Koordinatentransformation

Parameter	Kursinformationen
Veranstaltung:	`Robotik Projekt`
Semester	`Wintersemester 2024/25`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Koordinatentransformation in Robotikanwendungen`
Link auf GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_SoftwareprojektRobotik/blob/master/05_Koordinatentransformation/05_Koordinatentransformation.md
Autoren	Sebastian Zug & Georg Jäger

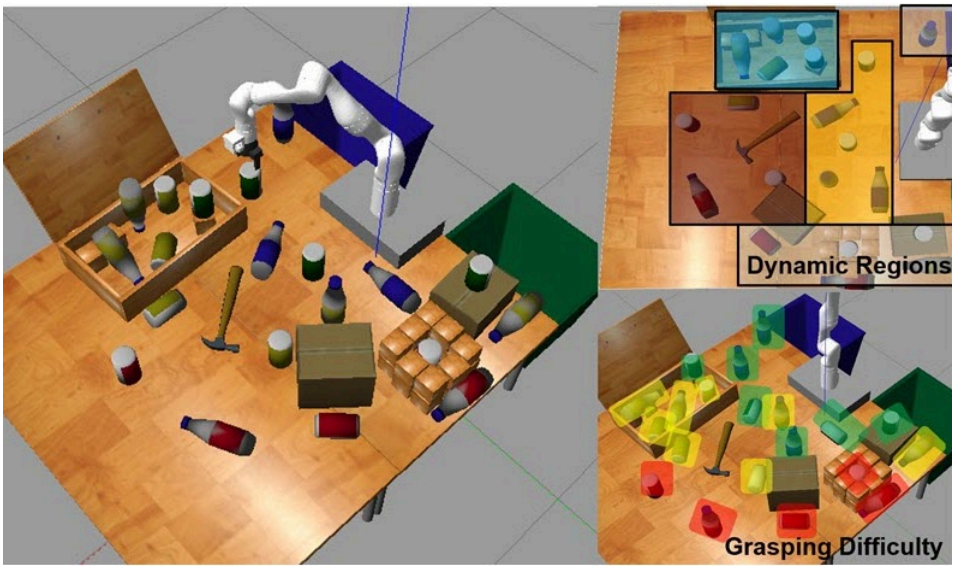
Zielstellung der heutigen Veranstaltung

- Wie kann man die Lage und Orientierung eines Roboters oder Objekts im Raum eindeutig beschreiben?
- Wie rechnet man Koordinaten von Punkten zwischen verschiedenen Bezugssystemen um?
- Was sind Translation, Rotation und homogene Koordinaten – und warum braucht man sie?
- Wie setzt man Transformationsmatrizen auf und verkettet sie?
- Was sind Euler-Winkel, wo liegen ihre Grenzen, und warum nutzt man manchmal Quaternionen?
- Wie werden Koordinatentransformationen in ROS (tf2, URDF) praktisch eingesetzt?

Motivation

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 Grundlagen

Objekte im Raum werden anhand ihrer Position und Orientierung beschrieben, die sich auf ein bestimmtes Koordinatensystem beziehen.

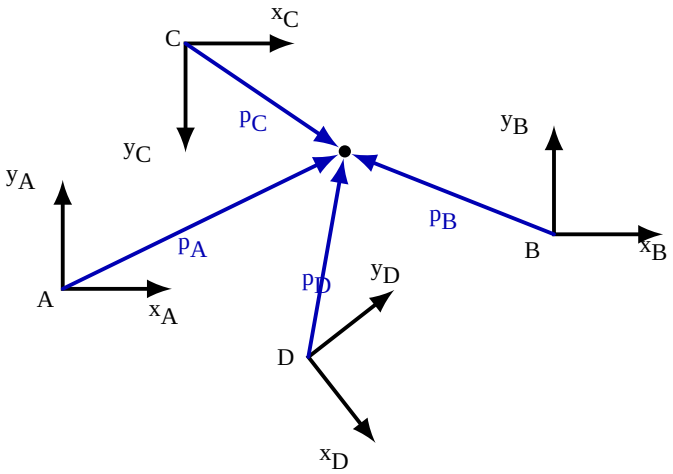
- in der Ebene sind dies kartesische Koordinaten (x, y) und die Orientierung als Winkel φ
- im Raum sind dies kartesische Koordinaten (x, y, z) und die Orientierung als Euler-Winkel (φ, θ, ψ) oder Quaternionen (q_w, q_x, q_y, q_z)

Das Bezugssystem bestimmt dabei den Wert der einzelnen Angaben.

