Hochschule Darmstadt

Lehrveranstaltung: Simulation von Robotersystemen
Prof. Dr. Thomas Horsch
Kalibrierung
Datum: 27.5.2016

Name	Matrikelnummer
Fabian Alexander Wilms	735162

Studiengang: Mechatronik
Abgabedatum: 3.6.2016

Testat	

Inhaltsverzeichnis

1 Transformation zwischen zwei Koordinatensystemen

Das Ziel dieser Aufgabe ist es, die Transformationsmatrix zwischen zwei Koordinatensystemen K0 und K1 zu bestimmen. Die beiden Koordinatensysteme sind durch jeweils 3 Punkte im Weltkoordinatensystem (KW) definiert: Den Ursprung (`K0o, K1o`), einen Punkt auf der positiven X-Achse (`K0x, K1x`) und einen Punkt auf der positiven Y-Achse (`K0y, K1y`). Die Z-Achse ist damit eindeutig bestimmt.

Durch Messungenauigkeiten kann es vorkommen, dass die jeweils 3 Punkte kein orthogonales Koordinatensystem bilden. Zudem haben die Punkte auf X- und Y-Achse nicht notwendigerweise den Abstand 1 vom Ursprung. Um aus diesen Punkten dennoch ein orthogonales Koordinatensystem zu erhalten, wird die Schmidtsche Orthonormalisierung angewendet.

Die neuen Vektoren x, y und z werden wie folgt berechnet:

$$
\begin{aligned}
x &= \frac{X - Origin}{|X - Origin|} \\
b_2 &= Y - Origin - \frac{(Y - Origin)^T \cdot (X - Origin)}{(X - Origin)^T \cdot (X - Origin)} \cdot (X - Origin) \\
y &= \frac{b_2}{|b_2|} \\
z &= x \times y
\end{aligned}
$$

Diese Orthonormalisierung wird für beide Koordinatensysteme durchgeführt, wodurch man für jedes orthogonale Koordinatensystem den Ursprung und die Vektoren x, y und z erhält.

Diese beiden Koordinatensysteme werden dann in jeweils eine Transformationsmatrix umgeformt, die den Übergang vom Weltkoordinatensystem nach K0 bzw. K1 beschreibt.

Die Transformationsmatrix $^{K0}T_{K1}$ kann daraufhin wie folgt bestimmt werden:

$$^{K0}T_{K1} = {}^{K0}T_{KW} \cdot {}^{KW}T_{K1} = ({}^{KW}T_{K0})^{-1} \cdot {}^{KW}T_{K1}$$

```
int main(int argc, const char* argv[])
{
    Position K0o, K0x, K0y, K1o, K1x, K1y;

    if (!LoadSys(DATA_FNAME1, K0o, K0x, K0y)
    ||  !LoadSys(DATA_FNAME2, K1o, K1x, K1y))
        return EXIT_FAILURE;

    // Ihr Code
```

```

    Position myPos[6] = { K0o, K0x, K0y, K1o, K1x, K1y };

    Vector3 x,y,z;
    Matrix T1;
    Matrix T2;
    Transform T1_Transform;
    Transform T2_Transform;
    Vector4 homogen(0, 0, 0, 1);
    Matrix homogen_t = mc_transpose(homogen);

    for (int i = 0; i <= 1; i++)
    {
        Vector3 x_minus_origin = myPos[1 + i * 3] - myPos[0 + i * 3];
        Matrix x_minus_origin_t = mc_transpose(x_minus_origin);
        Vector3 y_minus_origin = myPos[2 + i * 3] - myPos[0 + i * 3];
        Matrix y_minus_origin_t = mc_transpose(y_minus_origin);
        x = x_minus_origin / mc_length(x_minus_origin);
        Vector3 b2 = y_minus_origin - (Vector3)((double)((y_minus_origin_t*(Matrix)x_minus_origin) / (x_minus_origin_t*(Matrix)x_minus_origin)) * x_minus_origin);
        y = b2 / mc_length(b2);
        z = mc_vectorial_product(x, y);

        if (i == 0)
        {
            x.print("x␣System␣1");
            y.print("y␣System␣1");
            z.print("z␣System␣1");
            // T1 bestimmen
            T1 = x;
            T1 = mc_concatenate(T1, y, 1);
            T1 = mc_concatenate(T1, z, 1);
            T1 = mc_concatenate(T1, K0o, 1);
            T1 = mc_concatenate(T1, homogen_t, 0);
            T1.print("Transformationsmatrix␣System␣1");
        } else {
            x.print("x␣System␣1");
            y.print("y␣System␣1");
            z.print("z␣System␣1");
            // T1 bestimmen
            T2 = x;
            T2 = mc_concatenate(T2, y, 1);
            T2 = mc_concatenate(T2, z, 1);
            T2 = mc_concatenate(T2, K1o, 1);
            T2 = mc_concatenate(T2, homogen_t, 0);
            T2.print("Transformationsmatrix␣System␣2");
        }
    }

    T1_Transform = (Transform) T1;
```

```
    T2_Transform = (Transform) T2;

    Transform T1_Transform_inv = mc_pinv(T1_Transform);

    Transform T12 = T1_Transform_inv * T2_Transform;

    T12.print("Transformationsmatrix Sys 1 nach Sys 2");

    return EXIT_SUCCESS;
}
```

