Complete Guide: 2DOF Robot ROS 2 Simulation

Table of Contents

---

1. ROS 2 Fundamentals

What is ROS 2?

ROS 2 (Robot Operating System 2) is a middleware framework for robot software development. It's not an operating system, but a collection of tools, libraries, and conventions that simplify building complex robot behaviors.

Core Concepts

Nodes

- **Definition**: Independent processes that perform specific tasks
- **Your code**: `PendulumSimulator` is a node that simulates robot dynamics
- **Communication**: Nodes communicate via topics, services, or actions
- **Lifecycle**: Created → Running → Destroyed

Topics

- **Definition**: Named buses for message passing (publish/subscribe pattern)
- **Your code uses**: `/joint_states` topic
- **Decoupled**: Publishers don't know who subscribes, subscribers don't know who publishes
- **Message types**: Standardized (e.g., `sensor_msgs/JointState`)

Messages

- **Definition**: Data structures sent over topics
- **Your code**: `JointState` message contains:

```
Header header       # Timestamp and frame info
string[] name       # Joint names
float64[] position  # Joint angles (radians)
float64[] velocity  # Joint velocities (rad/s)
float64[] effort    # Joint torques (N·m) - empty in your code
```

Publishers

- **Definition**: Send messages to topics
- **Your code**: `self.joint_pub` publishes robot state every 0.01s
- **Queue size**: Buffer for messages (10 in your code)

Timers

- **Definition**: Trigger callbacks at regular intervals
- **Your code**: `self.create_timer(self.dt, self.step_and_publish)` runs at 100 Hz

---

2. Package Structure

Your ROS 2 package `urdf_2dof` follows this structure:

```
urdf_2dof/
├── CMakeLists.txt          # Build instructions
├── package.xml             # Package metadata and dependencies
├── launch/
│   └── display.launch.py   # Launch file to start system
├── config/
│   └── config.rviz         # RViz visualization settings
├── urdf/
│   └── 2dof.urdf           # Robot description (links, joints, geometry)
├── meshes/                 # 3D models for visualization
└── scripts/
    └── pendulum_sim_node.py # Your Python node
```

Purpose of Each Directory

- **launch/**: Scripts to start multiple nodes with configuration
- **config/**: Configuration files (RViz settings, parameters)
- **urdf/**: Robot model definition (Unified Robot Description Format)
- **meshes/**: Visual/collision geometry (STL, DAE files)
- **scripts/**: Executable Python/C++ programs

---

3. Python Node Deep Dive

File: `pendulum_sim_node.py`

A. Imports

```python
import rclpy                          # ROS 2 Python client library
from rclpy.node import Node           # Base class for nodes
from sensor_msgs.msg import JointState # Message type for joint data
from std_msgs.msg import Header       # Message header (timestamp, frame_id)
import numpy as np                    # Numerical computation
import pinocchio as pin               # Rigid body dynamics library
from pinocchio.robot_wrapper import RobotWrapper
from scipy.integrate import solve_ivp  # Ordinary Differential Equation solver
```

Why these libraries?

- `rclpy`: Core ROS 2 functionality (nodes, publishers, timers)
- `pinocchio`: Fast dynamics computation (forward kinematics, Jacobians, RNEA)
- `scipy`: Numerical integration of robot motion equations
- `numpy`: Matrix operations

B. Class Structure

```python
class PendulumSimulator(Node):
    def __init__(self):
        super().__init__('pendulum_simulator')  # Register node with name
```

Inheritance: Your class inherits from `Node`, gaining ROS 2 capabilities.

Node name: `'pendulum_simulator'` is how ROS 2 identifies this node. You'll see it in `ros2 node list`.

C. Parameters (Dependency Injection)

```python
self.declare_parameter('urdf_path', '/ThesisRosGITV1/src/urdf_2dof/urdf/2dof.urdf')
self.declare_parameter('mesh_dir', '/ThesisRosGITV1/src/urdf_2dof/meshes')

urdf_path = self.get_parameter('urdf_path').value
mesh_dir = self.get_parameter('mesh_dir').value
```

What are parameters?

