Proposed Architecture for Automated Robot Dynamics Dataset Generation

1. Introduction

This document outlines a revised architecture for the automated generation of robot dynamics datasets. The primary goal is to streamline the data collection process by maintaining a single, persistent Gazebo simulation instance while dynamically spawning and despawning robots. This approach significantly reduces overhead associated with repeatedly launching Gazebo, allowing for more efficient and scalable data generation, especially crucial for large datasets involving thousands of robots. The system is designed to be highly configurable via a centralized `settings.yaml` file, ensuring flexibility and ease of use.

2. Overall Architecture Flow

The data generation process can be conceptualized as a multi-stage pipeline, orchestrated by a top-level automation script. This script manages the lifecycle of individual robot simulations within a persistent Gazebo environment. The flow begins with the initial preparation of robot URDFs, followed by a sequential simulation and data collection phase for each robot, and concludes with an optional data aggregation step.

1. **URDF Preparation (Pre-simulation):** Before any simulation begins, a dedicated script (`gazebo_adapter.py`) modifies the original robot URDFs. This modification involves injecting randomized dynamic parameters (such as Coulomb and viscous friction coefficients) into the URDFs, based on ranges defined in `settings.yaml`. Concurrently, another script (`data_generator_node.py`) extracts these newly assigned dynamic parameters from the modified URDFs and saves them into individual YAML files, one for each robot. This ensures that the ground truth dynamic parameters for each simulated robot are recorded.

2. **Persistent Gazebo Environment:** A single Gazebo instance is launched and remains active throughout the entire data collection process. This eliminates the need to repeatedly start and stop the simulation environment, which is a time-consuming operation.

3. **Automated Simulation Loop (Per-Robot):** A higher-level Python script (`automate_data_collection.py`) acts as the orchestrator for the simulation loop. For each robot designated for the experiment, this script performs the following sequence:

    - **Gazebo Reset:** It sends a command to Gazebo to reset the world, clearing any previous robot models and simulation states.
    - **Robot Spawning:** It dynamically spawns the currently selected robot (with its modified URDF) into the Gazebo environment.
    - **Controller Activation & Data Collection:** It launches a ROS 2 node (`combined_controller_node.py`) responsible for generating target trajectories, controlling the robot using a PID controller, and collecting comprehensive trajectory data (joint positions, velocities, accelerations, torques, and link states). This node also handles internal logic for trajectory completion and timeout management.
    - **Robot Despawning:** Once the `combined_controller_node.py` signals completion (or a timeout occurs), the orchestrator script sends a command to Gazebo to despawn the current robot, preparing the environment for the next robot.

4. **Data Storage:** As each robot's simulation concludes, its trajectory data is saved into a dedicated CSV file. The dynamic parameters for each robot are stored in separate YAML files. These files are organized within a structured `dataset/` directory.

5. **Data Aggregation (Post-simulation):** After all robots have been simulated and their data collected, an optional script (`aggregate_csv.py`) can be run to combine all individual robot trajectory CSV files into a single, comprehensive CSV file. This facilitates bulk analysis and model training.

This architecture ensures a robust, efficient, and scalable solution for generating large-scale robot dynamics datasets, with clear separation of concerns between URDF preparation, simulation management, and data collection/storage.

3. Component Breakdown

3.1. `settings.yaml` (Configuration File)

This YAML file serves as the central configuration hub for the entire data generation pipeline. All relevant scripts and nodes will read parameters from this file to ensure consistent behavior and easy modification of experimental settings. Key parameters include:

- `num_trajectory_points_per_robot` : Defines the number of distinct target configurations the robot will attempt to reach during its simulation, influencing the length and complexity of the generated trajectory data.
- `coulomb_friction_coeff_range` : A two-element list specifying the minimum and maximum values for randomly assigning Coulomb friction coefficients to the robot's joints. This allows for controlled variation in the dynamic properties of each generated robot.
- `viscous_friction_coeff_range` : Similar to the Coulomb friction range, this defines the bounds for randomly assigning viscous friction coefficients, contributing to the diversity of the dataset.
- `timeout_trajectory_point` : A crucial parameter that sets a time limit (in milliseconds) for the robot to reach a specific trajectory point. If this timeout is exceeded, the current trajectory attempt is considered failed, the Gazebo simulation is reset, and the system proceeds to the next trajectory point or robot. This prevents simulations from getting stuck indefinitely.
- `num_robots_in_experiment` : Specifies the total number of robots from the `robots/` directory that will be processed in the experiment. This allows for running experiments on a subset of available robot models or scaling up to a large number of robots.
- `dataset_output_dir` : The base directory where all generated data (dynamic parameters YAMLs, trajectory CSVs) will be stored. This ensures a centralized and organized output structure.