Listing 1: Transformation zwischen zwei Koordinatensystemen

Der Einfachheit halber wurde nicht mit realen Messpunkten getestet, sondern es wurden zwei Koordinatensysteme vorgegeben.

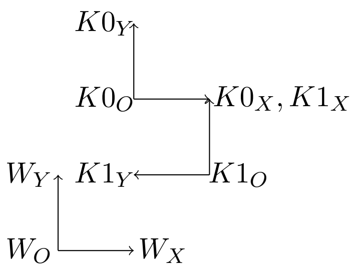

Abbildung 1: Die vorgegebenen Koordinatensysteme

```
1.01;2;0
2;2;0
1;3;0
```

Listing 2: Nicht orthogonale Messpunkte des ersten Koordinatensystems

```
2;1;0
2.01;2;0
1;1;0
```

Listing 3: Nicht orthogonale Messpunkte des zweiten Koordinatensystems

```
Lese Daten aus 4_1_System1.txt
Lese Daten aus 4_1_System2.txt
x System 1 3 x 1 Vector3
1.000000
0.000000
0.000000
y System 1 3 x 1 Vector3
0.000000
1.000000
0.000000
z System 1 3 x 1 Vector3
0.000000
```

```
0.000000
1.000000
Transformationsmatrix System 1 4 x 4 Matrix
1.000000        0.000000        0.000000        1.010000
0.000000        1.000000        0.000000        2.000000
0.000000        0.000000        1.000000        0.000000
0.000000        0.000000        0.000000        1.000000
x System 1 3 x 1 Vector3
0.010000
0.999950
0.000000
y System 1 3 x 1 Vector3
-0.999950
0.010000
0.000000
z System 1 3 x 1 Vector3
0.000000
-0.000000
1.000000
Transformationsmatrix System 2 4 x 4 Matrix
0.010000        -0.999950       0.000000        2.000000
0.999950        0.010000        -0.000000       1.000000
0.000000        0.000000        1.000000        0.000000
0.000000        0.000000        0.000000        1.000000
Transformationsmatrix Sys 1 nach Sys 2 4 x 4 Transform
0.010000        -0.999950       0.000000        0.990000
0.999950        0.010000        0.000000        -1.000000
0.000000        0.000000        1.000000        0.000000
0.000000        0.000000        0.000000        1.000000
```

Listing 4: Ausgabe

Es ergibt sich also folgende Transformationsmatrix:

$$T = \begin{bmatrix} 0.010000 & -0.999950 & 0.000000 & 0.990000 \\ 0.999950 & 0.010000 & 0.000000 & -1.000000 \\ 0.000000 & 0.000000 & 1.000000 & 0.000000 \\ 0.000000 & 0.000000 & 0.000000 & 1.000000 \end{bmatrix}$$