- Configuration values that can be changed without modifying code
- Can be set via command line, launch files, or YAML files

Example usage:

```bash
ros2 run urdf_2dof pendulum_sim_node.py --ros-args -p urdf_path:=/new/path.urdf
```

Why use them?

- Flexibility: Change paths without editing code
- Reusability: Same node works in different environments

D. Loading the Robot Model

```python
self.robot = RobotWrapper.BuildFromURDF(urdf_path, [mesh_dir])
self.model, self.data = self.robot.model, self.robot.data
self.model.gravity.linear = np.array([0.0, -9.81, 0.0])
```

Pinocchio's role:

- `model`: Contains robot structure (links, joints, inertias)
- `data`: Working memory for computations (preallocated arrays)
- `gravity`: Sets gravitational acceleration vector (Y-axis downward)

URDF parsing: Pinocchio reads your URDF file and builds:

- Kinematic tree (parent-child relationships)
- Inertia matrices
- Joint limits and types

E. Joint and Frame Identification

```python
self.joint_names = [self.model.names[i] for i in range(1, self.model.njoints)]
```

Why `range(1, ...)`?

- Index 0 is "universe" (root of kinematic tree)
- Actual joints start at index 1

```python
self.ee_frame = "EndEffector"
self.ee_id = self.model.getFrameId(self.ee_frame)
```

Frames in Pinocchio:

- **Joints**: Moving connections between links
- **Frames**: Reference points attached to links (e.g., end-effector, sensors)
- Your code tracks the "EndEffector" frame for control

F. Control Target and Gains

```python
self.x_des = np.array([0.2, 0.05])      # Desired position [x, y] in meters
self.xdot_des = np.zeros(2)            # Desired velocity (stationary)
self.xddot_des = np.zeros(2)           # Desired acceleration

self.Kp = np.diag([50, 50])            # Proportional gain matrix
self.Kd = np.diag([40, 40])            # Derivative gain matrix
```

Control law (PD controller):

```
τ = Kp * (x_desired - x_actual) + Kd * (ẋ_desired - ẋ_actual)
```

This creates a "spring-damper" system pulling the end-effector toward the target.

G. Joint Limits

```python
self.q_min = np.array([-np.pi, -np.pi])  # -180° both joints
self.q_max = np.array([np.pi, np.pi])    # +180° both joints
```

Purpose:

- Prevent unrealistic configurations
- Stop joints from spinning endlessly
- Enforce physical constraints (e.g., joint can't rotate 360°)

Implementation:

```python
self.y[:2] = np.clip(self.y[:2], self.q_min, self.q_max)
```

`np.clip` saturates joint angles to stay within bounds.

H. Simulation State

```python
self.dt = 0.01                    # Timestep (100 Hz)
self.time = 0.0                   # Simulation clock

q_init = np.array([0.5, 0.35])         # Initial joint angles (radians)
self.y = np.concatenate([q_init, np.zeros(2)])  # [q1, q2, q̇1, q̇2]
```

State vector `self.y`:

- Dimensions: `[nq + nv]` (positions + velocities)
- For 2DOF: `[q1, q2, v1, v2]` → 4 elements
- Updated every timestep by numerical integration

I. Publisher Setup

```python
self.joint_pub = self.create_publisher(JointState, 'joint_states', 10)
```

Breakdown:

- `JointState`: Message type (from `sensor_msgs`)
- `'joint_states'`: Topic name (standard ROS convention)
- `10`: Queue size (buffers 10 messages if subscribers are slow)

Who subscribes?

- `robot_state_publisher`: Converts joint states to TF transforms
- Your own monitoring nodes (optional)

J. Timer Creation

```python
self.timer = self.create_timer(self.dt, self.step_and_publish)
```

What happens:

1. Timer fires every `self.dt` seconds (0.01s = 100 Hz)
2. Calls `self.step_and_publish()` method
3. Runs in background thread (non-blocking)

Why use a timer instead of a loop?

- ROS 2 manages timing precisely
- Integrates with event loop (`rclpy.spin`)
- Other callbacks can run concurrently

K. Dynamics Function

```python
def robot_dynamics(self, t, y):
    q = y[:self.model.nq]   # Extract positions
    v = y[self.model.nq:]   # Extract velocities
```

Purpose: Computes `ẏ = [v, q̈]` for the ODE solver.

