

# Koordinatentransformation

Parameter	Kursinformationen
Veranstaltung:	Softwareprojekt Robotik
Semester	Wintersemester 2022/23
Hochschule:	Technische Universität Freiberg
Inhalte:	Umsetzung von ROS Paketen
Link auf GitHub:	<a href="https://github.com/TUBAF-lfl-LiaScript/VL_Softwareentwicklung/blob/master/10_Koordinatentransformation.md">https://github.com/TUBAF-lfl-LiaScript/VL_Softwareentwicklung/blob/master/10_Koordinatentransformation.md</a>
Autoren	Sebastian Zug & Georg Jäger



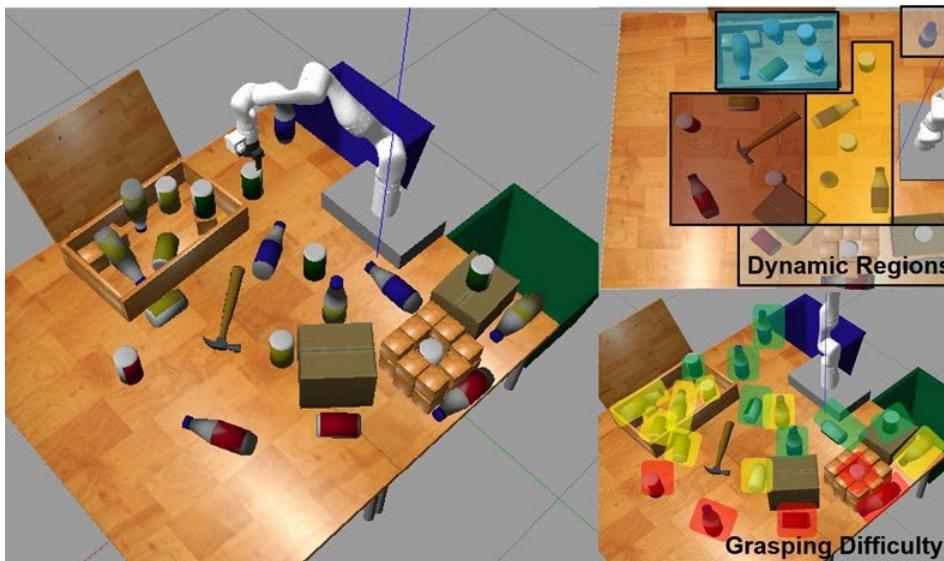
---

## Motivation

Roboter spannen mit ihren Sensoren und Aktoren jeweils eigene Koordinatensysteme auf. Diese dienen als Bezug für die Darstellung von erfassten Hindernissen, intendierten Wegpunkten oder Orientierungen im 2 oder 3D Raum.

Beispiel 1: Zwei Roboter operieren in einem Areal. Einer erkennt ein kritisches Hindernis - wo befindet es sich in Bezug auf den anderen Roboter?

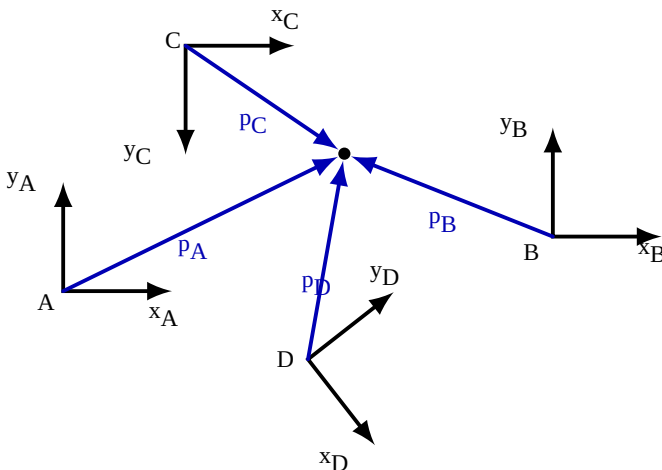
Beispiel 2: Ein stationärer Manipulator erfasst alle Objekte auf der Arbeitsfläche in dem er eine Kamera über diese bewegt. Entsprechend werden alle Objekte im Koordinatensystem der Kamera beschrieben. Für die Planung der Greifoperation müssen wir deren Lage aber auf das Basiskoordinatensystem des Roboters überführen.



