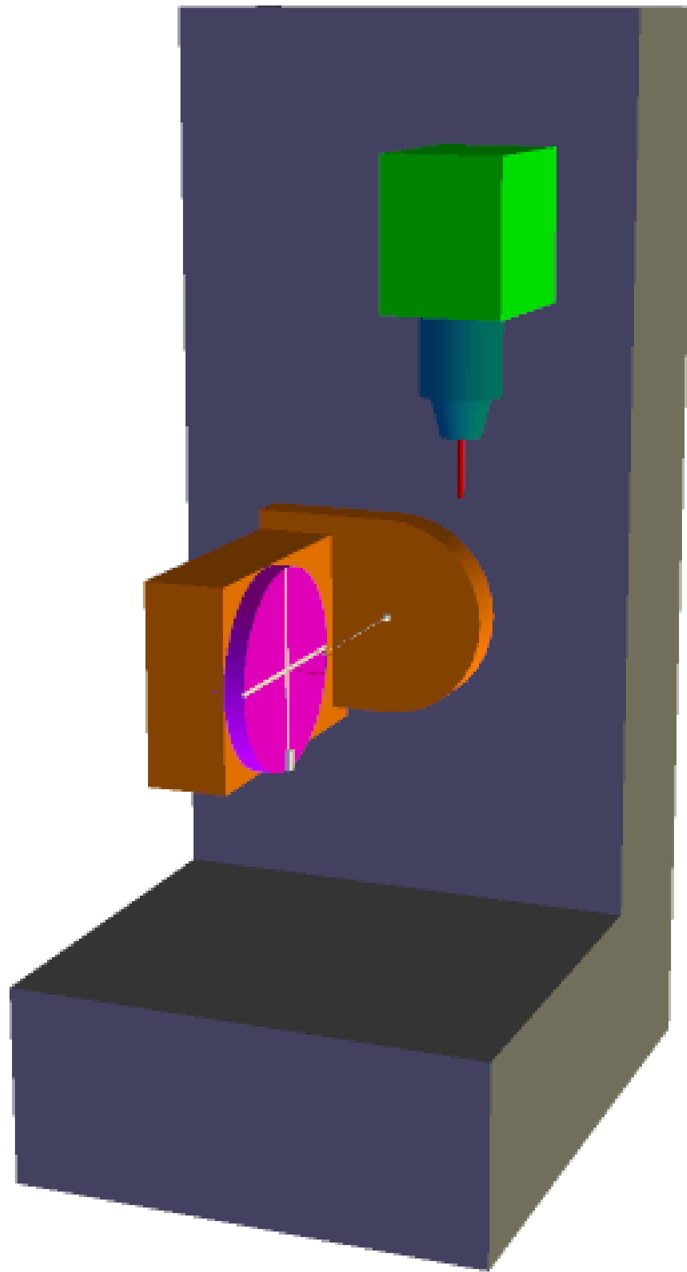
XYZAB_TDR kinematics for LinuxCNC

David Mueller

February 4, 2023



1 Introduction

This paper describes how to derive the kinematic model for a 5-axis machine tool in an XYZAB configuration with dual table rotation. In this example the B axis is the primary and the A axis is the secondary rotary axis. The primary being independent of the secondary axis. The model presented here will be named 'XYZAB-TDR'.

The method used here will be a step by step approach. Starting with a working kinematic model for a single rotary axis all the required elements will be added to build a complete kinematic model for the machine.

The final model includes tool-length compensation, compensation for mechanical offsets between the two rotational axis A and B and compensation for setups where the machine reference point is not located in the rotation-point of the rotary assembly.

In this document only basic mathematical functions are used so the kinematic models derived can be used directly in the 'userkins.comp' template file provided with the LinuxCNC installation. All calculations can be done whithout the use of any computer algebra systems, however the use of computer assistance like 'sage' will make the process of matrix multiplication much less error prone.

Note that there are other and potentially more computationally efficient ways to build custom kinematics using built in libraries like 'posemath'. Posemath provides many functions for efficient matrix manipulation and also offers functions for the use of quaternions. However the use of such a library would require a more indepth understanding of the mathematical theory that is beyond the scope of this presentation. Furthermore importing a library like 'posemath' into the 'userkins.comp' template would require substantially more programming skills than using the method applied in this paper.

A custom kinematic model in LinuxCNC is used to calculate cartesian coordinates from given machine joint positions (forward kinematics) and also to calculate the required machine joint positions to reach a given coordinate position (inverse kinematics). In the following description we will use vectors as mathematical representations of the two positions:

$$Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} \text{ Cartesian position} \qquad P = \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} \text{ Joint position} \tag{1}$$

Note that the fourth row is added to be able to multiply the vectors with a 4x4 transformation matrix.

2 TCP Kinematic model

For the tool to follow a point on the work piece we need a model that calculates where a given position on the work piece moves to when the rotary joints are rotated. In our example configuration the work piece will be mounted on the A rotary table and it is therefore here where we start to build our forward kinematic model. Note that in matrix multiplication the order is important that is $A \cdot B$ is generally not equal to $B \cdot A$.

2.1 Rotary A

2.1.1 Forward transformation

We start with the basic rotation around the A axis. In this case our forward transformation matrix ${}^{Q}A_{P}$ is equal to a rotation around the x-axis:

$$ {}^{Q}A_{P} = R_a \tag{2} $$

$$ R_a = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & Ca & -Sa & 0 \\ 0 & Sa & Ca & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

with $Sa = sin(\theta_a)$, $Ca = cos(\theta_a)$ and θ_a being the angle of rotation of joint A

To derive the coordinate position $Q(Qx, Qy, Qz)$ we now need to multiply the joint position vector $P(Px, Py, Pz)$ with our forward transformation matrix ${}^{Q}A_{P}$. Note that the input values P to our model need to be on the right hand side of the matrix multiplication.

$$ Q = {}^{Q}A_{P} \cdot P $$

$$ Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & Ca & -Sa & 0 \\ 0 & Sa & Ca & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} = \begin{pmatrix} Px \\ CaPy - PzSa \\ CaPz + PySa \\ 1 \end{pmatrix} $$

This shows the section of the TCP forward kinematics calculation in the file 'xyzab_tdr_kins.comp'. Note that P(px,py,pz) is equal to the joint position (j[0],j[1],j[2],) while $Q(Qx, Qy, Qz)$ is output to the coordinate position (pos->tran.x, pos->tran.y, pos->tran.z). The values 'ca', 'sa' are calculated and stored in the respective variables earlier in the component file.

```
case 1: // ======================== TCP kinematics FORWARD
        px          = j[0];
        py          = j[1];
        pz          = j[2];

        pos->tran.x = px;
        pos->tran.y = ca*py - sa*pz;
        pos->tran.z = ca*pz + sa*py;

        pos->a      = j[3];
        pos->b      = j[4];

        break;
```

2.1.2 Inverse transformation

To calculate the joint position P from given coordinate positions Q we need to 'unrotate' joint A. Mathematically this means we need to transpose the rotation part in our transformation matrix.

$$ {}^{P}A_{Q} = R_{a}^{T} \tag{3} $$

To derive the joint position P we then multiply the coordinate position vector Q from the right:

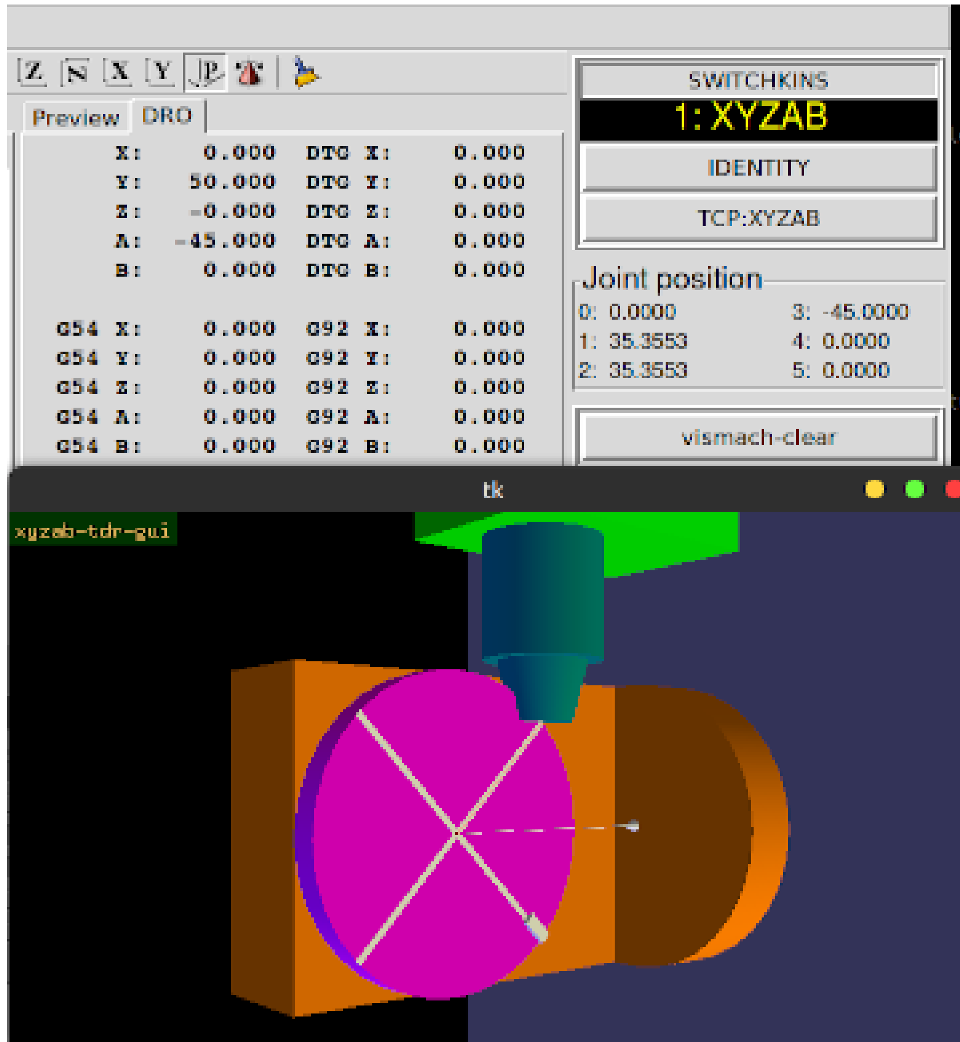
$$ P = {}^{P}A_{Q} \cdot Q \tag{4} $$

$$ P = \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & Ca & Sa & 0 \\ 0 & -Sa & Ca & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \begin{pmatrix} Qx \\ CaQy + QzSa \\ CaQz - QySa \\ 1 \end{pmatrix} $$

This shows the section of the TCP inverse kinematics calculation in the file 'xyzab_tdr_kins.comp'. Note that $Q(qx, qy, qz)$ is equal to the coordinate position (pos->tran.x, pos->tran.y, pos->tran.z) while $P(Px, Py, Pz)$ is output to the joint position (j[0],j[1],j[2]). The values 'ca', 'sa' are calculated and stored in the respective variables earlier in the component file.

```
case 1: // ========================= TCP kinematics INVERSE
    qx   = pos->tran.x;
    qy   = pos->tran.y;
    qz   = pos->tran.z;

    j[0] =   qx;

    j[1] =   ca*qy + qz*sa;

    j[2] =   ca*qz - qy*sa;

    j[3] = pos->a;
    j[4] = pos->b;
    break;
```

Testing in LinuxCNC shows a working TCP kinematic for rotations around A



2.2 Rotary A and B

2.2.1 Forward transformation

To add the B rotation we expand ${}^{Q}A_{P}$ by multiplying the rotation matrix R_b from the right:

$$ {}^{Q}A_{P} = R_a \cdot R_b \tag{5} $$

Note how the transformation matrix ${}^{Q}A_{P}$ is constructed from left to right. The first operation is on the left and the last operation is on the right as we work our way from the work side to the spindle side of the kinematic chain.

$$ R_b = \begin{pmatrix} Cb & 0 & Sb & 0 \\ 0 & 1 & 0 & 0 \\ -Sb & 0 & Cb & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

with $Sb = sin(\theta_b)$, $Cb = cos(\theta_b)$ and θ_b being the angle of rotation of joint B

$$^{Q}A_P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & Ca & -Sa & 0 \\ 0 & Sa & Ca & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Cb & 0 & Sb & 0 \\ 0 & 1 & 0 & 0 \\ -Sb & 0 & Cb & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$^{Q}A_P = \begin{pmatrix} Cb & 0 & Sb & 0 \\ SaSb & Ca & -CbSa & 0 \\ -CaSb & Sa & CaCb & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Note how the input data (this here being the forward kinematics the input is the joint position P is multiplied with the transformation matrix from the right to get the result (ie the coordinate position)

$$Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = {}^{Q}A_P \cdot P$$

$$Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \begin{pmatrix} Cb & 0 & Sb & 0 \\ SaSb & Ca & -CbSa & 0 \\ -CaSb & Sa & CaCb & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix}$$

$$Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \begin{pmatrix} Px \\ CaPy - PzSa \\ CaPz + PySa \\ 1 \end{pmatrix}$$

```
case 1: // ========================= TCP kinematics FORWARD
    px          = j[0];
    py          = j[1];
    pz          = j[2];

    pos->tran.x = cb*px + pz*sb;

    pos->tran.y = -cb*pz*sa + px*sa*sb + ca*py;

    pos->tran.z = ca*cb*pz -ca*px*sb + sa*py;

    pos->a      = j[3];
    pos->b      = j[4];

    break;
```

2.2.2 Inverse transformation

To calculate the joint position P from given coordinate positions Q we need to first unrotate joint B and then unrotate joint A. Mathematically we can write this:

$$ {}^{P}A_{Q} = R_b^T \cdot R_a^T \tag{6} $$

Note how the inverse transformation matrix ${}^{P}A_{A}$ is also constructed from left to right. The first operation is on the left and the last operation is on the right but here we start from the spindle side backwards to the work side.

To derive the joint position P we then multiply the coordinate position vector Q from the right:

$$ {}^{P}A_{Q} = \begin{pmatrix} Cb & 0 & -Sb & 0 \\ 0 & 1 & 0 & 0 \\ Sb & 0 & Cb & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & Ca & Sa & 0 \\ 0 & -Sa & Ca & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

$$ {}^{P}A_{Q} = \begin{pmatrix} Cb & SaSb & -CaSb & 0 \\ 0 & Ca & Sa & 0 \\ Sb & -CbSa & CaCb & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

$$ P = {}^{P}A_{Q} \cdot Q \tag{7} $$

Note again how the input data (this here being the inverse kinematics the input is the coordinate position Q) is multiplied with the transformation from the right.

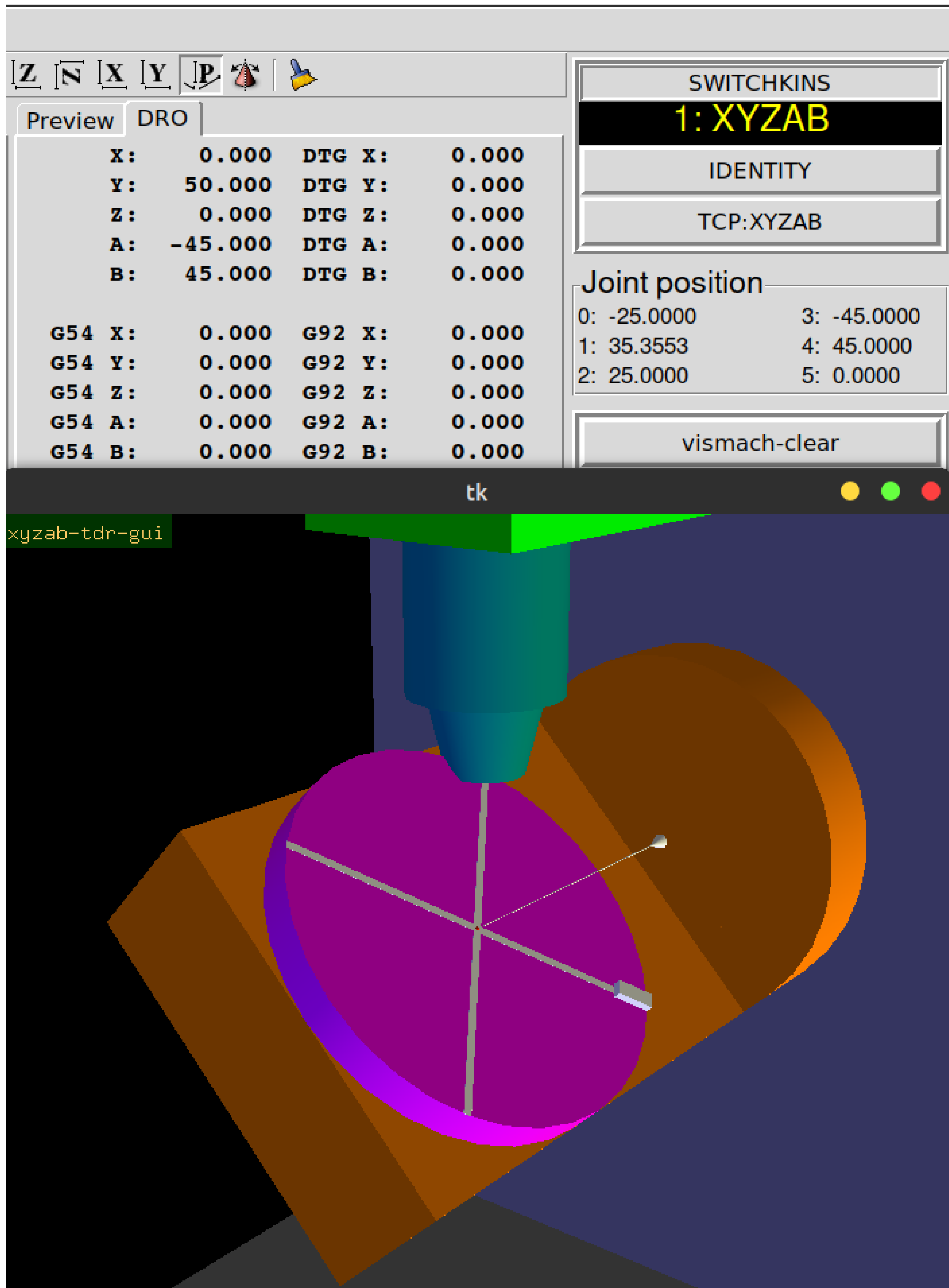
$$ P = \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} = \begin{pmatrix} Cb & SaSb & -CaSb & 0 \\ 0 & Ca & Sa & 0 \\ Sb & -CbSa & CaCb & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} $$

$$ P = \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} = \begin{pmatrix} -CaQzSb + QySaSb + CbQx \\ CaQy + QzSa \\ CaCbQz - CbQySa + QxSb \\ 1 \end{pmatrix} $$

```
case 1: // ========================= TCP kinematics INVERSE
    qx   = pos->tran.x;
    qy   = pos->tran.y;
    qz   = pos->tran.z;

    j[0] =   -ca*qz*sb + qy*sa*sb + cb*qx;

    j[1] =   ca*qy+ qz*sa;

    j[2] =   ca*cb*qz - cb*qy*sa + qx*sb;

    j[3] = pos->a;
```

```
j[4] = pos->b;
break;
```