Step 1: Forward Kinematics

```python
pin.forwardKinematics(self.model, self.data, q, v)
pin.updateFramePlacements(self.model, self.data)
```

- Computes link positions/orientations for current `q`
- Updates frame transformations (needed for end-effector position)

Step 2: Jacobian Computation

```python
J = pin.computeFrameJacobian(self.model, self.data, q, self.ee_id)[:2, :]
```

- **Jacobian matrix J**: Maps joint velocities to end-effector velocities

```
ẋ = J(q) * q̇
```

- `[:2, :]`: Extract only X-Y rows (ignore Z and rotation)

```python
J_dot = pin.getFrameJacobianTimeVariation(self.model, self.data, self.ee_id, pin.WORLD)[:2, :]
```

- **Time derivative of Jacobian**: Needed for acceleration mapping

```
ẍ = J * q̈ + J̇ * q̇
```

Step 3: Error Calculation

```python
x = self.data.oMf[self.ee_id].translation[:2]  # Current EE position
x_err = self.x_des - x                         # Position error
xdot_err = self.xdot_des - J @ v               # Velocity error
```

Step 4: Desired Acceleration (PD Control)

```python
x_acc_des = self.xddot_des + self.Kd @ xdot_err + self.Kp @ x_err
```

This is the task-space control law. In continuous time:

```
ẍ_desired = ẍ_ref + Kd * (ẋ_ref - ẋ) + Kp * (x_ref - x)
```

Step 5: Inverse Dynamics

```python
B = pin.crba(self.model, self.data, q)  # Mass matrix
n = pin.rnea(self.model, self.data, q, v, np.zeros(self.model.nv))  # Coriolis + gravity
```

- **CRBA** (Composite Rigid Body Algorithm): Computes inertia matrix `B(q)`
- **RNEA** (Recursive Newton-Euler Algorithm): Computes nonlinear terms `n(q, q̇)`

Equation of motion:

```
B(q) * q̈ + n(q, q̇) = τ
```

Step 6: Task-Space to Joint-Space Mapping

```python
qddot_task = np.linalg.pinv(J, rcond=1e-2) @ (x_acc_des - J_dot @ v)
```

From `ẍ = J * q̈ + J̇ * q̇`, solve for `q̈`:

```
q̈ = J⁺ * (ẍ_desired - J̇ * q̇)
```

`pinv` is the **pseudo-inverse** (handles redundancy/singularities).

Step 7: Compute Control Torque

```python
u = B @ qddot_task + n
```

This is the feedforward control law (model-based control).

Step 8: Actual Acceleration

```python
qddot = np.linalg.solve(B, u - n)
```

Solve `B * q̈ = u - n` for `q̈` (this is what the robot actually does).

Return derivative:

```python
return np.concatenate((v, qddot))  # [q̇, q̈]
```

L. Integration and Publishing

```python
def step_and_publish(self):
    # Clamp before integration
    self.y[:2] = np.clip(self.y[:2], self.q_min, self.q_max)

    # Integrate dynamics
    sol = solve_ivp(
        self.robot_dynamics,
        [self.time, self.time + self.dt],
        self.y,
        method='RK45',
        max_step=0.001
    )
```

`solve_ivp`: Scipy's ODE solver

- **Method RK45**: Runge-Kutta 4th/5th order (adaptive step size)
- **Time span**: `[t_start, t_end]`
- **Initial value**: Current state `self.y`
- **max_step**: Prevents integration from taking huge steps

Update state:

```python
self.y = sol.y[:, -1]  # Take final value from integration
```

Enforce limits again:

```python
self.y[:2] = np.clip(self.y[:2], self.q_min, self.q_max)

for i in range(2):
    if self.y[i] <= self.q_min[i] and self.y[2+i] < 0:
        self.y[2+i] = 0  # Stop moving into lower limit
    elif self.y[i] >= self.q_max[i] and self.y[2+i] > 0:
        self.y[2+i] = 0  # Stop moving into upper limit
```

This is a **collision response**: if joint hits limit and is still moving into it, set velocity to zero.

