - 3D Vision Sensor: A simulated depth sensor is added to the arm, demonstrating how Gazebo can incorporate complex sensory equipment.

3. Differential Wheeled Robot Simulation:
   - Basic Setup: The wheeled robot's URDF includes Gazebo plugins for differential drive capabilities, allowing it to navigate the simulated environment.
   - Laser Scanner: A laser scanner is simulated atop the robot, providing environmental data crucial for navigation and interaction.
   - Teleoperation: A teleoperation node is used to control the robot's movements within Gazebo, illustrating real-time interaction and control.

ROS Controllers:
1. Joint State Controller: Publishes the state of each joint in the robot, providing real-time feedback on positions, velocities, and efforts.
2. Position Controllers: Allow for precise control over the joint positions, utilizing PID loops for accurate and stable movements.
3. Differential Drive Controller: Manages the wheel joints of the differential wheeled robot, interpreting velocity commands to drive the robot.

Interactivity and Visualization:
1. RViz Integration: RViz is used alongside Gazebo to visualize sensor data, robot state, and environmental interactions. It provides a comprehensive view of the robot's functioning and its surroundings.
2. Teleoperation: Users interact with the simulation using teleoperation nodes, directing the robot's movement and observing its behavior in various scenarios.

Conclusion:
Chapter 4 of "Mastering ROS" provides an extensive guide to simulating robots in Gazebo, detailing the process from model preparation to real-time control and visualization. The

practices and codes offer a hands-on approach to understanding and implementing robot simulations, revealing the power and versatility of ROS and Gazebo in robotic development and research. The chapter lays a solid foundation for anyone looking to dive deeper into robotic simulations and control systems.

Chapter 5 - Simulating Robots Using ROS, CoppeliaSim, and Webots

CoppeliaSim with ROS:

1. Setup and Configuration: CoppeliaSim is installed and configured to communicate with ROS using the RosInterface plugin. This plugin allows CoppeliaSim to act as a ROS node, facilitating interaction via topics and services.

2. Simulation of Robotic Arm: The robotic arm simulation is set up by importing the URDF file into CoppeliaSim and enabling motors and control loops. The Lua scripts are modified to establish ROS publishers and subscribers, controlling the arm through ROS topics.

3. Understanding RosInterface Plugin: The RosInterface plugin's functionalities are explored, demonstrating how to interact with CoppeliaSim using ROS topics. Lua scripts are used to define how the simulation interacts with ROS, enabling control over simulated objects.

Webots with ROS:

1. Installation and Introduction: Webots is installed, and its integration with ROS is facilitated through the `webots_ros` package. The software's capabilities and structure, including world configuration files, controllers, and physical plugins, are introduced.

2. Simulating a Mobile Robot: A new simulation scene is created with Webots, containing objects and a mobile robot. The robot's motion and sensors are simulated, and a controller is implemented to manage its behavior based on sensor inputs.

3. Writing the First Controller: A new controller is written in C++ for the mobile robot. The controller's purpose is to command the wheel velocities based on predefined rules. The process involves creating the controller file, writing the controller code, compiling it, and then associating it with the robot.

4. Integrating ROS and Webots: The integration between ROS and Webots is established, allowing for the control of Webots simulations using ROS topics and services. The `webots_ros` package is utilized to provide a communication bridge, enabling the control of the robot's joints and the streaming of sensor data via ROS.

5. Teleoperation Node: A ROS node is implemented to control the wheels' velocity of the e-puck robot in Webots based on `geometry_msgs::Twist` messages. The node subscribes to the `/cmd_vel` topic, processes the received messages, and sends the appropriate commands to the robot in the Webots simulation.

6. Starting Webots with a Launch File: The process of starting Webots using a ROS launch file is described. This approach simplifies the launching of simulations and associated nodes, providing a streamlined way to initiate and control Webots environments.

In conclusion, Chapter 5 guides through the setup and use of CoppeliaSim and Webots with ROS, providing hands-on experience in simulating and controlling different robots. It covers the necessary tools, scripts, and techniques required to effectively integrate these powerful simulators with ROS, offering a solid foundation for more complex robotic simulations and applications.

Chapter 6 - utilizing ROS MoveIt! for robot manipulation and the navigation stack for autonomous navigation

ROS MoveIt! and Navigation Stack Overview:
- ROS MoveIt! is an advanced motion planning framework, allowing users to manipulate their robot in a 3D space with advanced motion planning algorithms, 3D perception, kinematics, and control.
- ROS Navigation Stack is designed for mobile robot navigation, offering a variety of tools and libraries to navigate robots autonomously in an environment.

Setting up and Configuring MoveIt!:
1. Installation: The MoveIt! core, plugins, and planners are installed for ROS Noetic.
2. MoveIt! Architecture Understanding: Detailed overview of the `move_group` node, which acts as an integrator of various robot components and interacts with the user through actions and services.
3. Motion Planning with MoveIt!: Understanding the process of setting up motion planning, including setting up start and goal poses, and using the RViz MotionPlanning plugin for interactive motion planning and execution.
4. Controller Configuration: Creating controller configuration files for both MoveIt! and Gazebo to handle trajectory following and joint control.
5. Launch Files Creation: Setting up launch files for starting trajectory controllers and interfacing MoveIt! with Gazebo for simulation.

