Welcome to the ROS Printing Station Documentation!

This documentation provides comprehensive information about the ROS Printing Station project, a system designed to automate printing tasks using a robotic manipulator integrated with the Robot Operating System (ROS).

Project Overview

The ROS Printing Station project aims to demonstrate and implement a robotic workcell capable of performing pick-and-place operations related to printing. Key aspects include:

- **Robot Control:** Utilizing ROS and MoveIt! for robot motion planning and execution.
- **Perception:** Incorporating camera feedback for task execution (e.g., identifying printing areas or paper).
- **Graphical User Interface (GUI):** Providing a user-friendly interface for monitoring status, sending commands, and viewing camera feeds.
- **Automated Workflows:** Implementing routines for automated printing tasks.

Whether you're a developer looking to contribute, a user wanting to understand how the system works, or just curious about ROS robotics, this documentation is your guide.

Getting Started

If you're new to the project, we recommend starting with the Installation Guide guide to set up the necessary environment and dependencies.

(Consider adding a diagram here if you have one!)

Table of Contents

Below is the table of contents, guiding you through the different sections of the documentation:

Documentation Contents:

Installation Guide

This document provides a high-level guide to installing and setting up the ROS Printing Station project. It assumes you have a working ROS Noetic environment installed on your system.

Prerequisites

Before proceeding with the project-specific installation, ensure you have the following prerequisites met:

- **ROS Noetic Installation:** The project requires ROS Noetic. If you do not have it installed, please follow the official ROS Wiki installation guide for your operating system:

  ROS Noetic Installation Guide

- **Catkin Workspace:** It is recommended to build this project within a Catkin workspace. If you don't have one, you can create one following the ROS tutorials.

Cloning the Repository

Navigate to the *src* directory of your Catkin workspace and clone the project repository:

```
git clone https://github.com/DavidSeyserGit/ROS_Printing_Station.git
```

Installing Dependencies

Once the repository is cloned, you need to install the project's dependencies.

1. **Update and Install ROS Dependencies with `rosdep`:**

   Navigate back to the root of your Catkin workspace and use *rosdep* to install any declared ROS package dependencies:

   ```
   cd ROS_Printing_Station/ # Replace with your actual workspace root
   sudo apt-get update  # Ensure your package list is up to date
   rosdep install --from-paths src --ignore-src -r -y
   ```

   This command will install necessary ROS packages based on the *package.xml* files in your source directories.

2. **Install Specific ROS Packages (if not covered by rosdep):**

   While *rosdep* should handle most ROS dependencies, explicitly installing common ones like controllers and MoveIt! components is good practice:

   ```
   sudo apt-get install ros-noetic-joint-trajectory-controller
   sudo apt-get install ros-noetic-robot-state-publisher
   sudo apt-get install ros-noetic-moveit
   ```

3. **Install Python Dependencies:**

   The GUI and other Python scripts have specific Python package requirements listed in *requirements.txt*. Install these using pip:

   ```
   cd ROS_Printing_Station # Navigate to the project directory
   pip install -r requirements.txt
   ```

Building the Workspace

After installing dependencies, build your Catkin workspace:

```
catkin_make -j16
```

Sourcing the Setup File

Source the workspace's setup file to make the installed packages and executables available in your current terminal session:

```
source devel/setup.bash
```

Running the Simulation

To launch the full simulation environment and GUI, use the provided start script:

```
cd ROS_Printing_Station # Navigate to the project directory
./scripts/start.sh
```

This script should launch all the necessary ROS nodes and the GUI.

Further Information

- **Known Issues:** For information on known issues, please refer to the project's GitHub repository:

  Open Issues on GitHub

- **Repository Link:** The project's source code is available on GitHub:

  ROS_Printing_Station Repository

GUI

This section provides documentation for the Graphical User Interface (GUI) module of the ROS Printing Station, which allows for interaction with the ROS nodes and robot control.

The GUI is implemented in a single Python file (e.g., *gui.py*) and utilizes the Tkinter library for the user interface and the *rospy* library for interacting with the ROS system.

Key Features:

- Visual feedback of robot status (via color indicator).
- Display of two camera feeds.
- Button to initiate the robot's automated routine.
- Entry fields and button to send joint commands (X, Y, Z values).
- Ability to toggle between the two camera views.
- Display of error messages received from the ROS system.