> Transformationen zwischen verschiedenen Koordinatensystemen (Polarkoordinaten, kartesische Koordinaten) werden hier nicht betrachtet.

Mathematische Beschreibung

Entsprechend beziehen sich Punkte als Vekoren $\mathbf{p} = [x, y]$ 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

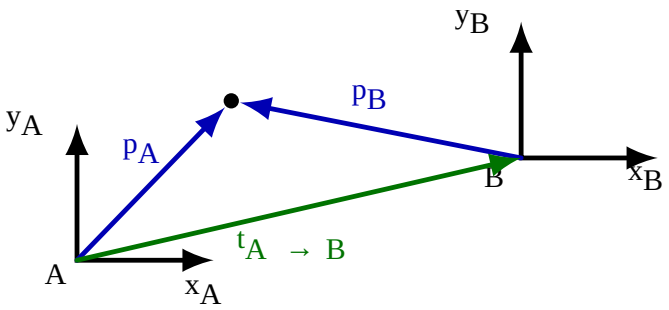
Relevante Transformationen

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

1. Translation

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

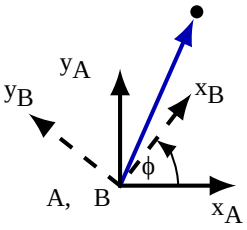
$$\mathbf{p}_A - \mathbf{p}_B = \mathbf{t}_{A \to B}$$
$$\mathbf{p}_B = \mathbf{p}_A - \mathbf{t}_{A \to B}$$



Translation von kartesischen Koordinatensystemen A und B

2. Rotation

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



Rotation von kartesischen Koordinatensystemen A und B

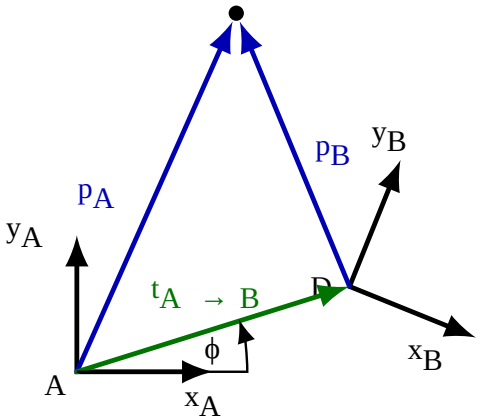
$$x_B = x_A \cos\varphi + y_A \sin\varphi,$$

$$y_B = -x_A \sin\varphi + y_A \cos\varphi,$$

In der Matrizenschreibweise bedeutet dies

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

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

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

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

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

2. Rotation

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

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 R und T abgebildet. Dabei bieten die spezfischen Eigenschaften der Matrix Vereinfachungsmöglichkeiten.

1. Translation

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

$$\begin{bmatrix} \mathbf{p}_B \\ 1 \end{bmatrix} = T_{A\to B} \cdot \begin{bmatrix} \mathbf{p}_A \\ 1 \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{p}_A \\ 1 \end{bmatrix} = T_{A\to B}^{-1} \cdot \begin{bmatrix} \mathbf{p}_B \\ 1 \end{bmatrix} = T_{B\to A} \cdot \begin{bmatrix} \mathbf{p}_B \\ 1 \end{bmatrix}$$

$$T_{A\to B}\cdot T_{A\to B}^{-1}=\begin{bmatrix}1&0&-t_x\\0&1&-t_y\\0&0&1\end{bmatrix}\cdot\begin{bmatrix}1&0&t_x\\0&1&t_y\\0&0&1\end{bmatrix}=E$$

2. Rotation

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

$$\begin{bmatrix}\mathbf{p}_B\\1\end{bmatrix}=R_{A\to B}\cdot\begin{bmatrix}\mathbf{p}_A\\1\end{bmatrix}$$
$$\begin{bmatrix}\mathbf{p}_A\\1\end{bmatrix}=R_{A\to B}^{-1}\cdot\begin{bmatrix}\mathbf{p}_B\\1\end{bmatrix}=R_{B\to A}\cdot\begin{bmatrix}\mathbf{p}_B\\1\end{bmatrix}$$

$$R_{A\to B}\cdot R_{A\to B}^{-1}=\begin{bmatrix}\cos\varphi&\sin\varphi&0\\-\sin\varphi&\cos\varphi&0\\0&0&1\end{bmatrix}\cdot\begin{bmatrix}\cos\varphi&-\sin\varphi&0\\\sin\varphi&\cos\varphi&0\\0&0&1\end{bmatrix}=E$$

> Die Berechnung einer inversen Matrix ist nicht nötig.