Visualisierung der Aufgabenstellung der *Autonomous Robot Manipulation Challenge* beim RoboCup 2022 [\[RoboCup\]](#)

## Mathematische Beschreibung

Entsprechend beziehen sich Punkte als Vektoren  $\mathbf{p} = [x, y]$  im Raum immer auf ein Bezugskoordinatensystem  $A$ , dass bei deren Spezifikation als Index angegeben wird  $\mathbf{p}_A$ .



Darstellung eines Punktes in verschiedenen kartesischen Koordinatensystemen  $A, B, C, D$

# Relevante Transformationen

Aus dem Kontext der *körpererhaltende Transformationen* (im Unterschied zu Scherung und Skalierung) müssen zwei Relationen berücksichtigt werden:

## 1. Translation

Die Darstellung eines Punkte im Koordinatensystem  $A$  kann mit dem Translationsvektor  $t_{A \rightarrow B}$  bestimmt werden.

$$\begin{aligned}\mathbf{p}_A - \mathbf{p}_B &= \mathbf{t}_{A \rightarrow B} \\ \mathbf{p}_B &= \mathbf{p}_A - \mathbf{t}_{A \rightarrow B}\end{aligned}$$

Translation von kartesischen Koordinatensystemen  $A$  und  $B$

## 2. Rotation

Bisher haben wir lediglich Konzepte der translatorischen Transformation betrachtet. Rotationen um den Winkel  $\varphi$  lassen sich folgendermaßen abbilden.

$$\begin{aligned}x_B &= x_A \cos \varphi + y_A \sin \varphi, \\ y_B &= -x_A \sin \varphi + y_A \cos \varphi,\end{aligned}$$

Rotation von kartesischen Koordinatensystemen  $A$  und  $B$

In der Matrizen Schreibweise bedeutet dies

$$\mathbf{p}_B = \begin{bmatrix} \cos \varphi & \sin \varphi \\ -\sin \varphi & \cos \varphi \end{bmatrix}_{A \rightarrow B} \cdot \mathbf{p}_A - \mathbf{t}_{A \rightarrow B}$$

# Homogene Koordinaten

Überlagerung von Translation und Rotation von kartesischen Koordinatensystemen  $A$  und  $B$

Fassen wir nun Translation und Rotation zusammen, so können wir eine 2D Koordinatentransformation mit

$$\mathbf{p}_B = \begin{bmatrix} \cos \varphi & \sin \varphi \\ -\sin \varphi & \cos \varphi \end{bmatrix}_{A \rightarrow B} \cdot \mathbf{p}_A - \mathbf{t}_{A \rightarrow B}$$

beschreiben. Problematisch ist, dass

- die Translation auf einer Addition und
- die Rotation auf der Multiplikation von Matrizen

beruhen.

Homogene Koordinaten lösen dieses Problem durch das Hinzufügen einer weiteren, virtuellen Koordinate. Der Wert der Matrix bleibt dabei unverändert!

### 1. Translation

$$\begin{bmatrix} \mathbf{p}_B \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{T}_{A \rightarrow B}} \cdot \begin{bmatrix} \mathbf{p}_A \\ 1 \end{bmatrix}_A$$

### 2. Rotation

$$\begin{bmatrix} \mathbf{p}_B \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} \cos \varphi & \sin \varphi & 0 \\ -\sin \varphi & \cos \varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{R}_{A \rightarrow B}} \cdot \begin{bmatrix} \mathbf{p}_A \\ 1 \end{bmatrix}_A$$

Beide Transformationsarten sind mit einer mathematischen Operation erklärt. Damit lassen sich auch die Rechenregeln für Matrizen anwenden.