ROSGUI Class

The main component of the GUI is the `ROSGUI` class.

```
class ROSGUI:
    def __init__(self, master):
        # ... initialization code ...

    def create_widgets(self):
        # ... widget creation code ...

    # ... other methods ...
```

The `ROSGUI` class is responsible for setting up the Tkinter window, initializing ROS nodes and subscribers/publishers, and handling user interactions and ROS message callbacks.

Initialization (*__init__*)

The *__init__* method sets up the main window, initializes the ROS node if necessary, creates ROS subscribers and publishers, and initializes variables for image handling and camera view tracking.

- **Parameters:** * `master` (`tkinter.Tk`): The root Tkinter window.
- **Key Actions:** * Sets the window title and geometry. * Configures the window's background color. * Initializes the ROS node named 'gui'. * Subscribes to `/response` (String), `/static_camera/image_raw` (Image), and `/camera2/image_raw` (Image). * Creates a publisher for the `/chatter` topic (String). * Initializes `cv_bridge` for image conversion. * Initializes variables for storing camera images and the active camera view. * Calls `create_widgets` to build the GUI elements.

Creating Widgets (*create_widgets*)

The `create_widgets` method constructs all the visual elements of the GUI using Tkinter.

- **Layout:** Uses a grid layout manager for the main frames and packed/gridded elements within frames.
- **Sections:** Organizes widgets into sections like Status, Controls, and Camera Views.
- **Widgets Created:** * Frames (`left_frame`, `image_frame`, `status_section`, `entry_frame`, `camera_controls`). * Labels (for status, error messages, coordinate entries, camera views). * A Canvas for the ROS status indicator. * Buttons (Start Robot Operation, Send Joint Values, Switch Camera View). * Entry fields (for X, Y, and Z coordinates). * Labels to display camera feeds (`image_label1`, `image_label2`).

Toggle Camera View (*toggle_camera_view*)

Switches the currently displayed camera feed between the first and second camera views.

- **Action:** Hides the currently active camera label and displays the other one. Updates the `active_camera` attribute and changes the appearance of the camera labels to indicate the active view.

ROS Message Callback (*callback*)

Callback function for the `/response` topic.

- **Parameters:** * `msg` (`std_msgs.msg.String`): The received String message.
- **Action:** Updates the color of the ROS status indicator circle based on the received message data.

Publish on Topic (*publish_on_topic*)

Publishes the values from the X, Y, and Z entry fields onto the `/chatter` topic.

- **Action:** Reads the text from the X, Y, and Z entry widgets, formats it into a single string, publishes it, and then clears the entry fields.

Error Callback (*error_callback*)

Callback function for receiving error messages. *(Note: Your code defines this method but does not subscribe to an error topic.)*

- **Parameters:** * `msg` (`std_msgs.msg.String`): The received error message.
- **Action:** Updates the text and color of the error label to display the received error message.

Image Callback 1 (*image_callback1*)

Callback function for the `/static_camera/image_raw` topic.

- **Parameters:** * `msg` (`sensor_msgs.msg.Image`): The received Image message.
- **Action:** Converts the ROS Image message to a format usable by Tkinter (using `cv_bridge` and PIL), and updates `image_label1` to display the image. Includes error handling for the conversion process.

Image Callback 2 (*image_callback2*)

Callback function for the `/camera2/image_raw` topic.

- **Parameters:** * `msg` (`sensor_msgs.msg.Image`): The received Image message.
- **Action:** Converts the ROS Image message to a format usable by Tkinter (using `cv_bridge` and PIL), and updates `image_label2` to display the image. Includes error handling for the conversion process.

Start Robot Operation (*start_robot*)

Initiates the main robot routine on a separate thread.

- **Action:** Creates and starts a new Python thread that executes the `run_robot_routine` method. This prevents the robot operation from freezing the GUI.

Run Robot Routine (*run_robot_routine*)

Executes the external robot pick-and-place routine.

- **Action:** Calls the `robot_main()` function imported from the `pick_and_place` module. Includes basic error logging for exceptions that occur during the routine execution.