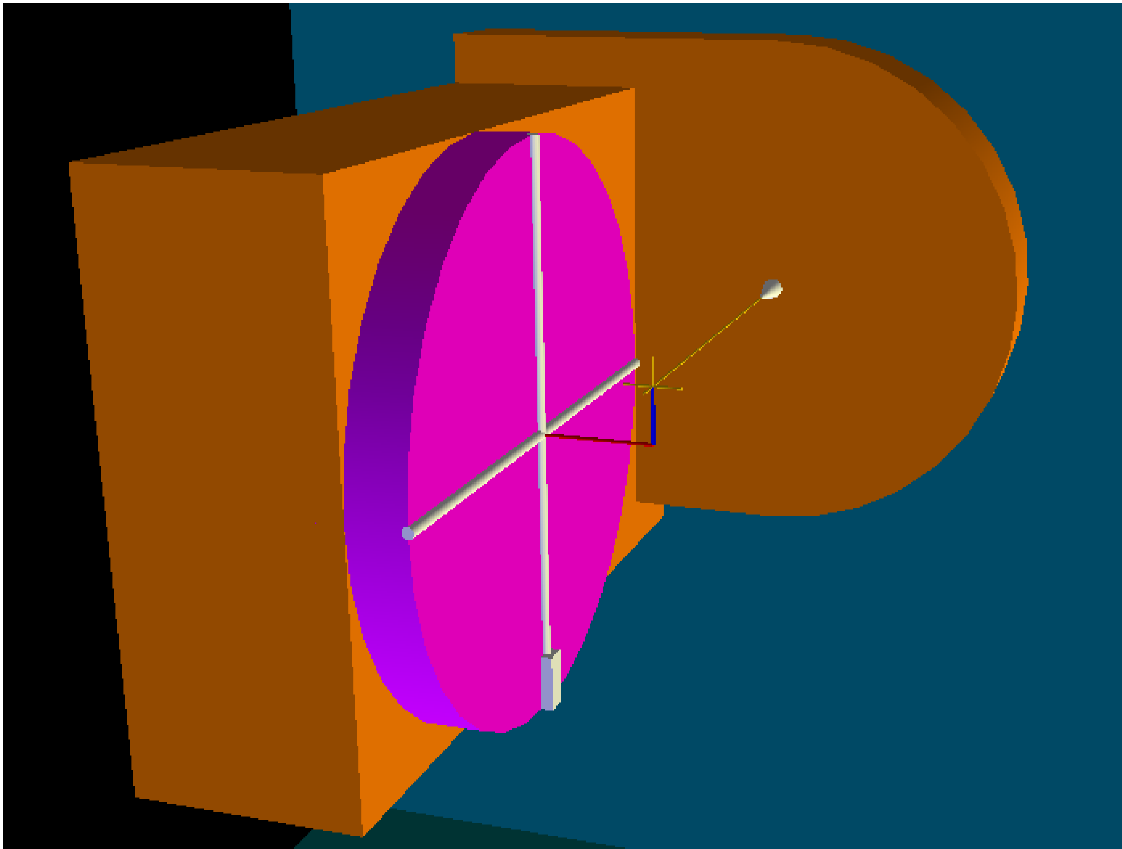
Testing in LinuxCNC now shows a working TCP kinematic for rotations of both joints A and B



2.3 Offsets in the rotary assembly

The rotational axes of a rotary assembly like the A/B type discussed here will always have an intended or unintented offset from one rotational axis to the other. In our case this is an offset in the x direction (red indicator) and in the z direction (blue indicator). In the image the yellow rod indicates the rotational axis of B and the yellow cross at it's end indicates where the axis A and axis B intersect when both x- and z-offset are zero this is also called the 'rotation point' of the rotary assembly. We define the values for the offsets in the example image as *x-offset = -20* and *z-offset = -10*. In our kinematic model this represents the situation where, starting from the rotation point of the rotary assembly, we need to travel 10 in the negative z-direction and 20 in the negative x-direction to reach the center of the face of the rotary A.

Note that the direction of travel when defining these offsets is arbitrary so in our case the offset situation in the image could also be defined as +20 in x and +10 in z. However once the definition is made we must keep it through the entire process of building the kinematic model.



2.3.1 Forward transformation

Because these offsets are located in between the two rotations they also need to be built in between the rotations in our transformation matrix. It may be helpful to view the offsets as the components

of a vector. In our case the vector components would be (-20,0,-10) so the vector would point from the rotation point to the face of the rotary table A. To build the transformation matrix that describes these offsets we need to keep in mind if we are moving with or against the offset vector as we travel from the work to the spindle. In our case we defined the offset vector to point towards the rotary A so we are travelling in the opposite direction and thus the vector components need to be entered in the negative.

$$^{Q}A_{P} = R_a \cdot T_o \cdot R_b \tag{8}$$

$$T_o = \begin{pmatrix} 1 & 0 & 0 & -Dx \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -Dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$^{Q}A_{P} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & Ca & -Sa & 0 \\ 0 & Sa & Ca & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -Dx \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -Dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Cb & 0 & Sb & 0 \\ 0 & 1 & 0 & 0 \\ -Sb & 0 & Cb & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$^{Q}A_{P} = \begin{pmatrix} Cb & 0 & Sb & -Dx \\ SaSb & Ca & -CbSa & DzSa \\ -CaSb & Sa & CaCb & -CaDz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = {}^{Q}A_{P} \cdot P$$

$$Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \begin{pmatrix} Cb & 0 & Sb & -Dx \\ SaSb & Ca & -CbSa & DzSa \\ -CaSb & Sa & CaCb & -CaDz \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix}$$

$$Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \begin{pmatrix} CbPx + PzSb - Dx \\ -CbPzSa + PxSaSb + CaPy + DzSa \\ CaCbPz - CaPxSb - CaDz + PySa \\ 1 \end{pmatrix}$$

```
case 1: // ========================= TCP kinematics FORWARD
    px          = j[0];
    py          = j[1];
    pz          = j[2];

    pos->tran.x = cb*px + pz*sb - dx;

    pos->tran.y = -cb*pz*sa + px*sa*sb + ca*py + dz*sa;

    pos->tran.z = ca*cb*pz - ca*px*sb - ca*dz + sa*py;
```

```
pos->a      = j[3];
pos->b      = j[4];

break;
```

2.3.2 Inverse transformation

For the inverse transformation we are moving in the opposite direction and need to reverse the vector translation:

$$ {}^{P}A_{Q} = R_b^T \cdot T_{io} \cdot R_a^T \tag{9} $$

$$ T_io = \begin{pmatrix} 1 & 0 & 0 & Dx \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & Dz \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

$$ {}^{P}A_{Q} = \begin{pmatrix} Cb & 0 & -Sb & 0 \\ 0 & 1 & 0 & 0 \\ Sb & 0 & Cb & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & Dx \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & Dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & Ca & Sa & 0 \\ 0 & -Sa & Ca & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

$$ {}^{P}A_{Q} = \begin{pmatrix} Cb & SaSb & -CaSb & CbDx - DzSb \\ 0 & Ca & Sa & 0 \\ Sb & -CbSa & CaCb & CbDz + DxSb \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

To derive the joint position P we then multiply the coordinate position vector Q from the right:

$$ P = {}^{P}A_{Q} \cdot Q \tag{10} $$

$$ P = \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} = \begin{pmatrix} Cb & SaSb & -CaSb & CbDx - DzSb \\ 0 & Ca & Sa & 0 \\ Sb & -CbSa & CaCb & CbDz + DxSb \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} $$

$$ P = \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} = \begin{pmatrix} -CaQzSb + QySaSb + CbDx + CbQx - DzSb \\ CaQy + QzSa \\ CaCbQz - CbQySa + CbDz + DxSb + QxSb \\ 1 \end{pmatrix} $$

```
case 1: // ======================== TCP kinematics INVERSE
    qx   = pos->tran.x;
    qy   = pos->tran.y;
    qz   = pos->tran.z;

    j[0] =   -ca*qz*sb + qy*sa*sb + cb*dx + cb*qx - dz*sb;
```

```
j[1] =   ca*qy+ qz*sa;

j[2] =   ca*cb*qz - cb*qy*sa + cb*dz + dx*sb + qx*sb;

j[3] = pos->a;
j[4] = pos->b;
break;
```

Testing in LinuxCNC shows a working TCP kinematic for rotations of joints A and B and the machine reference point has now shifted from the rotation point of the rotary assembly to the face center of the rotary A.