Kombination von Transformationen

Transformationen werden in der Robotik oft verkettet, um z. B. von Weltkoordinaten über Roboter- und Sensor-Frames bis zum Messpunkt zu gelangen.

```
world --> base_link --> camera --> sensor
```

Praxisbeispiel: Angenommen, ein Punkt ist im Sensor-Frame gegeben. Um seine Koordinaten im Welt-Frame zu berechnen, multipliziert man alle Transformationen entlang des Pfades:

$$\mathbf{p}_{world}=T_{base\to world}\cdot T_{camera\to base}\cdot T_{sensor\to camera}\cdot\mathbf{p}_{sensor}$$

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

$$\mathbf{p}_D=\underbrace{\boldsymbol{T}_{A\to B}\boldsymbol{T}_{B\to C}\boldsymbol{T}_{C\to D}}_{\boldsymbol{T}_{A\to D}}\cdot\mathbf{p}_A$$

$$\mathbf{p}_D=\underbrace{\boldsymbol{R}_{A\to B}\boldsymbol{R}_{B\to C}\boldsymbol{R}_{C\to D}}_{\boldsymbol{R}_{A\to D}}\cdot\mathbf{p}_A$$

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

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

Typische Fehlerquellen:

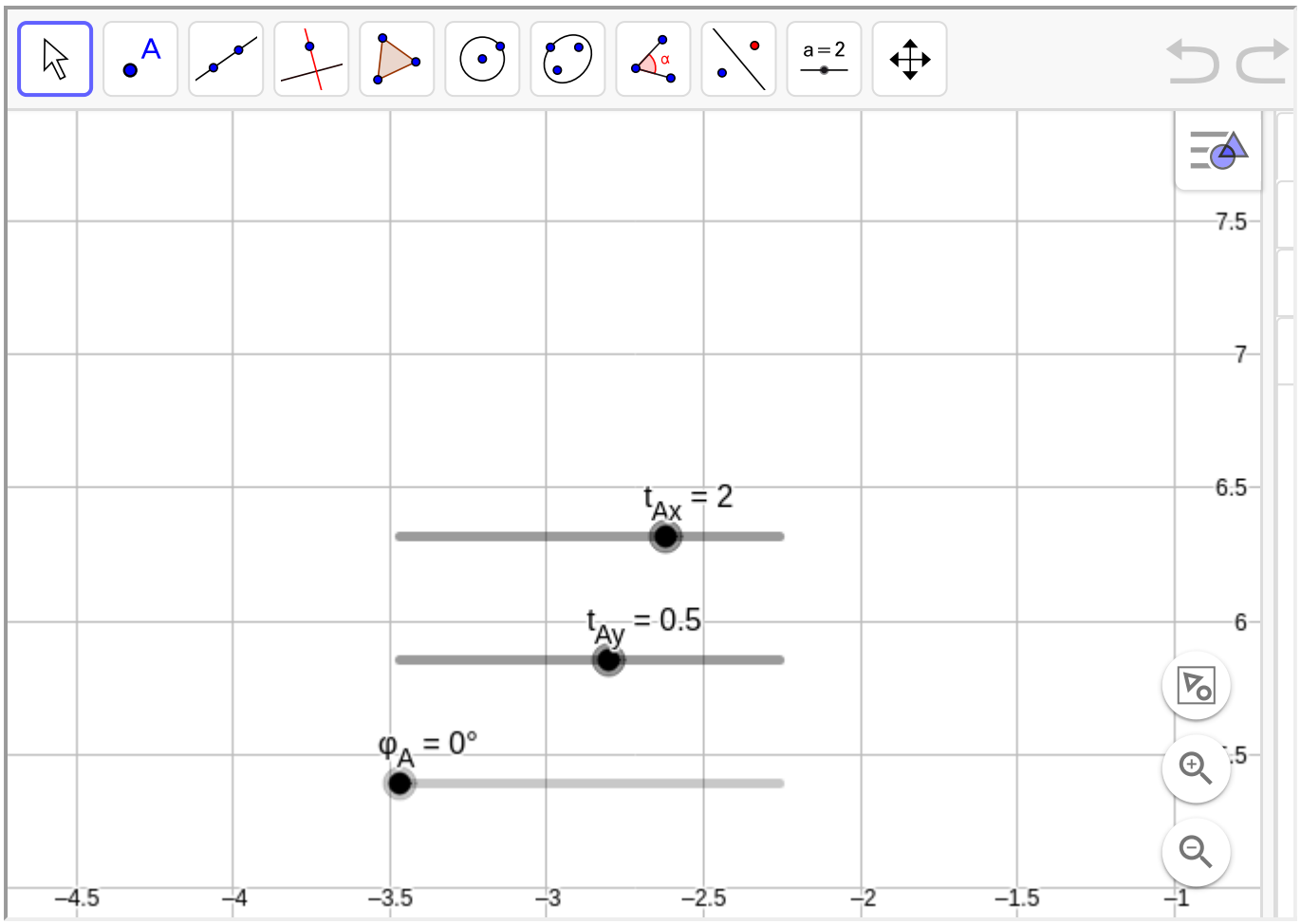
- Reihenfolge der Matrizen vertauscht
- Falsche Inverse verwendet
- Verwechslung von Frame- und Punkt-Notation (In welcher Richtung wird transformiert?)