2 Verkettung von Transformationen

Da der Arbeitsraum des MicroScribe-Gerätes begrenzt ist, benötigt man einen Weg, diesen zu erweitern. Dies ist möglich, indem man in Position 1 des Gerätes ein Koordinatensystem K0 vermisst, dann das Gerät an anderer Stelle platziert und die Messung desselben Koordinatensystems aus Position 2 wiederholt. Vermisst man nun ein zweites Koordinatensystem K1 von Position 2 aus, so lässt sich die Transformation relativ zu Position 1 berechnen, was eine Erweiterung des Arbeitsraumes bedeutet.

Man verkettet dieses Verfahren, indem man nun K2 von Position 2 und 3 aus misst, K3 von Position 3 und 4 usw. Sind das erste und das letzte gemessene Koordinatensystem dasselbe, so muss das Produkt der einzelnen Transformationsmatrizen idealerweise gleich der Einheitsmatrix sein:

$$K0 \cdot \sum_{i}^{n} {}^{i}T_{i+1} = K0 \cdot E = K0$$

Der Algorithmus aus Aufgabe 1 wird in eine Funktion integriert:

```
Transform get_Transform(Position K0o, Position K0x, Position K0y,
    Position K1o, Position K1x, Position K1y)
{
    // Ihr Code

    Position myPos[6] = { K0o, K0x, K0y, K1o, K1x, K1y };

    Vector3 x, y, z;
    Matrix T1;
    Matrix T2;
    Transform T1_Transform;
    Transform T2_Transform;
    Vector4 homogen(0, 0, 0, 1);
    Matrix homogen_t = mc_transpose(homogen);

    for (int i = 0; i <= 1; i++)
    {
        Vector3 x_minus_origin = myPos[1 + i * 3] - myPos[0 + i *
            3];
        Matrix x_minus_origin_t = mc_transpose(x_minus_origin);
        Vector3 y_minus_origin = myPos[2 + i * 3] - myPos[0 + i *
            3];
        Matrix y_minus_origin_t = mc_transpose(y_minus_origin);
        x = x_minus_origin / mc_length(x_minus_origin);
        Vector3 b2 = y_minus_origin - (Vector3)((double)((
            y_minus_origin_t*(Matrix)x_minus_origin) / (
            x_minus_origin_t*(Matrix)x_minus_origin)) *
            x_minus_origin);
        y = (Matrix) b2 / mc_length(b2);
        z = mc_vectorial_product(x, y);

        if (i == 0)
```

```
        {
            // T1 bestimmen
            T1 = x;
            T1 = mc_concatenate(T1, y, 1);
            T1 = mc_concatenate(T1, z, 1);
            T1 = mc_concatenate(T1, myPos[0], 1);
            T1 = mc_concatenate(T1, homogen_t, 0);
        }
        else {
            // T2 bestimmen
            T2 = x;
            T2 = mc_concatenate(T2, y, 1);
            T2 = mc_concatenate(T2, z, 1);
            T2 = mc_concatenate(T2, myPos[3], 1);
            T2 = mc_concatenate(T2, homogen_t, 0);
        }
    }
    T1_Transform = (Transform)T1;
    T2_Transform = (Transform)T2;
    Transform T1_Transform_inv = mc_pinv(T1_Transform);
    Transform T12 = T1_Transform_inv * T2_Transform;
    T12.print("\nTransformationsmatrix Sys1 nach Sys2");

    return T12;
}
```

Listing 5: Transformation zwischen zwei Koordinatensystemen

```
int main(int argc, const char* argv[])
{
    SysDefVector SysDefs;

    if (!LoadSysN(DATA_FNAME, SysDefs))
        return EXIT_FAILURE;

    // Ihr Code

    Transform myT;
    Transform T = mc_identity(4);

    for (int i = 0; i < SysDefs.size()-1; i=i+2)
    {
        //printf("\nSize: %d", (int)SysDefs.size());
        printf("\nSystem %d nach %d", i + 1, i+2);
        myT = get_Transform(SysDefs[i].Origin, SysDefs[i].X,
            SysDefs[i].Y, SysDefs[i+1].Origin, SysDefs[i+1].X,
            SysDefs[i+1].Y);
        T *= myT;
    }

    T.print("\nVerkettete Transformationen");

    Transform Tsoll = mc_identity(4);