Preview DRO

```
     X:    0.000  DTG X:    0.000
     Y:   -0.000  DTG Y:    0.000
     Z:   -0.000  DTG Z:    0.000
     A:    0.000  DTG A:    0.000
     B:    0.000  DTG B:    0.000

 G54 X:    0.000  G92 X:    0.000
 G54 Y:    0.000  G92 Y:    0.000
 G54 Z:    0.000  G92 Z:    0.000
 G54 A:    0.000  G92 A:    0.000
 G54 B:    0.000  G92 B:    0.000
 G54 R:    0.000

 TLO X:    0.000
 TLO Y:    0.000
 TLO Z:    0.000
 TLO A:    0.000
 TLO B:    0.000
  Vel:     0.000
```

Go

SWITCHKINS

1: XYZAB

IDENTITY

TCP:XYZAB

Joint position

0:	-20.0000	3:	0.0000
1:	-0.0000	4:	0.0000
2:	-10.0000	5:	0.0000

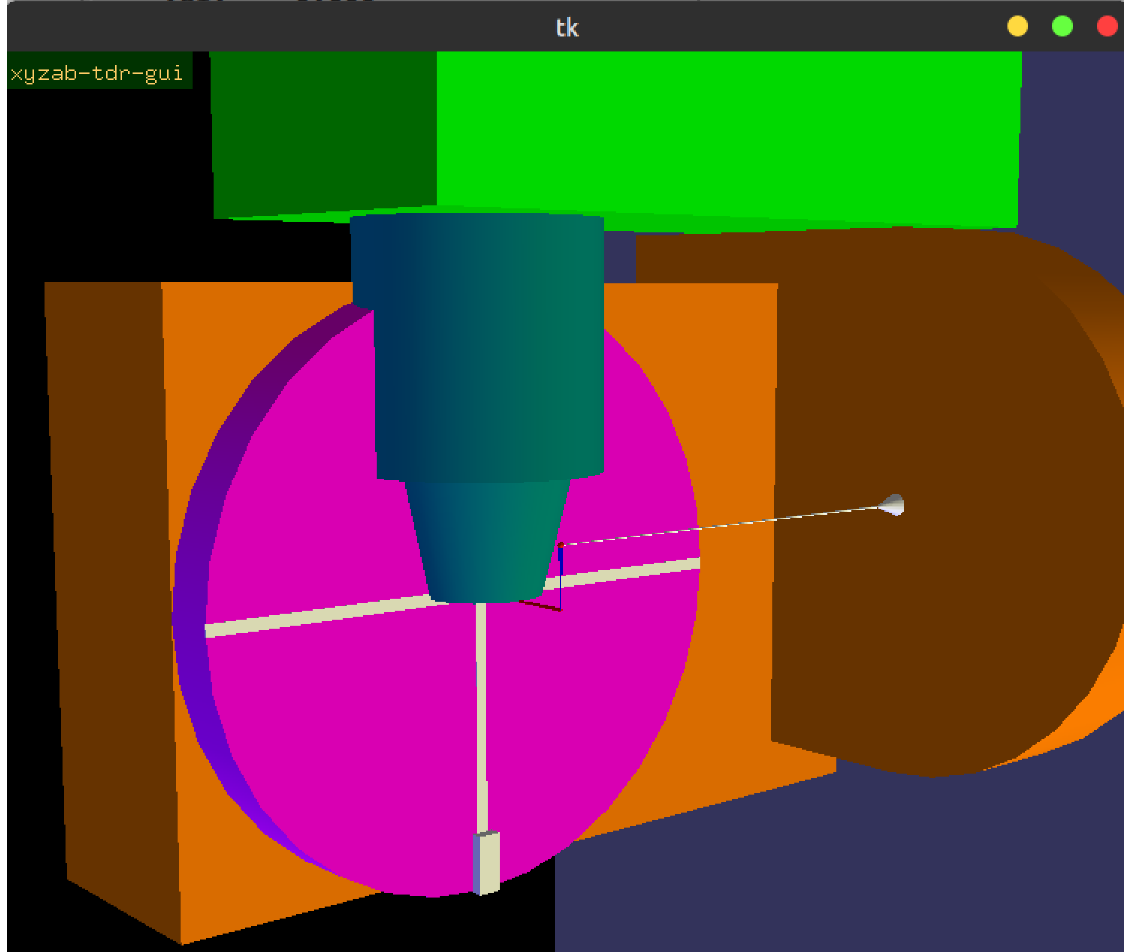
vismach-clear

Rotary Assembly Offset

x-offset: -20

z-offset: -10

Use hover + mouse wheel to change values



2.4 Shifting the reference point back to the rotation point

For the purpose of this example let's say we would like the machine reference to remain in the rotation point.

2.4.1 Forward kinematic

We set $P = (Px, Py, Pz) = (0, 0, 0)$ and $\theta_a = \theta_b = 0$ which gives $Sa = Sb = 0,\ Ca = Cb = 1$ With these input values our current forward kinematic

$$Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \begin{pmatrix} CbPx + PzSb - Dx \\ -CbPzSa + PxSaSb + CaPy + DzSa \\ CaCbPz - CaPxSb - CaDz + PySa \\ 1 \end{pmatrix}$$

results in $Q(-Dx, 0, -Dz)$ Which is the reason our machine reference point has been moved to the face center of the rotary A. So in order to move the machine reference back to the rotation-point of the rotary assembly we need to add the offset values (Dx, Dz) to the result of our forward kinematic which we can do in the form of a vector translation

$$T_io = \begin{pmatrix} 1 & 0 & 0 & Dx \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & Dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

that is multiplied with our forward transformation matrix from the left.

$$^{Q}A_P = T_io \cdot R_a \cdot T_o \cdot R_b \tag{11}$$

$$^{Q}A_P = \begin{pmatrix} 1 & 0 & 0 & Dx \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & Dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & Ca & -Sa & 0 \\ 0 & Sa & Ca & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -Dx \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -Dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Cb & 0 & Sb & 0 \\ 0 & 1 & 0 & 0 \\ -Sb & 0 & Cb & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$^{Q}A_P = \begin{pmatrix} Cb & 0 & Sb & 0 \\ SaSb & Ca & -CbSa & DzSa \\ -CaSb & Sa & CaCb & -CaDz + Dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = {}^{Q}A_P \cdot P$$

$$Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \begin{pmatrix} Cb & 0 & Sb & 0 \\ SaSb & Ca & -CbSa & DzSa \\ -CaSb & Sa & CaCb & -CaDz + Dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix}$$

$$Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} \begin{pmatrix} CbPx + PzSb \\ -CbPzSa + PxSaSb + CaPy + DzSa \\ CaCbPz - CaPxSb - CaDz + PySa + Dz \\ 1 \end{pmatrix}$$

```
case 1: // ======================== TCP kinematics FORWARD
    px          = j[0];
    py          = j[1];
    pz          = j[2];

    pos->tran.x = cb*px + pz*sb;

    pos->tran.y = -cb*pz*sa + px*sa*sb + ca*py + dz*sa;

    pos->tran.z = ca*cb*pz - ca*px*sb - ca*dz + sa*py
                + dz;

    pos->a      = j[3];
    pos->b      = j[4];

    break;
```

2.4.2 Inverse transformation

In the inverse transformation

$$P = {}^{P}A_{Q} \cdot Q = R_b^T \cdot T_{io} \cdot R_a^T \cdot Q \tag{12}$$

we have to subtract the offset values Dx, Dz from the input values Q:

$$Q = \begin{pmatrix} -Dx + Qx \\ Qy \\ -Dz + Qz \\ 1 \end{pmatrix}$$

This is essentially the same as multiplying a translation vector to right side of our transformation matrix. To keep the math more readable we choose to subtract the values directly from the input.

$${}^{P}A_{Q} = \begin{pmatrix} Cb & SaSb & -CaSb & CbDx - DzSb \\ 0 & Ca & Sa & 0 \\ Sb & -CbSa & CaCb & CbDz + DxSb \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$P = \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} = \begin{pmatrix} Cb & SaSb & -CaSb & CbDx - DzSb \\ 0 & Ca & Sa & 0 \\ Sb & -CbSa & CaCb & CbDz + DxSb \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} -Dx + Qx \\ Qy \\ -Dz + Qz \\ 1 \end{pmatrix}$$