Beispiel Anwendungsfall 2D

Die folgende Grafik stellt zwei Koordinatensysteme (A, B) dar. Diese sind in einem globalen Koordinatensystem O angeordnet. Die Translation zwisch 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 φ_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.

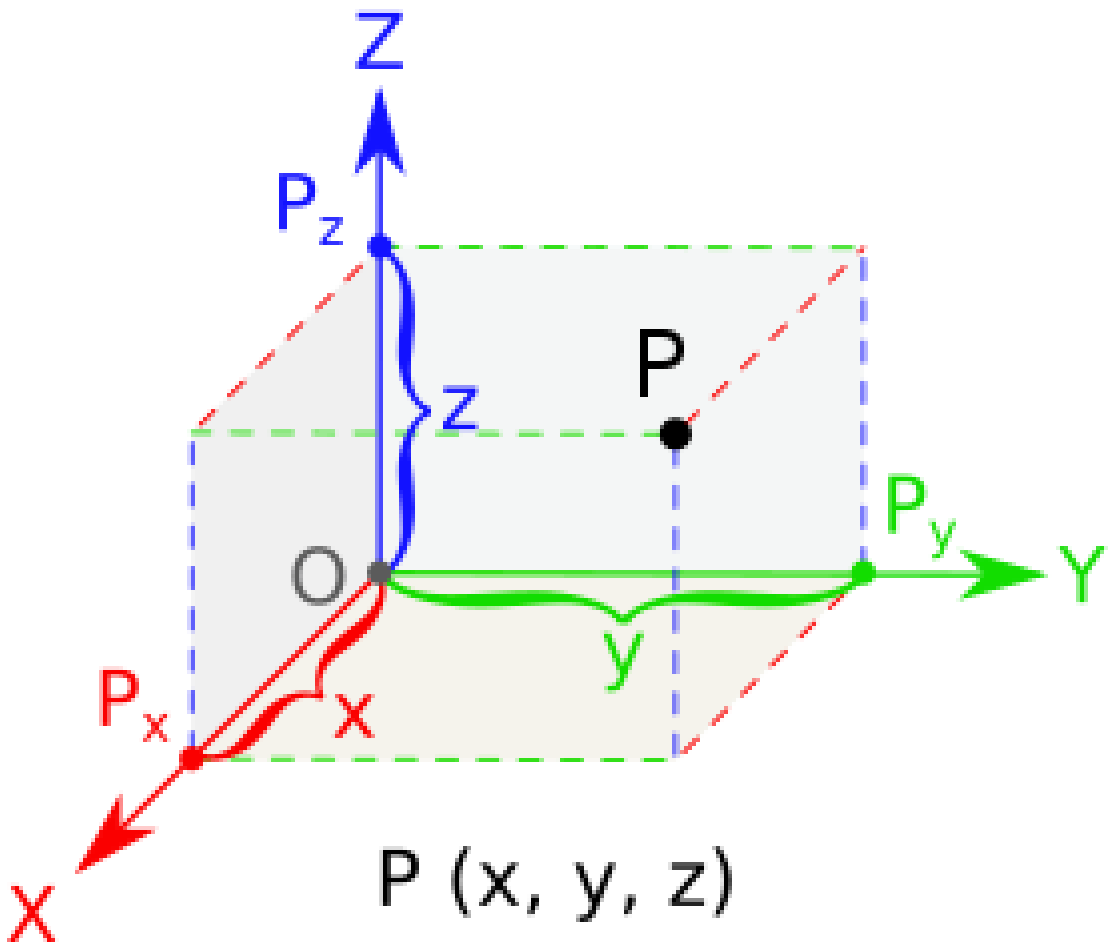
Koordinatentransformation.py

```python
import numpy as np
from numpy.linalg import inv

phi_A = 0 * np.pi /180      # phi in Radiant
t_A = [2, 0.5]
p_A = [2, 3]              # Angabe in homogenen Koordinaten
R_OB = np.array([[ np.cos(phi_A), -np.sin(phi_A),     0],
                 [ np.sin(phi_A), np.cos(phi_A),      0],
                 [             0,             0 ,     1]])
T_OB = np.array([[            1,             0,  t_A[0]],
                 [            0,             1,  t_A[1]],
                 [            0,             0,      1]])

# Determine transformation matrix
Xi_OB = np.dot(T_OB, R_OB)

# Determine f_global
p_O= Xi_OB.dot(np.append(p_A, 1))
print(p_O[0:2])
```

```
Waking up execution server ...
This may take up to 30 seconds ...
Please be patient ...
.......................
```