GUI Mover

This module (*GuiMover.py*) acts as a ROS node responsible for receiving joint commands from the GUI and controlling the SCARA robot using MoveIt!. It subscribes to the `/chatter` topic to get target joint values and uses the MoveIt! commander to plan and execute robot movements.

Code Structure

```
#!/usr/bin/env python
import sys
import rospy
import moveit_commander
from std_msgs.msg import String

status_pub = rospy.Publisher("response", String, queue_size=10)
error_pub = rospy.Publisher("error", String, queue_size=10)

def chatter_callback(message):
    # ... callback logic ...
    pass

if __name__ == '__main__':
    # ... initialization and main loop ...
    pass
```

Publishers

The module initializes two ROS publishers:

- `status_pub` (`rospy.Publisher`, topic: `/response`, message type: `std_msgs.msg.String`): Publishes status updates, typically the color of the status indicator, back to the GUI.
- `error_pub` (`rospy.Publisher`, topic: `/error`, message type: `std_msgs.msg.String`): Publishes error messages encountered during message processing or motion execution back to the GUI. *(Note: The provided GUI code has an `error_callback` but doesn't explicitly subscribe to an `/error` topic; this suggests a potential area for synchronization between the GUI and this module if you intend to display these errors).*

Callback Function (*chatter_callback*)

This function is the callback for messages received on the `/chatter` topic. It is triggered when the GUI sends joint commands.

- **Parameters:** * `message` (`std_msgs.msg.String`): The received String message containing the target joint values from the GUI. The message is expected to be in the format "q1: <value> q2: <value> q3: <value>".
- **Functionality:** * Parses the incoming string message to extract the target values for joints q1, q2, and q3. * Logs the received target joint values. * Gets the current joint values of the robot using MoveIt!. * Updates the target values for joints 0, 1, and 2 (corresponding to q1, q2, and q3) in the joint goal. * Attempts to move the robot to the target joint configuration using `move_group.go()`. * If the motion execution fails, it publishes "red" to the `/response` topic and the error message to the `/error` topic. * Stops any residual movement after the command using `move_group.stop()`. * Logs a success message if the motion completes without raising an exception. * Includes general error handling to catch exceptions during message processing.

Main Execution Block

This block is executed when the script is run directly.

- **Initialization:** * Initializes the MoveIt! commander with command-line arguments. * Initializes the ROS node named ‘moveit_scara_controller’. * Creates a `RobotCommander` and `PlanningSceneInterface` instance from MoveIt!. * Initializes a `MoveGroupCommander` for the “scara” planning group. * Logs a message indicating the node has been initialized and is waiting for commands.
- **Subscription:** * Subscribes to the `/chatter` topic with the `chatter_callback` function as the handler.
- **ROS Spin:** * Enters the ROS spin loop (`rospy.spin()`), which keeps the node alive and allows it to receive messages on subscribed topics. The script will remain active until it is shut down (e.g., by pressing Ctrl+C or a ROS shutdown command).
- **Shutdown:** * Performs necessary MoveIt! commander shutdown before the script exits.

Pick and Place Module

This module (*pick_and_place.py*) orchestrates the automated pick-and-place operation for the ROS Printing Station. It involves spawning a placeholder object in Gazebo, moving the “knick” arm to grasp it (simulated by reaching a pre-defined pose), and then moving a SCARA arm along waypoints for the printing task, and finally despawning the object.

Code Structure

```
#!/usr/bin/env python
import rospy
import moveit_commander
# ... other imports ...
from RobotMover import RobotMove, run_robot # <-- Imported from RobotMover
import random
from std_msgs.msg import String

pub = rospy.Publisher("response", String, queue_size=10)

# ... pre-defined poses and joint targets ...

def spawn_object(model_path, model_name, initial_pose):
    # ... spawn object logic ...
    pass

def main():
    # ... main pick and place routine ...
    pass

if __name__ == "__main__":
    # ... initialization and execution ...
    pass
```

Publishers

The module utilizes one ROS publisher:

- `pub` (`rospy.Publisher`, topic: `/response`, message type: `std_msgs.msg.String`): Publishes status updates (like “yellow”, “red”, “green”) back to the GUI to indicate the state of the pick-and-place process.

Pre-defined Poses and Joint Targets

The script defines several pre-configured poses and corresponding joint targets used during the pick-and-place sequence.