Publish message:

```python
msg = JointState()
msg.header = Header()
msg.header.stamp = self.get_clock().now().to_msg()  # Current ROS time
msg.name = self.joint_names
msg.position = q.tolist()
msg.velocity = v.tolist()
msg.effort = []

self.joint_pub.publish(msg)
```

Message structure:

- **header.stamp**: Timestamp (important for synchronization)
- **name**: Joint names (e.g., `['joint1', 'joint2']`)
- **position/velocity**: Current state
- **effort**: Empty (not used in simulation)

M. Main Entry Point

```python
def main(args=None):
    rclpy.init(args=args)  # Initialize ROS 2 system

    try:
        node = PendulumSimulator()  # Create node
        rclpy.spin(node)            # Run event loop (blocks here)
    except KeyboardInterrupt:
        pass
    finally:
        try:
            if rclpy.ok():
                node.destroy_node()  # Cleanup
                rclpy.shutdown()     # Disconnect from ROS 2
        except:
            pass

if __name__ == '__main__':
    main()
```

Execution flow:

1. `rclpy.init()`: Connects to ROS 2 middleware (DDS)
2. Create node: Registers with ROS 2 daemon
3. `rclpy.spin()`: **Blocks here** and runs event loop
   - Timer callbacks fire
   - Publisher sends messages
   - Handles Ctrl+C gracefully
4. Cleanup: Destroys resources when exiting

---

4. Launch File Explained

File: `display.launch.py`

Launch files automate starting multiple nodes with configuration. Written in Python for ROS 2.

A. Finding Package Path

```python
pkg_path = launch_ros.substitutions.FindPackageShare(package='urdf_2dof').find('urdf_2dof')
```

Purpose: Get absolute path to installed package (works after `colcon build`).

Why needed? Files are installed to system directories, not source folder.

B. Loading URDF

```python
urdf_model_path = os.path.join(pkg_path, 'urdf/2dof.urdf')

with open(urdf_model_path, 'r') as infp:
    robot_desc = infp.read()

params = {'robot_description': robot_desc}
```

`robot_description`: Standard ROS parameter name for URDF content.

Why load as string? Nodes receive it as a parameter, not a file path.

C. Node Definitions

1. Robot State Publisher

python

```python
robot_state_publisher_node = launch_ros.actions.Node(
    package='robot_state_publisher',
    executable='robot_state_publisher',
    output='screen',
    parameters=[params]
)
```

What it does:

- Subscribes to `/joint_states`
- Publishes TF transforms (link poses) based on URDF + joint angles
- Enables RViz to visualize robot

TF (Transform) System:

- Tree of coordinate frames (world → link1 → link2 → end-effector)
- Each frame's pose relative to parent
- RViz uses this to position 3D models

---

2. Joint State Publisher (non-GUI)

python

```python
joint_state_publisher_node = launch_ros.actions.Node(
    package='joint_state_publisher',
    executable='joint_state_publisher',
    condition=launch.conditions.UnlessCondition(LaunchConfiguration('gui'))
)
```

Purpose: Publishes dummy joint states (all zeros or fixed values).

When used: Headless mode or if no other source of joint states exists.

Condition: Only runs if `gui:=False` is passed to launch file.

---

3. Joint State Publisher GUI

```python
joint_state_publisher_gui_node = launch_ros.actions.Node(
    package='joint_state_publisher_gui',
    executable='joint_state_publisher_gui',
    condition=launch.conditions.IfCondition(LaunchConfiguration('gui'))
)
```

Purpose: Opens sliders to manually control joint angles.

When used: For testing URDF visualization without dynamics.

Condition: Only runs if `gui:=True` (default).

---

4. RViz2

```python
rviz_node = launch_ros.actions.Node(
    package='rviz2',
    executable='rviz2',
    arguments=['-d', rviz_config_path]
)
```

Purpose: 3D visualization tool.

Config file: Loads saved camera position, display settings, etc.

D. Launch Arguments

```python
gui_arg = launch.actions.DeclareLaunchArgument(
    name='gui',
    default_value='True',
    description='Flag to enable joint_state_publisher_gui'
)
```

How to use:

```bash
ros2 launch urdf_2dof display.launch.py gui:=False
```

This makes launch files flexible without editing code.

