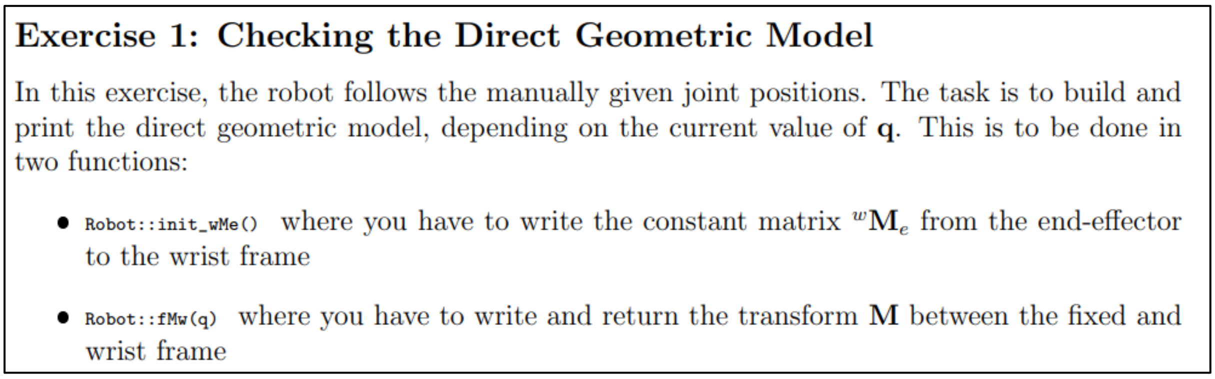
<u>Modelling of Manipulators</u>

Final Lab Report

Submitted by:

Anna Ricarda Hauschild

Sakthi Vikneswar Suresh Babu

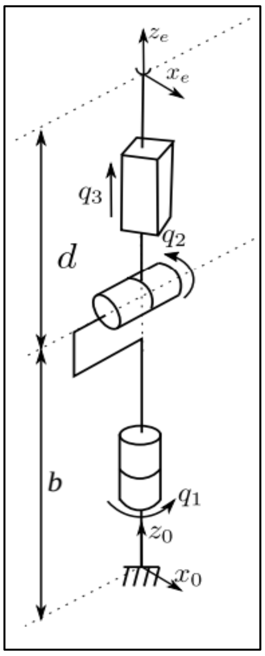


Exercise 1: Checking the Direct Geometric Model

In this exercise, the robot follows the manually given joint positions. The task is to build and print the direct geometric model, depending on the current value of $\mathbf{q}$. This is to be done in two functions:

- `Robot::init_wMe()` where you have to write the constant matrix ${}^{w}\mathbf{M}_e$ from the end-effector to the wrist frame
- `Robot::fMw(q)` where you have to write and return the transform $\mathbf{M}$ between the fixed and wrist frame

- Turret RRP Robot:



Joint	**α**	**a**	**θ**	**r**
1	0	0	q1	b
2	π/2	0	q2	0
3	-π/2	0	0	q3
End effector	0	0	0	d

