Dobot Magician E6 — Forward Kinematics via Custom DH Table

Working notes: problem, derivation, and result

Prepared by: Aditya Phadtare
Company: Simusoft Technology
Date: July 17–18, 2026
System: Dobot Magician E6, RoboDK API driver, DobotStudio

---

1 Problem Statement

Driving the robot through the API driver did not show accurate position, because of a model inconsistency between RoboDK and DobotStudio – specifically for the Dobot E6. Although RoboDK can accept a target as either a joint value or a pose, it does not allow a position to be sent in place of a joint value for this driver path. As a result, **the driver itself has to compute the robot's forward kinematics** from the joint values that RoboDK provides, in order to know the actual end-effector position.

2 Solution Approach

RoboDK already ships a DH table for the E6 internally, but RoboDK solves it using the **Product of Exponentials (POE)** method. Continuing with that method would require solving the exponential product form directly:

$$T(\theta) = e^{[\xi_1]\theta_1}\, e^{[\xi_2]\theta_2}\, e^{[\xi_3]\theta_3} \cdots M$$

Instead, the more familiar route was used: **reconstruct the DH diagram and homogeneous transformation matrices from scratch** using the classical Denavit–Hartenberg convention, which is built directly from observing the positive rotation direction of each joint axis and has no dependency on the POE formulation.

3 DH Parameter Definitions

α — rotation about the x-axis, carried out from x_{n-1} to x_n

θ — rotation about the z-axis, carried out from z_{n-1} to z_n

d — distance between x_n and x_{n-1}, such that both x_n and x_{n-1} are coplanar

a — distance between z_n and z_{n-1}, such that both z_n and z_{n-1} are coplanar

Sign convention for each rotation follows the right-hand / corkscrew rule.

4 Reference: RoboDK's Built-In DH Table (POE-based)

For comparison, this is the DH table already built into RoboDK for the E6 (the one RoboDK solves internally with POE):

Link	α (deg)	a (mm)	θ (deg)	d (mm)
Joint 1	90	0	0	168.2
Joint 2	0	−189	−90	0
Joint 3	0	−160	0	90.5
Joint 4	90	0	−90	101
Joint 5	90	0	180	101
Joint 6	0	0	180	66

5 Custom Frame Assignment & Derivation

Frames {0} through {6} were assigned directly on the robot by observing the positive rotation direction of each joint's z-axis (corkscrew rule), independent of RoboDK's own convention.

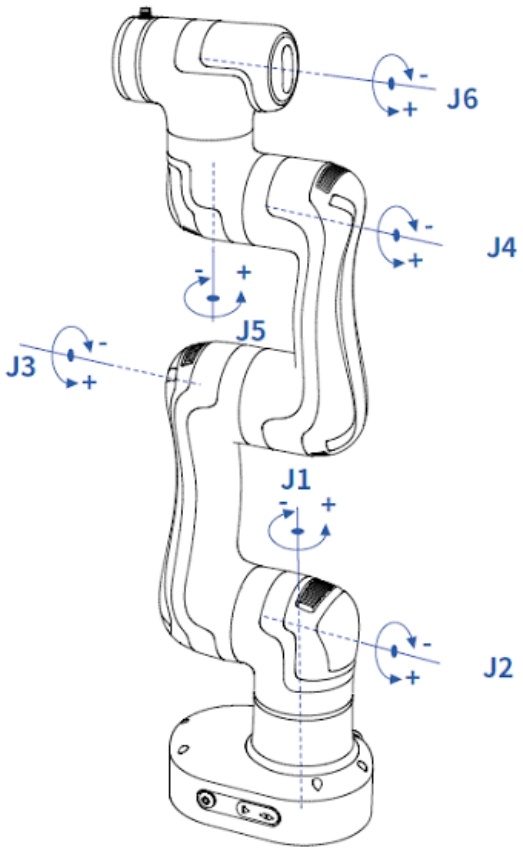


Figure: reference – positive/negative rotation direction for $J1$–$J6$ as marked on the physical Dobot E6.