---

5. Build System

package.xml

```xml
<package format="3">
  <name>urdf_2dof</name>
  <buildtool_depend>ament_cmake</buildtool_depend>

  <exec_depend>joint_state_publisher</exec_depend>
  <exec_depend>robot_state_publisher</exec_depend>
  <exec_depend>rviz2</exec_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

Purpose: Package metadata (like `package.json` in Node.js).

Dependency types:

- `buildtool_depend`: Needed to compile (e.g., CMake)
- `build_depend`: Needed at compile time (e.g., C++ libraries)
- `exec_depend`: Needed at runtime (e.g., other ROS nodes)

Your dependencies: All runtime (`exec_depend`) because you launch existing nodes.

CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.8)
project(urdf_2dof)

find_package(ament_cmake REQUIRED)

install(
  DIRECTORY launch config urdf meshes
  DESTINATION share/${PROJECT_NAME}
)

install(PROGRAMS
  scripts/pendulum_sim_node.py
  DESTINATION lib/${PROJECT_NAME}
)

ament_package()
```

What it does:

1. `find_package(ament_cmake)`: Loads ROS 2 build tools
2. `install(DIRECTORY ...)`: Copies folders to install space
3. `install(PROGRAMS ...)`: Installs Python script as executable
4. `ament_package()`: Generates package metadata

Install locations:

- Directories → `install/urdf_2dof/share/urdf_2dof/`
- Executables → `install/urdf_2dof/lib/urdf_2dof/`

Why needed? ROS 2 runs from install space, not source space.

---

6. Control Theory Implementation

Operational Space Control (Task-Space Control)

Your code implements **resolved acceleration control** in task space.

Key idea: Control end-effector position/velocity directly, not joint angles.

Step-by-Step Derivation

1. Forward Kinematics

```
x = f(q)  ← Position of end-effector given joint angles
```

2. Differential Kinematics

```
ẋ = J(q) * q̇  ← Velocity relationship
ẍ = J * q̈ + J̇ * q̇  ← Acceleration relationship
```

3. Task-Space Control Law

```
ẍ_desired = ẍ_ref + Kd * ė + Kp * e

where:
  e = x_desired - x       ← Position error
  ė = ẋ_desired - ẋ       ← Velocity error
```

4. Inverse Kinematics (Acceleration Level)

```
q̈ = J⁺ * (ẍ_desired - J̇ * q̇)
```

5. Inverse Dynamics (Feedforward Control)

```
τ = B(q) * q̈ + n(q, q̇)

where:
  B(q) = inertia matrix
  n(q, q̇) = Coriolis + gravity + friction
```

Why this works:

- **Feedforward term** (`B * q̈ + n`) cancels robot dynamics
- **Feedback term** (PD gains) corrects for model errors
- **Result**: Exact tracking if model is perfect, robust tracking otherwise

---

Numerical Integration

Your code uses **Runge-Kutta 4/5** (RK45) with adaptive step size.

Why not Euler integration?

```python
# Euler (bad for dynamics):
q_new = q + dt * v
v_new = v + dt * a

# Accumulates error quickly!
```

RK45:

- Estimates derivatives at multiple points within timestep
- Adjusts step size based on error tolerance
- More accurate for nonlinear systems

Your settings:

```python
method='RK45',
max_step=0.001  # Never take steps larger than 1ms
```

