

Arm Tutorial 2

Arm Tutorial 2 project build order

- A) If you have built all the other projects from the previous solutions step by step, you just need to build the projects in Column A.
- B) If you haven't built any of the previous tutorial solutions, you need to build all included projects in the order shown in Column B.

Column A

1. Transformation
2. KUKATutorial2MotionPlanning
3. KUKATutorial2Dashboard

Column B

1. ArticulatedArm (in abstract services)
2. Util
3. SimulatedLBR3Arm
4. KUKAArmTutorialSimulation
5. Transformation
6. KUKATutorial2MotionPlanning
7. KUKATutorial2Dashboard

Moving the end of arm to specific cartesian coordinates with a specific orientation

In arm tutorial 1, the robot was moved by setting target joint angles. Neither did we know the position and orientation of the end-of-arm nor could we move the robot to a desired cartesian coordinate.

In practice it's probably more useful to know about the Cartesian position and Euler angle orientation of the tool center point (TCP) than the knowledge about the joint angles of the robot. Furthermore it makes sense to command the robot via Cartesian coordinate with euler angle parameters instead joint angles. E.g. in practice you want to command 'move the TCP to Cartesian position $(0.3 \ 0.4 \ 0.3)^T$ with orientation $(0^\circ \ 90^\circ \ 0^\circ)^T$ '.

Note:

As you may noticed all robot arms in the **arm tutorials** have no manipulators or tools, that's why we use the notion 'end of arm' (EOA). As the last element of the robot kinematic chain, a mounted manipulator is called end-effector. An end-effector has usually a specific defined point at his end, called tool center point (TCP).

So for an application point of view it makes sense to transform from the joint angles robot arm state to cartesian coordinates with angular orientation and vice versa. Arm tutorial 2 implements a service that provides such transformations.

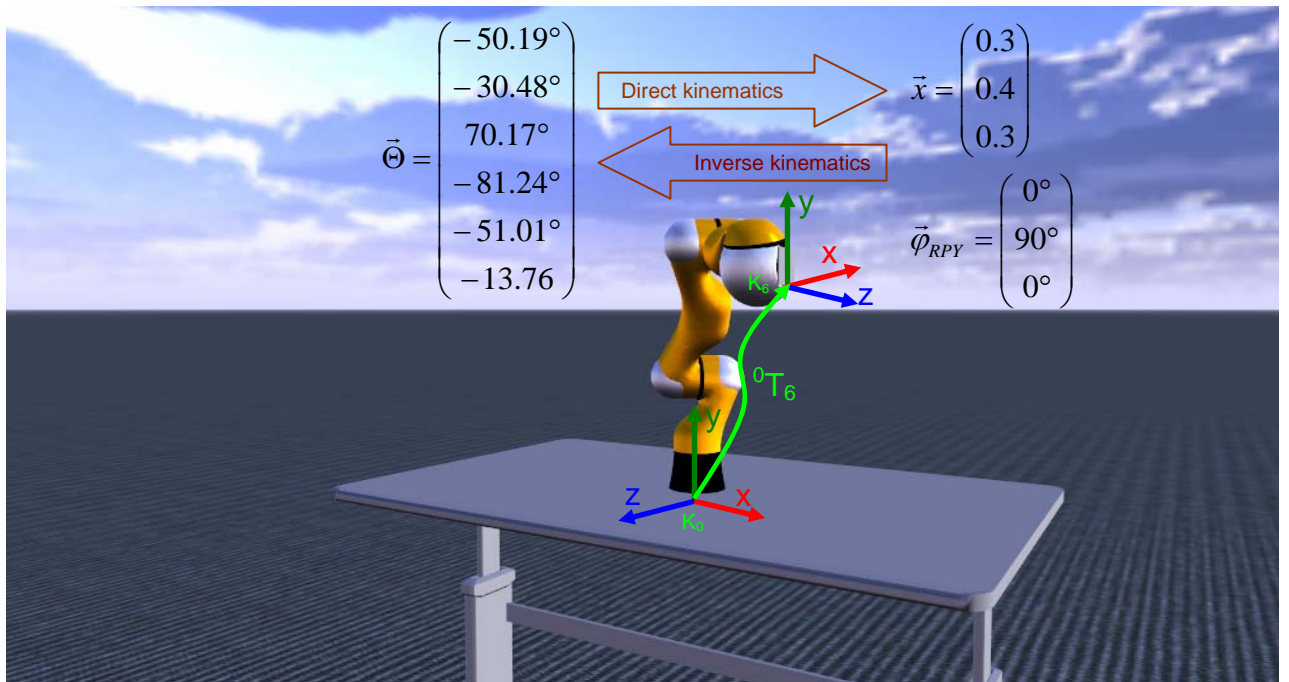
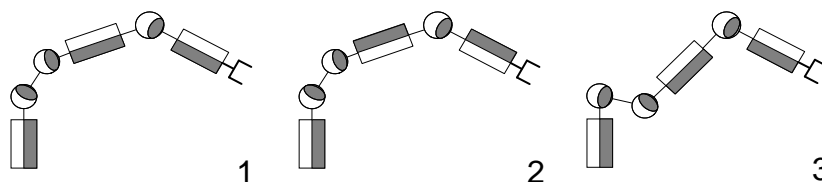


Fig. 1: Direct vs. inverse kinematic transformation

The transformation from the robot joint angles to a cartesian coordinate with angular orientation is called 'forward kinematic transformation' (Fig. 1). Mathematically this transformation is easy. One reason for it being easy is that we have no multiple solutions. We just take the axis values of each axis and calculate from one frame to the next.

The backward transformation from a Cartesian coordinate with angular orientation to the robot joint angle configuration is called 'inverse kinematic transformation' and is more complex. Like indicated in the following picture, multiple arm configurations are possible for a single TCP position and orientation. This impacts the complexity on the application design.