## Inverse Transformation

Die Umkehrung einer Transformation wird über die inverse Matrix von  $\mathbf{R}$  und  $\mathbf{T}$  abgebildet. Dabei bieten die spezifischen Eigenschaften der Matrix Vereinfachungsmöglichkeiten.

### 1. Translation

Die Umkehrung der Translation mit  $\mathbf{t}_{A \rightarrow B}$  kann über eine entgegengesetzte Verschiebung anhand von  $\mathbf{t}_{B \rightarrow A}$  realisiert werden. In homogenen Koordinaten bedeutet das:

$$\begin{aligned} \begin{bmatrix} \mathbf{p}_B \\ 1 \end{bmatrix} &= \mathbf{T}_{A \rightarrow B} \cdot \begin{bmatrix} \mathbf{p}_A \\ 1 \end{bmatrix} \\ \begin{bmatrix} \mathbf{p}_A \\ 1 \end{bmatrix} &= \mathbf{T}_{A \rightarrow B}^{-1} \cdot \begin{bmatrix} \mathbf{p}_B \\ 1 \end{bmatrix} = \mathbf{T}_{B \rightarrow A} \cdot \begin{bmatrix} \mathbf{p}_B \\ 1 \end{bmatrix} \\ \mathbf{T}_{A \rightarrow B} \cdot \mathbf{T}_{A \rightarrow B}^{-1} &= \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} = \mathbf{E} \end{aligned}$$

### 2. Rotation

Analog ergibt sich die Umkehrung der Rotation mit  $\varphi$  kann über eine entgegengesetzte Rotation mit  $-\varphi$  umgesetzt werden. In homogenen Koordinaten gilt:

$$\begin{aligned} \begin{bmatrix} \mathbf{p}_B \\ 1 \end{bmatrix} &= R_{A \rightarrow B} \cdot \begin{bmatrix} \mathbf{p}_A \\ 1 \end{bmatrix} \\ \begin{bmatrix} \mathbf{p}_A \\ 1 \end{bmatrix} &= R_{A \rightarrow B}^{-1} \cdot \begin{bmatrix} \mathbf{p}_B \\ 1 \end{bmatrix} = R_{B \rightarrow A} \cdot \begin{bmatrix} \mathbf{p}_B \\ 1 \end{bmatrix} \\ R_{A \rightarrow B} \cdot R_{A \rightarrow B}^{-1} &= \begin{bmatrix} \cos \varphi & \sin \varphi & 0 \\ -\sin \varphi & \cos \varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos \varphi & -\sin \varphi & 0 \\ \sin \varphi & \cos \varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} = E \end{aligned}$$

Die Berechnung einer inversen Matrix ist nicht nötig.

## Kombination von Transformationen

Sofern sich in dieser Kette weitere Koordinatensysteme wiederfinden können weitere Transformationsmatrizen  $T_{M \rightarrow N}$  oder  $R_{M \rightarrow N}$  integriert werden. Dabei sind sich wiederholende Verschiebungen oder Rotationen als Aggregation zu betrachten.

$$\begin{aligned} \mathbf{p}_D &= \underbrace{T_{A \rightarrow B} T_{B \rightarrow C} T_{C \rightarrow D}}_{T_{A \rightarrow D}} \cdot \mathbf{p}_A \\ \mathbf{p}_D &= \underbrace{R_{A \rightarrow B} R_{B \rightarrow C} R_{C \rightarrow D}}_{R_{A \rightarrow D}} \cdot \mathbf{p}_A \end{aligned}$$

Bedeutsamer ist die Kombination aus beiden Typen - dabei gilt die Aussage  $T \cdot R = R \cdot T$  nicht!

$$\begin{bmatrix} \mathbf{p}_B \\ 1 \end{bmatrix} = R_{A \rightarrow B} \cdot T_{A \rightarrow B} \cdot \begin{bmatrix} \mathbf{p}_A \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \varphi & \sin \varphi & -t_x \\ -\sin \varphi & \cos \varphi & -t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{p}_A \\ 1 \end{bmatrix}$$

## Beispiel Anwendungsfall 2D

Die folgende Grafik stellt zwei Koordinatensysteme ( $A, B$ ) dar. Diese sind in einem globalen Koordinatensystem  $O$  angeordnet. Die Translation zwischen  $O$  und  $A$  sowie  $B$  wird durch den Vektor  $t_A$  und  $t_B$  illustriert.

