Hochschule Darmstadt

Lehrveranstaltung: Simulation von Robotersystemen
Prof. Dr. Thomas Horsch
Robotermodellierung und kinematische Transformationen
Datum: 29.4.2016 und 13.5.2016

Name	Matrikelnummer
Fabian Alexander Wilms	735162

Studiengang: Mechatronik
Abgabedatum: 20.5.2016

Testat	

Inhaltsverzeichnis

1 Stabmodell

Der erste Schritt auf dem Weg zu einer realistischen Simulation des Roboters Kuka KR3 in EasyRob ist das Stabmodell. Dafür müssen die Transformationsmatrizen zwischen den einzelnen Gelenkkoordinatensystemen aufgestellt werden. Anschließend kann die kinematische Struktur in EasyRob via "Robotics > cRobot Kinematics > Kinematics Data" eingegeben werden. Es muss die Anzahl aktiver Gelenke angegeben werden, ob es sich jeweils um ein Rotations- oder Translationsgelenk handelt und auf welche Achse sich die Gelenkvariable bezieht. Schließlich werden die Rotationen und Translationen zwischen den Koordinatensystemen eingegeben. Das Ergebnis ist das folgende Stabmodell:

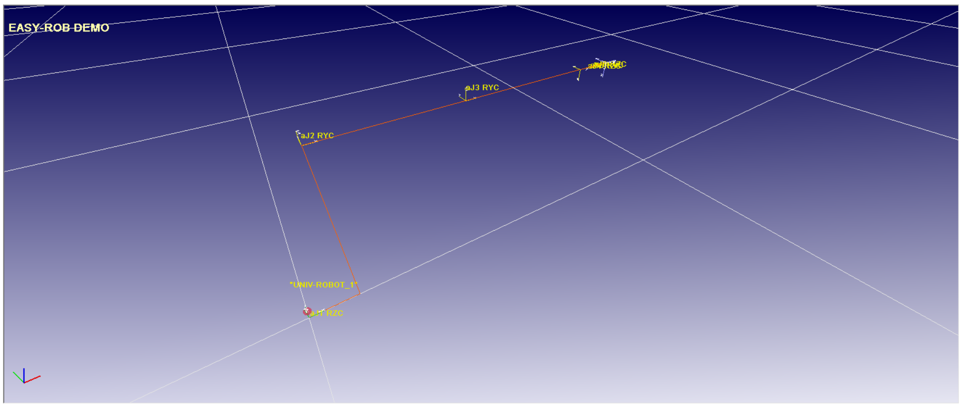

Abbildung 1: Stabmodell

Dieses wird vor dem nächsten Schritt auf Richtigkeit überprüft. Dazu werden in EasyRob verschiedene Gelenkwinkel angefahren und die Rotationen mit dem Stabmodell in den Versuchsunterlagen abgeglichen.

2 Realistisches Modell

Damit das Modell auch optisch einem realen KR3 ähnelt, können 3D-Körper zum Stabmodell hinzugefügt werden. Polygonmodelle werden an die einzelnen Armteile angeheftet und ermöglichen es somit, Kollisionen leichter zu erkennen. Die Darstellung des zugrundeliegenden Stabmodells lässt sich über "View > Coorsys > Show Robot Coorsys" deaktivieren. Die Polygonmodelle liegen als einzelne Dateien im IGP Format vor. Das Fenster zum Anheften von CAD-Objekten öffnet man via "3D-CAD > Open 3D-CAD Window". Dort muss man die Robot Group auswählen, damit die Objekte am Roboter fixiert werden. Über "Create Import" lassen sich IGP Teile importieren. Geschieht dies in der richtigen Reihenfolge von Armteil 0 bis Armteil 6 und mit dem Stabmodell in der Nullposition, so werden die Modelle automatisch an den korrekten Positionen eingefügt. Über den Wert des Schlüssels “Attach to active Joint” lässt sich das jeweilige Gelenk festlegen, an dem es fixiert wird.