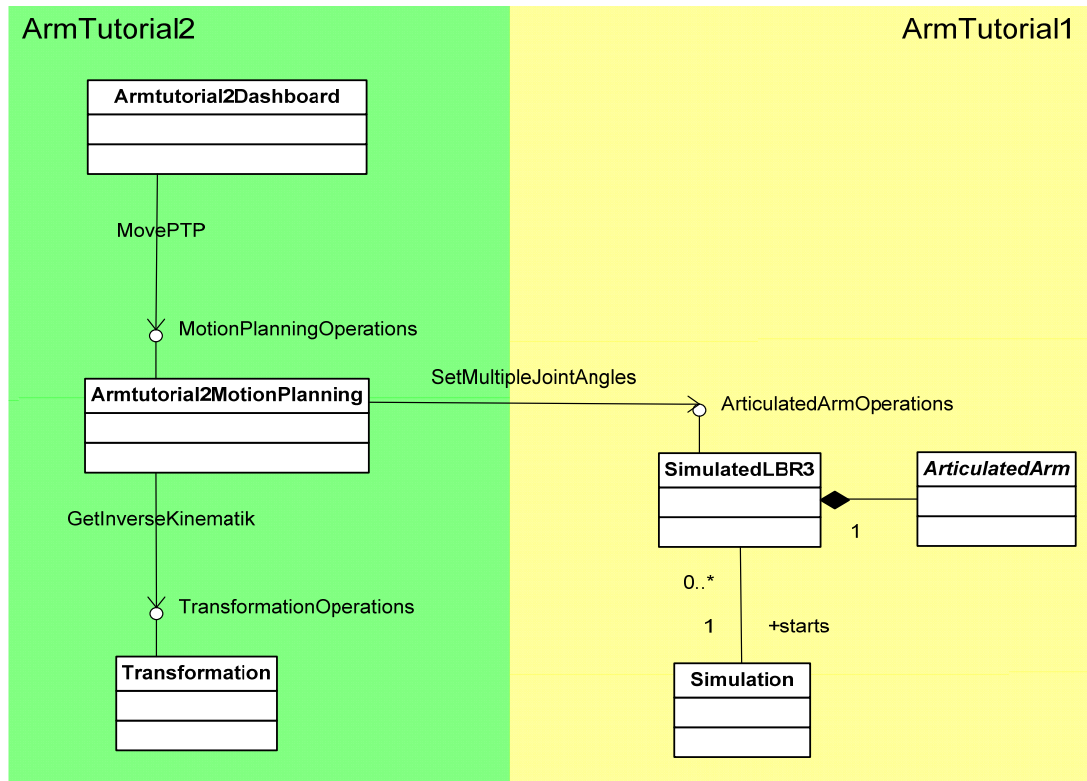


E.g.: A programmer wants to move the TCP to a target position with a specific orientation and the inverse kinematic transformation service returns up to 8 possible solutions. Which solution is the best in terms of collision, minimized joint state changes which implies e.g. minimized travel time?

Coding for Arm Tutorial 2:

In arm tutorial 2, we allow to define a destination for the end-of-arm in terms of a cartesian position. In this case, the motion planning service needs to find out the target axis values. This is done with a call to the newly introduced transformation service.

Arm Tutorial 2 Services Overview:



New services for arm tutorial 2:

- Armtutorial2Dashboard: User Interface with added fields to enter cartesian positions
- Armtutorial2MotionPlanning: Calculates the arm motion and sends axis values to the simulated LBR3
- Transformation: Calculates direct and inverse kinematics

Handling a PTP move with a cartesian destination:

As soon as a 'PTPMove' object with a cartesian destination is processed in the PTPMove Handler of the motion planning service, it will cause a request for an inverse kinematic calculation.

```

/// <summary>
/// Handler for PTP movements
/// </summary>
/// <param name="move">the PTPMove parameter</param>
/// <returns>tasks to perform</returns>
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public virtual IEnumerator<ITask> PTPMove(PTPMove move)
{
    . . .

    transformation.KinematikPose req = new transformation.KinematikPose();
    req.Position = new Vector3(position.X, position.Y, position.Z);
    req.Orientation = new Vector3(orientation.X, orientation.Y, orientation.Z);
    req.Joints = _startJoints;

    yield return Arbiter.Choice(
        _transformationPort.GetInverseKinematik(req),
        delegate(transformation.KinematikJoints resp) { _destinationJoints = resp.Joints; },
        FaultHandler
    );
}

```

This request is sent to the ‘_transformationPort’, which represents the transformation service. This request arrives at the ‘GetInverseKinematikHandler’ of the service.

```

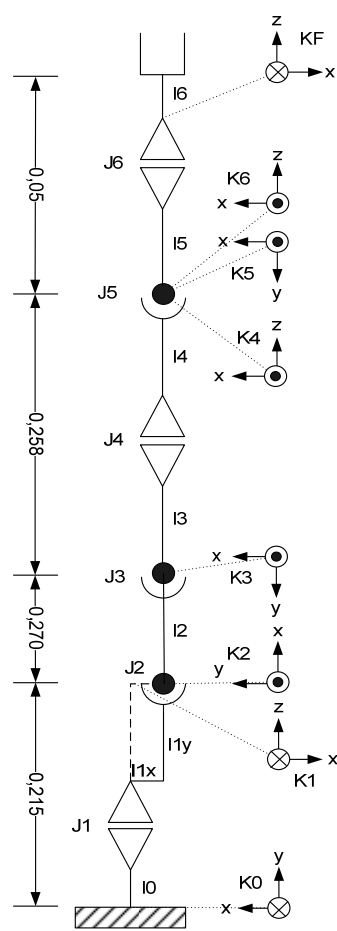
/// <summary>
/// Processes the inverse kinematics
/// (gets the position and orientation of the tool center point (TCP) and calculates the
/// corresponding joint angles)
/// </summary>
/// <param name="req">inverse kinematic parameter</param>
/// <returns></returns>
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public virtual IEnumerator<ITask> GetInverseKinematikHandler(GetInverseKinematik req)
{
}

```

The calculation of the inverse kinematic inside this handler as well as the calculation for the direct kinematic is shown in the following sections. It starts with the calculation of the general transformation matrix for the kinematic chain.

Calculation of the transformation matrix for the kinematic chain (class Matrix, method GetTransformationMatrix)

This method computes the translation matrix frame K_i to frame K_j . To each frame K_i a Denavit-Hartenberg parameter set is assigned to holding values for α_{i-1} , a_{i-1} , θ_i and d_i . The Denavit-Hartenberg parameter sets for all joints in Fig. 2 are listed in Table 1, for their definition refer to (1).



| Joint | Rotation | Link | Joint | Link |
|-------|----------|------|-------|------|
|-------|----------|------|-------|------|

Fig. 2: Robot arm configuration

Using [14] and [15] we get the complete transformation from frame K_i to frame K_j :

| i | α_{i-1} | length a_{i-1} | angle θ_i | offset d_i |
|------------|----------------|------------------|-----------------------|--------------|
| 1 | 0° | $l_{1,x}$ | 0° | $l_{1,y}$ |
| 2 | 90° | $l_{1,x}$ | $\Theta_2 + 90^\circ$ | 0 |
| 3 | 0° | l_2 | $\Theta_3 + 90^\circ$ | 0 |
| 4 | 90° | 0 | Θ_4 | $l_3 + l_4$ |
| 5 | -90° | 0 | Θ_5 | 0 |
| $i \leq j$ | 0° | a_{n-1} | Θ_n | 0 |
| 7 | 0° | 0 | 180° | l_z |

```

public static Matrix GetTransformationMatrix(int lowIndex, int highIndex, float[] theta, float[] alpha, float[] a, float[] d)
{
    Matrix[] matrices = new Matrix[highIndex];
    float[,] values;
    for (int i = lowIndex + 1; i < highIndex + 1; i++)
    {
        values = new float[4, 4];
        values[0, 0] = (float)(Math.Cos(theta[i]));
        values[0, 1] = -(float)(Math.Sin(theta[i]));
        values[0, 2] = 0;
        values[0, 3] = a[i - 1];
        values[1, 0] = (float)(Math.Cos(alpha[i - 1]) * Math.Sin(theta[i]));
        values[1, 1] = (float)(Math.Cos(alpha[i - 1]) * Math.Cos(theta[i]));
        values[1, 2] = -(float)(Math.Sin(alpha[i - 1]));
        values[1, 3] = -(float)(d[i] * Math.Sin(alpha[i - 1]));
        values[2, 0] = (float)(Math.Sin(alpha[i - 1]) * Math.Sin(theta[i]));
        values[2, 1] = (float)(Math.Sin(alpha[i - 1]) * Math.Cos(theta[i]));
        values[2, 2] = (float)(Math.Cos(alpha[i - 1]));
        values[2, 3] = (float)(d[i] * Math.Cos(alpha[i - 1]));
        values[3, 0] = 0;
        values[3, 1] = 0;
        values[3, 2] = 0;
        values[3, 3] = 1;
        matrices[i - (lowIndex + 1)] = new Matrix(values);
    }
}