Die Werte für  $t_{A_x}$ ,  $t_{A_y}$  und  $\varphi_A$  können über die Schieberegler am linken oberen Rand verändert werden.

Der grüne Vektor markiert die gesuchte Größe - die Abbildung des Hindernisses, dass in  $A$  am Punkt (2,3) liegt. Durch die Veränderung der Lage von  $A$  in  $O$  kann dieser Wert aus globaler Sicht verändert werden (vgl.  $F_O$ ).

<https://www.geogebra.org/classic/htug5qmn?embed>

Die nachfolgende Berechnung zeigt, wie die zuvor gezeigten Koordinatentransformationen für die Abbildung des Punktes F genutzt werden können.

```
1 import numpy as np
2 from numpy.linalg import inv
3
4 phi_A = 0 * np.pi /180      # phi in Radiant
5 t_A = [2, 0.5]
6 p_A = [2, 3]                 # Angabe in homogenen Koordinaten
7 R_OB = np.array([[ np.cos(phi_A), -np.sin(phi_A), 0],
8                  [ np.sin(phi_A), np.cos(phi_A), 0],
9                  [ 0, 0, 1]])
10 T_OB = np.array([[ 1, 0, t_A[0]],
11                  [ 0, 1, t_A[1]],
12                  [ 0, 0, 1]])
13
14 # Determine transformation matrix
15 Xi_OB = np.dot(T_OB, R_OB)
16
17 # Determine f_global
```

```
18 p_0= Xi_0B.dot(np.append(p_A, 1))
19 print(p_0[0:2])
```

Aufgabe: Berechnen Sie die die Koordinatenangabe im System  $B$

## 3D Koordinaten

3D Koordinatensystem

Merke: *The Axes display shows a set of axes, located at the origin of the target frame - red - x green - y blue - z*

Die entsprechende translatorische Transformation in homogenen Koordinaten ergibt sich nun mit einer erweiterten Matrix:

$$T_{A \rightarrow B} = \begin{bmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Wie ist es um die Rotationen bestellt?

Nunmehr müssen wir drei Rotationsachsen betrachten.

Bezeichnung	Achse	Transformationsmatrix
Gieren (Hochachse)	z	$R_{A \rightarrow B}^z = \begin{bmatrix} \cos \varphi & \sin \varphi & 0 & 0 \\ -\sin \varphi & \cos \varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Nicken (Querachse)	y	$R_{A \rightarrow B}^y = \begin{bmatrix} \cos \varphi & 0 & \sin \varphi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \varphi & 0 & \cos \varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Rollen (Längsachse)	x	$R_{A \rightarrow B}^x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \varphi & \sin \varphi & 0 \\ 0 & -\sin \varphi & \cos \varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Positiv:

- Intuitiv benutzbar
- Gut geeignet für die Bewegung in Animationen - die virtuelle Kamer lässt sich gut durch die Eulerwinkel interpolieren!

Negativ:

- Gimbal Lock (Unstetigkeit)
- Für allgemeine Rotationen bestimmen sich die Eulerwinkel nicht eindeutig

ROS nutzt für die Darstellung von Rotationen Quaternionen. Diese überwinden die Einschränkungen der Euler-Winkel Darstellung sind aber nicht so anschaulich. Entsprechend stellt die TF Bibliothek Transformationsvorschriften bereit, um zwischen beiden Formaten zu wechseln.



## Umsetzung in ROS

Die Handhabung der unterschiedlichen Koordinatensystem in ROS ist über das `tf`-verteilte System gelöst.