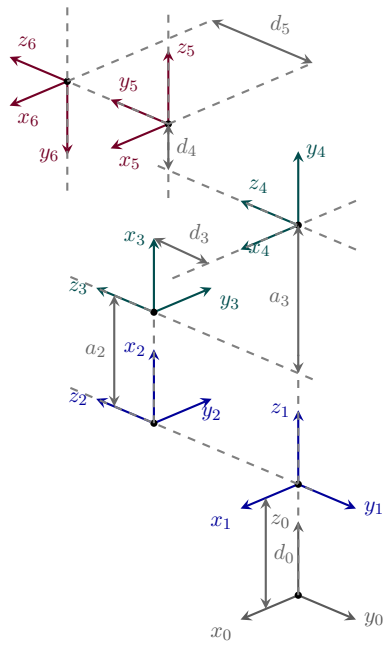
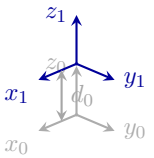
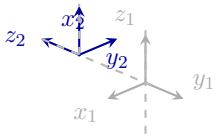


Figure: full kinematic diagram – frames {0}–{6}, joint axes (dashed) and DH offsets ($d_0, a_2, a_3, d_3, d_4, d_5$) in true 3D perspective (not to scale).

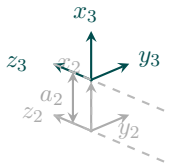
Each joint transition was worked through individually. α and θ for each step were read off directly from how much the x- and z-axes had to rotate between consecutive frames, using the corkscrew (right-hand) rule to fix the sign: curling the right-hand fingers from the old axis toward the new one, the thumb points along the positive rotation axis, and a positive angle follows that thumb direction.



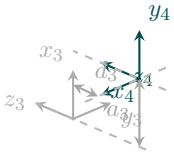
Step 1: $J_0 \to J_1$
α=0, θ=0
d=168.2, a=0



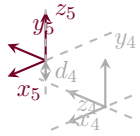
Step 2: $J_1 \to J_2$
α=90, θ=90
d=0, a=0



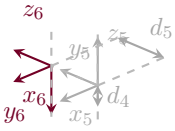
Step 3: $J_2 \to J_3$
α=0, θ=0
d=0, a=189



Step 4: $J_3 \to J_4$
α=0, θ=−90
d=90.5, a=160



Step 5: $J_4 \to J_5$
α=−90, θ=0
d=101, a=0



Step 6: $J_5 \to J_6$
α=90, θ=0
d=144, a=0
(suction-cup tool)

Figure: coordinate frames for Steps 1–6, taken directly from the full kinematic diagram above – solid arrows are the axes, dashed lines are joint axes / common normals, and double-headed arrows mark the DH offsets relevant to that step (not to scale).

Each step diagram shows the outgoing frame $(x_{i-1}, y_{i-1}, z_{i-1})$ and the incoming frame (x_i, y_i, z_i) together, using the same global coordinates as the full diagram; α_i is read as the angle swept from x_{i-1} to x_i about the connecting x-axis, and θ_i as the angle swept from z_{i-1} to z_i about the connecting z-axis, both signed by the corkscrew rule above. The resulting values per step are summarised below:

Step	**Transition**	α (deg)	θ (deg)	d (mm)	a (mm)
1	$J_0 \to J_1$	0	0	168.2	0
2	$J_1 \to J_2$	90	90	0	0
3	$J_2 \to J_3$	0	0	0	189
4	$J_3 \to J_4$	0	-90	90.5	160
5	$J_4 \to J_5$	-90	0	101	0
6	$J_5 \to J_6$	90	0	144	0

Note on Step 6: the flange frame was skipped and the end-effector frame taken directly, since the tool is a simple suction cup with no wrist offset to account for separately ($d = 66 + 80 = 144$ mm).