```

Direct kinematics calculation (file class `dirKinematik`, method `dirKinematik`)

This method computes the position in cartesian coordinates and the orientation in RPY - angles of the TCP.

We start with the calculation of the transformation matrix 0T_7 for the whole kinematic chain that ends with the tool frame K_7 . Through a call of the method `GetTransformationMatrix` we evaluate [14]:

$${}^0T_7 = {}^0D_1(\alpha_0, a_0, \theta_1, d_1) \cdot {}^1D_2(\alpha_1, a_1, \theta_2, d_2) \cdot \dots \cdot {}^6D_7(\alpha_6, a_6, \theta_7, d_7)$$

```
Matrix T6 = Matrix.GetTransformationMatrix(0, 7, theta, state.Alpha, state.A, state.D);
```

The TCP position can be read out directly from matrix 0T_7 :

$$\vec{p} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} t_{14} \\ t_{24} \\ t_{34} \end{pmatrix}$$

```
_x = T6.Values[0, 3];
_y = T6.Values[1, 3];
_z = T6.Values[2, 3];
```

The orientation in roll γ , pitch β and yaw α angles are calculated from the rotational sub matrix of 0T_7 using [9]:

```
Pitch = (float)(Math.Atan2(-T6.Values[2, 0],
    Math.Sqrt(T6.Values[0, 0] * T6.Values[0, 0] + T6.Values[1, 0] * T6.Values[1, 0])));
if (Pitch == (float)(Math.PI / 2))
{
    Roll = 0;
    Yaw = (float)(Math.Atan2(T6.Values[0,1], T6.Values[1,1]));
}
else if (Pitch == -(float)(Math.PI / 2))
{
    Roll = 0;
    Yaw = (float)(-Math.Atan2(T6.Values[0,1], T6.Values[1,1]));
}
else
{
    Roll = (float)(Math.Atan2(T6.Values[2, 1], T6.Values[2, 2]));
    Yaw = (float)(Math.Atan2(T6.Values[1, 0], T6.Values[0, 0]));
}
```

Inverse kinematics calculation through a geometric approach (class `InvKinematik`, method `InvKinematik`)

From a given cartesian position and RPY - orientation of the TCP this method calculates the joint angles for a seven axis robot. For a single TCP position the method computes all eight possible arm configurations, see Fig. 3. For more information about the inverse kinematic calculation refer to (2).

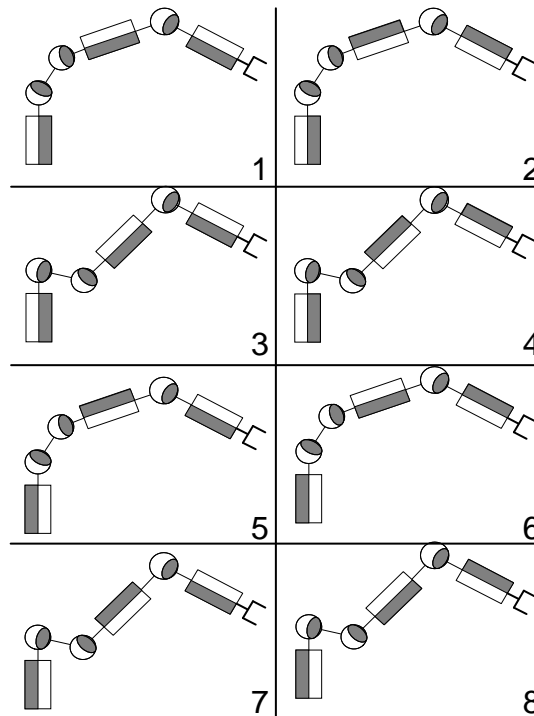


Fig. 3: Possible solutions of the inverse kinematics calculation

Joint 1 angle

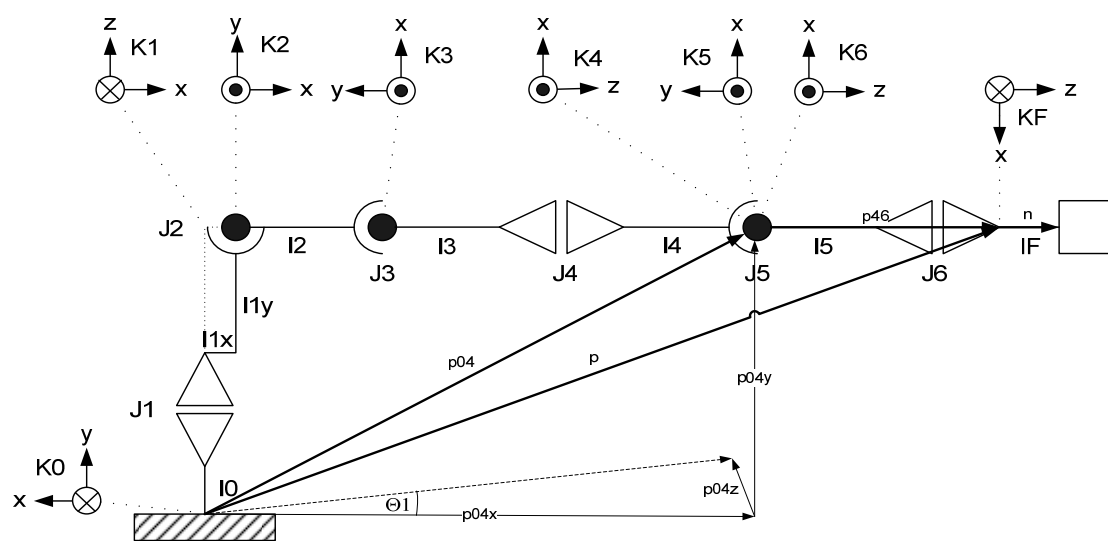


Fig. 4: Joint angle 1

Angle Θ_1 can be calculated from the position projection of frame K_4 on the x-z plane of frame K_0 , see Fig. 4. Hence at first we have to compute the position of K_4 . Twist joints don't change the position but the orientation of its subsequent frame. We use this property to simplify our calculations. In our robot arm we have three twist joints: Joint 1, 4 and 6. As shown in Fig. 4, the angular state of joint 6 influences just the orientation of the TCP, not its position. Through $\vec{p}_{46} = (l_5 + l_F) \cdot \vec{n}$ we get the position vector \vec{p}_{04} . The unit vector \vec{n} points to the z - axis of K_6 is retrieved from the third column of the rotation sub matrix of ${}^0\mathbf{T}_6$:

$$\vec{p}_{04} = \vec{p} - \vec{p}_{46} = \vec{p} - (l_5 + l_F) \cdot \vec{n}$$

```
p04K0[0] = (float)(Math.Round(x - state.D[7] * n[0], 5));
p04K0[1] = (float)(Math.Round(y - state.D[7] * n[1], 5));
p04K0[2] = (float)(Math.Round(z - state.D[7] * n[2], 5));
p04K0[3] = 1;
```

