IK System Architecture

Technical Documentation

Version 1.0

Inverse Kinematics Framework

June 15, 2025

Contents

1 Introduction

The Inverse Kinematics (IK) system has been redesigned with the IK Context as the central API hub. This architecture provides a clean, ROS-like separation of concerns where IK Context manages all solver operations and automatically handles TCP (Tool Center Point) integration.

1.1 Key Features

- Central IK Context as single source of truth
- Automatic TCP tool integration
- Dynamic solver loading from `/public/IKSolvers/`
- Standardized solver interface
- Real-time end effector tracking
- Extensible architecture for custom solvers

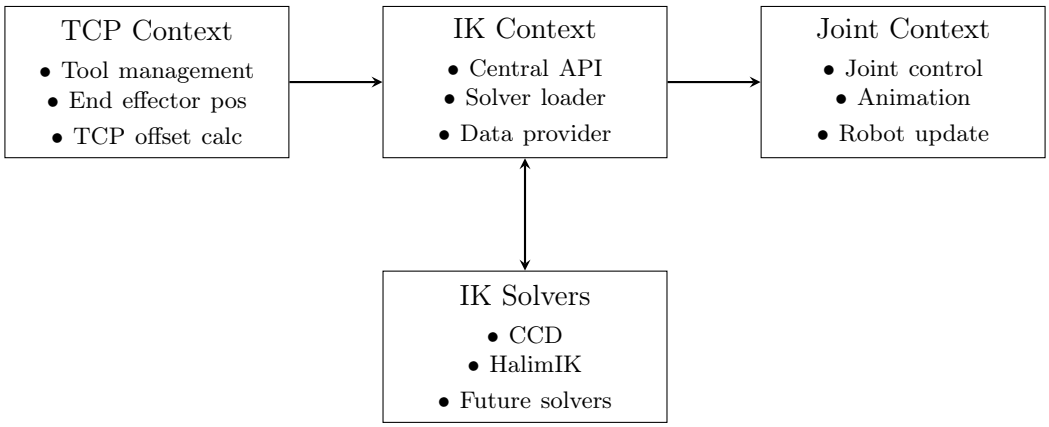
1.2 Design Principles

1. **Centralized Control**: IK Context manages all IK operations
2. **TCP Transparency**: Tools work automatically without solver modifications
3. **Dynamic Loading**: Add solvers without recompiling
4. **Clean Interface**: Standardized solver API
5. **Performance Optimized**: Efficient data flow and minimal redundancy

2 Architecture Overview

2.1 System Architecture

The IK system follows a hierarchical architecture with clear data flow:



2.2 Data Flow

The system implements a unidirectional data flow pattern:

1. **TCP Tool Attached**
   - TCP Context calculates offset
   - Emits `'tcp:endeffector-updated'`
   - IK Context receives update
   - Stores TCP-aware position
2. **IK Execution Request**
   - IK Context prepares data
   - Calls `solver.solve()` with all data
   - Solver returns joint angles
   - IK Context emits to Joint Context
   - Joint Context animates robot

3 Core Components

3.1 IK Context

The IK Context serves as the central hub for all IK operations:

3.1.1 Responsibilities

- Automatic TCP integration
- Dynamic solver loading
- Data management
- State tracking
- Event coordination

3.1.2 Key Methods

Method	Description
`loadSolver()`	Dynamically loads IK solver from file
`executeIK()`	Executes IK calculation with current solver
`configureSolver()`	Updates solver configuration
`setCurrentSolver()`	Switches active solver
`stopAnimation()`	Halts ongoing IK animation

Table 1: IK Context Core Methods

3.2 Standardized Solver Interface

All IK solvers implement a standardized interface for consistency:

```
class IKSolver {
  static metadata = {
    name: "Solver Name",
    description: "Description",
    author: "Author Name",
    version: "1.0.0"
  };

  static defaultConfig = {
    // Solver-specific configuration
    maxIterations: 100,
    tolerance: 0.001
  };

  constructor(config = {}) {
    Object.assign(this, IKSolver.defaultConfig, config);
  }

  async solve(params) {
    const {
      robot,              // Robot model
      currentPosition,    // Current end effector position (includes
          TCP)
      currentOrientation, // Current orientation quaternion
      targetPosition,     // Target position
      targetOrientation   // Target orientation (euler angles)
    } = params;

    // Calculate joint angles
    const solution = this.calculateJointAngles();

    return solution; // { jointName: angle, ... }
  }
}

export default IKSolver;
```