- `traget_pose_1` to `traget_pose_5`: Cartesian poses (X, Y, Z, Quaternion) representing potential pick-up locations for the object.
- `drop_off_pose`: A Cartesian pose representing the location where the object is notionally placed.
- `joint_target_1` to `joint_target_5`: Joint space configurations (for the “knick” arm) corresponding to reaching the respective `traget_pose_X` locations.
- `joint_target_drop_off`: A joint space configuration for the “knick” arm at the drop-off position.

Spawning Object Function (*spawn_object*)

This function handles the spawning of a specified SDF model into the Gazebo simulation environment.

- **Parameters:** * `model_path` (`str`): The file path to the SDF model file. * `model_name` (`str`): The de-

sired name for the spawned model in Gazebo. * `initial_pose` (`geometry_msgs.msg.Pose`): The initial pose (position and orientation) where the model should be spawned.
- **Functionality:** * Waits for the `/gazebo/spawn_sdf_model` service to be available. * Reads the SDF model XML from the specified file. * Calls the spawn service with the provided model name, XML, and initial pose. * Logs success or failure of the spawning operation.

Main Execution (*main*)

This is the primary function that orchestrates the automated pick-and-place process. It is typically called by the GUI's start routine.

- **Process Flow:** 1. Publishes "yellow" to the `/response` topic to indicate the start of the process. 2. Randomly selects one of the five target pick-up locations (`traget_pose_1` to `traget_pose_5`). 3. Determines the corresponding initial pose for the object in Gazebo (slightly below the target pick-up pose). 4. Attempts to spawn the specified object model at the determined initial pose using the `spawn_object` function. Logs a warning if spawning fails. 5. Initializes the MoveIt! commander (if not already initialized by the ROS node). 6. Creates a `MoveGroupCommander` for the "knick" arm and configures planning parameters. 7. Sets the joint target for the "knick" arm to the selected pick-up joint configuration (corresponding to the randomly chosen target pose). 8. Plans and attempts to execute the motion for the "knick" arm to reach the pick-up position. Publishes "red" if planning fails. 9. Sets the joint target for the "knick" arm to the drop-off joint configuration. 10. Plans and attempts to execute the motion for the "knick" arm to reach the drop-off position. Publishes "red" if planning fails. 11. Selects the appropriate SCARA waypoints file (*scara_wp.json* or *scara_wp2.json*) based on whether the randomly chosen target index was even or odd. 12. **Initializes a robot control object:**

    Uses the **`RobotMove`** class from the RobotMover to create an instance for controlling the "scara" arm with the selected waypoints file.

    ```
    scara_robot = RobotMove(
        robot_name="scara",
        move_group_name="scara",
        waypoints_file=scara_waypoints_file,
    )
    ```

13. **Runs the SCARA routine:** Calls the **`run_robot()`** helper function (also from the RobotMover) to execute the waypoints defined in the selected file for the SCARA arm.

    ```
    run_robot(scara_robot)
    ```

14. Attempts to delete the spawned object model from Gazebo using the `/gazebo/delete_model` service. Logs errors if despawning fails.

15. Publishes "green" to the `/response` topic to indicate successful completion of the entire routine.

16. Includes general error handling for issues during MoveIt! planning or execution.

Main Execution Block

This block is executed when the script is run directly (though it's typically called as a function from the GUI).

- Initializes the `pub` publisher (though it's already initialized at the module level, which is slightly redundant).
- Calls the `main()` function within a try-except block to catch potential ROS interrupt exceptions and other unexpected errors.
- Ensures `moveit_commander.roscpp_shutdown()` is called in a `finally` block to clean up MoveIt! resources when the script exits.

Robot Mover Module

This module (*RobotMover.py*) provides the `RobotMove` class and a helper function `run_robot` for controlling a robot arm using MoveIt! and executing pre-defined sequences of joint waypoints loaded from a JSON file.