Two solutions Θ_1 and $\Theta_1 + \pi$ yield to the same K_4 position see Fig. 4:

$$\Theta_{1,1} = \Theta_{1,2} = \Theta_{1,3} = \Theta_{1,4} = \arctan2(-p_{04,z}, p_{04,x})$$

$$\Theta_{1,5} = \Theta_{1,6} = \Theta_{1,7} = \Theta_{1,8} = \arctan2(-p_{04,z}, p_{04,x}) + \pi$$

```
_theta[0][0] = _theta[1][0] = _theta[2][0] = _theta[3][0]
= (float)(Math.Atan2(-p04K0[2], p04K0[0]));
_theta[4][0] = _theta[5][0] = _theta[6][0] = _theta[7][0]
= (float)(Math.Atan2(-p04K0[2], p04K0[0]) + Math.PI);
```

Note:

For the sake of clarity we use a simple geometric approach for our inverse kinematic algorithm. Hence we didn't consider all special cases that might result in ambiguous results, e.g. for the $\arctan2(y, x)$ function. An improvement to the algorithm is left for the user as an exercise.

Joint angle 3

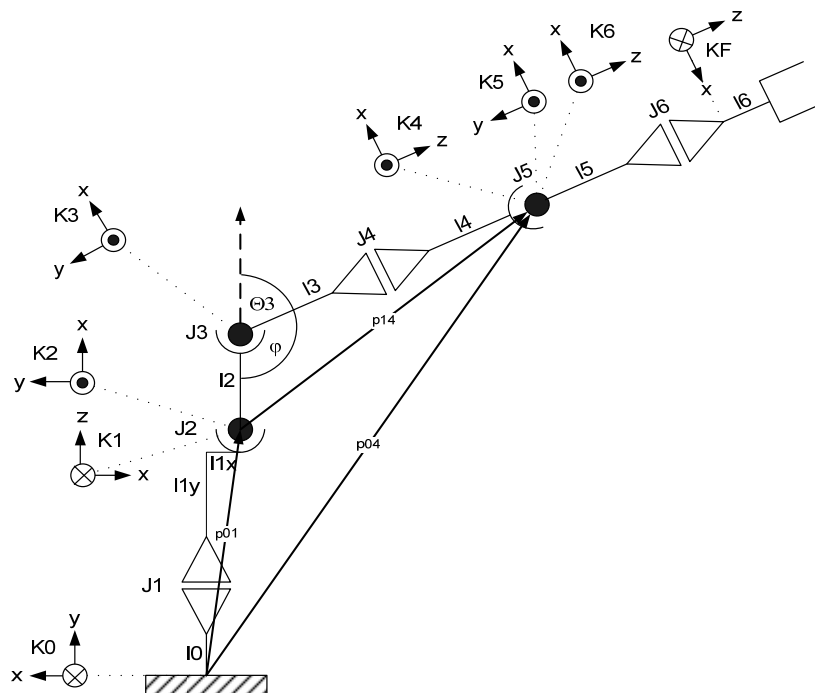


Fig. 5: Joint angle 3

Via the knowledge of \vec{p}_{14} we get angle φ which in turns leads to joint angle Θ_3 . Again we take advantage of a twist joint property. As shown in Fig. 5, the angular state of joint 4 influences the orientation of frame K_4 , not its position. Hence vector \vec{p}_{14} can be calculated. We get the position \vec{p}_{01} from the translation part of ${}^0\mathbf{T}_1$:

$$\vec{p}_{14} = \vec{p}_{04} - \vec{p}_{01}$$

```
Matrix T01 = Matrix.GetTransformationMatrix(0, 1,
new float[] {0, _theta[0][0]},
state.Alpha, state.A, state.D);
float[] p14K0 = new float[4];
p14K0[0] = (float)(p04K0[0] - T01.Values[0, 3]);
p14K0[1] = (float)(p04K0[1] - T01.Values[1, 3]);
p14K0[2] = (float)(p04K0[2] - T01.Values[2, 3]);
p14K0[3] = 1;
```

We compute angle φ through the law of cosine:

$$\varphi = \arccos\left(\frac{l_2^2 + (l_3 + l_4)^2 - |\vec{p}_{14}|^2}{2 \cdot l_2 \cdot (l_3 + l_4)}\right)$$

```
float p14BetQuad = (float)(p14K0[0] * p14K0[0] + p14K0[1] * p14K0[1] + p14K0[2] * p14K0[2]);
float help3 = (float)((state.A[2] * state.A[2]) + (state.D[4] * state.D[4]) - p14BetQuad)
/ (2 * Math.Abs(state.A[2]) * Math.Abs(state.D[4]));
float phi = (float)(Math.Acos(Math.Round(help3, 2)));
```

Finally knowledge of φ offers two possible solutions for Θ_3 :

$$\Theta_{3,1} = \Theta_{3,2} = \pi - \varphi$$

$$\Theta_{3,3} = \Theta_{3,4} = \pi + \varphi$$

$$\Theta_{3,5} = \Theta_{3,6} = -(\pi - \varphi)$$

$$\Theta_{3,7} = \Theta_{3,8} = -(\pi + \varphi)$$

```
_theta[0][2] = _theta[1][2] = (float)(Math.PI - phi);
_theta[4][2] = _theta[5][2] = -_theta[0][2];
_theta[2][2] = _theta[3][2] = (float)(Math.PI + phi);
_theta[6][2] = _theta[7][2] = -_theta[2][2];
```

Joint angle 2:

As shown in Fig. 6, angle Θ_2 can be computed via β_1 and β_2 which in turn we get through the vector \vec{p}_{14} . For an easy calculation of β_1 we want to look at $\vec{p}_{14}^{(1)}$. As seen from K_1 , vector \vec{p}_{14} only has an x and z component. Therefore we transform \vec{p}_{14} from K_0 to K_1 using the inverse of the sub matrix ${}^0\mathbf{R}_1$ of ${}^0\mathbf{T}_1$:

$${}^0\mathbf{T}_1 = {}^0\mathbf{D}_1(\alpha_0, a_0, \theta_1, d_1) = \begin{pmatrix} {}^0\mathbf{R}_1 & \vec{p} \\ \vec{0}^T & 1 \end{pmatrix}$$

$$\vec{p}_{14}^{(1)} = ({}^0\mathbf{R}_1)^{-1} \cdot \vec{p}_{14}$$

Because ${}^0\mathbf{R}_1$ is orthogonal we get its inverse through its transpose:

$$({}^0\mathbf{R}_1)^T \cdot {}^0\mathbf{R}_1 = \mathbf{I} \rightarrow ({}^0\mathbf{R}_1)^{-1} = ({}^0\mathbf{R}_1)^T$$

```

T01.Values[0, 3] = 0;
T01.Values[1, 3] = 0;
T01.Values[2, 3] = 0;
float[] p14K1 = T01.transpose().times(p14K0);