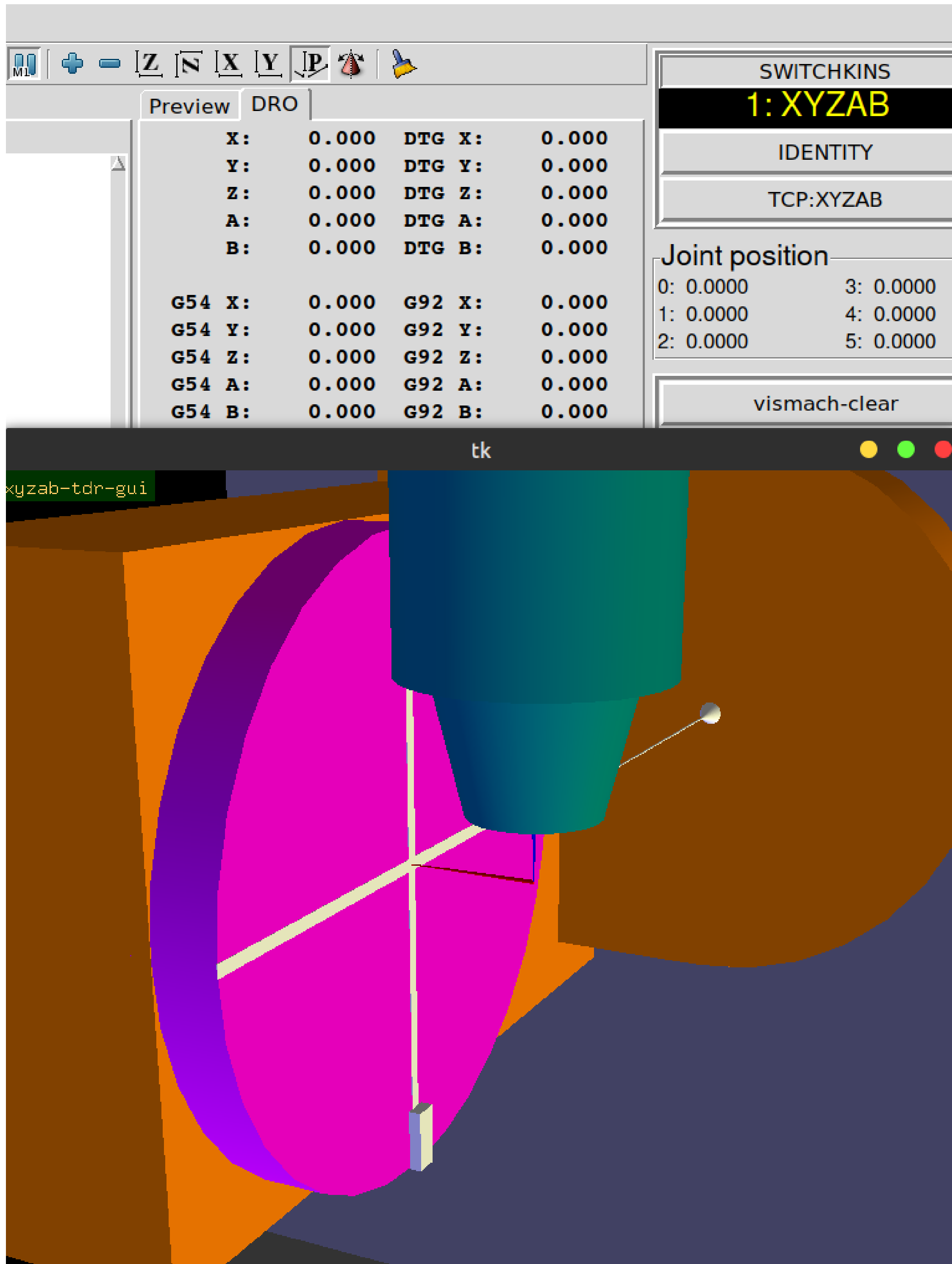
$$P = \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} = \begin{pmatrix} Ca(Dz - Qz)Sb + QySaSb - Cb(Dx - Qx) + CbDx - DzSb \\ CaQy - (Dz - Qz)Sa \\ -CaCb(Dz - Qz) - CbQySa + CbDz - (Dx - Qx)Sb + DxSb \\ 1 \end{pmatrix}$$

Note that in the kinematic component Dx is subtracted from the coordinate value *pos->tran.x* in the line *qx = pos->tran.x - dx;* and Dz is subtracted from the coordinate value *pos->tran.z* in the line *qz = pos->tran.z - dz;*

Also note that $(Dz - Qz) = -(Qz - Dz)$

```
case 1: // ======================== TCP kinematics INVERSE
    qx   = pos->tran.x - dx;
    qy   = pos->tran.y;
    qz   = pos->tran.z - dz;

    j[0] =   -ca*qz*sb + qy*sa*sb + cb*dx + cb*qx - dz*sb;

    j[1] =   ca*qy+ qz*sa;

    j[2] =   ca*cb*qz - cb*qy*sa + cb*dz + dx*sb + qx*sb;

    j[3] = pos->a;
    j[4] = pos->b;
    break;
```

Testing in LinuxCNC shows the machine reference point has now been shifted back to rotation point of the rotary assembly.

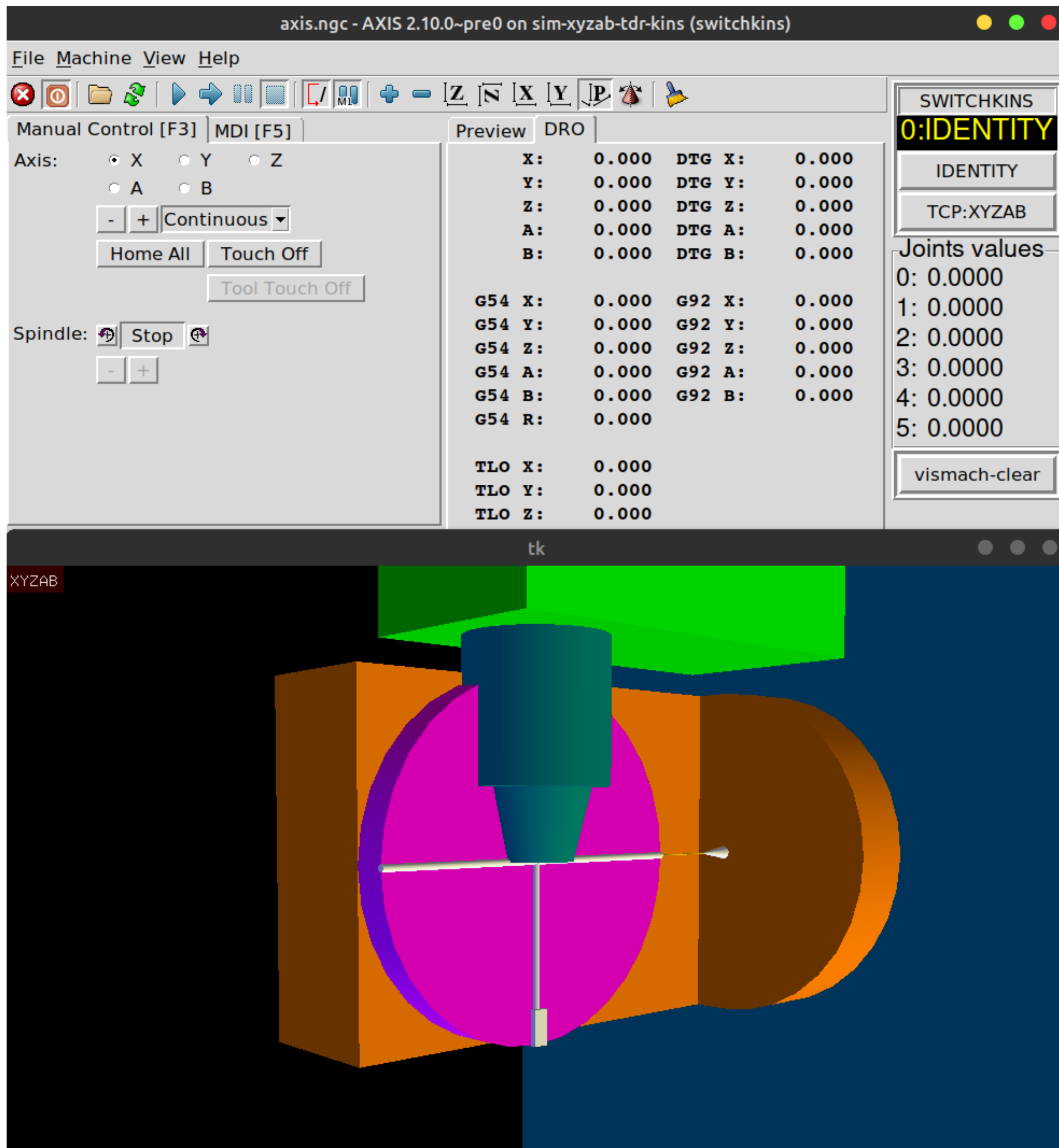

Preview
DRO
X: 0.000 DTG X: 0.000
Y: 0.000 DTG Y: 0.000
Z: 0.000 DTG Z: 0.000
A: 0.000 DTG A: 0.000
B: 0.000 DTG B: 0.000
G54 X: 0.000 G92 X: 0.000
G54 Y: 0.000 G92 Y: 0.000
G54 Z: 0.000 G92 Z: 0.000
G54 A: 0.000 G92 A: 0.000
G54 B: 0.000 G92 B: 0.000
SWITCHKINS
1: XYZAB
IDENTITY
TCP:XYZAB
Joint position
0: 0.0000
1: 0.0000
2: 0.0000
3: 0.0000
4: 0.0000
5: 0.0000
vismach-clear
tk
xyzab-tdr-gui

2.5 Tool length offset (TLO)

Tool-length compensation is applied automatically in LinuxCNC by subtracting the tool-length value Dt stored in the tool table from the the z-axis coordinate position while the joint position value remains unchanged. In it's current form our forward kinematic model assumes that the joint position P is identical to the tool position and uses the joint position value $P(Px, Py, Pz)$ to calculate the coordinate value $Q(Qx, Qy, Qz)$. Hence our kinematic model does not see a tool position change when TLO is activated by G43.