Listing 1: IK Solver Interface

4 TCP Integration

4.1 Automatic TCP Handling

The TCP integration is completely transparent to users and solver developers:

4.1.1 Tool Attachment Flow

```
// User attaches a gripper tool
tcpContext.attachTool(robotId, 'gripper');

// IK automatically uses gripper tip position
const { currentPosition } = useIK(); // Returns gripper tip position

```

```
// Execute IK - solver automatically gets TCP position
moveToTarget(targetPosition); // Solver receives TCP-aware position
```

Listing 2: TCP Tool Attachment Example

4.1.2 TCP State Management

State	Behavior
TCP Tool Attached	IK Context uses tool tip position
TCP Tool Hidden	IK Context still uses tool position
TCP Tool Removed	IK Context falls back to robot end effector
No TCP Tool	IK Context uses robot's default end effector

Table 2: TCP State Management

4.2 Event Flow

The system uses events for loose coupling between contexts:

```
// TCP Context emits when tool state changes
EventBus.emit('tcp:endeffector-updated', {
  robotId: 'ur5_robot',
  endEffectorPoint: { x: 0.5, y: 0.3, z: 0.8 },
  endEffectorOrientation: { x: 0, y: 0, z: 0, w: 1 },
  hasTCP: true
});

// IK Context listens and updates automatically
EventBus.on('tcp:endeffector-updated', (data) => {
  if (data.robotId === activeRobotId) {
    updateEndEffectorPosition(data.endEffectorPoint);
    updateEndEffectorOrientation(data.endEffectorOrientation);
  }
});
```

Listing 3: TCP Event Flow

5 Creating IK Solvers

5.1 Solver Development Guide

Creating a new IK solver involves implementing the standardized interface:

5.1.1 Step 1: Create Solver File

Create a new file in `/public/IKSolvers/YourSolver.jsx`:

```
import * as THREE from 'three';

class YourSolver {
  static metadata = {
```

```
    name: "Your Solver Name",
    description: "What your solver does",
    author: "Your Name",
    version: "1.0.0"
  };

  static defaultConfig = {
    maxIterations: 100,
    tolerance: 0.001,
    // Add your custom parameters
    customParam1: 1.0,
    customParam2: true
  };

  constructor(config = {}) {
    Object.assign(this, YourSolver.defaultConfig, config);

    // Initialize reusable objects
    this.tempVector = new THREE.Vector3();
    this.tempQuaternion = new THREE.Quaternion();
  }

  // Optional: Configuration methods
  getConfig() {
    return {
      maxIterations: this.maxIterations,
      tolerance: this.tolerance,
      customParam1: this.customParam1,
      customParam2: this.customParam2
    };
  }

  configure(config) {
    Object.assign(this, config);
  }

  async solve(params) {
    const {
      robot,
      currentPosition,    // Already includes TCP offset!
      currentOrientation,
      targetPosition,
      targetOrientation
    } = params;

    // Validate inputs
    if (!robot || !robot.joints) {
      console.error('[YourSolver] Invalid robot model');
      return null;
    }

    // Your IK algorithm implementation
    const solution = this.calculateSolution(params);

    return solution; // { joint1: angle1, joint2: angle2, ... }
  }
```

```
61
62    // Private helper methods
63    calculateSolution(params) {
64      // Implementation details
65      const jointAngles = {};
66
67      // Calculate joint angles...
68
69      return jointAngles;
70    }
71  }
72
73  export default YourSolver;
```

Listing 4: Custom IK Solver Template

5.1.2 Step 2: Solver Best Practices

1. **Use Provided Positions**: Don't calculate end effector position - use `currentPosition`
2. **Return Joint Map**: Return an object mapping joint names to angles
3. **Handle Errors**: Return `null` if no solution found
4. **Optimize Performance**: Use reusable objects to reduce garbage collection
5. **Document Configuration**: Clearly document all configuration parameters

5.1.3 Step 3: Testing Your Solver

```
1  // Configure and test your solver
2  const { configureSolver, setCurrentSolver } = useIK();
3
4  // Select your solver
5  setCurrentSolver('YourSolver');
6
7  // Configure parameters
8  configureSolver('YourSolver', {
9    maxIterations: 200,
10   customParam1: 0.5
11 });
12
13 // Test execution
14 moveToTarget(targetPosition);
```