3.2. `gazebo_adapter.py` (URDF Pre-processor)

This Python script is executed once at the very beginning of the data generation process, before any ROS 2 nodes or Gazebo simulations are initiated. Its primary function is to prepare the robot URDF files by injecting randomized dynamic parameters. The script performs the following key operations:

1. **Reads Settings:** It reads the `coulomb_friction_coeff_range` and `viscous_friction_coeff_range` from `settings.yaml`.
2. **Iterates Robot Folders:** It scans the `robots/` directory, identifying each robot's subfolder. For each robot, it locates the `robot.urdf` file.
3. **Modifies URDF:** For each joint in the `robot.urdf`, it randomly generates Coulomb and viscous friction coefficients within the specified ranges. These values are then programmatically inserted or updated within the `<dynamics>` tag of the corresponding joint in the URDF. It also ensures that the URDF is compatible with Gazebo and ROS 2 control by adding necessary `<transmission>` and `<ros2_control>` tags, and converting DAE meshes to STL if necessary.
4. **Saves Modified URDF:** The modified URDF is saved as `robotGA.urdf` (Gazebo Adapted URDF) within the respective robot's folder. This `robotGA.urdf` will be used for simulation.

This script ensures that each robot simulated has unique, randomly assigned dynamic properties, which is essential for creating a diverse dataset for dynamic parameter estimation.

3.3. `data_generator_node.py` (Dynamic Parameter Extractor)

Similar to `gazebo_adapter.py`, this script is also executed once at the beginning of the process, immediately after `gazebo_adapter.py`. Its role is to extract the ground truth dynamic parameters that were just injected into the `robotGA.urdf` files. The steps are:

1. **Reads Modified URDFs:** It iterates through the `robotGA.urdf` files generated by `gazebo_adapter.py`.
2. **Extracts Parameters:** For each `robotGA.urdf`, it parses the XML to extract:

- **Joint Parameters:** Coulomb friction coefficient and viscous friction coefficient for each joint.
- **Link Parameters:** Mass, inertia tensor components (ixx, ixy, ixz, iyy, iyz, izz), and center of mass (COM) for each link.

3. **Saves to YAML:** The extracted parameters for each robot are then saved into a separate YAML file (e.g., `robot_X_dynamic_parameters.yaml`) within the `dataset/` directory. This YAML file serves as the ground truth for the dynamic parameters of that specific robot, which can be used later for model training.

3.4. `bringup.launch.py` (Persistent Gazebo Launcher)

This ROS 2 launch file is responsible for initiating the core simulation environment. It is designed to launch Gazebo only once and keep it running throughout the entire data collection process. Key features:

1. **Launches Gazebo:** It starts a Gazebo simulation instance with a predefined empty world.
2. **Initial Robot State Publisher:** It includes the `robot_state_publisher` node, which is necessary for publishing the robot's state based on its URDF.
3. **Initial Controller Manager:** It launches the `controller_manager` node, which is essential for managing the robot's controllers (e.g., joint state broadcaster, torque controller).
4. **ROS-Gazebo Bridge:** It sets up the `ros_gz_bridge` for communication between ROS 2 and Gazebo, specifically for services like spawning and deleting entities, and world control (e.g., resetting the simulation).
5. **No Initial Robot Spawn:** Crucially, this launch file *does not* initially spawn any robot. Robot spawning will be handled dynamically by the top-level automation script.

3.5. `combined_controller_node.py` (Trajectory Generation, PID Control, and Data Recorder)

This is the central ROS 2 node that performs the actual simulation and data collection for a single robot. It combines the functionalities previously found in `pid.py` and

`target_gen.py`, and adds robust data logging. This node is launched by the `automate_data_collection.py` script for each robot.

1. **Reads Settings:** It reads `num_trajectory_points_per_robot` and `timeout_trajectory_point` from `settings.yaml`.
2. **URDF Retrieval:** It retrieves the robot's URDF from the `robot_description` parameter provided by the `controller_manager`.
3. **PyBullet for Target Generation:** It uses PyBullet (in DIRECT mode) to generate a sequence of valid target joint configurations. This involves checking for collisions and ensuring the robot's end-effector stays within a reasonable workspace. The number of target points is determined by `num_trajectory_points_per_robot`.
4. **PID Control Loop:** It subscribes to the `/joint_states` topic to get real-time joint positions and velocities from Gazebo. It then calculates the necessary torque commands using a PID controller to drive the robot towards the current target joint configuration.
5. **Trajectory Interpolation:** It interpolates between the current joint state and the next target point, breaking down the movement into smaller steps to ensure smooth trajectories.
6. **Data Recording:** For each time step of the simulation, it records comprehensive data into internal buffers:
    - `time`: Simulation time.
    - `index`: The index of the current target trajectory point.
    - `Joint_Position`: Position of each joint.
    - `Joint_Velocity`: Velocity of each joint.
    - `Joint_Acceleration`: Acceleration of each joint (currently a placeholder, would require numerical differentiation).
    - `Joint_Torque`: Commanded torque for each joint.
    - `link_Position`: Position (x, y, z) of each link.
    - `link_Velocity`: Linear velocity (x, y, z) of each link.
    - `link_Acceleration`: Linear acceleration (x, y, z) of each link (currently a placeholder).

