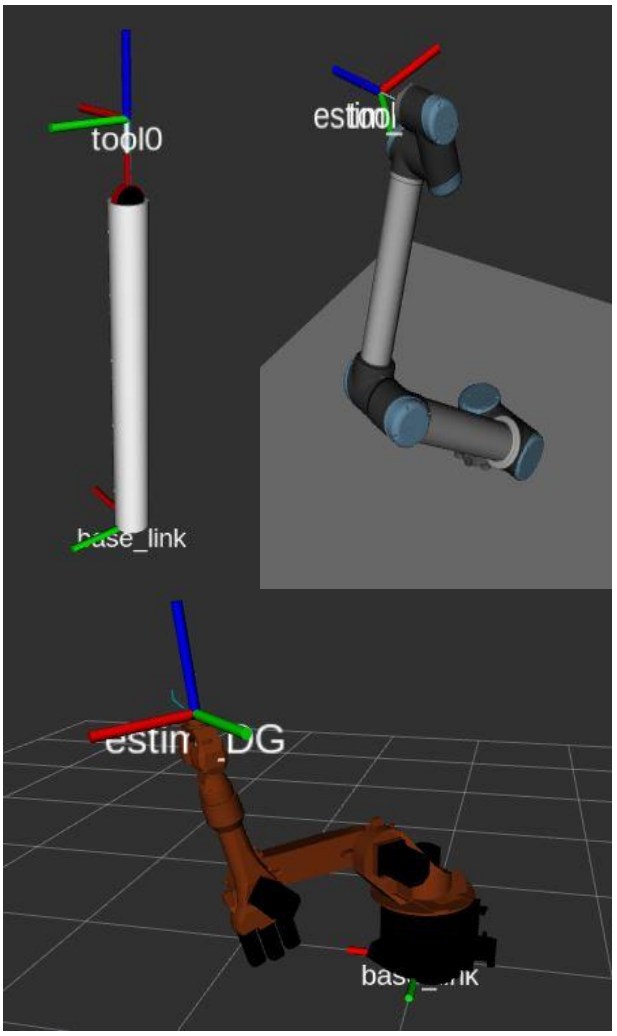
Manipulator Modeling and Control

Bianchi Matteo and Jecklin Erich



Direct Geometric Models

The first expected work on this lab was to obtain the Modified Denavit-Hartenberg parameters, for each of the robots.