```

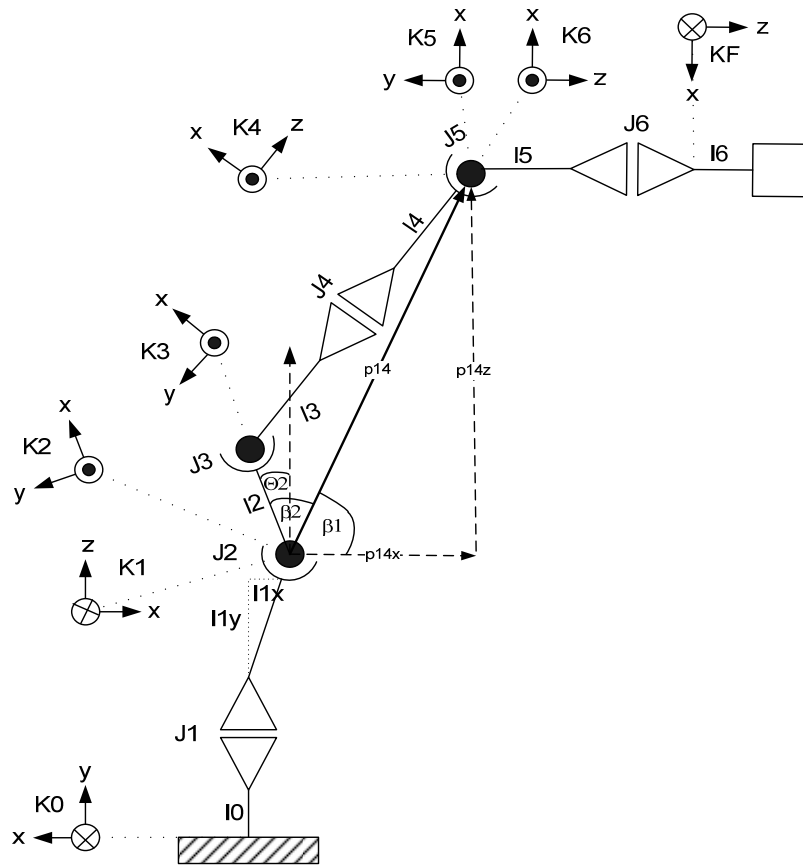


Fig. 6: Joint angle 2

We get β_1 and β_2 through (see Fig. 6):

$$\beta_1 = \arctan(\vec{p}_{14,x}^{(1)}, \vec{p}_{14,z}^{(1)})$$

$$\beta_2 = \arccos\left(\frac{l_2^2 + |\vec{p}_{14}|^2 - (l_3 + l_4)^2}{2 \cdot l_2 \cdot |\vec{p}_{14}|}\right)$$

```

float help2 = (float)((state.A[2] * state.A[2] + p14BetQuad - state.D[4] * state.D[4]) /
(2 * Math.Abs(state.A[2]) * Math.Sqrt(p14BetQuad)));
float beta1 = (float)(Atan2(p14K1[0], p14K1[2]));
float beta2;
...
beta2 = (float)(Math.Acos(help2));

```

Finally all solutions for Θ_2 come out through:

$$\Theta_{2,1} = \Theta_{2,2} = -(\beta_1 + \beta_2)$$

$$\Theta_{2,3} = \Theta_{2,4} = -(\beta_1 - \beta_2)$$

$$\Theta_{2,5} = \Theta_{2,6} = \beta_1 + \beta_2$$

$$\Theta_{2,7} = \Theta_{2,8} = \beta_1 - \beta_2$$

```

Theta[0][1] = _theta[1][1] = -(beta1 + beta2);
Theta[2][1] = _theta[3][1] = -(beta1 - beta2);
Theta[4][1] = Theta[5][1] = -Theta[0][1];
Theta[6][1] = Theta[7][1] = -Theta[2][1];

```

Joint angle 5:

Joint angle 5 can be calculated through the inner product between the unit vectors \vec{n} and \vec{z}_4 (Fig. 7):

$$\vec{z}_4 \cdot \vec{n} = |\vec{z}_4| \cdot |\vec{n}| \cdot \cos(\Theta_5)$$

$$\Theta_5 = \arccos(\vec{z}_4 \cdot \vec{n})$$

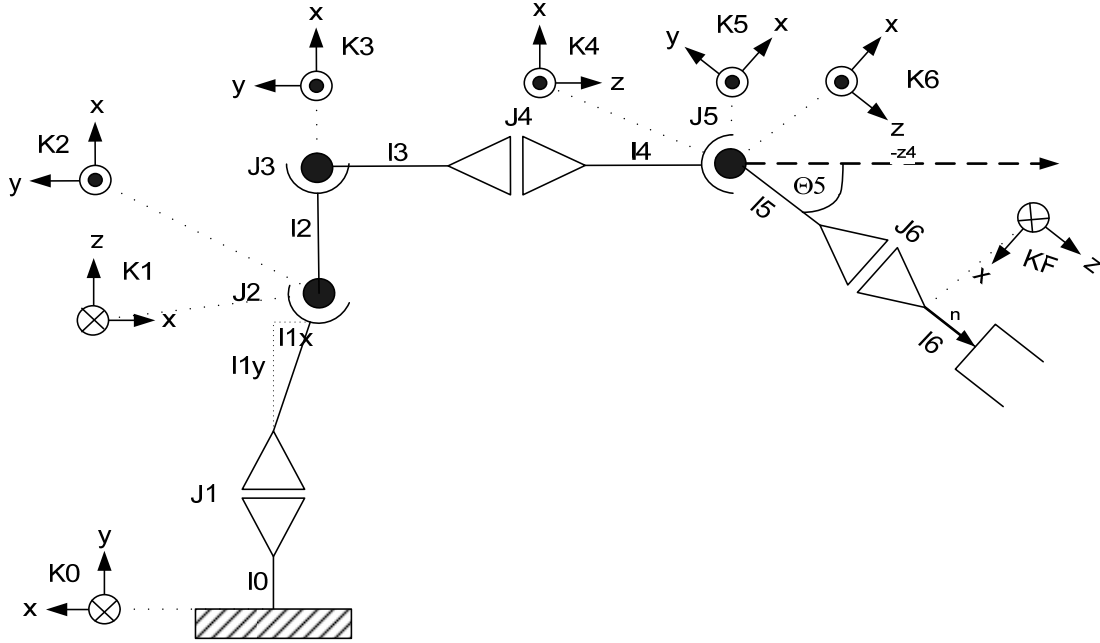


Fig. 7: Joint angle 5

The orientation of vector \vec{z}_4 can be retrieved from the rotation sub matrix of the transformation 0T_4 . Matrix 0T_4 can be calculated via the angle results we already have. Angle Θ_4 doesn't influence the position of \vec{z}_4 therefore we assume $\Theta_4 = 0$. Using the prior results Θ_1, Θ_2 and Θ_3 we evaluate [14] and [15]:

$${}^0T_4 = {}^0D_1(\alpha_0, a_0, \theta_1, d_1) \cdot {}^1D_2(\alpha_1, a_1, \theta_2, d_2) \cdot {}^2D_3(\alpha_2, a_2, \theta_3, d_3) \cdot {}^3D_4(\alpha_3, a_3, \theta_4, d_4)$$