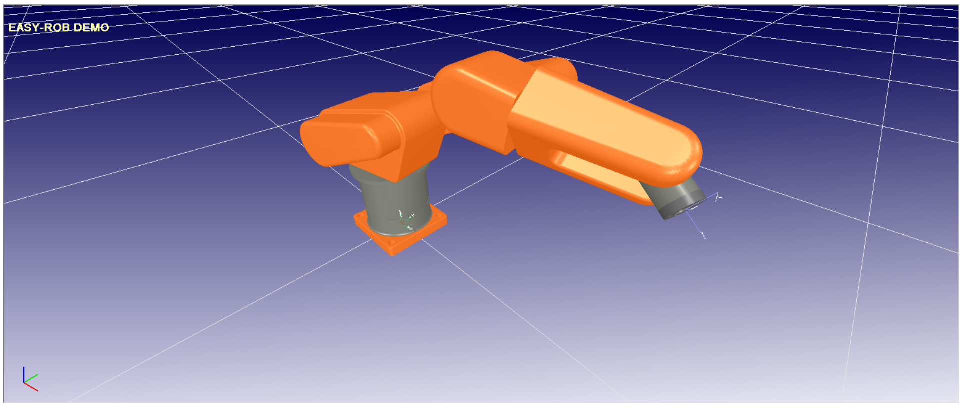

Abbildung 2: Realistisches Modell

3 Vorwärtstransformation

Mit der Vorwärtstransformation erhält man Position und Orientierung des TCP im kartesischen Koordinatensystem basierend auf den Gelenkstellungen. Zur Implementierung der Vorwärtstransformation in C++, unabhängig von EasyRob wurde die Vectmath-Bibliothek aus dem Microb-Projekt verwendet. Dieses wurde vom Unternehmen Hydro-Québec entwickelt. Vectmath stellt eine Reihe von Klassen bereit, welche für Berechnungen mit Matrizen benötigt werden. In einem ersten Schritt wird die kinematische Struktur des KR3 im Code dargestellt. Dazu dient das Array KR3[6] des Typs KinematicData mit 6 Feldern. Der Datentyp KinematicData beschreibt für jedes Gelenk den Typ und die jeweils drei Rotations- sowie Translationskomponenten der Transformation zum nächsten Koordinatensystem. Jedem Feld der Variablen KR3 werden mit einer Schleife jeweils die Werte für Typ, Translation und Rotation zugewiesen. Dafür werden zuerst 7 Arrays angelegt, je eines für jede Member-Variable, mit jeweils 6 Feldern, einem pro Gelenk.

Gelenk	Art	Achse	Trans X	Trans Y	Trans Z	Rot X	Rot Y	Rot Z
1	ROT	z	100	265	270	0	0	0
2	ROT	y	0	0	0	0	0	0
3	ROT	y	350	0	0	0	75	0
4	ROT	z	0	0	0	0	0	0
5	ROT	y	0	0	90	0	0	0
6	ROT	z	0	0	0	0	0	0

```
double kr3_trans_x[6] = {  100, 265, 270, 0, 0, 0 };
double kr3_trans_y[6] = { 0, 0, 0, 0, 0, 0 };
double kr3_trans_z[6] = { 350, 0, 0, 0, 75, 0 };
double kr3_rot_x[6] = { 0, 0, 0, 0, 0, 0 };
double kr3_rot_y[6] = { 0, 0, 90, 0, 0, 0 };
double kr3_rot_z[6] = { 0, 0, 0, 0, 0, 0 };

/*---------------------------------------------------------------------------*/
/*              HAUPTPROGRAMM                 */
/*---------------------------------------------------------------------------*/
int main(void)
{
  KinematicData KR3[6];
  Transform T, WK2Basis;
  Matrix R;
  int dof = 6;

  // Frame vom Weltkoordinatensystem zur Roboterbasis
  WK2Basis = mc_identity(4);
  //WK2Basis = Transform(mc_Rz(90.0 * DEG2RAD)) * Position(100.0, 200.0, 300.0);

  // Gelenkwinkel
  // Vector6 axes(0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
  // Vector6 axes(10.0, 0.0, 0.0, 0.0, 0.0, 0.0);
  // Vector6 axes(10.0, 20.0, -30.0, 40.0, 50.0, 60.0);
  // Vector6 axes(-10.0, 20.0, -30.0, 40.0, -50.0, 60.0);
  Vector6 axes(0.0, -142.65, 96.1763, 0.0, 51.4737, 0.0);

  // Initalisierung der kinematischen Struktur des KR3
  int kr3_typ[6] = { ROT_Z, ROT_Y, ROT_Y, ROT_Z, ROT_Y, ROT_Z };

  for(int i = 0; i < dof; i++)
  {
    KR3[i].typ = kr3_typ[i];
    KR3[i].trans_x = kr3_trans_x[i];
    KR3[i].trans_y = kr3_trans_y[i];
    KR3[i].trans_z = kr3_trans_z[i];
    KR3[i].rot_x = kr3_rot_x[i];
    KR3[i].rot_y = kr3_rot_y[i];
    KR3[i].rot_z = kr3_rot_z[i];
  }

  printf("Vorwaertstransformation");
  // Vorwärtstransformation via Wrapper-Funktion
  T = RobotPose(KR3, WK2Basis, axes);
  axes.print("\nAxes = ");
  T.print("\nT");
```