Where,

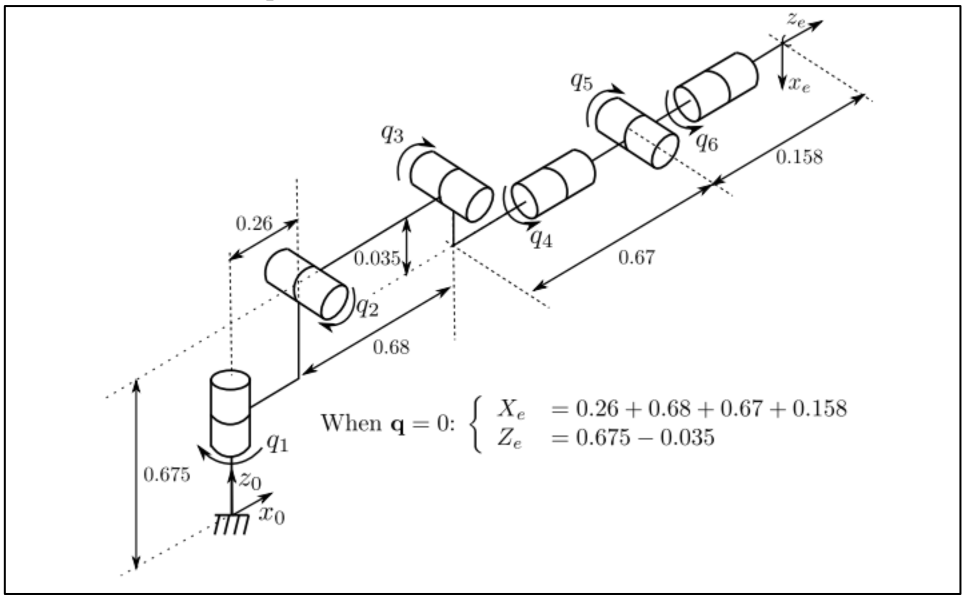
α: Rotation along X-axis

a: Translation along X-axis

θ: Rotation along Z-axis

r: Translation along Z-axis

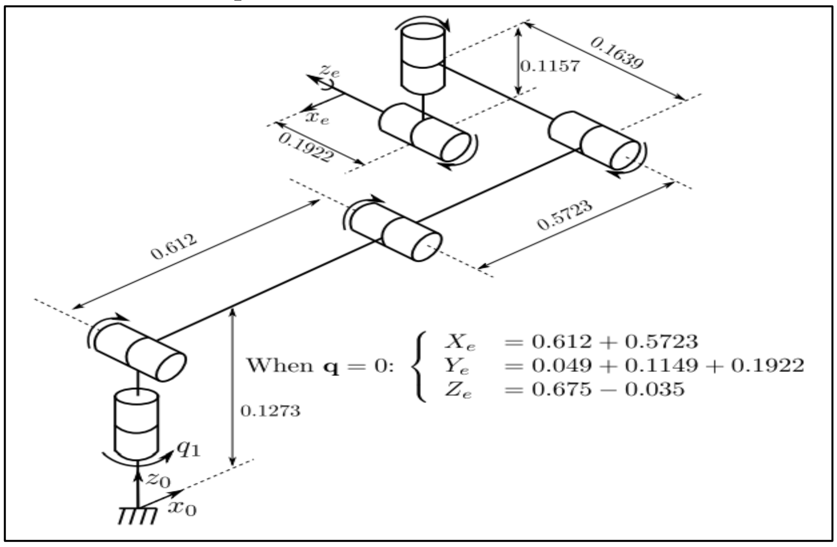
- KUKA KR-16 Anthromorphic Robot


z_e
x_e
q_5
q_6
q_3
0.158
0.26
0.035
q_4
0.67
q_2
0.68
q_1
0.675
z_0
x_0
When $\mathbf{q} = 0$: $\begin{cases} X_e = 0.26 + 0.68 + 0.67 + 0.158 \\ Z_e = 0.675 - 0.035 \end{cases}$

Joint	α	a	θ	r
1	0	0	-q1	r1
2	-π/2	a2	q2	0
3	0	a3	q3- π/2	0
4	- π/2	-a4	-q4	r4
5	π/2	0	q5	0
6	- π/2	0	-q6	0
End effector	0	0	π	re

Where,
α: Rotation along X-axis
a: Translation along X-axis
θ: Rotation along Z-axis
r: Translation along Z-axis