2.5.1 Forward kinematics

To illustrate the situation we assume that the machine coordinate system origin (ie G53 X0Y0Z0) coincides with the rotation point of our rotary assembly. Further assuming that a tool could be positioned in this location we would expect that without any tool-length compensation (G49) a tool positioned here (G0 X0Y0Z0) would be unaffected by any rotation of either A or B and that is indeed the case:



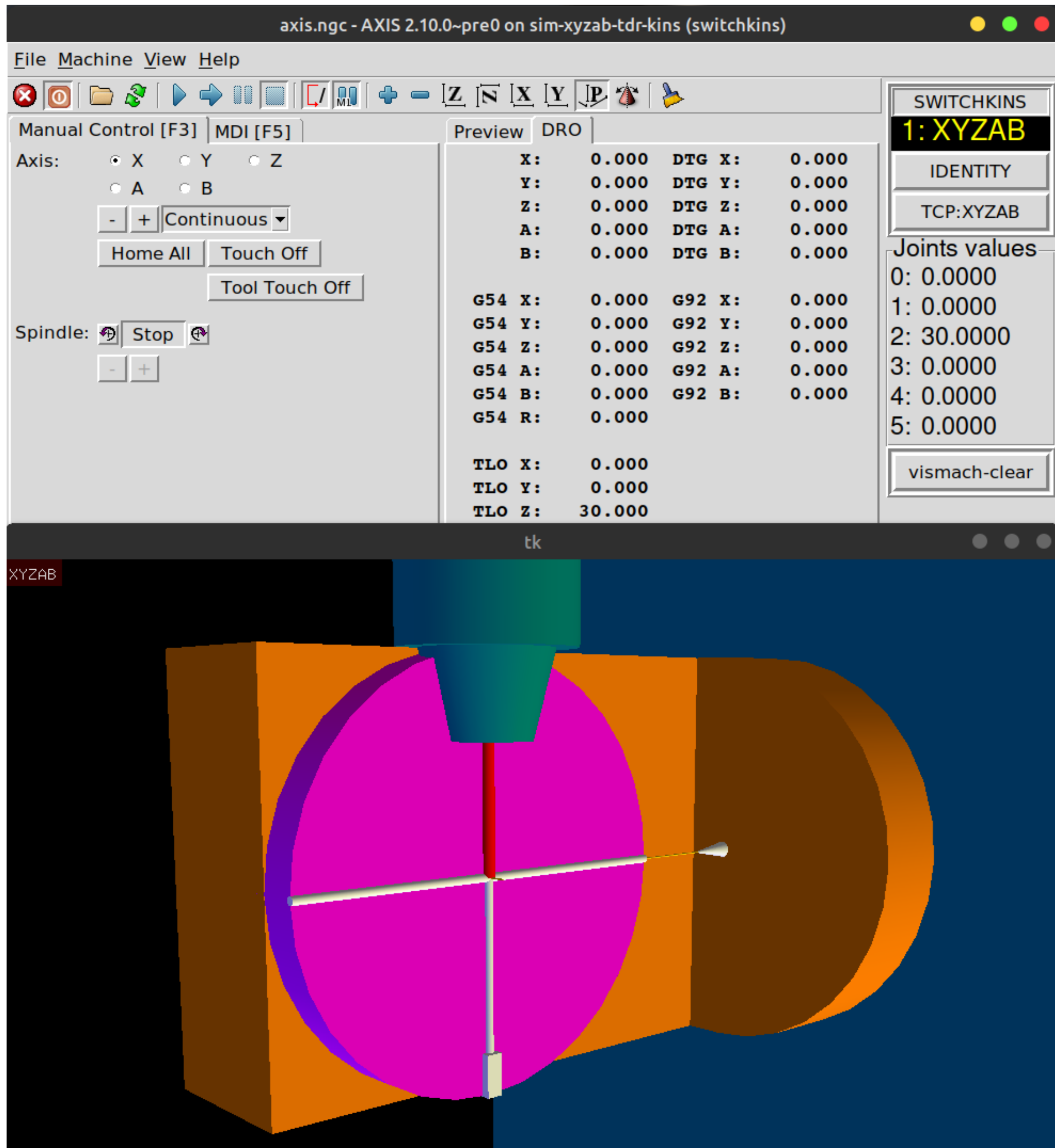
Let us consider this with our forward kinematic:

$$Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \begin{pmatrix} CbPx + PzSb \\ -CbPzSa + PxSaSb + CaPy + DzSa \\ CaCbPz - CaPxSb - CaDz + PySa + Dz \\ 1 \end{pmatrix}$$

It is easy to see that for $Dx = Dz = 0$ and $P(0,0,0)$ the result is indeed $Q(0,0,0)$ and this is what we expected.

However if we apply tool-length compensation (ie G43) and we move the tool to G0 X0Y0Z0 so the tool center point (TCP) is again positioned at the rotation-point we see that the input to our

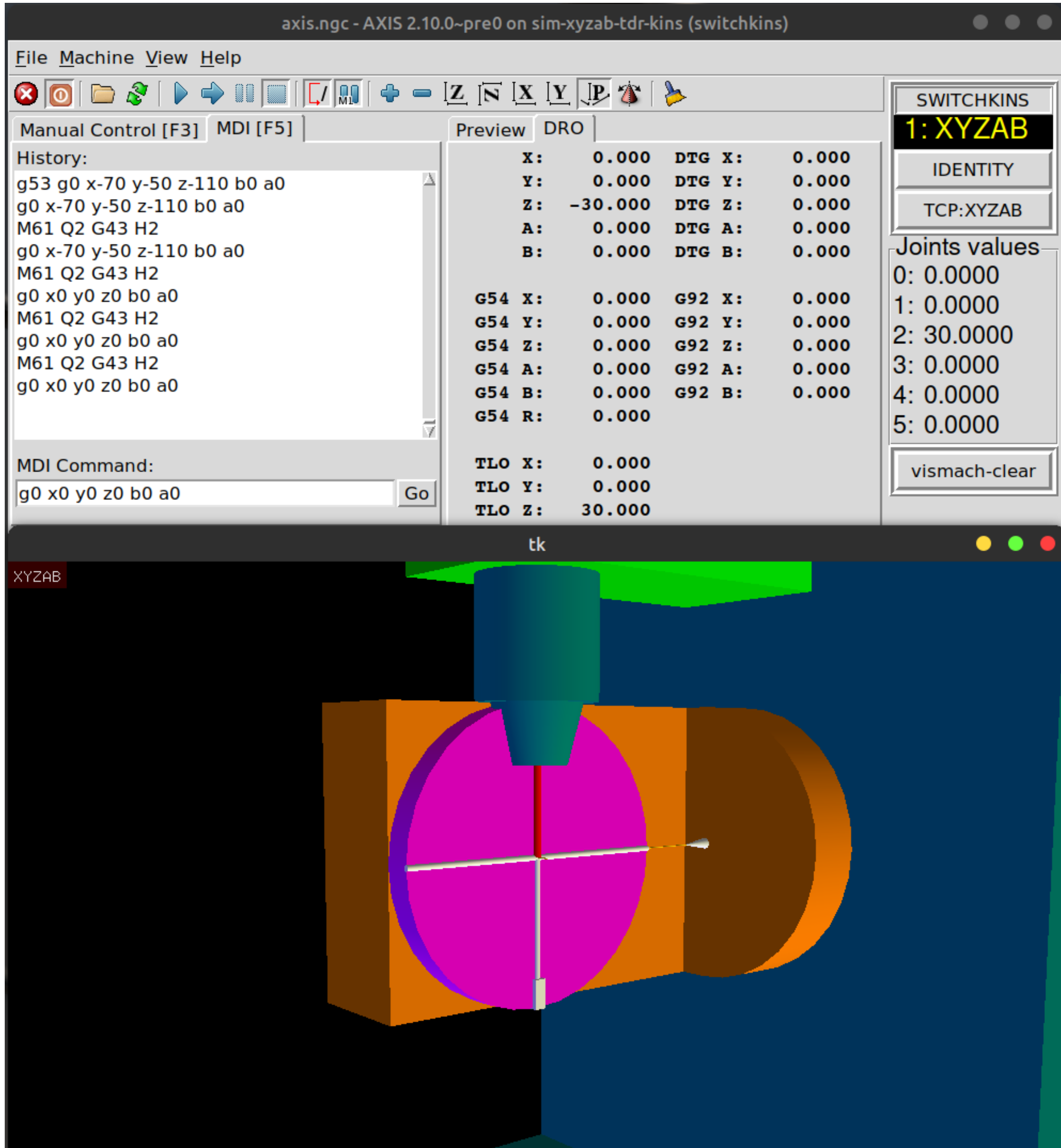
forward kinematic (ie the joint position) is not at $P(0,0,0)$ anymore:



Due to the way TLO is handled by LinuxCNC our joint-position is now *P(0,0,Dt)* while the DRO reads *Z: 0.000*. To solve this problem we need to subtract the tool-offset value *Dt* from the z joint position before feeding it into the forward kinematic. In this way our joint-position becomes $P(0,0,0)$ which will then give us the desired result of $Q(0,0,0)$.

$$P = \begin{pmatrix} Px \\ Py \\ -Dt + Pz \\ 1 \end{pmatrix}$$

Now however there is a new problem in that the DRO in LinuxCNC will show a z-axis position value equivalent to -*Dt*.