Listing 1: kinematische Struktur

Das Ziel der Vorwärtstransformation ist es, aus den Gelenkwinkeln zu Position und Orientierung des Tool Center Point (TCP) in kartesischen Weltkoordinaten zu gelangen. Dazu muss das i-te Gelenk um den Wert der Gelenkvariablen q_i um die Gelenkachse gedreht, bzw. auf ihr verschoben werden und anschließend die Transformation zum nächsten Gelenkkoordinatensystem durchgeführt werden. Dazu muss basierend auf dem Gelenktyp und dem Wert der Gelenkvariablen je Gelenk eine Transformationsmatrix erstellt werden und eine weitere, basierend auf den Werten, welche im Array KR3[] für jedes Gelenk gespeichert sind, mit der man Position und Orientierung des nächsten Gelenkkoordinatensystems erhält. Dabei muss darauf geachtet werden, dass die Winkelangaben der kinematischen Struktur und die Gelenkvariablen in Grad angegeben werden und daher in Radians umgerechnet werden müssen.

```
/* Berechnung der Vorwärtstransformation von Gelenk qi bis Gelenk qn ******************/
Transform ForwardKinematics(KinematicData robot[], Vector axes, int qi, int qn)
{
  Transform T;

  assert(qi < axes.dim() && qn < axes.dim());

  Euler_angle_xyz Rot;
  Euler_angle_xyz Transf;

  while (qi <= qn)
  {
    // Transformation aufgrund der Gelenkvariablen
    switch (robot[qi].typ) {
    case ROT_X:
      Rot = Euler_angle_xyz(axes[qi] * DEG2RAD, 0, 0, 0, 0, 0);
      break;
    case ROT_Y:
      Rot = Euler_angle_xyz(0, axes[qi] * DEG2RAD, 0, 0, 0, 0);
      break;
    case ROT_Z:
      Rot = Euler_angle_xyz(0, 0, axes[qi] * DEG2RAD, 0, 0, 0);
      break;
    case TRANS_X:
      Rot = Euler_angle_xyz(0, 0, 0, axes[qi], 0, 0);
      break;
    case TRANS_Y:
      Rot = Euler_angle_xyz(0, 0, 0, 0, axes[qi], 0);
      break;
    case TRANS_Z:
      Rot = Euler_angle_xyz(0, 0, 0, 0, 0, axes[qi]);
      break;
    }

    // Transformation zum nächsten Gelenk
```

```
    Transf = Euler_angle_xyz(robot[qi].rot_x * DEG2RAD, robot[qi].
        rot_y * DEG2RAD, robot[qi].rot_z * DEG2RAD, robot[qi].
        trans_x, robot[qi].trans_y, robot[qi].trans_z);

    T *= (Transform) Rot;
    T *= (Transform) Transf;

    ++qi;
  }

  return T;
}
```