```

```
    Tsoll.print("\nSoll");

    return EXIT_SUCCESS;
}
```

Listing 6: Verkettung von Transformationen

Getestet wurde der Algorithmus mit 8 vorgegebenen Transformationsmatrizen:

```
Lese Datensatz aus 4_2_System1.txt
Lese Datensatz aus 4_2_System2.txt
Lese Datensatz aus 4_2_System3.txt
Lese Datensatz aus 4_2_System4.txt
Lese Datensatz aus 4_2_System5.txt
Lese Datensatz aus 4_2_System6.txt
Lese Datensatz aus 4_2_System7.txt
Lese Datensatz aus 4_2_System8.txt

System 1 nach 2
Transformationsmatrix Sys1 nach Sys 2 4 x 4 Transform
0.985230        0.171207        0.003152        -252.307704
-0.171214       0.985231        0.002277        -174.796067
-0.002715       -0.002783       0.999992        0.462192
0.000000        0.000000        0.000000        1.000000

System 3 nach 4
Transformationsmatrix Sys1 nach Sys 2 4 x 4 Transform
-0.517979       0.855393        -0.000047       236.817210
-0.855389       -0.517977       0.003032        -234.055211
0.002569        0.001611        0.999995        0.353946
0.000000        0.000000        0.000000        1.000000

System 5 nach 6
Transformationsmatrix Sys1 nach Sys 2 4 x 4 Transform
-0.329056       -0.944310       0.001261        -233.017251
0.944308        -0.329052       0.002781        79.421043
-0.002211       0.002106        0.999995        0.073087
0.000000        0.000000        0.000000        1.000000

System 7 nach 8
Transformationsmatrix Sys1 nach Sys 2 4 x 4 Transform
0.923591        -0.383376       0.001418        -264.349646
0.383375        0.923593        0.000398        242.670564
-0.001462       0.000176        0.999999        0.445352
0.000000        0.000000        0.000000        1.000000

Verkettete Transformationen 4 x 4 Transform
0.999905        -0.012206       0.006358        -1.205218
0.012191        0.999923        0.002337        1.215314
-0.006386       -0.002259       0.999977        -0.128655
0.000000        0.000000        0.000000        1.000000

Soll 4 x 4 Transform
```

```
1.000000        0.000000        0.000000        0.000000
0.000000        1.000000        0.000000        0.000000
0.000000        0.000000        1.000000        0.000000
0.000000        0.000000        0.000000        1.000000
```

Listing 7: Ausgabe

Die aus der Verkettung der Transformationen resultierende Matrix

$$T_{Ist} = \begin{bmatrix} 0.999905 & -0.012206 & 0.006358 & -1.205218 \\ 0.012191 & 0.999923 & 0.002337 & 1.215314 \\ -0.006386 & -0.002259 & 0.999977 & -0.128655 \\ 0.000000 & 0.000000 & 0.000000 & 1.000000 \end{bmatrix}$$

entspricht annähernd genau der idealen Lösung (d.h. Einheitsmatrix) T_{Soll}

$$T_{Soll} = \begin{bmatrix} 1.000000 & 0.000000 & 0.000000 & 0.000000 \\ 0.000000 & 1.000000 & 0.000000 & 0.000000 \\ 0.000000 & 0.000000 & 1.000000 & 0.000000 \\ 0.000000 & 0.000000 & 0.000000 & 1.000000 \end{bmatrix}$$

3 Kalibrierung

Wird an einem Roboter ein Werkzeug angebracht, muss die Transformation vom alten zum neuen TCP bestimmt werden. Da die neue Messspitze parallel zur ursprünglichen verläuft, enthält die Transformation keine Rotation, sondern nur den Translationsvektor P_t.