---

7. Workflow Diagram

System Architecture

```
┌─────────────────────────────────────────────────────────────┐
│              Launch File Execution                          │
│  (display.launch.py)                                        │
└──────────────────────────┬──────────────────────────────────┘
                           │
          ┌────────────────┼──────────────────┐
          │                │                  │
          ▼                ▼                  ▼
    ┌──────────┐    ┌──────────┐    ┌──────────┐
    │  Node 1  │    │  Node 2  │    │  Node 3  │
    │  Robot   │    │  Joint   │    │  RViz2   │
    │  State   │    │  State   │    │          │
    │Publisher │    │Publisher │    │          │
    └────┬─────┘    └────┬─────┘    └────┬─────┘
         │               │               │
         │ subscribes    │ publishes     │ subscribes
         │ to            │ to            │ to
         ▼               ▼               ▼
    /joint_states topic          /tf topic
         ▲                           │
         │                           │
         │ publishes                 │ visualizes
         │                           ▼
    ┌────┴──────────────────────┐  ┌──────────┐
    │  Your Custom Node         │  │ 3D Model │
    │  PendulumSimulator        │  │ in RViz  │
    │  (pendulum_sim_node)      │  └──────────┘
    └───────────────────────────┘
         │
         │ Timer (100 Hz)
         ▼
    ┌──────────────────┐
    │ step_and_publish()│
    │  1. Integrate    │
    │  2. Enforce      │
    │     limits       │
    │  3. Publish state │
    └──────────────────┘
```

Execution Timeline

```
t=0.00s: rclpy.init()
        ├─> Connect to ROS 2 middleware (DDS)
        └─> Initialize logging system

t=0.01s: PendulumSimulator.__init__()
        ├─> Load URDF from file
        ├─> Build Pinocchio model
        ├─> Create publisher
        └─> Start 100Hz timer

t=0.01s: rclpy.spin(node)  [BLOCKS HERE]
        └─> Enter event loop

t=0.02s: Timer fires → step_and_publish()
        ├─> Integrate dynamics (solve_ivp)
        │   └─> Calls robot_dynamics() multiple times
        ├─> Clamp joints
        ├─> Build JointState message
        └─> Publish to /joint_states

t=0.02s: robot_state_publisher receives message
        ├─> Compute forward kinematics
        ├─> Generate TF transforms
        └─> Publish to /tf

t=0.02s: RViz receives TF transforms
        └─> Update 3D visualization

t=0.03s: Timer fires again...
        [Repeat every 10ms]

User presses Ctrl+C:
        ├─> KeyboardInterrupt exception
        ├─> node.destroy_node()
        └─> rclpy.shutdown()
```

Advanced Topics

Why Pinocchio?

Pinocchio is a fast rigid body dynamics library. Compared to alternatives:

Library	Speed	Features
Pinocchio	Fastest	Analytical derivatives
PyBullet	Medium	Full physics sim
MuJoCo	Fast	Contact dynamics
KDL	Slow	Basic kinematics

Your use case: You only need kinematics/dynamics (no collisions), so Pinocchio is perfect.

Message Synchronization

ROS 2 uses **DDS** (Data Distribution Service) for communication:

- **Reliable QoS**: Messages guaranteed to arrive (default for most topics)
- **Best-effort QoS**: Occasional drops OK (for sensor data like camera images)
- **Timestamps**: `header.stamp` ensures time-consistent data

Your code: Uses reliable QoS (default for `JointState`).

Coordinate Frames

Your robot has these frames (from URDF):

```
world (fixed)
  └─> link1 (rotates around joint1)
        └─> link2 (rotates around joint2)
              └─> EndEffector (fixed to link2)
```

Pinocchio tracks:

- Joint positions: `q = [q1, q2]`
- Link poses: `self.data.oMf[i]` (4×4 transformation matrices)
- End-effector position: `self.data.oMf[self.ee_id].translation`

Common Issues & Solutions

1. "Could not find frame 'EndEffector'"

Cause: Frame name in Python doesn't match URDF.

Solution: Check your URDF:

```xml
<link name="EndEffector">  <!-- Must match exactly -->
```

---

2. Joints spinning wildly

Cause: No joint limits enforced.

Solution: Your code already fixes this:

```python
self.y[:2] = np.clip(self.y[:2], self.q_min, self.q_max)
```

---

3. Robot vibrating/jittering

Causes:

- Gains too high (`Kp`, `Kd`)
- Integration timestep too large
- Singularities in Jacobian

Solutions:

- Reduce gains
- Decrease `max_step` in `solve_ivp`
- Add damping: `u = B @ qddot_task + n - damping * v`

---

4. RViz shows nothing

Checklist:

1. Is `robot_state_publisher` running? → `ros2 node list`
2. Is `/joint_states` being published? → `ros2 topic echo /joint_states`
3. Is RV