Listing 2: `ForwardKinematics()`

Ein Aufruf der Funktion `ForwardKinematics()` mit den Gelenkwinkeln `axes(0.0, -142.65, 96.1763, 0.0, 51.4737, 0.0)` als Argument ergab folgende homogene Transformationsmatrix:

$$T = \begin{bmatrix} -0.087156 & 0.000000 & 0.996195 & 149.999965 \\ 0.000000 & 1.000000 & 0.000000 & 0.000000 \\ -0.996195 & 0.000000 & -0.087156 & 699.999891 \\ 0.000000 & 0.000000 & 0.000000 & 1.000000 \end{bmatrix}$$

Gibt man obige Gelenkwinkel in EasyRob ein und öffnet das Fenster mit der Ausgabe der Pose in kartesischen Weltkoordinaten via “View > Open Online Output Data”, so sieht man, dass die Werte mit den selbst errechneten übereinstimmen.

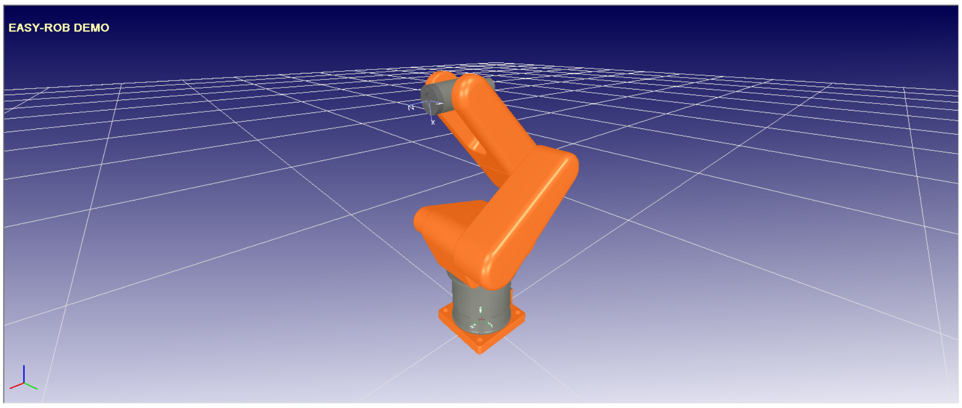

Abbildung 3: Pose in EasyRob

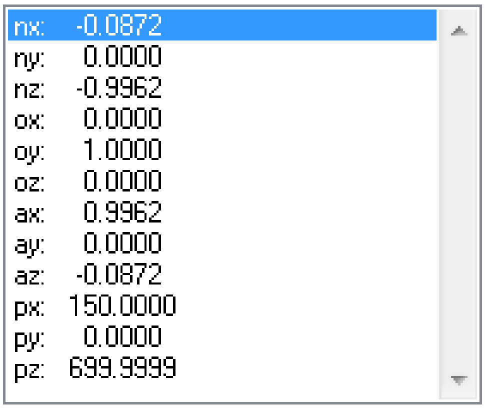

Abbildung 4: Ergebnisse des obigen Algorithmus stimmen mit EasyRob überein

4 Rückwärtstransformation

Das Ziel der Rückwärtstransformation (inversen Kinematik) ist die Bestimmung der Gelenkwinkel bei gegebener Position und Orientierung des TCP in kartesischen Weltkoordinaten. Die inverse Kinematik kann geometrisch oder numerisch berechnet werden. Hier wurde die zweite Möglichkeit gewählt. Der erste Schritt der numerischen Berechnung der inversen Kinematik ist die Bestimmung der Jacobi-Matrix im Arbeitspunkt, welcher durch die Gelenkvariable `axes` gegeben ist. Dies linearisiert den Zusammenhang zwischen Gelenkvariablen und kartesischen Koordinaten in der Nähe des Arbeitspunktes.

$$\begin{aligned} J(\underline{q}) \cdot \Delta \underline{q} &= \Delta \underline{x} \quad | \cdot J^{-1}(\underline{q}) \\ \Delta \underline{q} &= J^{-1}(\underline{q}) \cdot \Delta \underline{x} \end{aligned}$$

Spalte i der Jacobi-Matrix besteht aus der numerisch nach der Gelenkvariablen q_k differenzierten Spalte i der Vorwärtstransformation. Die Berechnung der Jacobi-Matrix bei gegebener kinematischer Struktur und Gelenkwinkeln geschieht in der Funktion `ikNumDiff()`.