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 __*

Homogene Koordinaten in 3D

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

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

Wie ist es um die Rotationen bestellt?

Nunmehr müssen wir drei Rotationsachsen betrachten.

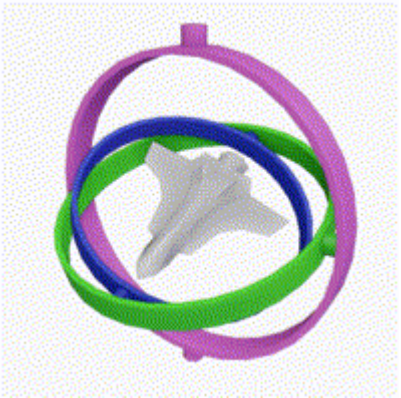
Bezeichnung	**Achse**	**Transformationsmatrix**
Gieren (Hochachse)	z	$R^z_{A\to B} = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 & 0 \\ -\sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Nicken (Querachse)	y	$R^y_{A\to B} = \begin{bmatrix} \cos\varphi & 0 & \sin\varphi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

Positiv:

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

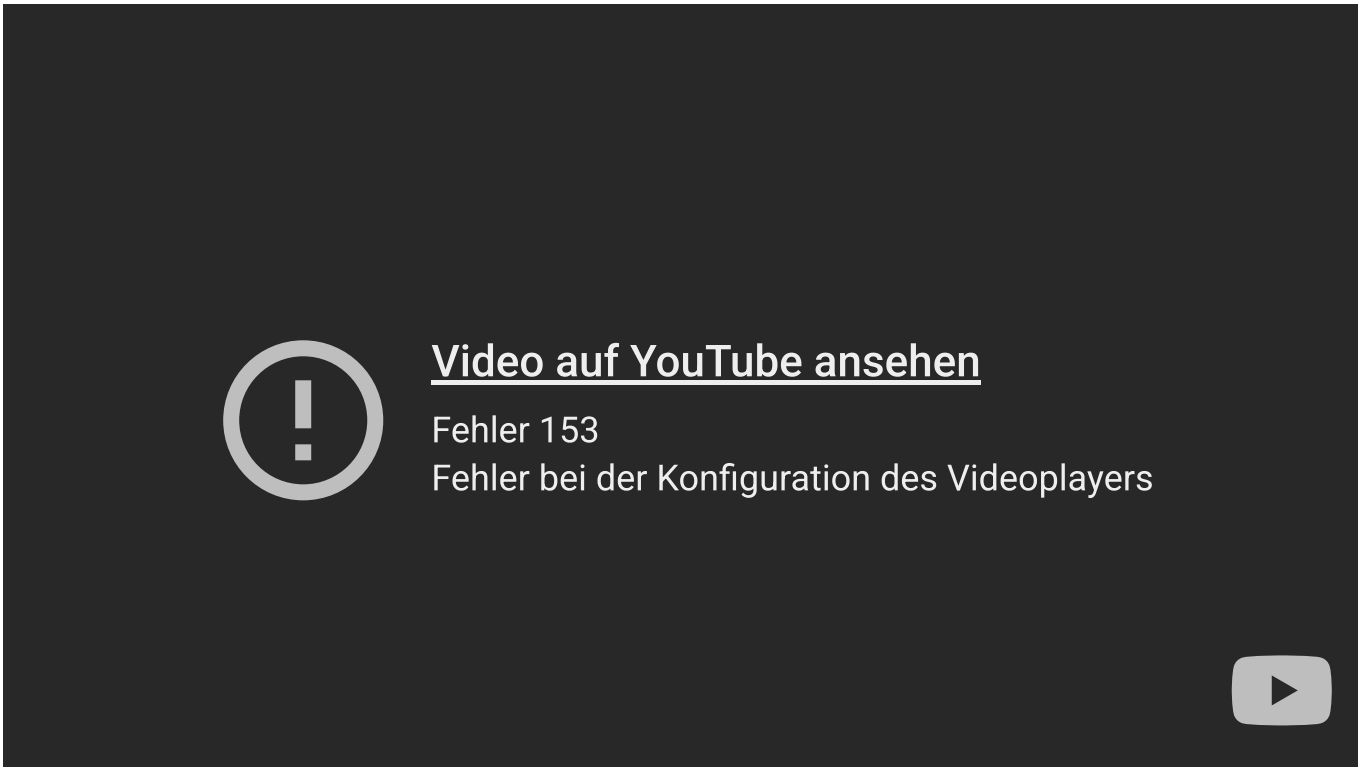
Negativ:

- Gimbal Lock (Unstetigkeit)



- Die Eulerwinkel sind nicht injektiv: Mehrere Winkeltriple können zur gleichen Orientierung führen.

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.



Quaternionen

Motivation: Euler-Winkel sind anschaulich, aber sie haben Schwächen: Gimbal Lock, Mehrdeutigkeit und numerische Instabilität. In der Robotik und Computergrafik werden daher oft **Quaternionen** verwendet, um Rotationen im Raum zu beschreiben.

Ein Quaternion ist eine Erweiterung komplexer Zahlen auf vier Dimensionen:

$$q = q_w + q_x i + q_y j + q_z k$$

Oft wird q als Vektor geschrieben:

$$q = \begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix}$$

Eigenschaften:

- q_w ist der Skalaranteil, q_x, q_y, q_z der Vektoranteil.
- Ein Quaternion beschreibt eine Drehung um eine Achse $\vec{u}$ mit Winkel θ:

$$q = \begin{bmatrix} \cos(\theta/2) \\ u_x \sin(\theta/2) \\ u_y \sin(\theta/2) \\ u_z \sin(\theta/2) \end{bmatrix}$$

- Keine Singularitäten (kein Gimbal Lock)
- Eindeutige, stabile Repräsentation
- Effiziente Interpolation (SLERP)
- Kompakte Speicherung (4 Werte statt 9 für Rotationsmatrix)

In ROS und vielen Bibliotheken gibt es fertige Funktionen für die Umrechnung. Prinzipiell gilt:

Von Quaternion zu Rotationsmatrix:

$$R(q) = \begin{bmatrix} 1 - 2(q_y^2 + q_z^2) & 2(q_x q_y - q_z q_w) & 2(q_x q_z + q_y q_w) \\ 2(q_x q_y + q_z q_w) & 1 - 2(q_x^2 + q_z^2) & 2(q_y q_z - q_x q_w) \\ 2(q_x q_z - q_y q_w) & 2(q_y q_z + q_x q_w) & 1 - 2(q_x^2 + q_y^2) \end{bmatrix}$$

Von Rotationsachse $\vec{u}$ und Winkel θ zu Quaternion:

$$q = \begin{bmatrix} \cos(\theta/2) \\ u_x \sin(\theta/2) \\ u_y \sin(\theta/2) \\ u_z \sin(\theta/2) \end{bmatrix}$$

Eine Drehung um φ um die z-Achse:

$$q = \begin{bmatrix} \cos(\varphi/2) \\ 0 \\ 0 \\ \sin(\varphi/2) \end{bmatrix}$$

Quaternionen sind die Standardrepräsentation für Rotationen in ROS, Computergrafik und moderner Robotik. Sie vermeiden die Probleme der Euler-Winkel und sind für Interpolation und numerische Berechnungen optimal geeignet.