Implementing Navigation Stack:
1. Introduction to Navigation Stack: Overview of the components, algorithms, and hardware requirements for effective autonomous navigation.
2. Building a Map using SLAM: Configuration and launching of the `slam_gmapping` node to create a 2D occupancy grid map of the environment using laser scan data and robot odometry.
3. Autonomous Navigation: Implementing autonomous navigation using the `amcl` package for localization and the `move_base` node for goal-oriented movement.
4. Configuring and Running Navigation Nodes: Details on configuring and launching various navigation nodes, including setting parameters for local and global planning and recovery behaviors.
5. Debugging and Testing: Techniques for ensuring proper communication and functionality between MoveIt!, Gazebo, and the Navigation stack.

Conclusion
Chapter 6 provides a comprehensive guide on robot manipulation with MoveIt! and autonomous navigation using the ROS Navigation stack. The practical examples and configurations offer insights into the advanced capabilities of ROS for robot control, motion planning, and navigation in complex environments. With these tools, users can simulate sophisticated robot behaviors and plan movements in dynamic and unpredictable settings.

Chapter 7 - Exploring the Advanced Capabilities of ROS-MoveIt!

Chapter 7 of "Mastering ROS" dives deep into the advanced functionalities of ROS-MoveIt! and focuses on sophisticated robotic operations such as collision avoidance, 3D perception, grasping, and hardware interfacing. Here's a very detailed technical report based on the practice examples and the provided information.

Overview:
ROS-MoveIt! is an advanced motion planning framework that supports complex robot manipulation and navigation tasks. This chapter demonstrates how to leverage MoveIt!'s capabilities to perform intricate operations like custom path planning, collision detection, and object manipulation. It also discusses interfacing robotic manipulator hardware with MoveIt!, particularly focusing on DYNAMIXEL servos for high-performance applications.

Random Motion Planning:
1. Functionality: This example demonstrates the use of MoveIt! to plan and execute a random motion path for a robotic arm.
2. Implementation:
    - Use of `move_group_interface::MoveGroupInterface` for interacting with the move_group node.
    - The `setRandomTarget()` method randomly chooses a feasible target for the robot's end effector.
    - The `move()` method is called to execute the planned path.
3. Applications: Useful for testing the robot's workspace and understanding the range of motion without specific target constraints.

Custom Motion Planning:
1. Functionality: Illustrates how to move the robot's end effector to a custom, predefined goal position.
2. Implementation:
    - Inclusion of necessary MoveIt! header files for planning and visualization.
    - Configuration of the target pose with specific orientation and position values.
    - Execution of the motion plan towards the set target.
3. Applications: Essential for tasks where the robot needs to reach specific points in space, e.g., in assembly tasks or where the object's location is known in advance.

Collision Object Addition:
1. Functionality: Demonstrates adding a collision object to the MoveIt! planning scene and observing how motion planning adapts to avoid it.
2. Implementation:
    - Use of `PlanningSceneInterface` to access and modify the current planning scene.
    - Creation of a `CollisionObject` message to define the shape, size, and position of the new obstacle.

    - Addition of the collision object to the planning scene to influence subsequent motion planning.
3. Applications: Crucial for realistic motion planning where the robot needs to avoid obstacles or operate in cluttered environments.

Attaching/Detaching Objects:

1. Functionality: Shows how to attach and detach objects to the robot's body, extending obstacle avoidance to grasped objects.
2. Implementation:
    - Creation of `AttachedCollisionObject` instances specifying the target link and the object to attach.
    - Utilization of the `applyAttachedCollisionObject` method to perform attachment and detachment.
3. Applications: Vital for manipulation tasks where the robot needs to pick up, carry, and place objects while avoiding collisions.

DYNAMIXEL Servos Interface:

1. Functionality: Discusses interfacing DYNAMIXEL-based robotic arms with ROS MoveIt! for high-end robotic applications.
2. Implementation:
    - Introduction to DYNAMIXEL servos, known for their performance and networking capabilities.
    - Use of `dynamixel_motor` ROS stack to interface with the servos.
    - Configuration and launch files setup for controlling individual joints and handling complex multi-joint systems.
3. Applications: Used in high-performance robotics requiring precise position and torque control, such as in humanoid robots, robotic arms, and other complex multi-joint systems.

Advanced Collision Detection:

1. Functionality: Explores advanced collision detection techniques, including self-collision and environment collision checks.
2. Implementation:
    - Use of the MoveIt! planning scene and collision detection APIs.
    - Setting up collision request and response objects.
    - Performing self-collision checks with the robot's current state and full collision checks considering the environment.
3. Applications: Essential for ensuring safe robot operation by avoiding collisions with itself and the surrounding environment.

