

# Controlling Robot Manipulators

Alexis Maldonado, Andreas Fedrizzi

Lehrstuhl für Informatik IX  
Technische Universität München

Sommersemester 2008

# Outline of the Course

- Introduction
  - Robot-Components
  - Manipulators
  - Configuration Space
- Software
  - Player
  - Gazebo
- Position Analysis
  - Vectors in Space
  - Representation of Rigid Bodies
  - Representation of Frame Transformations
- Software
  - Roboop
  - GLroboop
- Forward and Inverse Kinematics
- Modeling of Manipulators
  - DH Parameters
- Further topics
  - Trajectory Planning
  - Modern Control Methods

# Outline

---

- 1 Inverse Kinematics
- 2 Inverse Kinematics in Roboop

# Outline

---

- 1 Inverse Kinematics
- 2 Inverse Kinematics in Roboop

# Inverse Kinematics

- **Task:** Compute the manipulator's joint angles in configuration space, when a workspace pose of the endeffector is given.
- **Input:** Position and orientation (=pose) of the endeffector.
- **Output:** All joint states of the manipulator.

# Inverse Kinematics

- **Task:** Compute the manipulator's joint angles in configuration space, when a workspace pose of the endeffector is given.
- **Input:** Position and orientation (=pose) of the endeffector.
- **Output:** All joint states of the manipulator.
- **Example:** Inverse kinematics for a cartesian robot
- Joint state of cartesian robot:  $Cart(t_n, t_o, t_a) = \langle l_1, l_2, l_3 \rangle$ .

- **Input:**  $Position_{cart} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}$

- **Solution:**

$$l_1 = p_x$$

$$l_2 = p_y$$

$$l_3 = p_z$$

## Inverse Kinematics for a spherical robot

- Joint state of spherical robot:  $Sph(r_a, r_o, t_a) = \langle \alpha_1, \alpha_2, l_3 \rangle$ .

- Input:  $Position_{sph} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}$

- Solution: I.  $l_3 S\alpha_2 C\alpha_1 = p_x$ ,

- II.  $l_3 S\alpha_2 S\alpha_1 = p_y$ ,

- III.  $l_3 C\alpha_2 = p_z$

$$\frac{\text{II.}}{\text{I.}} \rightarrow \frac{S\alpha_1}{C\alpha_1} = \frac{p_y}{p_x} \rightarrow \alpha_1 = \text{atan2}\left(\frac{p_y}{p_x}\right)$$

$$\frac{\text{I.}}{\text{III.}} \rightarrow \frac{S\alpha_2}{C\alpha_2} = \frac{p_x}{p_z * C\alpha_1} \rightarrow \alpha_2 = \text{atan2}\left(\frac{p_x}{p_z * C\alpha_1}\right)$$

$$\text{III.} \rightarrow l_3 = \frac{p_z}{C\alpha_2}$$

## In which quadrant is our solution angle?

- **Caution:** An angle of  $60^\circ$  leads to a sine of 0.87 and a cosine of 0.5, whereas an angle of  $240^\circ$  leads to a sine of -0.87 and a cosine of -0.5. The fraction  $\frac{p_y}{p_x}$  is 1.74 in both cases.
- But because only one of the angles can be correct, the arctangent function has to choose between the possible solutions. The arctangent function can choose the right angle, if it knows the signs of the sine and the cosine.
- C/C++ provides the function `double atan2(double, double);` which does exactly that.

# Inverse Kinematics for RPY angles

- Joint state of RPY angles:  $RPY(r_a, r_o, r_n) = \langle \alpha_1, \alpha_2, \alpha_3 \rangle$ .
- Solution:

- We **decouple the rotation** around the  $a$ -axis by premultiplying with  $rot(a, \alpha_1)^{-1}$

$$rot(a, \alpha_1)^{-1} \times RPY(\alpha_1, \alpha_2, \alpha_3) = rot(o, \alpha_2) \times rot(n, \alpha_3)$$

- Assuming that we **know the final orientation**, achieved by RPY, we have

$$rot(a, \alpha_1)^{-1} \times \begin{bmatrix} n_x & o_x & a_x & 0 \\ n_y & o_y & a_y & 0 \\ n_z & o_z & a_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = rot(o, \alpha_2) \times rot(n, \alpha_3)$$

- Multiplying the matrices**, we get

$$\begin{bmatrix} n_x C\alpha_1 + n_y S\alpha_1 & o_x C\alpha_1 + o_y S\alpha_1 & a_x C\alpha_1 + a_y S\alpha_1 & 0 \\ n_y C\alpha_1 - n_x S\alpha_1 & o_y C\alpha_1 - o_x S\alpha_1 & a_y C\alpha_1 - a_x S\alpha_1 & 0 \\ n_z & o_z & a_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} C\alpha_2 & S\alpha_2 S\alpha_3 & S\alpha_2 C\alpha_3 & 0 \\ 0 & C\alpha_3 & -S\alpha_3 & 0 \\ -S\alpha_2 & C\alpha_2 S\alpha_3 & C\alpha_2 C\alpha_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## Inverse Kinematics for RPY angles (cont.)