Umsetzung in ROS

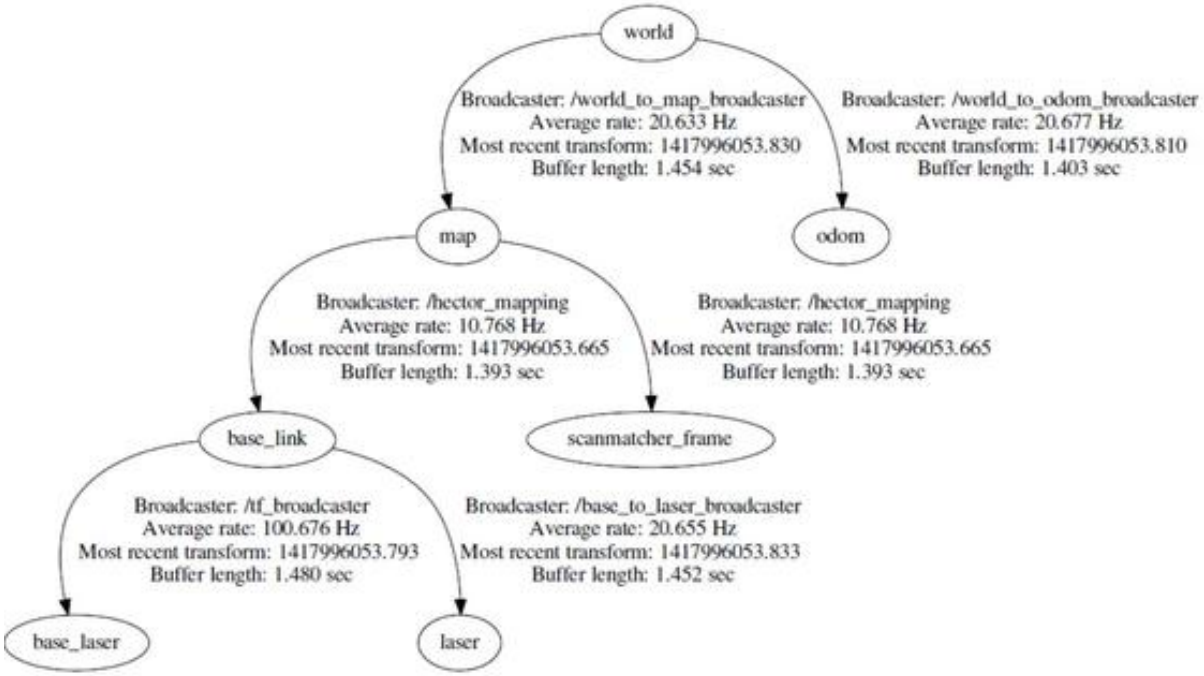
Die Handhabung der unterschiedlichen Koordinatensystem in ROS ist über das `tf`-verteitle System gelöst.

 *RoboterSystem*

Darstellung verschiedener Koordinatensysteme innerhalb eines Roboterszenarios (Autor Bo im ROS Forum unter 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 realsieren 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
  'secs')
# * stamp.nsec: nanoseconds since stamp_secs (in Python the variable is cal
  '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

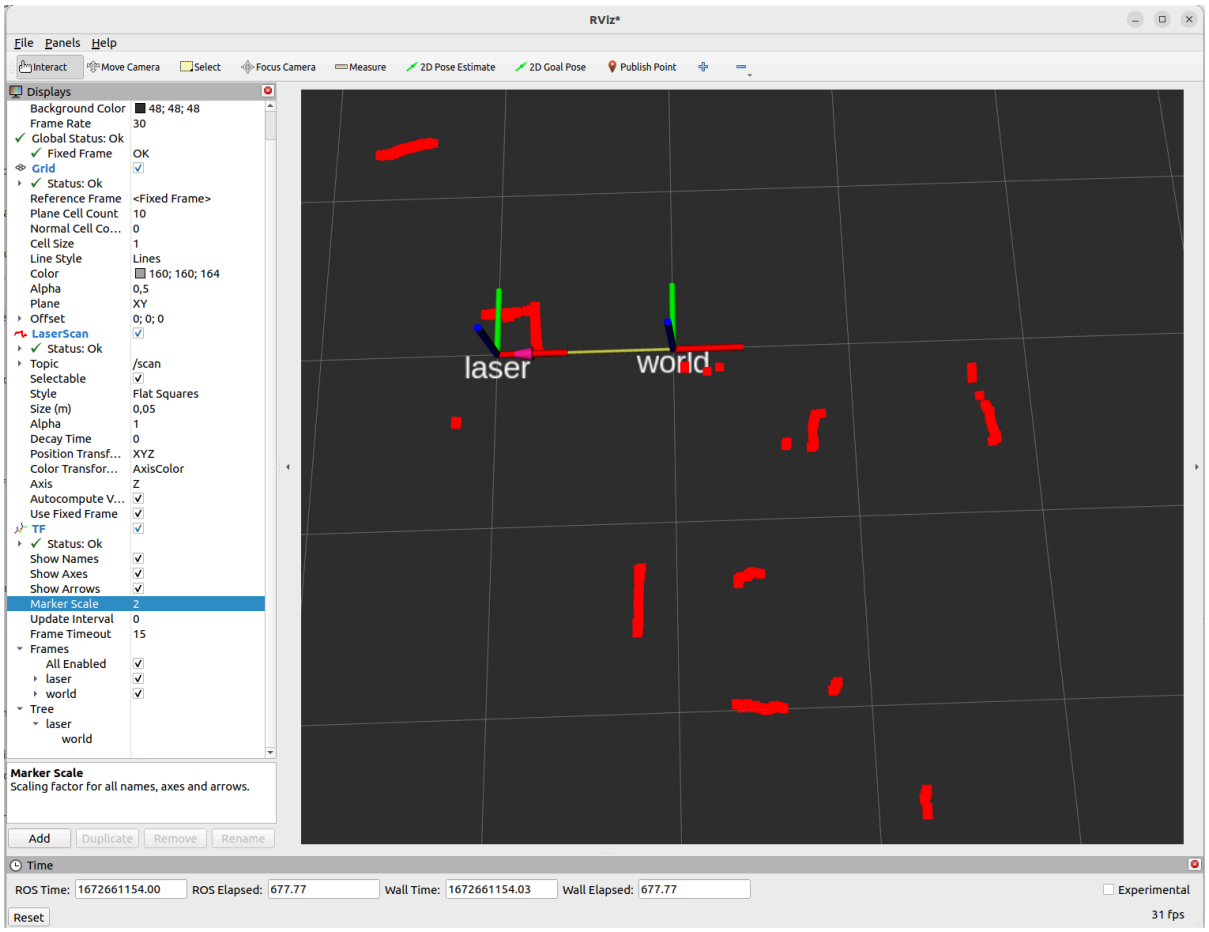
RunLaserScanner

```
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'
```