Two arm configurations yield to the same position for frame K_4 , see Fig. 3. Hence we calculate the transformation 0T_4 for both situations resulting in ${}^0T_{4,A}$ and ${}^0T_{4,B}$:

```
Matrix A0_4S1 = Matrix.GetTransformationMatrix(0, 4, new float[] {state.Theta[0],
    _theta[0][0] + state.Theta[1],
    _theta[0][1] + state.Theta[2],
    _theta[0][2] + state.Theta[3],
    state.Theta[4]},
    state.Alpha, state.A, state.D);
Matrix A0_4S2 = Matrix.GetTransformationMatrix(0, 4, new float[] {state.Theta[0],
    _theta[2][0] + state.Theta[1],
    _theta[2][1] + state.Theta[2],
    _theta[2][2] + state.Theta[3],
    state.Theta[4]},
    state.Alpha, state.A, state.D);
```

Now we extract the rotation sub matrix from 0T_4 which column vectors are the base unit vectors of frame 4:

$${}^0\mathbf{T}_4 = \begin{pmatrix} {}^0\mathbf{R}_4 & \vec{p} \\ \vec{0}^T & 1 \end{pmatrix}$$

$${}^0\mathbf{R}_4 = (\vec{x}_4 \ \vec{y}_4 \ \vec{z}_4) = {}^0\mathbf{R}_1(\Theta_1) \cdot {}^1\mathbf{R}_2(\Theta_2) \cdot {}^2\mathbf{R}_3(\Theta_3) \cdot {}^3\mathbf{R}_4(\Theta_4)$$

With the column vector \vec{z}_4 of ${}^0\mathbf{R}_4$ we finally compute all solutions for angle Θ_5 :

$$\Theta_{5,1} = \Theta_{5,5} = \arccos(\vec{z}_{4,A} \cdot \vec{n})$$

$$\Theta_{5,2} = \Theta_{5,6} = -\arccos(\vec{z}_{4,A} \cdot \vec{n})$$

$$\Theta_{5,3} = \Theta_{5,7} = \arccos(\vec{z}_{4,B} \cdot \vec{n})$$

$$\Theta_{5,4} = \Theta_{5,8} = -\arccos(\vec{z}_{4,B} \cdot \vec{n})$$

```
float[] z4K0S1 = new float[] { -A0_4S1.Values[0, 2], -A0_4S1.Values[1, 2], -A0_4S1.Values[2, 2], 0 };
float[] z4K0S2 = new float[] { -A0_4S2.Values[0, 2], -A0_4S2.Values[1, 2], -A0_4S2.Values[2, 2], 0 };
float help5S1 = (float)((z4K0S1[0] * n[0]) + (z4K0S1[1] * n[1]) + (z4K0S1[2] * n[2]));
float help5S2 = (float)((z4K0S2[0] * n[0]) + (z4K0S2[1] * n[1]) + (z4K0S2[2] * n[2]));
Theta[0][4] = (float)Math.Acos(help5S1);
Theta[1][4] = -Theta[0][4];
Theta[2][4] = (float)Math.Acos(help5S2);
Theta[3][4] = -Theta[2][4];
Theta[4][4] = Theta[0][4];
Theta[5][4] = -Theta[4][4];
Theta[6][4] = Theta[2][4];
Theta[7][4] = -Theta[6][4];
```

Joint angles 4 and 6:

The total rotation can be composed as follows:

$${}^0\mathbf{R}_7 = {}^0\mathbf{R}_4 \cdot {}^4\mathbf{R}_7 \Rightarrow {}^4\mathbf{R}_7 = ({}^0\mathbf{R}_4)^{-1} \cdot {}^0\mathbf{R}_7 = {}^4\mathbf{R}_0 \cdot {}^0\mathbf{R}_7$$

Because matrix \mathbf{R} is orthogonal its inverse can also be calculated through its transpose:

$$({}^0\mathbf{R}_4)^T \cdot {}^0\mathbf{R}_4 = \mathbf{I} \Rightarrow {}^4\mathbf{R}_0 = ({}^0\mathbf{R}_4)^T = ({}^0\mathbf{R}_4)^{-1}$$

Matrix ${}^4\mathbf{R}_7$ is composed by three sequential rotations around Z-Y-Z:

$${}^4\mathbf{R}_7 = \text{Rot}_z(\Theta_4) \cdot \text{Rot}_y(\Theta_5) \cdot \text{Rot}_z(\Theta_6)$$

Hence we state:

$${}^4\mathbf{R}_7 = ({}^0\mathbf{R}_4)^T \cdot {}^0\mathbf{R}_7$$

$$= \begin{pmatrix} c\Theta_4 \cdot c\Theta_5 \cdot c\Theta_6 - s\Theta_4 \cdot s\Theta_6 & -c\Theta_4 \cdot c\Theta_5 \cdot s\Theta_6 - s\Theta_4 \cdot c\Theta_6 & c\Theta_4 \cdot s\Theta_5 \\ s\Theta_4 \cdot c\Theta_5 \cdot c\Theta_6 + c\Theta_4 \cdot s\Theta_6 & -s\Theta_4 \cdot c\Theta_5 \cdot s\Theta_6 - c\Theta_4 \cdot c\Theta_6 & s\Theta_4 \cdot s\Theta_5 \\ -s\Theta_5 \cdot c\Theta_6 & s\Theta_5 \cdot s\Theta_6 & c\Theta_5 \end{pmatrix}$$

Again we have to calculate two matrices to retrieve all possible solutions:

$${}^4\mathbf{R}_{7,A} = ({}^0\mathbf{R}_{4,A})^T \cdot {}^0\mathbf{R}_7$$

$${}^4\mathbf{R}_{7,B} = ({}^0\mathbf{R}_{4,B})^T \cdot {}^0\mathbf{R}_7$$

```
Matrix A4_0S1 = A0_4S1.transpose();
Matrix A4_0S2 = A0_4S2.transpose();
Matrix A7_4S1 = A4_0S1.times(rot0_7);
Matrix A7_4S2 = A4_0S2.times(rot0_7);
```