This is because LinuxCNC automatically subtracts the TLO value from the value of *pos->tran.z* which is the result of our forward kinematic. This can be fixed by adding *Dt* back to the result of our forward kinematic calculation which we can do by creating a vector translation:

$$Tt = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & Dt \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and multiply that to the left of our transformation matrix:

$$^{Q}A_P = \ T_t \cdot T_{io} \cdot R_a \cdot T_o \cdot R_b \tag{13}$$

$$^{Q}A_P \ = \ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & Dt \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & Dx \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & Dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & Ca & -Sa & 0 \\ 0 & Sa & Ca & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -Dx \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -Dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Cb & 0 & Sb & 0 \\ 0 & 1 & 0 & 0 \\ -Sb & 0 & Cb & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$^{Q}A_P = \begin{pmatrix} Cb & 0 & Sb & 0 \\ SaSb & Ca & -CbSa & DzSa \\ -CaSb & Sa & CaCb & -CaDz + Dt + Dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \ ^{Q}A_P \cdot P$$

$$Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \begin{pmatrix} Cb & 0 & Sb & Drpx \\ SaSb & Ca & -CbSa & DzSa + Drpy \\ -CaSb & Sa & CaCb & -CaDz + Drpz + Dt + Dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} -Drpx + Px \\ -Drpy + Py \\ -Drpz - Dt + Pz \\ 1 \end{pmatrix}$$

$$Q = \begin{pmatrix} -Cb(Drpx - Px) - (Drpz + Dt - Pz)Sb + Drpx \\ Cb(Drpz + Dt - Pz)Sa - (Drpx - Px)SaSb - Ca(Drpy - Py) + DzSa + Drpy \\ -CaCb(Drpz + Dt - Pz) + Ca(Drpx - Px)Sb - CaDz - (Drpy - Py)Sa + Drpz + Dt + Dz \\ 1 \end{pmatrix}$$

Note that in the kinematic component Dt is subtracted from the joint value *j[2]* in the line * pz = j[2]- dt;*

```
case 1: // ========================= TCP kinematics FORWARD
    px          = j[0];
    py          = j[1];
    pz          = j[2]- dt;

    pos->tran.x = cb*px + pz*sb;

    pos->tran.y = -cb*pz*sa + px*sa*sb + ca*py + dz*sa;

    pos->tran.z = ca*cb*pz - ca*px*sb - ca*dz + sa*py
                + dz + dt;

    pos->a      = j[3];
```

```
    pos->b      = j[4];

    break;
```

2.5.2 Inverse transformation

For the inverse transformation to match the new forward transformation we need to subtract Dt from the coordinate position Q and add it back to the result which we do by multiplying the translation by the vector Dt from the left:

$$P = \ {}^{P}A_{Q} \cdot Q = \ T_t \cdot R_b^T \cdot T_{io} \cdot R_a^T \cdot Q \tag{14}$$

$$Q = \begin{pmatrix} -Dx + Qx \\ Qy \\ -Dt - Dz + Qz \\ 1 \end{pmatrix}$$

$${}^{P}A_{Q} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & Dt \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Cb & 0 & -Sb & 0 \\ 0 & 1 & 0 & 0 \\ Sb & 0 & Cb & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & Dx \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & Dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & Ca & Sa & 0 \\ 0 & -Sa & Ca & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$${}^{P}A_{Q} = \begin{pmatrix} Cb & SaSb & -CaSb & CbDx - DzSb \\ 0 & Ca & Sa & 0 \\ Sb & -CbSa & CaCb & CbDz + DxSb + Dt \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$P = \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} = \ {}^{P}A_{Q} \cdot Q$$

$$P = \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} = \begin{pmatrix} Cb & SaSb & -CaSb & CbDx - DzSb \\ 0 & Ca & Sa & 0 \\ Sb & -CbSa & CaCb & CbDz + DxSb + Dt \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} -Dx + Qx \\ Qy \\ -Dt - Dz + Qz \\ 1 \end{pmatrix}$$

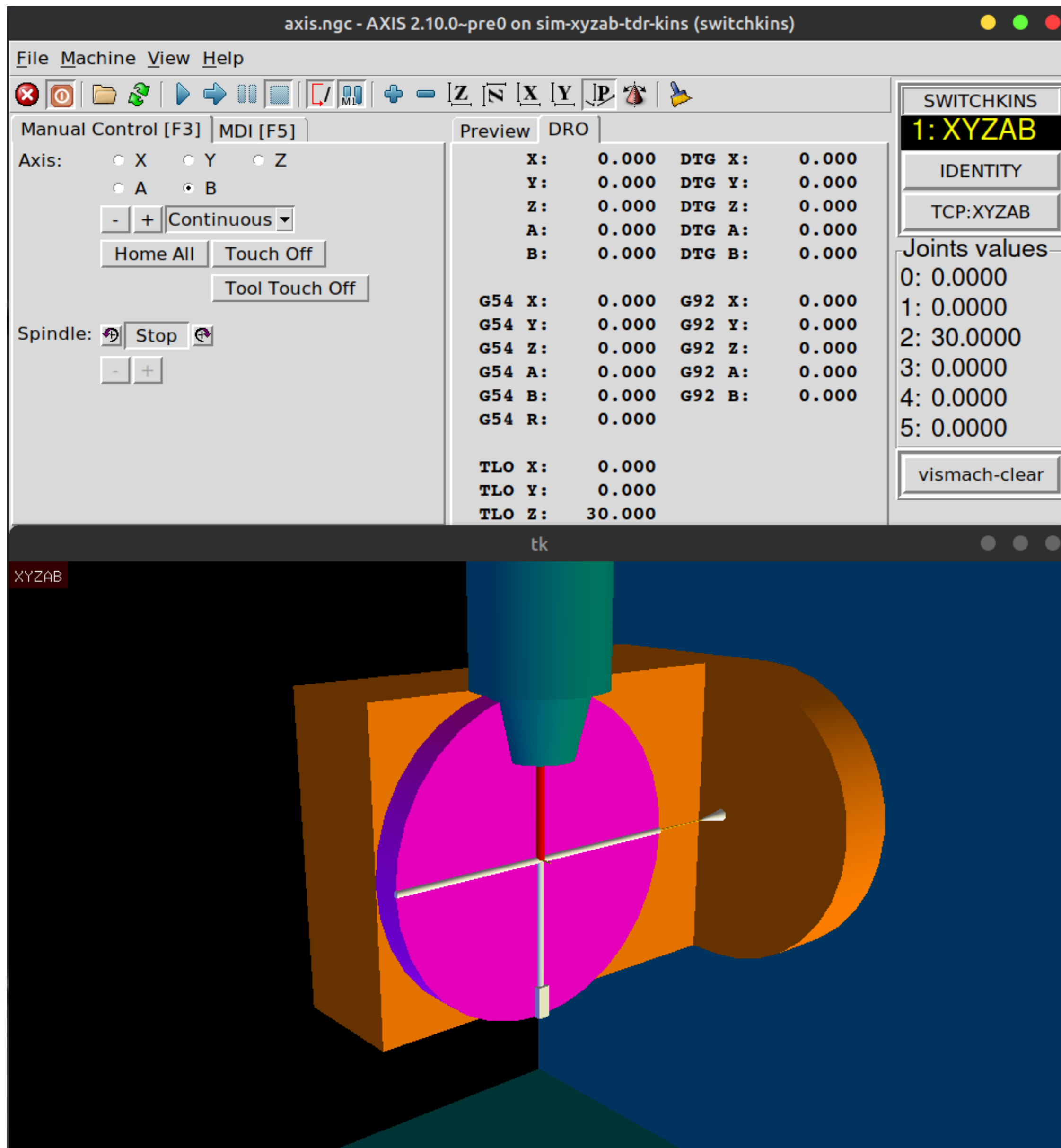
$$P = \begin{pmatrix} Ca(Dt + Dz - Qz)Sb + QySaSb - Cb(Dx - Qx) + CbDx - DzSb \\ CaQy - (Dt + Dz - Qz)Sa \\ -CaCb(Dt + Dz - Qz) - CbQySa + CbDz - (Dx - Qx)Sb + DxSb + Dt \\ 1 \end{pmatrix}$$

Note that in the kinematic component Dt is subtracted from the coordinate value *pos->tran.z* in the line *qz = pos->tran.z - dz - dt;*

```
case 1: // ======================== TCP kinematics INVERSE
    qx   = pos->tran.x - dx;
    qy   = pos->tran.y;
    qz   = pos->tran.z - dz - dt;
```

```
j[0] =   -ca*qz*sb + qy*sa*sb + cb*dx + cb*qx - dz*sb;

j[1] =   ca*qy+ qz*sa;

j[2] =   ca*cb*qz - cb*qy*sa + cb*dz + dx*sb + qx*sb
         + dt;

j[3] = pos->a;
j[4] = pos->b;
break;
```

Now everything looks correct in the DRO.