a) Turret RRP Robot

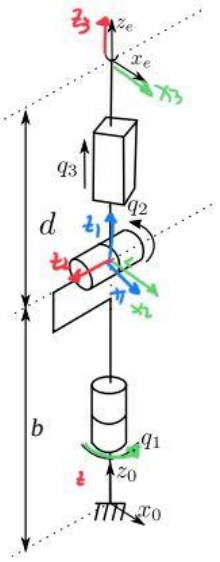


Figure 1. Turret RRP reference frames

Joint	a	α	θ	r
1	0	0	q1	b
2	0	$\pi/2$	q2	0
3	0	$-\pi/2$	0	d + q3
e	0	0	0	0

Table 1. DH table for Turret RRP

b = 0.5 m
d = 0.1 m

b) Kuka KR16 anthropomorphic robot

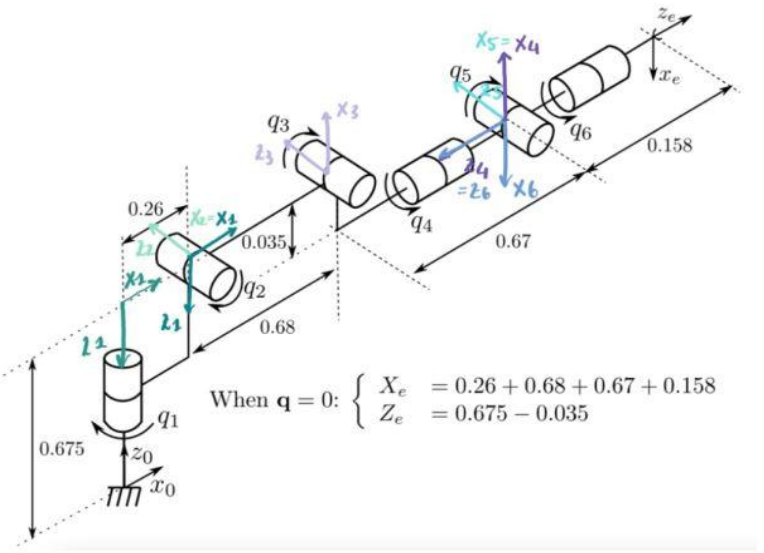


Figure 2. Kuka KR16 anthropomorphic robot

Joint	a	α	θ	r
1	0	π	q1	-0.675
2	0.26	$\pi/2$	q2	0
3	0.68	0	$-\pi/2$ +q3	0
4	-0.035	$\pi/2$	q4	-0.67
5	0	$-\pi/2$	q5	0
6	0	$\pi/2$	π+q6	0
e	0	π	0	0.158

Table 2. DH table for KR16

c) Universal Robot UR-10 anthropomorphic robot

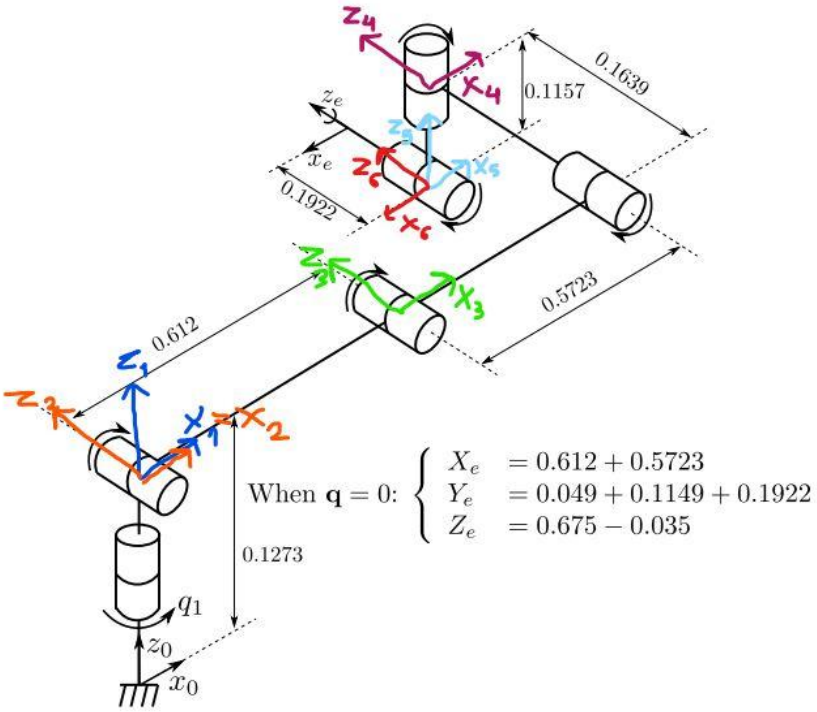


Figure 3. Universal Robot UR-10 anthropomorphic robot

Joint	a	α	θ	r
1	0	0	q1	0.1273
2	0	-π/2	q2	0
3	0.612	0	q3	0
4	0.5723	0	q4	0.1639
5	0	π/2	-q5	-0.1157
6	0	-pi/2	-pi + q6	0
e	0	0	0	0.1922