$$\begin{bmatrix} n_x C\alpha_1 + n_y S\alpha_1 & o_x C\alpha_1 + o_y S\alpha_1 & a_x C\alpha_1 + a_y S\alpha_1 & 0 \\ n_y C\alpha_1 - n_x S\alpha_1 & o_y C\alpha_1 - o_x S\alpha_1 & a_y C\alpha_1 - a_x S\alpha_1 & 0 \\ n_z & o_z & a_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} C\alpha_2 & S\alpha_2 S\alpha_3 & S\alpha_2 C\alpha_3 & 0 \\ 0 & C\alpha_3 & -S\alpha_3 & 0 \\ -S\alpha_2 & C\alpha_2 S\alpha_3 & C\alpha_2 C\alpha_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- (2,1) I.  $n_y C\alpha_1 - n_x S\alpha_1 = 0$   
 $\rightarrow \alpha_1 = \text{atan2}\left(\frac{n_y}{n_x}\right)$  or  $\alpha_1 = \text{atan2}\left(\frac{-n_y}{-n_x}\right)$
- (3,1) and (1,1) I.  $-S\alpha_2 = -n_z$ ,  
 II.  $C\alpha_2 = n_x C\alpha_1 + n_y S\alpha_1$   
 $\rightarrow \alpha_2 = \text{atan2}\left(\frac{-n_z}{n_x C\alpha_1 + n_y S\alpha_1}\right)$
- (2,2) and (2,3) I.  $C\alpha_3 = o_y C\alpha_1 - o_x S\alpha_1$ ,  
 II.  $S\alpha_3 = a_x S\alpha_1 - a_y C\alpha_1$   
 $\rightarrow \alpha_3 = \text{atan2}\left(\frac{a_x S\alpha_1 - a_y C\alpha_1}{o_y C\alpha_1 - o_x S\alpha_1}\right)$

# Inverse Kinematics for Euler angles

- The inverse kinematic solution for Euler angles is very similar to that of RPY angles.
- See the book of Saeed B. Niku for further details.

# Outline

---

- 1 Inverse Kinematics
- 2 Inverse Kinematics in Roboop

## How Roboop computes IK queries

- Roboop has **analytical IK solutions** for the manipulators Rhino (100 loc), Puma (160 loc), and Schilling (100 loc).
- For other devices, Roboop uses the **Newton-Raphson** algorithm to compute a solution for an IK query.
- Newton-Raphson performs an **iterative gradient descent**.
- This means that Roboop **does not always converge** to a solution, even if there is one. But Roboop is **more general** than analytical solutions in a way that it can be used for any manipulator where the DH parameters are known.
- The initial pose affects whether the Newton-Raphson algorithm converges for a certain goal pose or not.

## How Roboop changes the angles in an iteration step

- Roboop computes the **Jacobian matrix** for the manipulator.
- In each iteration, Roboop computes the forward kinematics and compares it with the goal pose.
- Subsequently Roboop reassigns joint angles with values that minimize the pose error with respect to the Jacobian matrix.
- As other local search, techniques Newton-Raphson can be trapped in **local minima**, or the **Jacobian matrix can become nearly  $\vec{0}$**  even if the manipulator is dextrous at the location.
- The method is **deterministic**, so the solution angles are always the same if the initial pose and goal pose are identical.
- As stated in a prior lecture the user can choose to use the **Taylor expansion** instead of the Jacobian matrix by setting the function parameter  $m_j$  to 1. The webpage says that Taylor expansion does find a solution more often, but is slower than the default algorithm by a factor of  $\approx 1.8$ .

# Performing an inverse kinematics query

## Listing 1: Roboop example for IK queries

```
// see forward kinematics program in lecture 6 on which headers to include
int main(int args, char** argv)
{
    Matrix solve_this_pose(4,4);
    ColumnVector theta(6);
    ColumnVector angles(6);
    ColumnVector translation(3);
    ColumnVector rotation(3);
    bool converged;
    ... // see forward kinematics program in lecture 6 on how to create rs_T_rg

    theta = rs_T_rg.get_q(); // storing initial theta values

    translation << -0.19 << -0.47 << 0.09;
    rotation << -2.31 << -0.62 << -2.03;
    solve_this_pose = trans(translation) * rpy(rotation);

    angles = rs_T_rg.inv_kin(solve_this_pose, 0, rs_T_rg.get_dof(), converged);
    if (!converged) // check if Roboop found a valid solution
        std::cout << "Did_not_converge." << std::endl;
    else {
        std::cout << "Solution_angles_(roboop-notation)" << angles << std::endl;
        std::cout << "Solution_angles_(pi-actarray)" << angles - theta << std::endl;
    }
}
```

## Opportunities to make Roboop converge more often

- Increase the number of Roboop's **convergence steps** (not that promising; default: NITMAX 1000 in *invkine.cpp*).
- Add a **simulated annealing component** to Roboops algorithm.
- Change between Jacobian matrix and Taylor expansion.
- As stated on a prior slide, the **initial pose affects convergence**, so trying different initial poses might help.
- Build a **loop**, where you set the initial pose of the manipulator with *set\_q(...)* to different initial poses before performing the IK query with *inv\_kin(...)*. The advantage of this opportunity is that you do not have to patch Roboop source code.
- **Which initial poses** could be used?
  - Random poses.
  - Count how often a random pose was a successful initial pose for IK queries and reuse promising poses.
  - Construct promising initial poses by hand.

## Hint for comparing angles in Roboop and GLrooop

- GLrooop labels rpy angles strangely. The Inverse Kinematics panel has the order  $\langle \text{yaw}, \text{pitch}, \text{roll} \rangle$  and calls the rpy function with the values in the order yaw, pitch, roll.
- We learned that the first rpy-transformation is around the a-axis by an angle of yaw, the second rpy-transformation is around the o-axis by an angle of pitch, and the third rpy-transformation is around the n-axis by an angle of roll.
- The rpy-function in Roboop expects a ColumnVector that stores the angles in the order  $\langle \text{roll}, \text{pitch}, \text{yaw} \rangle$ .
- So GLrooop's yaw-angle is the roll-angle in our and Roboop's notation, GLrooop's pitch-angle is the pitch angle, and GLrooop's roll-angle is the yaw-angle.
- **Keep this in mind**, when you compare results obtained by inverse kinematics queries from GLrooop and Roboop, because otherwise the results will be different!