Combined DH Table

Collecting all six steps into a single table gives the custom DH parameters used for forward kinematics:

Link	α (deg)	θ (deg)	d (mm)	a (mm)
Joint 1	90	0	168.2	0
Joint 2	90	90	0	0
Joint 3	0	0	0	189
Joint 4	0	-90	90.5	160
Joint 5	-90	0	101	0
Joint 6	90	180	144	0

6 Homogeneous Transformation

Each link transformation follows the standard DH form,

$$A_i = R_{z,\theta_i}\,\mathrm{Trans}_{z,d_i}\,\mathrm{Trans}_{x,a_i}\,R_{x,\alpha_i},$$

which, evaluated symbolically for Joint 1 (θ variable, $\alpha = 90^\circ$, $a = 0$, $d = 168.2$), gives:

$$T_0^1 = \begin{bmatrix} \cos\theta & -\sin\theta\cos\alpha & \sin\theta\sin\alpha & 0 \\ \sin\theta & \cos\theta\cos\alpha & -\cos\theta\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 168.2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The remaining link transforms (T_1^2 through T_5^6) follow the same pattern using the parameters from the combined table above, and the full chain $T_0^6 = T_0^1\,T_1^2\,T_2^3\,T_3^4\,T_4^5\,T_5^6$ is evaluated programmatically in the driver to get the live end-effector pose from RoboDK's joint values.

7 Driver Implementation

The forward-kinematics chain above is implemented directly in the robot driver as a `ForwardKin` class: each joint's rotation matrix J_i is built from the live joint angle, each fixed link transform $T_{i-1,i}$ is hard-coded from the DH table, and `Position()` chains them together and converts the result to the pose format (`X, Y, Z, RX, RY, RZ`) the robot controller expects. `MovJ`, `MovL`, and `DO` then call into this to accept either joint or pose targets transparently.

Listing 1: `ForwardKin` class and updated `MovJ`/`MovL`/`DO` driver methods

```
class ForwardKin:
    def __init__(self, J1:float, J2:float, J3:float, J4:float, J5:float, J6:float):
        self.J1 = np.array([[c(r(J1)),-s(r(J1)), 0, 0],[ s(r(J1)),c(r(J1)), 0, 0],[0, 0, 1,
            0],[0,0, 0,1]
        ])
        self.J2 = np.array([[c(r(J2)),-s(r(J2)), 0, 0],[ s(r(J2)),c(r(J2)), 0, 0],[0, 0, 1,
            0],[0,0, 0,1]
        ])
        self.J3 = np.array([[c(r(J3)),-s(r(J3)), 0, 0],[ s(r(J3)),c(r(J3)), 0, 0],[0, 0, 1,
            0],[0,0, 0,1]
        ])
        self.J4 = np.array([[c(r(J4)),-s(r(J4)), 0, 0],[ s(r(J4)),c(r(J4)), 0, 0],[0, 0, 1,
            0],[0,0, 0,1]
        ])
        self.J5 = np.array([[c(r(J5)),-s(r(J5)), 0, 0],[ s(r(J5)),c(r(J5)), 0, 0],[0, 0, 1,
            0],[0,0, 0,1]
        ])
        self.J6 = np.array([[c(r(J6)),-s(r(J6)), 0, 0],[ s(r(J6)),c(r(J6)), 0, 0],[0, 0, 1,
            0],[0,0, 0,1]
        ])
        self.T01 = np.array([[1,0,0,0],[0,1,0,0],[0,0,1,168.2],[0,0,0,1]],dtype=float)
        self.T12 = np.array([[0,-1,0,0],[0,0,-1,0],[1,0,0,0],[0,0,0,1]],dtype=float)
        self.T23 = np.array([[1,0,0,189],[0,1,0,0],[0,0,1,0],[0,0,0,1]],dtype=float)
        self.T34 = np.array([[0,1,0,160],[-1,0,0,0],[0,0,1,90.5],[0,0,0,1]],dtype=float)
        self.T45 = np.array([[1,0,0,0],[0,0,1,101],[0,-1,0,0],[0,0,0,1]],dtype=float)
        self.T56 = np.array([[-1,0,0,0],[0,0,-1,-146],[0,-1,0,0],[0,0,0,1]],dtype=float)

    def Position(self) -> list:
        result = (
            self.T01 @ self.J1 @
            self.T12 @ self.J2 @
            self.T23 @ self.J3 @
            self.T34 @ self.J4 @
            self.T45 @ self.J5 @
            self.T56 @ self.J6
        )

        x, y, z = result[:3, 3]

        rx = round(d(atan2(result[2, 1], result[2, 2])), 3)
        ry = round(d(atan2(-result[2, 0], sqrt(result[2, 1]**2 + result[2, 2]**2))), 3)
        rz = round(d(atan2(result[1, 0], result[0, 0])), 3)

        pose = [round(float(x), 3),
                round(float(y), 3),
                round(float(z), 3),
                rx, ry, rz]

        print(
            f"X = {pose[0]:.3f}, "
            f"Y = {pose[1]:.3f}, "
            f"Z = {pose[2]:.3f}, "
            f"RX = {pose[3]:.3f}, "
            f"RY = {pose[4]:.3f}, "
            f"RZ = {pose[5]:.3f}"
        )
        return pose