Code Structure

```
#!/usr/bin/env python
import os
import sys
import json
import time
import threading # Note: Not directly used in the provided code snippet
import rospy
import moveit_commander

class RobotMove:
    def __init__(self, robot_name, move_group_name, waypoints_file):
        # ... initialization ...
        pass

    def load_waypoints(self, file_path):
        # ... waypoint loading logic ...
        pass

    def move_to_position(self, joint_positions):
        # ... movement logic ...
        pass

    def print_current_pose(self):
        # ... print pose logic ...
        pass

    def execute_waypoints(self):
        # ... waypoint execution logic ...
        pass

def run_robot(robot):
    # ... helper function ...
    pass
```

RobotMove Class

The `RobotMove` class encapsulates the functionality for controlling a specific robot arm defined within MoveIt!.

```
class RobotMove:
    def __init__(self, robot_name, move_group_name, waypoints_file):
        # ... initialization code ...
        pass
    # ... other methods ...
```

Initialization (*__init__*)

Initializes the `RobotMove` object, setting up the connection to MoveIt! and loading the waypoints.

- **Parameters:** * `robot_name` (`str`): A name for the robot instance (e.g., used for ROS node name). * `move_group_name` (`str`): The name of the MoveIt! planning group for this robot (e.g., “scara”, “knick”). * `waypoints_file` (`str`): The file path to the JSON file containing the waypoint definitions.

- **Key Actions:** * Initializes a unique ROS node if one is not already initialized. * Initializes the MoveIt! commander. * Creates a `MoveGroupCommander` for the specified planning group. * Sets velocity and acceleration scaling factors for planning and execution. * Loads the waypoints from the specified JSON file using the `load_waypoints` method.

Load Waypoints (*load_waypoints*)

Loads the waypoint data from a specified JSON file.

- **Parameters:** * `file_path` (`str`): The path to the JSON waypoints file.
- **Returns:** * `list`: A list of dictionaries, where each dictionary represents a waypoint. Returns an empty list if the file is not found or has an invalid format.
- **Expected Waypoint JSON Structure:** Each item in the list should be a dictionary with: * `"description"` (`str`): A description of the waypoint action. * `"position"` (`list[float]`): A list of joint values for the target position. The length of this list must match the number of joints in the MoveIt! planning group. * `"pause"` (`float`, optional): The duration to pause in seconds after reaching this waypoint. * `"action"` (`str`, optional): An optional message to log after reaching the waypoint.
- **Error Handling:** Includes error handling for *FileNotFoundError*, *json.JSONDecodeError*, and other potential exceptions during file reading.

Move to Position (*move_to_position*)

Commands the robot's joints to move to a specified set of joint positions.

- **Parameters:** * `joint_positions` (`list[float]`): A list of target joint values.
- **Returns:** * `bool`: `True` if the movement successfully completes, `False` otherwise.
- **Functionality:** * Checks if the length of the provided joint positions matches the number of joints in the move group. Logs an error and returns `False` if they don't match. * Sets the target joint values for the MoveIt! planning group. * Executes the motion plan using `move_group.go()` with `wait=True`. * Calls `move_group.stop()` after the movement attempt.

Print Current Pose (*print_current_pose*)

Logs the current Cartesian pose (position and orientation) of the robot's end effector (TCP - Tool Center Point).

- **Action:** Retrieves the current pose using `move_group.get_current_pose()` and logs its position.

Execute Waypoints (*execute_waypoints*)

Iterates through the loaded waypoints and commands the robot to move to each target joint position sequentially, including pauses and action logging as defined in the waypoint data.

- **Action:** * Checks if waypoints were successfully loaded. * Loops through each waypoint in the loaded list. * Extracts the description, joint positions, optional pause time, and optional action message. * Logs the waypoint description. * Calls `move_to_position` to move the robot to the waypoint's joint positions. * Logs success or failure for each movement. * Calls `print_current_pose` after a successful move. * Logs the action message if provided. * Pauses execution for the specified duration if a pause time is defined. * Includes error handling for missing keys in waypoint data or other exceptions during waypoint processing. * Logs a completion message after all waypoints are executed.

Helper Function (*run_robot*)

A simple helper function to execute the waypoints for a given `RobotMove` object.

```
    def run_robot(robot):
        """Helper function to run a robot's waypoint execution."""
        robot.execute_waypoints()
```

- **Parameters:** * `robot` (`RobotMove`): An instance of the `RobotMove` class.
- **Action:** Calls the `execute_waypoints` method of the provided `RobotMove` object.