# 6-DOF Task-Space Teleoperation Architecture

## 1. Overview

This document describes the architecture and data flow of a **6-DOF task-space keyboard teleoperation system** implemented in **ROS 2**. The system enables Cartesian (task-space) control of a 6-DOF robotic manipulator using keyboard inputs, converts Cartesian increments into joint motion via **Jacobian-based inverse kinematics**, and visualizes the motion in **RViz** through the /joint_states interface.

The design is intentionally **ROS-light** (no ros2_control, no MoveIt) and is intended primarily for:

- RViz visualization
- Algorithm validation and debugging
- Understanding task-space kinematics and Jacobian-based IK flow

---

## 2. High-Level System Architecture

```
 Keyboard Input
        |
        v
Task-Space Increment (Δx)
        |
        v
World to End-Effector Frame Conversion
        |
        v
Jacobian Computation (J)
        |
        v
Pseudoinverse (J⁺)
        |
        v
```

```
    Joint Increment (Δq)
              |
              v
    Joint State Update
              |
              v
  /joint_states → robot_state_publisher → RViz
```

This pipeline cleanly separates **input handling**, **kinematics**, and **visualization**.

---

# 3. Core Components

## 3.1 Keyboard Interface Layer

**Purpose**
Capture raw keyboard input and map it to task-space motion commands.

**Key Characteristics**

- Uses termios and tty for raw, non-blocking keyboard input
- Each key corresponds to a small Cartesian or rotational increment
- Task-space mapping:
    - Translation: X, Y, Z
    - Rotation: Roll (Rx), Pitch (Ry), Yaw (Rz)
- Independent of ROS topics to maintain deterministic and low-latency control

---

## 3.2 Task-Space Teleop Node (TaskSpaceTeleop6DOF)

**ROS Node Name:** task_space_teleop_6dof

**Responsibilities**

- Load robot kinematic model
- Maintain joint-space and task-space state
- Convert task-space commands to joint motion
- Publish joint states for visualization

---

# 4. Robot Model & Kinematics

## 4.1 URDF Loading

- Robot description is loaded via **XACRO**
- ament_index_python is used to locate the description package
- XACRO is expanded into URDF XML
- A temporary URDF file is generated for **IKPy** compatibility

**Pipeline**
XACRO –> URDF XML –> Temporary File –> IKPy Chain

---

## 4.2 IK Chain Construction

- IKPy.Chain.from_urdf_file() is used
- base_link defined as the fixed base
- Active links correspond to joints 1–6
- Tool link included for correct end-effector visualization

This chain acts as the **source** for both forward and inverse kinematics.

---

# 5. State Representation

## 5.1 Joint Space

- Joint vector: **q** $\in \mathbb{R}^6$

  Initialized to a non-singular configuration:
  q = [90°, 0°, 90°, 0°, 90°, 0°]

## 5.2 Task Space

- End-effector position: **ee_pos** $\in \mathbb{R}^3$
- End-effector orientation: **ee_rot** $\in$ **SO(3)**
- Continuously updated using forward kinematics

---

# 6. Forward Kinematics (FK)

### Purpose

- Compute current end-effector pose from joint angles

### Required For

- Frame transformations
- Jacobian computation
- State consistency

### Method

- IKPy.forward_kinematics()
- Returns a homogeneous transformation matrix

---

# 7. Jacobian Computation

## 7.1 Numerical Jacobian

- Jacobian is computed numerically
- Jacobian matrix: $\mathbf{J} \in \mathbb{R}^{6\times6}$

### Structure

J = [ v

ω ]

### Procedure

- Apply a small change ε to each joint
- Measure resulting change in:
    - Position → linear velocity
    - Orientation → angular velocity (rotation vector)

This avoids analytical Jacobian derivation and keeps the implementation **robot-agnostic**.

---

# 8. Inverse Kinematics (Where IK Happens)

Inverse kinematics is solved incrementally using the Jacobian:

$$\Delta q = J^{+} \cdot \Delta x$$

Where:

- **$\Delta x$** = task-space increment (6D twist)
- **$J^{+}$** = Moore–Penrose pseudoinverse of the Jacobian
- **$\Delta q$** = joint-space increment

**Benefits**

- Smooth, continuous motion
- Local linearization
- Real-time teleoperation capability

---

# 9. Frame Handling

### 9.1 World → End-Effector Frame Conversion

- Keyboard translation inputs are defined in the **world frame**

  Converted into the **end-effector frame** before IK:
  $\Delta x_{ee} = R^{T} \cdot \Delta x_{world}$

**Ensures**

- Intuitive tool-relative motion
- Consistent Cartesian behavior regardless of orientation

---

# 10. Joint State Publishing

**Topic:** /joint_states

**Purpose**

- RViz visualization
- Input to robot_state_publisher

**Published Data**

- Joint names
- Joint positions
- Timestamped header

No controllers, trajectories, or hardware interfaces are involved.

---

# 11. Visualization Pipeline (RViz)

```
TaskSpaceTeleop6DOF
       |
       v
 /joint_states
       |
       v
robot_state_publisher
       |
       v
      TF
       |
       v
     RViz
```

---

# 12. Control Strategy Used

## 12.1 Control Strategy

The system uses **Task-Space Incremental Control with Jacobian Pseudoinverse IK**.