Um diesen zu bestimmen sucht man sich einen festen Punkt P_{pivot} im Weltkoordinatensystem, dessen Koordinaten nicht bekannt sein müssen und fährt mit dem Roboter verschiedene Posen an, wobei immer die Position des neuen TCP identisch zu P_{pivot} sein muss (dies nennt sich „Quirlen“).

Da der direkte Weg zum Drehpunkt P_{pivot} mit dem indirekten über P_i und anschließend $R_i \cdot P_t$ identisch sein soll muss gelten:

$$P_i + R_i \cdot P_t - P_{pivot} \cong 0$$

mit $i = 1..n$.

Bringt man P_i auf die rechte Seite und drückt $R_i \cdot P_t - P_{pivot}$ durch eine Matrixmultiplikation aus, so erhält man

$$\underbrace{\begin{pmatrix} R_1 & -I \\ \vdots & \vdots \\ R_n & -I \end{pmatrix}}_{\text{Koeffizientenmatrix A}} \cdot \underbrace{\begin{pmatrix} P_t \\ P_{pivot} \end{pmatrix}}_{\text{Parameter x}} \cong \underbrace{\begin{pmatrix} -P_1 \\ \vdots \\ -P_n \end{pmatrix}}_{\text{Beobachtungen b}}$$

$$\Leftrightarrow \qquad A \cdot x \cong b$$

Da die Messungen fehlerbehaftet sind, muss auf der rechten Seite der Residuenvektor r hinzu addiert werden, welcher die Fehler kompensiert.

$$A \cdot x = b + r$$

Gesucht sind jetzt die Werte für P_t und P_{pivot}, für die der Fehler $r^T \cdot r$ minimal wird.

Dafür stellt man obige Gleichung nach r um, differenziert $r^T \cdot r$ nach x und setzt dies gleich 0. Daraus ergibt sich

$$x = (A^T \cdot A)^{-1} \cdot A^T \cdot b \tag{1}$$

Da es 6 Unbekannt gibt, werden auch mindestens 2 Messposen benötigt, um das Gleichungssystem lösen zu können.

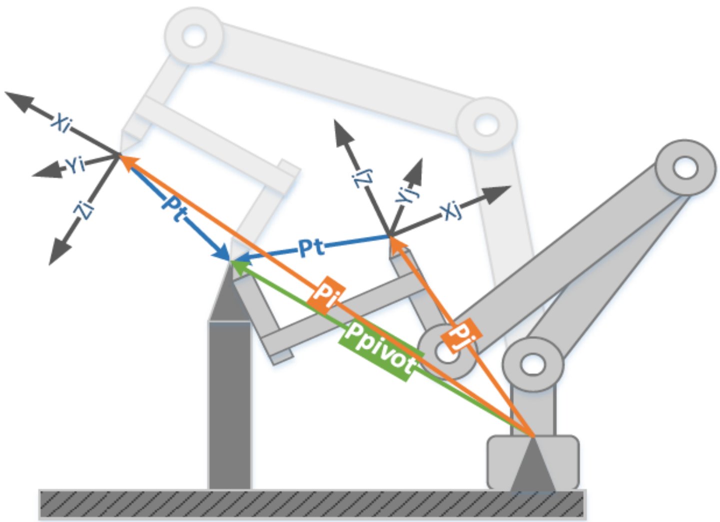

Abbildung 2: „Quirlen“ um einen festen Punkt im Raum

Die für P_t und P_{pivot} berechneten Werte lassen sich validieren, indem man den direkten Weg zum Drehpunkt P_{pivot} mit dem indirekten über P_i und anschließend $R_i \cdot P_t$ vergleicht. Idealerweise ist die Differenz $\vec{0}$.

$$P_{test} = R_i \cdot P_t + P_i - P_{pivot} \stackrel{!}{=} \vec{0} \tag{2}$$

Der folgende Algorithmus extrahiert zunächst für jeden gemessenen Frame die Position und die Rotation des ursprünglichen TCP und baut daraus die Matrix A sowie den Vektor b auf. Anschließend wird Gleichung (1) angewendet und die berechneten Werte für P_t und P_{pivot} mit Gleichung (2) validiert. Zum Validieren muss wieder für jeden Frame Rotation und Position extrahiert werden, um sie in die Gleichung einsetzen zu können.