7. **Timeout Management:** It monitors the time taken to reach each trajectory point. If `timeout_trajectory_point` is exceeded, it signals the `automate_data_collection.py` script (e.g., via a ROS 2 topic or service) that the current trajectory failed, prompting a Gazebo reset and progression to the next robot or trajectory point.
8. **CSV Saving:** Once all trajectory points for the current robot are completed (or a timeout occurs), it saves the collected data into a CSV file (e.g., `robot_X_trajectory_data.csv`) within the `dataset/` directory. It also creates a subfolder for each robot within the `dataset/` directory to store its specific CSV and dynamic parameters YAML.

3.6. `automate_data_collection.py` (Top-Level Orchestrator)

This is the main Python script that drives the entire data generation process. It is executed from the command line and manages the sequential simulation of multiple robots within the persistent Gazebo environment. Its responsibilities include:

1. **Reads Settings:** It reads `num_robots_in_experiment` from `settings.yaml`.
2. **Iterates Robots:** It identifies the `robotGA.urdf` files for each robot that needs to be simulated.
3. **Dynamic Robot Management:** For each robot:
    - **Despawn Previous Robot:** It uses `ros2 service call /world/default/remove` to delete the previously simulated robot from Gazebo. It will need to dynamically get the model ID using `ign model -m robot_name" -p`. * **Spawn Current Robot:** It then uses `ros2 service call /world/default/create` to spawn the new robot into Gazebo, referencing its `robotGA.urdf`. * **Launch `combined_controller_node.py`:** It launches the `combined_controller_node.py` as a subprocess, passing the necessary parameters (e.g., the robot's URDF path, though the node now retrieves it from the controller manager). * **Monitor and Reset:** It monitors the `combined_controller_node.py` for completion signals or timeouts. If a timeout occurs, it will send a `gz service -s /world/default/control --reqtype gz.msgs.WorldControl --reptype gz.msgs.Boolean --timeout 3000 --req 'reset: {all: true}'` command to reset the Gazebo world before proceeding to the next robot.

This script is crucial for automating the entire data collection process, allowing for unattended generation of large datasets.

3.7. `aggregate_csv.py` (Data Aggregator)

This utility script is designed to be run after all individual robot simulations are complete. Its purpose is to consolidate the trajectory data from all robots into a single, unified CSV file. This simplifies the subsequent data analysis and machine learning model training steps.

1. **Scans Dataset Directory:** It iterates through the `dataset/` directory, identifying all individual trajectory CSV files generated by `combined_controller_node.py`.
2. **Concatenates Data:** It reads each CSV file into a pandas DataFrame and then concatenates all DataFrames into a single, large DataFrame.
3. **Saves Aggregated CSV:** The combined data is then saved into a new CSV file (e.g., `aggregated_trajectory_data.csv`) within the `dataset/` directory.

4. Data Flow and Interactions

- **Initial Setup:** `settings.yaml` provides configuration to `gazebo_adapter.py` and `data_generator_node.py`.
- **URDF Transformation:** `gazebo_adapter.py` takes `Original URDFs` and `settings.yaml` to produce `Modified URDFs (robotGA.urdf)`.
- **Parameter Extraction:** `data_generator_node.py` reads `Modified URDFs` and outputs `Dynamic Parameters (YAML)` files.
- **Simulation Orchestration:** `automate_data_collection.py` reads `settings.yaml` and iterates through `Modified URDFs`.
- **Gazebo Interaction:** `automate_data_collection.py` interacts with `bringup.launch.py` to manage the persistent `Gazebo Simulation`. It sends commands to spawn/despawn robots and reset the world.
- **Real-time Simulation & Control:** `combined_controller_node.py` runs within the `Gazebo Simulation`. It receives `Joint States` and `Link States` from Gazebo, generates `Torque Commands`, and sends `Reset Signals` back to Gazebo (via `automate_data_collection.py` or directly if implemented).

- **Trajectory Data Collection:** `combined_controller_node.py` records `Trajectory Data (CSV per robot)`.
- **Data Aggregation:** `aggregate_csv.py` takes all `Trajectory Data (CSV per robot)` files and combines them into `Aggregated Trajectory Data (CSV)`.

5. Conclusion

This proposed architecture provides a robust, efficient, and scalable framework for generating comprehensive robot dynamics datasets. By centralizing configuration, automating URDF modification, maintaining a persistent Gazebo instance, and orchestrating dynamic robot simulations, the system minimizes manual intervention and maximizes throughput. The clear separation of concerns among different scripts and nodes ensures maintainability and extensibility. The generated dataset, comprising both dynamic parameters and corresponding trajectory data, will be invaluable for training machine learning models to estimate robot dynamic properties from observed motion.