Listing 5: Testing Custom Solver

6 API Reference

6.1 useIK Hook

The `useIK()` hook provides a clean interface for IK operations:

Property/Method	Description
State Properties	
`currentPosition`	Current end effector position (TCP-aware)
`currentOrientation`	Current orientation quaternion
`targetPosition`	Target position for IK
`targetOrientation`	Target orientation for IK
`isAnimating`	Animation in progress flag
`solverStatus`	Current solver status message
`currentSolver`	Active solver name
`availableSolvers`	List of available solver names
Methods	
`setTargetPosition()`	Set target position
`setTargetOrientation()`	Set target orientation
`setCurrentSolver()`	Change active solver
`moveToTarget()`	Execute IK to target
`moveRelative()`	Move relative to current
`rotateRelative()`	Rotate relative to current
`syncTargetToCurrent()`	Sync target with current position
`stopAnimation()`	Stop current animation
`configureSolver()`	Configure solver parameters
`getSolverSettings()`	Get current solver settings

Table 3: useIK Hook API Reference

6.2 Event Reference

6.2.1 IK Context Listens For

Event	Description
`'tcp:endeffector-updated'`	Updates end effector position with TCP data
`'ik:animation-complete'`	Animation finished notification
`'joint:stop-animation'`	Stop animation request

Table 4: IK Context Input Events

6.2.2 IK Context Emits

Event	Description
`'ik:joint-values-calculated'`	Sends calculated joint values to Joint Context
`'joint:stop-animation'`	Requests animation stop from Joint Context

Table 5: IK Context Output Events

7 Benefits and Features

7.1 Architectural Benefits

1. **Centralized Control**: IK Context manages all IK operations

2. **TCP Transparency**: Tools work automatically without solver awareness
3. **Dynamic Loading**: Add new solvers without recompiling
4. **Clean Interface**: Standardized solver API for consistency
5. **Better Testing**: Isolated solvers are easier to test
6. **Easy Extension**: Drop in new solver files to extend functionality

7.2 Performance Optimizations

- Reusable object pools in solvers reduce garbage collection
- Efficient event-based communication between contexts
- Cached end effector positions avoid redundant calculations
- Optimized matrix operations using Three.js utilities

8 Example Usage

8.1 Basic IK Controller

```
function IKController() {
  const {
    currentPosition,
    targetPosition,
    moveToTarget,
    isAnimating,
    currentSolver,
    setCurrentSolver,
    availableSolvers
  } = useIK();

  const handleMove = () => {
    // IK automatically uses TCP position if tool attached
    moveToTarget(true); // Animate to target
  };

  return (
    <div className="ik-controller">
      <div className="status">
        <p>Current Position:
          X: {currentPosition.x.toFixed(3)},
          Y: {currentPosition.y.toFixed(3)},
          Z: {currentPosition.z.toFixed(3)}
        </p>
        <p>Active Solver: {currentSolver}</p>
      </div>

      <select
        value={currentSolver}
        onChange={(e) => setCurrentSolver(e.target.value)}
      >
```

```
        {availableSolvers.map(solver => (
          <option key={solver} value={solver}>{solver}</option>
        ))}
      </select>

      <button onClick={handleMove} disabled={isAnimating}>
        Move to Target
      </button>
    </div>
  );
}
```

Listing 6: IK Controller Implementation

8.2 Advanced Configuration

```
function AdvancedIKControl() {
  const {
    configureSolver,
    getSolverSettings,
    currentSolver,
    moveToTarget,
    setTargetOrientation
  } = useIK();

  const settings = getSolverSettings(currentSolver);

  const updateSolverConfig = (key, value) => {
    configureSolver(currentSolver, {
      ...settings,
      [key]: value
    });
  };

  const executeWithOrientation = () => {
    // Set target orientation
    setTargetOrientation({
      roll: 0,
      pitch: Math.PI / 4,  // 45 degrees
      yaw: 0
    });

    // Execute IK with orientation
    moveToTarget(true);
  };

  return (
    <div>
      {Object.entries(settings).map(([key, value]) => (
        <div key={key}>
          <label>{key}:</label>
          <input
            type="number"
            value={value}
```

```
              onChange={(e) => updateSolverConfig(key, parseFloat(e.
                  target.value))}
            />
          </div>
        ))}

        <button onClick={executeWithOrientation}>
          Move with Orientation
        </button>
      </div>
    );
}
```

Listing 7: Advanced IK Configuration