Zur Berechnung der numerischen Differentiation bildet man den Vorwärtsdifferenzenquotienten. Dazu wird zunächst die momentane kartesische Pose in einer Variablen vom Typ `Angle_axis_xyz` gespeichert. Dann werden nacheinander die einzelnen Gelenkvariablen um den Wert `delta` erhöht, damit eine zweite Vorwärtskinematik berechnet und daraus der Spaltenvektor der Differenzenquotienten gebildet. Die einzelnen Spalten werden nebeneinander gesetzt und bilden so eine Matrix der Dimension 6x6.

Die so bestimmte Jacobi-Matrix wird von der Funktion zurückgegeben.

```
/* Berechnet die Jakobi-Matrix mit Hilfe des Vorwärtsdifferenzenquotienten *************/
Matrix ikNumDiff(KinematicData robot[], Vector axes)
{
  Matrix Jacobi;
  double delta = 1e-4;

  Angle_axis_xyz currentCartesian = (Angle_axis_xyz) ForwardKinematics(robot, axes, 0, axes.dim()-1);

  for (int i = 0; i < axes.dim(); i++)
  {
    Vector diffAxes(axes);

    diffAxes[i] += delta;

    Angle_axis_xyz deltaCartesian = (Angle_axis_xyz) ForwardKinematics(robot, diffAxes, 0, diffAxes.dim() - 1) - currentCartesian;

    if (i == 0)
      Jacobi = deltaCartesian;
    else
      Jacobi = mc_concatenate(Jacobi, deltaCartesian, 1);
  }

  Jacobi /= delta;
  Jacobi.print("\nJacobi");

  return Jacobi;
}
```

Listing 3: `ikNumDiff()`

Die oben erläuterte Funktion zur Bestimmung der Jacobi-Matrix im Arbeitspunkt wird innerhalb der Funktion `InverseKinematics()` aufgerufen. Zunächst wird die Jacobi-Matrix zum übergebenen Gelenkwinkelvektor bestimmt und dann invertiert. Die dazu benutzte Funktion `ikSVD()` ist robust gegenüber Rangdefekten, welche in der Nähe von singulären Stellungen auftreten können. Dann wird in einer Schleife `dq` mithilfe der invertierten Jacobi-Matrix und der kartesischen Differenz zwischen gewünschten und TCP-Koordinaten und den aus den aktuellen Gelenkstellungen berechneten aktuellen Koordinaten bestimmt.

Da jedoch alle Gelenke rotatorisch sind, erreicht man nach einem Durchlauf der Schleife meist nicht die gewünschten TCP-Koordinaten. Ob die neuen Koordinaten nah genug an den gewünschten sind, wird mithilfe einer Größe ϵ bestimmt. Sind der erreichte Punkt nah genug am Ziel und die benötigten Gelenkänderungen ebenfalls klein genug, so wird die Schleife abgebrochen und die neuen Gelenkvariablen werden von der Funktion zurückgegeben. Sind die Bedingungen noch nicht erfüllt, so wird eine neue