Table 3. DH table for UR10

3.3 Inverse Geometric Model

Turret:

```
Building pose C++ code...

    // Generated pose code
    const auto c1{cos(q[0])};
    const auto c2{cos(q[1])};
    const auto s1{sin(q[0])};
    const auto s2{sin(q[1])};
    M[0][0] = c1*c2;
    M[0][1] = -s1;
    M[0][2] = -s2*c1;
    M[0][3] = (-d - q[2])*s2*c1;
    M[1][0] = s1*c2;
    M[1][1] = c1;
    M[1][2] = -s1*s2;
    M[1][3] = (-d - q[2])*s1*s2;
    M[2][0] = s2;
    M[2][1] = 0;
    M[2][2] = c2;
    M[2][3] = b + (d + q[2])*c2;
    M[3][0] = 0;
    M[3][1] = 0;
    M[3][2] = 0;
    M[3][3] = 1.;
    // End of pose code
```

1. We found q1 through the type 3 equation (X1 = -1, Y1 = 0, Z1 = Yx, X2 = 0, Y2 = 1,
Z2 = Yy)
2. We found q2 through the type 3 equation (X1 = 1, Y1 = 0, Z1 = Xz, X2 = 0, Y2 = 1,
Z2 = Zz)
3. q3 was found with:
• if C2 != 0, q3 = (tz−0,5)/C2
•if S1 !=0, q3= -ty/(S1S2)
• else if C1 != 0 and S2 != 0, q3 =−tx/(S2∗C1)

KUKA KR16:

```
Decomposing DGM with regards to frame 3:

Translation from root to wrist frame 0T6 (should only depend on q1 q2 q3):

⎡(-0.035·sin(q₂ + q₃) + 0.68·cos(q₂) + 0.67·cos(q₂ + q₃) + 0.26)·cos(q₁)⎤
⎢                                                                       ⎥
⎢(0.035·sin(q₂ + q₃) - 0.68·cos(q₂) - 0.67·cos(q₂ + q₃) - 0.26)·sin(q₁) ⎥
⎢                                                                       ⎥
⎣    -0.68·sin(q₂) - 0.67·sin(q₂ + q₃) - 0.035·cos(q₂ + q₃) + 0.675     ⎦

Rotation 3R6 from frame 3 to wrist frame (should only depend on q4 q5 q6):

⎡sin(q₄)·sin(q₆) - cos(q₄)·cos(q₅)·cos(q₆)   sin(q₄)·cos(q₆) + sin(q₆)·cos(q₄)·cos(q₅)  sin(q₅)·cos(q₄)⎤
⎢                                                                                                    ⎥
⎢             -sin(q₅)·cos(q₆)                                sin(q₅)·sin(q₆)                -cos(q₅)    ⎥
⎢                                                                                                    ⎥
⎣-sin(q₄)·cos(q₅)·cos(q₆) - sin(q₆)·cos(q₄)  sin(q₄)·sin(q₆)·cos(q₅) - cos(q₄)·cos(q₆)  sin(q₄)·sin(q₅)⎦
```

From the MDH table, the direct model can be decomposed in three parts:
t06 , R03 , R36 = (R03)^T∗R06 (using also the command --wrist).

q1, q2 and q3 are found using the translation vector t06 , while q4, q5 and q6, belonging to the spherical wrist, are found using the rotation matrix R36.

• To obtain q1, we can observe that the first equation is equal to the second despite a minus. We can divide the second to the first one and e obtain q1= arctan2(-ty,tx) or from the type 2 equation (X = tx, Y = ty, Z = 0).
– if abs(C1) > abs(S1) then α = (tx)/C1.
– else α = (−ty)/S1.
• q2 and q23 from equation type 7, with i=2 and j=23 (X = -0,68, Y = 0, Z1 = α − 0, 26, Z2 = 0,675 - tz, W1 = 0,67, W2 = -0,035).
• q3 = q23 - q2

For the last three joints the solution depends on the value of q5:
• if q5 = null:
– q5 = 0
– q46 = atan2(−Xz1, −Xx1)