```

```
    def MovJ(self, a1, b1, c1, d1, e1, f1, coordinateMode, user=-1, tool=-1, a=-1, v=-1, cp=-1):
        """
        Move from the current position to the target position through joint motion.
        Required parameter:
        Parameter name      Type      Description
        P      string      Target point (joint variables or posture variables)
        coordinateMode      int       Coordinate mode of the target point, 0: pose, 1: joint
        Optional parameter:
        Parameter name      Type      Description
        user      int      user coordinate system
        tool      int      tool coordinate system
        a      int      acceleration rate of the robot arm when executing this command. Range: (0,100].
        v      int      velocity rate of the robot arm when executing this command. Range: (0,100].
        cp      int      continuous path rate. Range: [0,100].
        """
        string = ""
        if coordinateMode == 1 or coordinateMode == 0:
            if all(v == 0 for v in [a1, b1, c1, d1, e1, f1]):
                string = "MovJ(joint={{{:f},{:f},{:f},{:f},{:f},{:f}}}".format(a1, b1, c1, d1, e1, f1)
            else:
                FK = ForwardKin(a1, b1, c1, d1, e1, f1)
                Pose = FK.Position()
                string = "MovJ(pose={{{:f},{:f},{:f},{:f},{:f},{:f}}}".format(Pose[0], Pose[1], Pose[2], Pose[3], Pose[4], Pose[5])
        params = []
        if user != -1:
            params.append('user={:d}'.format(user))
        if tool != -1:
            params.append('tool={:d}'.format(tool))
        if a != -1:
            params.append('a={:d}'.format(a))
        if v != -1:
            params.append('v={:d}'.format(v))
        if cp != -1:
            params.append('cp={:d}'.format(cp))
        for ii in params:
            string = string + ',' + ii
        string = string + ')'
        return self.sendRecvMsg(string)

    def MovL(self, a1, b1, c1, d1, e1, f1, coordinateMode, user=-1, tool=-1, a=-1, v=-1, speed=-1, cp=-1, r=-1):
        """
        Move from the current position to the target position in a linear mode.
        Required parameter:
        Parameter name      Type      Description
        P      string      Target point (joint variables or posture variables)
        coordinateMode      int       Coordinate mode of the target point, 0: pose, 1: joint
        Optional parameter:
        Parameter name      Type      Description
        user      int      user coordinate system
        tool      int      tool coordinate system
        a      int      acceleration rate of the robot arm when executing this command. Range: (0,100].
        v      int      velocity rate, incompatible with "speed". Range: (0,100].
        speed      int      target speed, incompatible with "v". If both exist, speed takes precedence. Unit: mm/s.
        cp      int      continuous path rate, incompatible with "r". Range: [0,100].
        r      int      continuous path radius, incompatible with "cp". If both exist, r takes precedence. Unit: mm.
        """
        string = ""
        if coordinateMode == 1 or coordinateMode == 0:
            if all(v == 0 for v in [a1, b1, c1, d1, e1, f1]):
                string = "MovL(joint={{{:f},{:f},{:f},{:f},{:f},{:f}}}".format(a1, b1, c1, d1, e1, f1)
```

```
            else:
                FK = ForwardKin(a1, b1, c1, d1, e1, f1)
                Pose = FK.Position()
                string = "MovL(pose={{{:f},{:f},{:f},{:f},{:f},{:f}}}".format(Pose[0], Pose
                    [1], Pose[2], Pose[3], Pose[4], Pose[5])
        params = []
        if user != -1:
            params.append('user={:d}'.format(user))
        if tool != -1:
            params.append('tool={:d}'.format(tool))
        if a != -1:
            params.append('a={:d}'.format(a))
        if v != -1 and speed != -1:
            params.append('speed={:d}'.format(speed))
        elif speed != -1:
            params.append('speed={:d}'.format(speed))
        elif v != -1:
            params.append('v={:d}'.format(v))
        if cp != -1 and r != -1:
            params.append('r={:d}'.format(r))
        elif r != -1:
            params.append('r={:d}'.format(r))
        elif cp != -1:
            params.append('cp={:d}'.format(cp))
        for ii in params:
            string = string + ',' + ii
        string = string + ')'
        return self.sendRecvMsg(string)

    def DO(self, index, status, time=-1):
        """
        Set the status of digital output port (queue command).
        Required parameter:
        Parameter name      Type      Description
        index      int      DO index
        status      int      DO status, 1: ON, 0: OFF
        Optional parameter:
        Parameter name      Type      Description
        time      int      continuous output time, range: [25,60000]. Unit: ms.
        If set, the system auto-inverts the DO after the specified time (asynchronous,
        does not block the command queue).
        """
        if index == 101:
            return self.sendRecvMsg("ToolDO(1,{:d})".format(status))
        string = "DO({:d},{:d}".format(index, status)
        params = []
        if time != -1:
            params.append('{:d}'.format(time))
        for ii in params:
            string = string + ',' + ii
        string = string + ')'
        return self.sendRecvMsg(string)
```