Zuletzt wird die Standardabweichung des Ergebnisses mit folgender Formel bestimmt:

$$s_0 = \sqrt{\frac{r^T \cdot r}{n}}$$

```
int main(int argc, const char* argv[])
{
    EulerAngleXYZVector Frames;
    Position Pt, Ppivot;

    if (!LoadCalFrames(DATA_FNAME_IN, Frames))
        return EXIT_FAILURE;

    // A * x = b
    //
    // | R1,nx R1,ox R1,ax -1  0  0 |                    | -P1,x |
```

```
    // | R1,ny R1,oy R1,ay  0 -1  0 |   | Pt,x       |   | -P1,y |
    // | R1,nz R1,oz R1,az  0  0 -1 |   | Pt,y       |   | -P1,z |
    // | R2,nx R2,ox R2,ax -1  0  0 | * | Pt,z       | = | -P2,x |
    // | R2,ny R2,oy R2,ay  0 -1  0 |   | Ppivot,x   |   | -P2,y |
    // | R2,nz R2,oz R2,az  0  0 -1 |   | Ppivot,y   |   | -P2,z |
    // | R3,nx R3,ox R3,ax -1  0  0 |   | Ppivot,z   |   | -P3,x |
    // | ...                        |                    | ...   |
    //
    // x = (A' * A)^(-1) * A' * b

    // Ihr Code

    Matrix A;
    Vector b;

    for (unsigned int i = 0; i < Frames.size(); i++) {
        if (i == 0) {
            b =  - (Position) Frames[i];
            A = (Rotation) Frames[i];
            A = mc_concatenate(A, - mc_identity(3), 1);
        }
        else {
            b = mc_concatenate(b, - (Position) Frames[i], 0);
            Matrix Block36 = (Rotation)Frames[i];
            Block36 = mc_concatenate(Block36, - mc_identity(3), 1);
            A = mc_concatenate(A, Block36, 0);
        }
    }

    Matrix AT = mc_transpose(A);
    Vector6 x = mc_inv(AT*A)*AT*b;

    x.print("x");

    // Validierung der Ergebnisse
    Pt = { x[0], x[1], x[2] };
    Ppivot = { x[3], x[4], x[5] };

    Pt.print("\nPt");
    Ppivot.print("\nPpivot");

    Position Ptest;
    Rotation R;
    Position p;

    for (unsigned int i = 0; i < Frames.size(); i++) {
            R = (Rotation) Frames[i];
            p = (Position) Frames[i];
            Ptest = R * Pt + p - Ppivot;
            Ptest.print("\nPtest");
    }

    // Standardabweichung berechnen
```

```
    Vector r;
    r = Vector (A * x) - b;
    double s0 = sqrt((double) (mc_transpose(r) * r) / (int) Frames
        .size());

    printf("\nStandardabweichung %f \n", s0);

    if (!SaveCal(DATA_FNAME_OUT, Pt, Ppivot))
        return EXIT_FAILURE;

    return EXIT_SUCCESS;
}
```

Listing 8: Kalibrierung

P_{test} wurde für jede in `Kreisel.txt` enthaltene Messpose berechnet:

```
Lese Datensatz aus Kreisel.txt
x 6 x 1 Vector6
-1.747805
44.586360
-87.648108
-253.032904
107.040686
-3.215224

Pt 3 x 1 Position
-1.747805
44.586360
-87.648108

Ppivot 3 x 1 Position
-253.032904
107.040686
-3.215224

Ptest 3 x 1 Position
-0.106450
0.191828
-0.098577
```

Listing 9: Ausgabe

```
Ptest 3 x 1 Position
0.043934
-0.236217
-0.109208

Standardabweichung 0.443922
Schreibe Datensatz nach Kalibrierung.txt
```

Listing 10: Ausgabe

Wie erwartet gilt für jede Messpose $P_{test} \approx \vec{0}$.