- User inputs generate small Cartesian increments:
  (Δx, Δy, Δz, ΔRx, ΔRy, ΔRz)
- These increments represent a desired end-effector twist

  Joint updates are computed as:
  $\Delta q = J^{+} \cdot \Delta x$

- 

This makes the controller:

- Incremental

- Smooth
- Locally stable

---

## 13. Why This Control Strategy Is Safe

**a) Incremental Motion Only**

- No large target jumps
- Small Cartesian steps per keypress
- Prevents sudden joint accelerations

**b) No Trajectory Precomputation**

- Motion is purely reactive
- Eliminates risks from outdated goals

**c) Local Linearization**

- Jacobian IK operates locally
- Avoids joint flips common in closed-form IK

**d) Implicit Singularity Awareness**

- Near singularities, pseudoinverse reduces motion authority
- Robot naturally slows down

**e) Visualization-Only Output**

- Only /joint_states are published
- No torque, velocity, or effort commands
- Inherently safe during development

---

## 14. Where Force-Compliance Would Plug In

Force-compliance can be integrated **cleanly and modularly** into the existing task-space teleoperation pipeline **without altering the core kinematic architecture**. The compliance logic operates entirely in **Cartesian space**, before the Jacobian-based inverse kinematics step.

---

### 14.1 Compliance Insertion Point in the Control Loop

The compliance layer modifies the **task-space command**, not the joint-space solution.

**Control Flow:**

User Command ($\Delta x\_user$)
+
Compliance Response ($\Delta x\_compliance$)
↓
Modified Task-Space Command ($\Delta x\_compliant$)
↓
Jacobian Pseudoinverse IK
↓
Joint Increment

$$\Delta q = J^{+} \cdot \Delta x_{compliant}$$

where,

- $\Delta q$ — Joint Increment
- $J^{+}$ — Jacobian Pseudoinverse
- $\Delta x\_compliant$ — Compliant Task-Space Increment (6×1 Cartesian motion vector)

This placement ensures:

- Compliance is **independent of robot kinematics**
- The same IK pipeline remains valid
- Safety behavior is enforced **before joint motion is generated**

---

### 14.2 Admittance Control (Recommended for Teleoperation)

**Admittance control** converts external forces into motion.
It is the **preferred strategy for teleoperated medical robots**, especially when interacting with soft human tissue.

**Control Law:**

$$\Delta x_{compliant} = \Delta x_{user} + M^{-1}(F_{ext} - D\dot{x} - Kx)$$

Where:

- $F_{ext}$ — measured external force/torque (FT sensor)
- M — virtual mass (inertia shaping)
- D — virtual damping (motion smoothness)
- K — virtual stiffness (contact firmness)

**Behavior:**

- Robot **yields when pushed**
- Maintains contact without instability
- Filters operator tremor and patient motion
- Allows safe surface following (e.g., Abdomen ultrasound scanning)

**Why admittance fits this architecture**

- Operates directly on **Δx**
- No torque control required
- Compatible with velocity or position-controlled robots
- Ideal for **human-in-the-loop** systems

---

### 14.3 Impedance Control Variant (Torque-Controlled Robots)

**Impedance control** regulates the relationship between motion and force by **generating forces**, not motion.

**Control Concept:**

1. Compute Cartesian pose error:

$$e = x_{desired} - x_{actual}$$

2. Generate interaction force:

$$F=Ke+D\dot{e}$$

where,

- F → Cartesian force at the end effector
- e → Cartesian position error
- $\dot{e}$ → Cartesian velocity error
- K → Stiffness matrix (spring behavior)
- D → Damping matrix (damper behavior)

3. Map Cartesian force to joint torques:

$$\tau=J^T.F$$

where,

- $\tau$ → joint torque vector
- $J^T$ → transpose of the Jacobian
- F → Cartesian force at the end effector

**Requirements:**

- Torque-controlled robot
- Real-time dynamics model
- Low-latency force feedback loop

**Not used in this RViz-only setup**, as it publishes only joint positions and has no hardware torque interface.

---

### 14.4 Hybrid Admittance–Impedance Architecture (Medical Robotics)

For real medical systems, the most effective approach is **hybrid compliance**:

- **Admittance control** at the Cartesian command level
  → handles patient interaction and safety

- **Impedance control** at the low-level actuator interface
  → ensures stable force rendering

This architecture naturally extends to:

- Ultrasound probe contact regulation
- Patient-induced disturbance rejection
- Safe tele-echography
- Force-guided surface following
- 

### 14.4 Medical Robotics Relevance

This architecture directly supports:

- Tissue contact control
- Patient-induced disturbance rejection
- Safe human–robot interaction
- Hybrid admittance–impedance control

---

## 15. Design Characteristics

### Advantages

- Fully task-space driven
- No MoveIt or ros2_control
- Transparent and deterministic IK flow

### Limitations

- No joint-limit enforcement
- No explicit singularity avoidance
- No dynamics or force control
- Visualization-only (not hardware-safe)

---

## 16. Final Summary

- Uses incremental task-space Jacobian IK
- Safe due to small-step motion and visualization-only output
- Force-compliance integrates naturally at the Cartesian level
- Scales cleanly from RViz simulation to real medical robots