Note: `MovJ`/`MovL` now branch on the target: an all-zero joint vector is sent straight through as `joint={...}`, otherwise `ForwardKin(...).Position()` computes the pose from the joint values and the command is sent as `pose={...}` – this is the fix for the original problem, since RoboDK only supplies joint values and the driver now derives the corresponding pose itself before talking to the robot.

8 Connecting to the Physical Robot via RoboDK

Once the driver above is in place, the physical E6 is connected to RoboDK as follows:

1. **Use the same IP address as DobotStudio.** Whatever IP address was used to connect to the robot in DobotStudio (over the network) must be entered into RoboDK's connection settings as well

– both programs are just talking to the same controller.

2. **Port 29999.** This is the command port used by the driver and is not occupied by DobotStudio, so DobotStudio can be left running at the same time without any conflict.

3. **Switch the robot to TCP/IP mode.** The controller has two modes – *Auto* and *TCP/IP*. TCP/IP mode must be enabled for the RoboDK driver (which talks over the network socket) to be able to send commands.

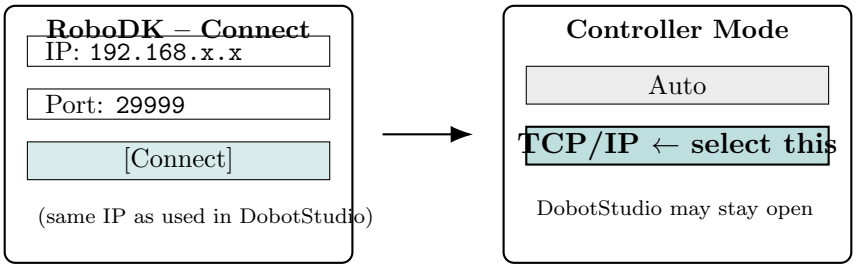


Figure 1: Connection procedure schematic: enter the DobotStudio IP address and port 29999 in RoboDK, then switch the controller from Auto to TCP/IP mode.

Note: this is a schematic of the settings, not a screenshot of the actual RoboDK or DobotStudio interface.

9 Reference Material

The classical DH convention and the general form of A_i (link length, link twist, link offset, joint angle) were cross-checked against the Denavit–Hartenberg chapter of Spong's *Robot Modeling and Control*, specifically the definition of A_i as a function of a single joint variable q_i, and the chain rule $T_j^i = A_{i+1}A_{i+2}\cdots A_j$ for composing transformations along the kinematic chain from base to tool.

These are working notes assembled directly from project whiteboard sessions (July 17–18, 2026). Let me know if you want the programmatic chain-multiplication step or the RoboDK-vs-custom table comparison expanded further.