*Darstellung verschiedener Koordinatensysteme innerhalb eines Roboterszenarios (Autor Bo im ROS Forum unter [answers.ros.org](https://answers.ros.org))*

ROS stellt unter dem namen `tf2` mehrere Pakete bereit, um Transformationen zu handhaben. Um die Abbildung von Daten aus einem Koordinatensystem in ein anderes zu realisieren müssen diese in einer Baumstruktur verbunden sein. Ausgehend davon können wir eine Information aus einem *Frame* (Knoten im Baum) in einen anderen überführen.

- eindeutige Zuordnung zu Frames
- mathematische Darstellung der Translations-/Rotationsparameter
- ggf. Kommunikation von Änderungen der Translation-/Rotationsparameter

Grundlage dieser Lösung ist die Integration einer Frame-ID in jeden Datensatz. Jede `sensor_msgs` enthält entsprechend einen header, der folgendermaßen strukturiert ist.

## std\_msgs/Header Message

```
# sequence ID: consecutively increasing ID
uint32 seq
#Two-integer timestamp that is expressed as:
# * stamp.sec: seconds (stamp_secs) since epoch (in Python the variable is called 'secs')
# * stamp.nsec: nanoseconds since stamp_secs (in Python the variable is called 'nsecs')
time stamp
#Frame this data is associated with
string frame_id
```

Eine Transformation kann entweder

- statisch (ändert sich nicht im Laufe der Zeit) oder
- dynamisch (kann sich im Laufe der Zeit ändern, muss es aber nicht)

sein. Die Unterscheidung ist aus Fehlertoleranzgründen wichtig, robuste Systeme müssen wissen, ob ihre Informationen ggf. veraltet sind. Statische Transformationen können einmal gesendet werden und können dann als bekannt vorausgesetzt werden.

## Beispiel 1 Laserscanner

### RunLaserScanner

```
ros2 run urg_node urg_node_driver --ros-args --params-file ./examples
/07_ROS_Pakete/startupHokuyo/config/hokuyo_config.yml
```

```
urg_node:
  ros__parameters:
    serial_port: "/dev/ttyACM0"
    serial_baud: 115200
    laser_frame_id: laser
    angle_min: -2.356194496154785
    angle_max: 2.0923497676849365
    angle_increment: 0.006135923322290182
    scan_time: 0.1
    range_min: 0.02
    range_max: 5.0
    publish_intensity: false
    publish_multiecho: false
```

## StartTransformPublisher

```
ros2 run tf2_ros static_transform_publisher 1 0 0 0 0 0 laser world
[WARN] [1672660636.223769001] []: Old-style arguments are deprecated; see
  for new-style arguments
[INFO] [1672660636.231239054] [static_transform_publisher_VY5dCRuDnDi6Md07]
  Spinning until stopped - publishing transform
translation: ('1.000000', '0.000000', '0.000000')
rotation: ('0.000000', '0.000000', '0.000000', '1.000000')
from 'laser' to 'world'
```

Screenshot aus rviz2 mit aktivierten TF Knotendarstellungen

**Aufgabe** Ermitteln Sie die neue Konfiguration der Argumentenübergabe für die Translations- und Rotations-elemente in ROS Humble!

## Beispiel 2 - Turtle

Dabei werden die Relationen zwischen den Koordinatensystemen über eigene Publisher und Listener ausgetauscht. Am Ende bildet jede Applikation eine Baumstruktur aus den Transformationen zwischen den Frames. Isolierte Frames können entsprechend nicht in Verbindung mit anderen gesetzt werden.

*Beispielhafte Darstellung des tf-Trees eines ROS-Turtle Szenarios, das zwei Turtle umfasst. Die individuellen Posen werden jeweils gegenüber den globalen `world`-Frame beschrieben.*

Nehmen wir nun an, dass wir die Positionsinformation von `turtle2` im Koordinatensystem von `turtle1` darstellen wollen, um zum Beispiel eine Richtungsangabe zu bestimmen. Dafür subscribieren wir uns für deren Frame und berechnen die Koordinate im eigenen Koordinatensystem.