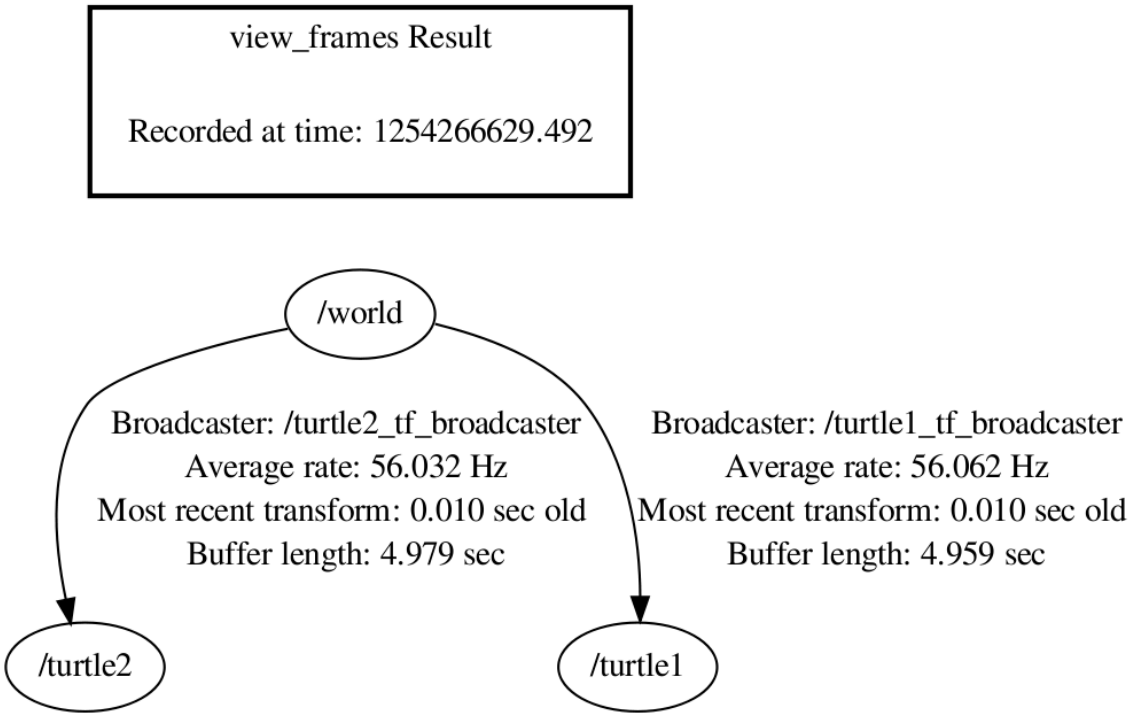


Screenshot aus rviz2 mit aktivierten TF Knotendarstellungen

> **Aufgabe** Ermitteln Sie die neue Konfiguration der Argumentenübergabe für die Translations- und Rotationselemente 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 Koordindatensystem von `turtle1` darstellen wollen, um zum Beispiel eine Richtungsangabe zu bestimmen. Dafür subskribieren 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_s
      tf_, target_frame_, 10);
    tf_filter_->registerCallback( boost::bind(&PoseDrawer::msgCallback, thi
      );
  } ;

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 transfo
    are available
  void msgCallback(const boost::shared_ptr<const geometry_msgs::PointStampe
    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:%
               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 caugh
    }
  };
};

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

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

```
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>

  <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 gezeigt 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.