Object Manipulation:

1. Functionality: Details the process of performing pick-and-place operations using MoveIt!.
2. Implementation:

    - Creation of collision objects representing the objects to manipulate and the environment.
    - Planning and execution of robot motion to approach, grasp, and move objects to new locations.
    - Attaching and detaching objects to the robot's end effector during manipulation.
3. Applications: Fundamental for tasks where the robot needs to interact with and manipulate objects, such as in assembly lines, packaging, or material handling.

Perception Integration:
1. Functionality: Demonstrates integrating 3D vision sensors with MoveIt! for enhanced environmental perception.
2. Implementation:
   - Use of plugins like `PointCloudOctomapUpdater` and `DepthImageOctomapUpdater` for building 3D occupancy maps from sensor data.
   - Configuration of sensor topics, range, and other parameters for accurate

map building.
   - Launch file setup for initiating the sensor manager and octomap monitor.
3. Applications: Crucial for tasks requiring awareness of the environment, such as navigation in unknown spaces, obstacle avoidance, and complex manipulation.

Conclusion:
Chapter 7 provides a comprehensive exploration of the advanced functionalities of ROS-MoveIt!, showcasing its versatility in handling complex robotic tasks. From basic motion planning to sophisticated object manipulation and hardware interfacing, this chapter equips readers with the knowledge and skills to implement advanced robot behaviors and operations using ROS-MoveIt!. The detailed examples and technical insights pave the way for developing high-end robotic applications in research and industry.

Chapter 8 - ROS for Aerial Robots

Chapter 8 of "Mastering ROS" focuses on the simulation and control of aerial robots, particularly quadcopters, within the ROS environment. This chapter provides insights into the tools and techniques required to simulate, interface, and control aerial robots using ROS frameworks like PX4 and RotorS.

ROS and Aerial Robotics:
ROS (Robot Operating System) provides a flexible framework for writing software for robotic systems, including aerial robots. It offers libraries and tools to help software developers create robotic applications. Aerial robots, especially quadcopters, present unique challenges due to their dynamic environment and the need for stable flight control. ROS addresses these with its robust communication and sensor integration capabilities.

Simulation and Control of Aerial Robots:
1. PX4 and MAVROS:
    - PX4: An open-source flight control software for drones and other unmanned vehicles. It's widely used in the industry and academia for controlling quadcopters and other types of UAVs.
    - MAVROS: A ROS package that provides communication between ROS and PX4 via MAVLink, allowing for easy sending of commands and receiving of data.

2. Creating a ROS-PX4 Application:
    - Package Creation: A new ROS package, `px4_ros_ctrl`, is created to hold the source and launch files needed for UAV communication.
    - Key Operations:
        - Arming: The quadrotor is armed using the `/mavros/cmd/arming` service, allowing the motors to start spinning.
        - Mode Switching: Switching to OFFBOARD mode via `/mavros/set_mode` allows the acceptance of external commands.
        - Position Sending: The desired position is published on `/mavros/setpoint_position/local` to direct the UAV.
        - Landing: Utilizing `/mavros/cmd/land` to safely land the UAV.
    - Implementation: The ROS node in `px4_ctrl_example.cpp` performs these operations, ensuring a structured approach to UAV control.

3. RotorS Simulation Framework:
   - Purpose: RotorS is a MAV gazebo simulator that provides a set of ROS packages for simulating multirotor systems and their dynamics.
   - Installation: The process involves installing necessary dependencies and cloning the RotorS repository into the ROS workspace.
   - Launching Simulation: Using provided models and launch files, users can simulate various types of UAVs, including custom configurations.

- Controller and Node Operation: Nodes like `lee_position_controller_node` and `hovering_example` are used to generate propeller velocities and control the UAV, respectively.

Key Components and Considerations:
1. State Estimation: Accurate state estimation is crucial for aerial robots. Tools like EKF (Extended Kalman Filter) are commonly used for this purpose.
2. Control Strategies: Control of aerial robots is complex due to their dynamics. Controllers need to be robust and responsive to ensure stable flight.
3. Sensor Integration: Aerial robots often rely on a variety of sensors like IMUs, GPS, LiDAR, and cameras to understand and interact with their environment. ROS's flexibility in integrating various sensors is a significant advantage.

Challenges and Solutions:
1. Dynamic Environments: Aerial robots operate in three-dimensional, often unpredictable environments. Solutions include advanced motion planning algorithms and real-time obstacle avoidance techniques.
2. Communication: Maintaining reliable communication between the UAV and the control station is crucial, especially for tasks involving autonomy or remote operation.
3. Safety and Reliability: Ensuring safe operation, especially in failure scenarios, is a critical aspect. This involves creating fail-safe mechanisms and thoroughly testing the systems in simulation and real-world scenarios.

Conclusion:
Chapter 8 of "Mastering ROS" effectively guides through the complexities of aerial robotics within the ROS framework. It provides the foundational knowledge and practical examples needed to simulate, control, and interface with aerial robots, particularly focusing on quadcopters. The use of tools like PX4, MAVROS, and RotorS within the ROS ecosystem exemplifies the powerful capabilities available for developing sophisticated aerial robotic systems. Whether for hobbyists, researchers, or industrial applications, the knowledge gained from this chapter offers a solid foundation for advancing in the field of aerial robotics.