Jacobi-Matrix um den neuen Punkt bestimmt und die Schleife wird ein weiteres mal durchlaufen. Um Endlosschleifen bei unerreichbaren TCP-Positionen oder Sprüngen der Gelenkvariablen aufgrund von numerischen Fehlern zu vermeiden, bricht die Schleife nach spätestens 100 Durchläufen ab.

```
/* Numerische Berechnung der inversen Kinematik
    ****************************************/
Vector InverseKinematics(Transform T, KinematicData robot[],
    Transform WK2Basis, Vector axes)
{
  // T ist Zielposition (4x4)
  // axes ist Startposition (Vector)

  Matrix Jacobi = ikNumDiff(robot, axes);

  Matrix invJacobi = ikSVD(Jacobi);

  invJacobi.print("\ninvJacobi");

  Vector dq;
  double epsilon = 0.00001;

  Angle_axis_xyz T_angle = (Angle_axis_xyz)T;
  Angle_axis_xyz dx;

  for (int i = 0; i < 100; i++) {
    dx = T_angle - (Angle_axis_xyz) ForwardKinematics(robot, axes,
        0, axes.dim() - 1);

    // dq: 6x1
    // invJacobi: 6x6
    // dx: 6x1

    dq = invJacobi * dx;

    // Algorithmus zum Berechnen der Gelenkstellungen
    // q[k + 1] = q[k] + J^-1(q_k)*(x_(k + 1) - x_k);
    axes += dq;

    if (dx.norm() < epsilon && dq.norm() < epsilon)
    {
      printf("\nbenoetigte Iterationen: %d\n",i);
      break;
    }

    Jacobi = ikNumDiff(robot, axes);
    invJacobi = mc_pinv(Jacobi);
  }

  return axes;
}
```

Listing 4: `InverseKinematics()`

5 Linearbewegung

Die Funktion `IpoLin()` erzeugt aus einer gegebenen kinematischen Struktur, Start- und Endposition in Gelenkkoordinaten und der gewünschten Anzahl an Interpolationsschritten ein EasyRob-Programm, dass eine Linearbahn zwischen beiden Positionen fährt.

Dazu werden aus den gewünschten Gelenkwinkeln für Start- und Endposition mit der Vorwärtskinematik die kartesischen Koordinaten des TCP berechnet. Zwischen beiden Punkten werden n Punkte linear interpoliert. n wird als Argument beim Aufruf von `IpoLin()` übergeben. Für jeden dieser Punkte werden die Gelenkstellungen mittels der inversen Kinematik berechnet.

```
// Gelenkwinkel für Linearbahn
// Vector6 qs(0.0, 0.0, 0.0, 0.0, 0.0, 0.0); // Singuläre Stellung -> Fehler bei IK
Vector6 qs(-30.0, -100.0, 80.0, 0.0, 0.0, 0.0);
Vector6 qe(0.0, -142.65, 96.1763, 0.0, 51.4737, 0.0);

// Linearbewegung von qs nach qe
if (IpoLin(KR3, qs, qe, 100, "kr3.prg", true)) // 100 Schritte, numerische Lösung
  fprintf(stderr, "Fehler beim Oeffnen der Datei\n");
```

Listing 5: `IpoLin()`

Das erzeugte EasyRob-Programm sieht wie folgt aus:

```
JUMP_TO_AX      0.0000      0.0000     -0.0000     -0.0000      0.0000     -0.0000
JUMP_TO_AX     -0.0000 -4867953837.5816 9645760434.4445  5220.0000 -4777806614.1060 -5220.0000
JUMP_TO_AX     -0.0000 -4867953834.5121 9645760427.5833  5220.0000 -4777806617.9476 -5220.0000
```

Listing 6: Anfang kr3.prg

```
JUMP_TO_AX     -0.0000 -4867953887.9687 9645760352.6409  5220.0000 -4777806751.1467 -5220.0000
JUMP_TO_AX      0.0000 -4867953889.4072 9645760353.7269  5220.0000 -4777806751.5492 -5220.0000
JUMP_TO_AX     -0.0000 -4867953890.8431 9645760354.8456  5220.0000 -4777806751.9164 -5220.0000
```