– q4 = q0[3] (it’s an arbitrary choice).
– q6 = q46 - q4
• else q5 !=0:
– q4 from equation type 2 (X = Zx1, Y = -Zz1, Z = 0)
– q5 from equation type 3 (X1 = C4, Y1 = 0, Z1 = Zx1, X2 = 0, Y2 = -1, Z2 = Zy1)
– q6 from equation type 3 (X1 = 0, Y1 = -S5, Z1 = Xy1, X2 = S5, Y2 = 0, Z2 = Yy1)

3.4 Interpolated point-to-point control

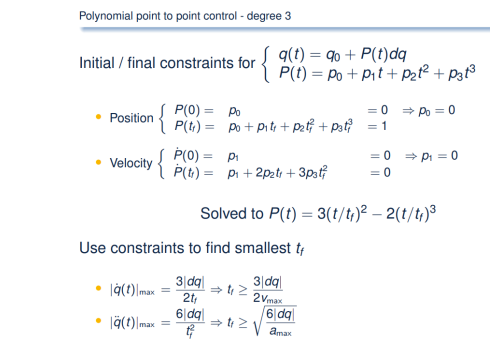
```
else if(robot->mode() == ControlMode::POLYNOM_P2P)
{
    // reach Md with polynomial joint trajectory
    // use q0 (initial position), qf (final), aMax and vMax
    // if reference has changed, compute new tf
    if(robot->newRef())
    {
        q0 = robot->inverseGeometry(M0, q);
        qf = robot->inverseGeometry(Md, q);
        double tf=0;
        for(int i=0;i<6;i++){
            auto dq = qf[i] - q0[i];
            auto tfv = (3*abs(dq))/(2*vMax[i]);
            auto tfa = sqrt((6*abs(dq))/(aMax[i]));
            auto a = max(tfv,tfa);

            if(a>tf){
                tf=a;

            }
        }

    }
    auto pt = 3*pow(min((t-t0)/tf,1.0),2)- 2*pow(min((t-t0)/tf,1.0),3);
    auto qCommand= q0+pt*(qf-q0);
    // TODO: compute qCommand from q0, qf, t, t0 and tf
    robot->setJointPosition(qCommand);
}
```

"POLYNOM_P2P" (polynomial point-to-point) control mode. It checks if there is a new reference for the robot, and if so, it calculates a trajectory based on initial (q0) and final (qf) joint positions, maximum acceleration (aMax), and maximum velocity (vMax). The time (tf) required to reach the final joint positions is computed. Subsequently, a trajectory parameter (pt) is calculated based on the current time (t), initial time (t0), and tf. The desired joint position (qCommand) is then determined as an interpolation between the initial and final joint positions. Finally, this calculated joint position is set for the robot to execute the desired point-to-point motion. We compute the current setpoint qc = q0 + p(t)(qf-q0). The interpolation function p(t) was defined during the lectures and should take into account the maximum joint velocity and acceleration that are stored in the vMax and aMax vectors. Basically this amounts to computing the minimum time tf needed to reach qf from q0.



(We based our code on this slide explained during the class).

3.5 Operational Control through Inverse Geometric Model

Exercises 3 and 4 lead to the correct pose but not in a straight 3D line. Indeed the robot is only controlled in the joint space with no constraints between the two extreme poses. In this exercise, the goal is to perform the interpolation not in the joint space but in the Cartesian space. In practice, we will compute many poses between M0 and Md and command the robot to reach sequentially all these poses. As they will be pretty close to each other, the resulting motion will be close to a straight line. The robot->intermediaryPose(M0, Md, alpha) function returns a pose between M0 and Md , where alpha is between 0 and 1 and should be computed from t, t0 and tf = 1 s. Then, the inverse geometry of this pose should be sent to the robot as a joint position setpoint.We compute many poses between the current and the desired joint poses and command the robot to reach them. Since the computed poses are in Cartesian space, we get the Inverse Geometry to be sent as a joint position.

```
else if(robot->mode() == ControlMode::STRAIGHT_LINE_P2P)
{
    // go from M0 to Md in 1 sec

    tf = 1;
    // TODO: compute qCommand from M0, Md, t, t0 and tf
    // use robot->intermediaryPose to build poses between M0 and Md
    auto alpha = (t-t0)/tf;
    auto M = robot->intermediaryPose(M0,Md,alpha);
    qCommand = robot->inverseGeometry(M,q);
    robot->setJointPosition(qCommand);
}
```