## MessageTfTransform.cpp

```
#include "ros/ros.h"
#include "tf/transform_listener.h"
#include "tf/message_filter.h"
#include "message_filters/subscriber.h"

class PoseDrawer
{
public:
    PoseDrawer() : tf_(), target_frame_("turtle1")
    {
        point_sub_.subscribe(n_, "turtle_point_stamped", 10);
        tf_filter_ = new tf::MessageFilter<geometry_msgs::PointStamped>(point_sub_,
            tf_, target_frame_, 10);
        tf_filter_->registerCallback( boost::bind(&PoseDrawer::msgCallback, this, _1) );
    } ;

private:
    message_filters::Subscriber<geometry_msgs::PointStamped> point_sub_;
    tf::TransformListener tf_;
    tf::MessageFilter<geometry_msgs::PointStamped> * tf_filter_;
    ros::NodeHandle n_;
    std::string target_frame_;

    // Callback to register with tf::MessageFilter to be called when transforms
    // are available
    void msgCallback(const boost::shared_ptr<const geometry_msgs::PointStamped>
        point_ptr)
    {
        geometry_msgs::PointStamped point_out;
        try
        {
            tf_.transformPoint(target_frame_, *point_ptr, point_out);

            printf("point of turtle 2 in frame of turtle 1 Position(x:%f y:%f z:%f)\n",
                point_out.point.x,
                point_out.point.y,
                point_out.point.z);
        }
        catch (tf::TransformException &ex)
        {
            printf ("Failure %s\n", ex.what()); //Print exception which was caught
        }
    };
};

int main(int argc, char ** argv)
{

```

```
ros::init(argc, argv, "pose_drawer"); //Init ROS
PoseDrawer pd; //Construct class
ros::spin(); // Run until interrupted
};
```

```
point of turtle 2 in frame of turtle 1 Position(x:-0.603264 y:4.276489 z:0.
)
point of turtle 2 in frame of turtle 1 Position(x:-0.598923 y:4.291888 z:0.
)
point of turtle 2 in frame of turtle 1 Position(x:-0.594828 y:4.307356 z:0.
)
point of turtle 2 in frame of turtle 1 Position(x:-0.590981 y:4.322886 z:0.
)
```

Dann können wir darauf reagieren und die Position von Schildkröte 1 ansteuern.

## Beispiel 3 - URDF für statische Relationen

Die Konfiguration der Transformationen lässt sich auch abstrakt anhand konkreter Modelle in einer eigenen Sprache darstellen. Die beschreiben

1. geometrische Dimensionen einzelner Komponenten und
2. physikalische Eigenschaften deren
3. Relationen anhand sog. Joints
4. integrierte Sensoren/Aktoren

Das Unified Robot Description Format (URDF) ist eine Variante davon. Auf der Basis eines XML formates können die oben genannten Parameter beschrieben werden. Die Beschreibungssprache XACO erweitert die darstellbaren Inhalte und ermöglicht insbesondere die Referenzierung von Sub-Dokumenten.

```
<?xml version="1.0"?>
<robot name="origins">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
    </visual>
  </link>

  <link name="right_leg">
    <visual>
      <geometry>
        <box size="0.6 0.1 0.2"/>
      </geometry>
      <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
    </visual>
  </link>
</robot>
```

```
    </visual>
  </link>

  <joint name="base_to_right_leg" type="continuous">
    <parent link="base_link"/>
    <child link="right_leg"/>
    <origin xyz="0 -0.22 0.25"/>
  </joint>
</robot>
```

Das zeigt Listing [example.urdf](#) finden Sie im Projektrepository.

```
ros2 launch urdf_tutorial display.launch.py model:=examples
/10_Sensordatenhandling/urdf_example/example.urdf
```

Ein umfangreicheres Modell lässt sich zum Beispiel nach dem Aufruf von

```
ros2 launch urdf_tutorial display.launch.py model:=`ros2 pkg prefix --share
urdf_tutorial`/urdf/06-flexible.urdf
```

evaluieren.