Using the relation $\tan(\alpha) = \sin(\alpha) \cdot \cos^{-1}(\alpha)$ angle Θ_4 can be calculated through the elements r_{23} and r_{13} of ${}^4\mathbf{R}_7$:

$$\Theta_4 = \text{atan2}(\pm r_{23}, \pm r_{13})$$

Now all possible solutions for Θ_4 can be calculated:

$$\begin{aligned}\Theta_{4,1} &= \Theta_{4,6} = \text{atan2}(r_{23,A}, r_{13,A}) \\ \Theta_{4,2} &= \Theta_{4,5} = \text{atan2}(r_{23,A}, r_{13,A}) + \pi \\ \Theta_{4,3} &= \Theta_{4,8} = \text{atan2}(r_{23,B}, r_{13,B}) \\ \Theta_{4,4} &= \Theta_{4,7} = \text{atan2}(r_{23,B}, r_{13,B}) + \pi\end{aligned}$$

```
_theta[0][3] = (float)(Math.Atan2(A7_4S1.Values[1, 2], A7_4S1.Values[0, 2]));
_theta[2][3] = (float)(Math.Atan2(A7_4S2.Values[1, 2], A7_4S2.Values[0, 2]));
_theta[4][3] = (float)(_theta[0][3] + Math.PI);
_theta[6][3] = (float)(_theta[2][3] + Math.PI);
_theta[1][3] = _theta[4][3];
_theta[3][3] = _theta[6][3];
_theta[5][3] = _theta[0][3];
_theta[7][3] = _theta[2][3];
```

The same way we get all solution for Θ_6 through the elements r_{23} and r_{13} of the matrix ${}^4\mathbf{R}_7$:

$$\begin{aligned}\Theta_{6,1} &= \Theta_{6,5} = \text{atan2}(-r_{32,A}, r_{31,A}) \\ \Theta_{6,2} &= \Theta_{6,6} = \text{atan2}(-r_{32,A}, r_{31,A}) + \pi \\ \Theta_{6,3} &= \Theta_{6,7} = \text{atan2}(-r_{32,B}, r_{31,B}) \\ \Theta_{6,4} &= \Theta_{6,8} = \text{atan2}(-r_{32,B}, r_{31,B}) + \pi\end{aligned}$$

```
_theta[0][5] = (float)(-Math.Atan2(-A7_4S1.Values[2, 1], A7_4S1.Values[2, 0]));
_theta[2][5] = (float)(-Math.Atan2(-A7_4S2.Values[2, 1], A7_4S2.Values[2, 0]));
_theta[4][5] = _theta[0][5];
_theta[6][5] = _theta[2][5];
_theta[1][5] = _theta[0][5] + (float)Math.PI;
_theta[3][5] = _theta[2][5] + (float)Math.PI;
_theta[5][5] = _theta[4][5] + (float)Math.PI;
_theta[7][5] = _theta[6][5] + (float)Math.PI;
```

As you can see in Fig. 7, for Θ_4 and Θ_6 unlimited solutions are possible if Θ_5 becomes zero. In that case the mean value of Θ_4 and Θ_6 is assigned to Θ_4 and Θ_6 :

```
for (int i = 0; i < _theta.GetLength(0); i++)
{
    if (Math.Round(Theta[i][4], 2) == 0)
    {
        help46 = (_theta[i][3] + _theta[i][5]) % (float)(2f * Math.PI);
        _theta[i][3] = _theta[i][5] = help46 / 2f;
        if (i % 2 == 1)
        {
            _theta[i][3] = _theta[i][5] += (float)Math.PI;
        }
    }
}
```

Finally all angles are wrapped to the closed interval $[-\pi, \pi]$:

```
for (int i = 0; i < _theta.GetLength(0); i++)
{
    for (int j = 0; j < _theta[i].GetLength(0); j++)
    {
        Theta[i][j] = (float)(Theta[i][j] % (Math.PI * 2));
        if (Theta[i][j] < -(float)Math.PI)
            Theta[i][j] += (float)(2f * Math.PI);
        else if (Theta[i][j] > (float)Math.PI)
            Theta[i][j] -= (float)(2f * Math.PI);
    }
}
```

At last the `GetInverseKinematikHandler` selects one of the eight possible arm configurations. For each possible arm configuration $i, i \in [1,8]$ a quality function value is computed. The calculation takes the prior joint angle states $\tilde{\Theta}_1, \tilde{\Theta}_2, \dots, \tilde{\Theta}_6$ into account:

$$\phi_i = \sum_{n=1}^6 |\Theta_{n,i} - \tilde{\Theta}_n|$$

The arm configuration with the smallest quality function value becomes the new target arm configuration:

```

deviation = 0;
k = 0;
for (int j = 0; j < invKin.Theta[i].Length; j++)
{
    if (!_state.IsUseable[k]) k++;
    jointangle = req.Body.Joints[k] % (float)(Math.PI * 2f);
    if (jointangle > Math.PI)
        jointangle -= (float)(Math.PI * 2f);
    if (jointangle < -Math.PI)
        jointangle += (float)(Math.PI * 2f);
    deviation += Math.Abs(invKin.Theta[i][j] - jointangle);
    k++;
}

if (deviation < min_deviation)
{
    min_deviation = deviation;
    min_deviationID = i;
}

```

Linear coordinate transformation

\vec{p}_i := Basis vectors $\vec{p}_i \in \mathfrak{R}^3$ of the new coordinate system

\mathbf{P} := Transformation matrix $\mathbf{P} \in \mathfrak{R}^{3 \times 3}$

Coordinate transformation matrix:

$$\mathbf{P} = (\vec{p}_1 \quad \vec{p}_2 \quad \vec{p}_3) = \begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{pmatrix} \quad [1]$$

Coordinate transformation:

$$\vec{y} = \mathbf{P} \cdot \vec{x} \quad [2]$$

Orthogonality:

\mathbf{P} is orthogonal \Leftrightarrow

$$\vec{p}_i^T \cdot \vec{p}_j = 0 \quad (i \neq j)$$

$$|\vec{p}_i| = 1$$

$$\mathbf{P}^T \cdot \mathbf{P} = \mathbf{I}$$

\mathbf{P} is orthogonal \Rightarrow

\mathbf{P}^T is orthogonal

$$\mathbf{P}^{-1} = \mathbf{P}^T$$

$$\det(\mathbf{P}) = \pm 1$$

$$|\lambda_i| = 1 \text{ for all eigenvalues } \mathbf{P} \cdot \vec{x} = \lambda_i \cdot \vec{x}$$

[3]

Coordinate system orientation:

Base of \mathbf{P} is a clockwise oriented orthogonal coordinate system $\Leftrightarrow \det(\mathbf{P}) = 1$

[4]

Rotational transformation around axis x:

$$\mathbf{R}_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & c\alpha & -s\alpha \\ 0 & s\alpha & c\alpha \end{pmatrix} \quad [5]$$