If our machine is setup so that the machine reference point coincides with the rotation point of our rotary-assembly then our forward kinematic is complete. Otherwise one more step is necessary to derive a correct kinematic model.

2.6 Position offset of the rotary assembly to the machine reference point

Up to this point we have assumed that the machine reference point coincides with the rotation point of our rotary-assembly or in case of applied geometric offsets (Dx, Dz) maybe the center of the face of our secondary rotary table A. For such a case our forward kinematic is complete. However, if the machine at hand is setup in a way where there is an offset between the machine reference point and the rotary-assembly then this will need to be taken into account in the kinematic model. What does such an offset mean for our forward kinematic?

2.6.1 Forward kinematic

Let us assume that we have a setup with no TLO $(dt = 0)$, no geometric offset $(Dz = Dx = 0)$ and no offset of the rotation-point. A tool positioned at the rotation-point would have a joint-position of $P(0, 0, 0)$ and that would give the expected resulting coordinate position of $Q(0, 0, 0)$. If we now assume that the rotation-point of our rotary-assembly is offset from the machine reference point by (rp_x, rp_y, rp_z) then our joint-position would be equal to the offset $P(rp_x, rp_y, rp_z)$ which would clearly not give us the required result of Q(0,0,0). So we need to subtract the offset (rp_x, rp_y, rp_z) from the joint-position $P = (Px - rp_x, Py - rp_y, Pz - rp_z)$ or in other words we need to translate the joint-position vector in the opposite direction along the offset vector.

$$P = \begin{pmatrix} -Drpx + Px \\ -Drpy + Py \\ -Drpz - Dt + Pz \\ 1 \end{pmatrix}$$

However some more consideration is needed for now a joint position of $P(rp_x, rp_y, rp_z)$ will result in a coordinate position of $Q(0, 0, 0)$ which is of course not the value we can hand back to LinuxCNC because if the rotation point is offset from machine zero then the coordinate position would be $Q(rp_x, rp_y, rp_z)$. So to be consistent we need to add the offset values back to the results of our calculations which we do again by multiplying a vector translation to the left of our forward kinematic:

$$^{Q}A_P = T_{rp} \cdot T_t \cdot T_{io} \cdot R_a \cdot T_o \cdot R_b \tag{15}$$

$$Trp = \begin{pmatrix} 1 & 0 & 0 & Drpx \\ 0 & 1 & 0 & Drpy \\ 0 & 0 & 1 & Drpz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$^{Q}A_P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & Dt \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & Dx \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & Dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & Ca & -Sa & 0 \\ 0 & Sa & Ca & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -Dx \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -Dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot$$

$$\begin{pmatrix} Cb & 0 & Sb & 0 \\ 0 & 1 & 0 & 0 \\ -Sb & 0 & Cb & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$^{Q}A_P = \begin{pmatrix} Cb & 0 & Sb & Drpx \\ SaSb & Ca & -CbSa & DzSa + Drpy \\ -CaSb & Sa & CaCb & -CaDz + Drpz + Dt + Dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = {}^{Q}A_P \cdot P$$

$$Q = \begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \begin{pmatrix} Cb & 0 & Sb & Drpx \\ SaSb & Ca & -CbSa & DzSa + Drpy \\ -CaSb & Sa & CaCb & -CaDz + Drpz + Dt + Dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} -Drpx + Px \\ -Drpy + Py \\ -Drpz - Dt + Pz \\ 1 \end{pmatrix}$$

$$Q = \begin{pmatrix} -Cb(Drpx - Px) - (Drpz + Dt - Pz)Sb + Drpx \\ Cb(Drpz + Dt - Pz)Sa - (Drpx - Px)SaSb - Ca(Drpy - Py) + DzSa + Drpy \\ -CaCb(Drpz + Dt - Pz) + Ca(Drpx - Px)Sb - CaDz - (Drpy - Py)Sa + Drpz + Dt + Dz \\ 1 \end{pmatrix}$$

Note that in the kinematic component $Drp(x, y, z)$ is named *(x,y,z)_rot_point* and subtracted from the respective joint values in the lines *px = j[0] - x_rot_point; py = j[1] - y_rot_point; pz = j[2] - z_rot_point - dt;*

```
case 1: // ========================= TCP kinematics FORWARD
    px          = j[0] - x_rot_point;
    py          = j[1] - y_rot_point;
    pz          = j[2] - z_rot_point - dt;

    pos->tran.x =   cb*px + sb*pz
                  + x_rot_point;

    pos->tran.y =   sa*sb*px + ca*py - cb*sa*pz + sa*dz
                  + y_rot_point;

    pos->tran.z = - ca*sb*px + sa*py + ca*cb*pz - ca*dz
                  + z_rot_point + dz + dt;

    pos->a      = j[3];
    pos->b      = j[4];
    pos->c      = j[5];
    break;
```

2.6.2 Inverse transformation

For the inverse transformation to match the new forward transformation we need mirror the modifications in the forward kinematics. That means to subtract $Drp(x, y, z)$ from the coordinate position Q

$$Q = \begin{pmatrix} -Drpx - Dx + Qx \\ -Drpx + Qy \\ -Drpx - Dz + Qz \\ 1 \end{pmatrix}$$

and add it back to the result which we do by multiplying the translation vector $Drp(x, y, z)$ to the inverse transformation from the left:

$$^{P}A_{Q} = T_{rp} \cdot \; T_t \cdot R_b^T \cdot T_{-o} \cdot R_a^T \cdot Q \tag{16}$$

$$^{P}A_{Q} = \begin{pmatrix} 1 & 0 & 0 & Drpx \\ 0 & 1 & 0 & Drpy \\ 0 & 0 & 1 & Drpz \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & Dt \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Cb & 0 & -Sb & 0 \\ 0 & 1 & 0 & 0 \\ Sb & 0 & Cb & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & Dx \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & Dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & Ca & Sa & 0 \\ 0 & -Sa & Ca & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$^{P}A_{Q} = \begin{pmatrix} Cb & SaSb & -CaSb & CbDx - DzSb + Drpx \\ 0 & Ca & Sa & Drpy \\ Sb & -CbSa & CaCb & CbDz + DxSb + Drpz + Dt \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$P = \; ^{P}A_{Q} \cdot Q \tag{17}$$

$$P = \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} = \begin{pmatrix} Cb & SaSb & -CaSb & CbDx - DzSb + Drpx \\ 0 & Ca & Sa & Drpy \\ Sb & -CbSa & CaCb & CbDz + DxSb + Drpz + Dt \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} -Drpx - Dx + Qx \\ -Drpx + Qy \\ -Drpx - Dz + Qz \\ 1 \end{pmatrix}$$

$$P = \begin{pmatrix} Ca(Drpx + Dz - Qz)Sb - (Drpx - Qy)SaSb - Cb(Drpx + Dx - Qx) + CbDx - DzSb + Drpx \\ -Ca(Drpx - Qy) - (Drpx + Dz - Qz)Sa + Drpy \\ -CaCb(Drpx + Dz - Qz) + Cb(Drpx - Qy)Sa + CbDz - (Drpx + Dx - Qx)Sb + DxSb + Drpz + Dt \\ 1 \end{pmatrix}$$

Note that in the kinematic component $Drp(x, y, z)$ is named *(x,y,z)_rot_point* and subtracted from the respective coordinate values in the lines *qx = pos->tran.x - x_rot_point - dx; qy = pos->tran.y - y_rot_point; qz = pos->tran.z - z_rot_point - dz - dt;*

```
case 1: // ======================== TCP kinematics INVERSE =====================
    qx   = pos->tran.x - x_rot_point - dx;
    qy   = pos->tran.y - y_rot_point;
    qz   = pos->tran.z - z_rot_point - dz - dt;
```

```
j[0] =   cb*qx + sa*sb*qy - ca*sb*qz + cb*dx - sb*dz
       + x_rot_point;

j[1] =   ca*qy + sa*qz
       + y_rot_point;

j[2] =   sb*qx - sa*cb*qy + ca*cb*qz + sb*dx + cb*dz
       + z_rot_point + dt;

j[3] = pos->a;
j[4] = pos->b;
break;
```

This completes the kinematic model.