Listing 7: Ende kr3.prg

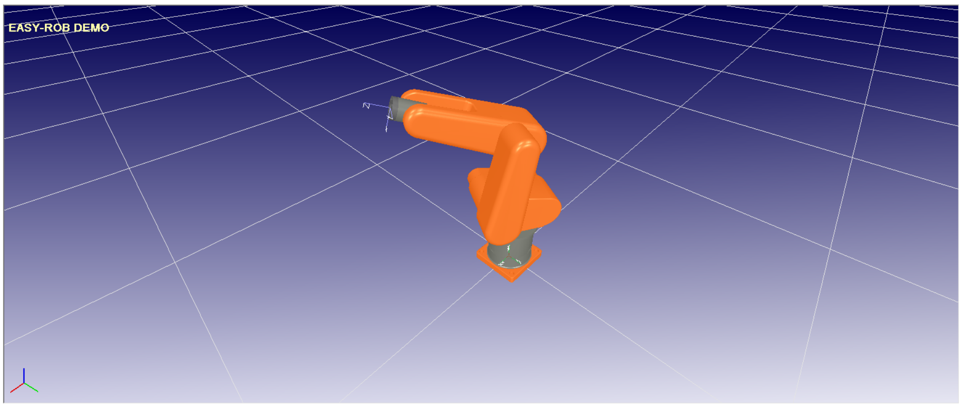

Abbildung 5: Startposition

Die Linearbahn des TCP wurde mithilfe der Funktion “View > TCP Trace > TCP Trace ON/OFF” sichtbar gemacht.

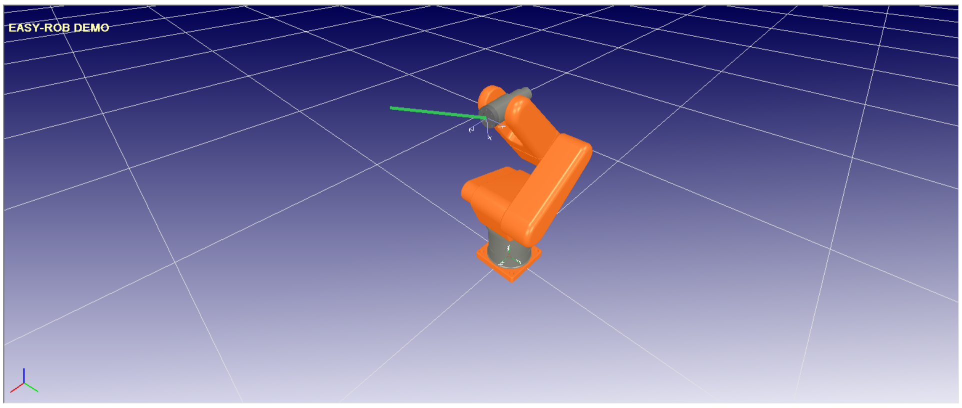

Abbildung 6: Endposition

Beim ersten Versuch, eine Linearbahn zu fahren trat ein Fehler auf: Der Arm machte einen relativen großen Sprung und schien sich danach nicht mehr zu bewegen. Grund war, dass als Ausgangsposition die Nullposition gewählt worden war. Da dann die Gelenkachsen 4 und 6 in einer Flucht liegen verliert man einen Freiheitsgrad: man befindet sich in einer Randsingularität. Eine Lösung des Problems wäre es, einen Algorithmus zu schreiben, der Singularitäten anhand der resultierenden äußerst großen Gelenkwinkeln erkennt und dann einzelne Gelenke minimal zu verfahren, um die Singularität zu eliminieren.

```
JUMP_TO_AX      0.0000      0.0000     -0.0000     -0.0000      0.0000
       -0.0000
JUMP_TO_AX     -0.0000 -4867953837.5816 9645760434.4445  5220.0000
    -4777806614.1060 -5220.0000
```

```
JUMP_TO_AX    -0.0000 -4867953834.5121 9645760427.5833  5220.0000
    -4777806617.9476 -5220.0000
```

Listing 8: Anfang kr3-singularitaet.prg

```
JUMP_TO_AX    -0.0000 -4867953887.9687 9645760352.6409  5220.0000
    -4777806751.1467 -5220.0000
JUMP_TO_AX     0.0000 -4867953889.4072 9645760353.7269  5220.0000
    -4777806751.5492 -5220.0000
JUMP_TO_AX    -0.0000 -4867953890.8431 9645760354.8456  5220.0000
    -4777806751.9164 -5220.0000
```

Listing 9: Ende kr3-singularitaet.prg