Rotational transformation around axis y:

$$\mathbf{R}_y(\alpha) = \begin{pmatrix} c\alpha & 0 & s\alpha \\ 0 & 1 & 0 \\ -s\alpha & 0 & c\alpha \end{pmatrix} \quad [6]$$

Rotational transformation around axis z:

$$\mathbf{R}_z(\alpha) = \begin{pmatrix} c\alpha & -s\alpha & 0 \\ s\alpha & c\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad [7]$$

Roll, pitch and yaw (RPY) angles

RPY - rotation matrix (X, Y', Z" rotation):

$$\mathbf{R}_{rpy}(\gamma, \beta, \alpha) = \mathbf{R}_z(\alpha) \cdot \mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\gamma)$$

$$= \begin{pmatrix} c\alpha \cdot c\beta & c\alpha \cdot s\beta \cdot s\gamma - s\alpha \cdot c\gamma & c\alpha \cdot s\beta \cdot c\gamma + s\alpha \cdot s\gamma \\ s\alpha \cdot c\beta & s\alpha \cdot s\beta \cdot s\gamma + c\alpha \cdot c\gamma & s\alpha \cdot s\beta \cdot c\gamma - c\alpha \cdot s\gamma \\ -s\beta & c\beta \cdot s\gamma & c\beta \cdot c\gamma \end{pmatrix}$$

[8]

RPY angles from rotation a matrix \mathbf{R} :

$$\beta = \text{Atan2}\left(-r_{31}, \sqrt{r_{11}^2 + r_{21}^2}\right)$$

$$\alpha = \begin{cases} 0 & |\beta| = \frac{\pi}{2} \\ \text{Atan2}\left(\frac{r_{21}}{\cos(\beta)}, \frac{r_{11}}{\cos(\beta)}\right) & |\beta| \neq \frac{\pi}{2} \end{cases}$$

$$\gamma = \begin{cases} \frac{\beta}{|\beta|} \text{Atan2}(r_{12}, r_{22}) & |\beta| = \frac{\pi}{2} \\ \text{Atan2}\left(\frac{r_{32}}{\cos(\beta)}, \frac{r_{33}}{\cos(\beta)}\right) & |\beta| \neq \frac{\pi}{2} \end{cases}$$

[9]

Frames

\vec{p} := Translation vector $\vec{p} \in \mathfrak{R}^3$

\mathbf{R} := Rotation matrix $\mathbf{R} \in \mathfrak{R}^{3 \times 3}$

${}^a\mathbf{T}_b$:= Transformation matrix ${}^a\mathbf{T}_b \in \mathfrak{R}^{4 \times 4}$

$\vec{x}^{(a)}$:= Vector in frame K_a : $\vec{x}^{(a)} = \begin{pmatrix} \vec{v}^{(a)} & 1 \end{pmatrix}^T \in \mathfrak{R}^4$

$\vec{x}^{(b)}$:= Vector in frame K_b : $\vec{x}^{(b)} = \begin{pmatrix} \vec{v}^{(b)} & 1 \end{pmatrix}^T \in \mathfrak{R}^4$

Frame rotation ("Orientate \vec{v} via \mathbf{R} "):

$$\text{Rot}(\mathbf{R}) = \begin{pmatrix} \mathbf{R} & \vec{0} \\ \vec{0}^T & 1 \end{pmatrix}$$

[10]

Frame translation ("Translate \vec{v} by \vec{p} "):

$$\text{Trans}(\vec{p}) = \begin{pmatrix} \mathbf{I} & \vec{p} \\ \vec{0}^T & 1 \end{pmatrix}$$

[11]

Transformation matrix from frame K_a to frame K_b ("Orientate \vec{v} via \mathbf{R} and translate it by \vec{p} "):

$${}^a\mathbf{T}_b = \begin{pmatrix} \mathbf{R} & \vec{p} \\ \vec{0}^T & 1 \end{pmatrix}, \quad {}^a\mathbf{T}_b \in \mathfrak{R}^{4 \times 4}$$

[12]

Transformation from frame K_a to frame K_b :

$$\vec{x}^{(b)} = {}^a\mathbf{T}_b \cdot \vec{x}^{(a)}$$

[13]

Chained transformation from frame K_i to frame K_j (note that the multiplication is not commutative and must happen from left to right):

$${}^i\mathbf{T}_j = \prod_{n=i+1}^j {}^{n-1}\mathbf{T}_n, \quad i < j$$

[14]

Denavit-Hartenberg transformation convention

As described in (1), four operations, two rotations and two translations are sufficient to compute a transformation matrix that transforms from frame K_{i-1} to frame K_i within a robot kinematic link chain based on the Denavit-Hartenberg convention:

- Step 1: **Rotation by link twist** α_{i-1} - Rotate frame K_{i-1} around its axis x_{i-1} until z_{i-1} becomes parallel to axis z_i of the next frame K_i .
- Step 2: **Translation by link length** a_{i-1} - From its current position after step 1, move frame K_{i-1} into the direction of axis x_{i-1} until z_{i-1} comes to overlap with axis z_i of the next frame K_i .
- Step 3: **Rotation by joint angle** θ_i - From its current position after step 2, rotate frame K_{i-1} around axis z_i of the next frame K_i until x_{i-1} becomes parallel to axis x_i .
- Step 4: **Translation by link offset** d_i - From its current position after step 3, move frame K_{i-1} into the direction of axis z_i until its origin matches the origin of the next frame K_i .

The total transformation matrix for frame K_{i-1} to frame K_i is calculated through multiplication of the four single transformations. Note that the multiplication is not commutative and must happen from left to right:

$$\begin{aligned} & {}^{i-1}\mathbf{D}_i(\alpha_{i-1}, a_{i-1}, \theta_i, d_i) \\ &= \begin{pmatrix} \mathbf{R}_x(\alpha_{i-1}) & \vec{0} \\ \vec{0}^T & 1 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{I} & (a_{i-1} \ 0 \ 0)^T \\ \vec{0}^T & 1 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{R}_z(\theta_i) & \vec{0} \\ \vec{0}^T & 1 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{I} & (0 \ 0 \ d_i)^T \\ \vec{0}^T & 1 \end{pmatrix} \\ &= \begin{pmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ s\theta_i \cdot c\alpha_{i-1} & c\theta_i \cdot c\alpha_{i-1} & -s\alpha_{i-1} & -s\alpha_{i-1} \cdot d_i \\ s\theta_i \cdot s\alpha_{i-1} & c\theta_i \cdot s\alpha_{i-1} & c\alpha_{i-1} & c\alpha_{i-1} \cdot d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

[15]

Symbols:

$c\alpha$:= $\cos(\alpha)$

$s\alpha$:= $\sin(\alpha)$

\mathbf{M} := Matrix

\mathbf{I} := Unit matrix

m_{rc} := Element row r column c of a matrix \mathbf{M}

\vec{v} := Vector

References

- (1) J. Craig, "Introduction to Robotics: Mechanics and Control", Addison Wesley Longman Publishing Co, 1986
- (2) W. Weber, "Industrieroboter: Methoden der Steuerung und Regelung", Fachbuchverlag Leipzig, 2002