"STRAIGHT_LINE_P2P" (straight-line point-to-point) control mode. It sets a fixed time (tf) of 1 second for the robot to travel from the initial pose (M0) to the destination pose (Md). Using an interpolation parameter (alpha), the code computes an intermediary pose (M) between M0 and Md with robot->intermediaryPose(). The inverse kinematics (robot->inverseGeometry()) is then applied to obtain joint positions (qCommand) corresponding to the intermediary pose. Finally, these calculated joint positions are set for the robot using robot->setJointPosition(), directing the robot to execute the desired straight-line motion within the specified time frame.

3.6 Operational Control through Jacobian

In this exercise the goal is still to follow a straight line between the two points, but this time by sending a joint velocity command.

As seen in class, this command should make the end-effector approach its desired pose. The steps are:

1. Compute the pose error in the desired frame: ${}^{e*}\mathbf{M}_e = {}^f\mathbf{M}_{e*}^{-1}.{}^f\mathbf{M}_e$
   - in the code, ${}^f\mathbf{M}_{e*}$ corresponds to `Md` and ${}^f\mathbf{M}_e$ to `M`
   - In practice we use the $(\mathbf{t}, \theta\mathbf{u})$ representation with `p.buildFrom()`
2. Compute the desired linear and angular velocities: $v = -\lambda {}^f\mathbf{M}_{e*}\mathbf{t}, \quad \omega = -\lambda {}^f\mathbf{M}_e(\theta\mathbf{u})$
3. Map these velocities to the joint space: $\dot{\mathbf{q}} = \mathbf{J}^+ \begin{bmatrix} v \\ \omega \end{bmatrix}$

λ is a gain that can be changed from the GUI. Increasing it will lead to a faster convergence. It can be obtained through `robot->lambda()` .

Instead of sending the joint position, here we send the joint velocities to the robot to follow a straight line between two points. The position error between the current and desired positions of end-effector is derived as:

$$ {}^eM_{e^*} = {}^fM_e^{-1} * {}^fM_{e^*} $$

Then we decompose the position error matrix as a column vector of translation and rotation (t,u). We get the desired velocity in the fixed frame and finally we get the joint velocities using jacobian.
This process enables the robot to dynamically adjust its joint velocities based on the pose error, allowing it to smoothly follow a straight line between the specified points. The utilization of the Jacobian facilitates this control strategy by translating desired end-effector velocities into corresponding joint velocities.

```
else if(robot->mode() == ControlMode::VELOCITY_P2P)
{

    auto lambda = robot->lambda();
    vpColVector epose = p.buildFrom(Md.inverse()*M);
    vpMatrix fRe = M.getRotationMatrix();
    vpMatrix fRed = Md.getRotationMatrix();
    vpMatrix vMatrix(6,6);


    ecn::putAt(vMatrix, fRed, 0, 0);
    ecn::putAt(vMatrix, fRed, 3, 3);
    vpMatrix vw = -lambda*vMatrix*epose;
    vpMatrix Jplus = robot->fJe(q).pseudoInverse();
    auto vCommand = Jplus*vw;

 robot->setJointVelocity(vCommand);
}
```

"VELOCITY_P2P" (velocity point-to-point) control mode in a robot control system. It calculates the pose error between the desired pose (Md) and the current pose (M), incorporating rotational matrices and a weighting matrix. The end effector velocity (vw) is then computed based on this pose error and a damping factor (lambda). Using the pseudo-inverse of the Jacobian, the desired joint velocity (vCommand) is obtained. Finally, this calculated joint velocity is set for the robot using robot->setJointVelocity(), directing it to execute the desired velocity-controlled point-to-point motion.