- Universal UR-10 Anthromorphic Robot

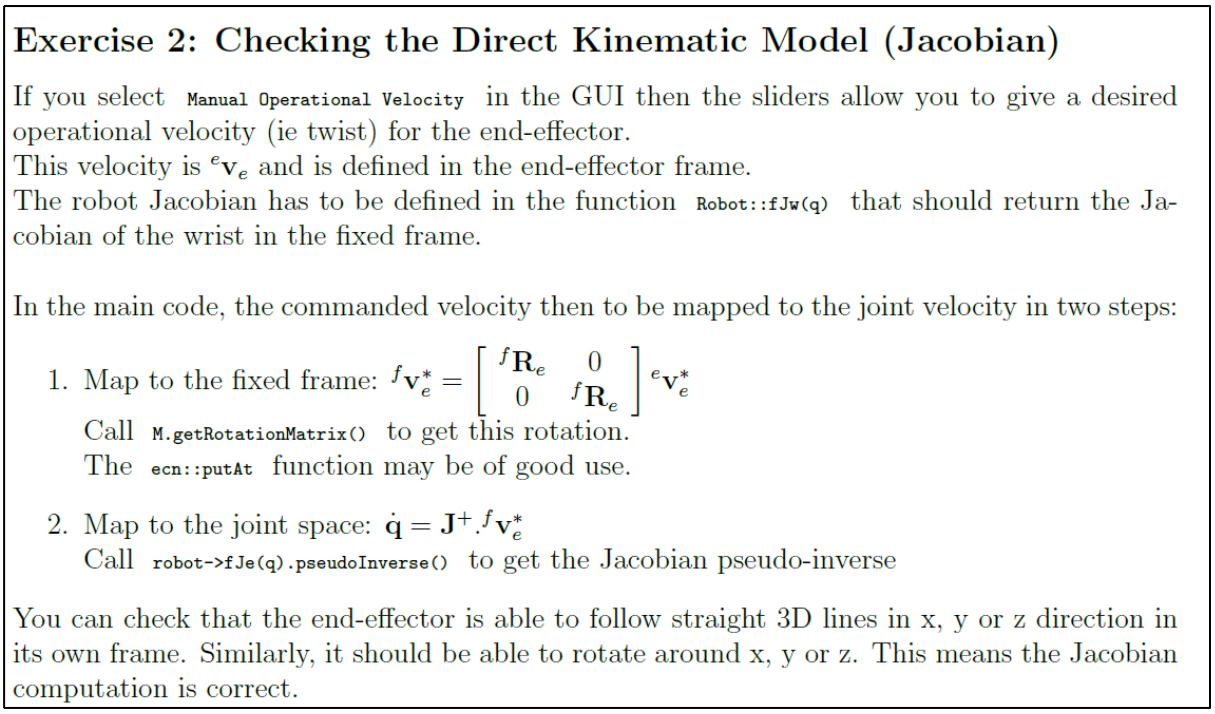

0.1157
0.1639
z_e
x_e
0.1922
0.612
0.5723
When $\mathbf{q} = 0$: $\begin{cases} X_e = 0.612 + 0.5723 \\ Y_e = 0.049 + 0.1149 + 0.1922 \\ Z_e = 0.675 - 0.035 \end{cases}$
0.1273
q_1
z_0
x_0

Joint	α	a	θ	r
1	0	0	q1	r1
2	-π/2	0	q2	0
3	0	a3	q3	0
4	0	a4	q4	r4
5	-π/2	0	q5+ π	r5
6	- π/2	0	q6	0
End effector	0	0	0	re

Where,
α: Rotation along X-axis
a: Translation along X-axis
θ: Rotation along Z-axis
r: Translation along Z-axis

After the calculation of the DH parameters, the following steps have to be followed:

- Transfer the DH parameters to the yml file.
- Launch the yml file to generate the corresponding pose code and end effector code.
- Implement those to the CPP file.
- Launch the RViz simulator for the corresponding robot and simultaneously execute the CPP file.
- We have to ensure that the calculated end effector pose is the same as the previous one.



Exercise 2: Checking the Direct Kinematic Model (Jacobian)

If you select `Manual Operational Velocity` in the GUI then the sliders allow you to give a desired operational velocity (ie twist) for the end-effector.
This velocity is ${}^e\mathbf{v}_e$ and is defined in the end-effector frame.
The robot Jacobian has to be defined in the function `Robot::fJw(q)` that should return the Jacobian of the wrist in the fixed frame.

In the main code, the commanded velocity then to be mapped to the joint velocity in two steps:

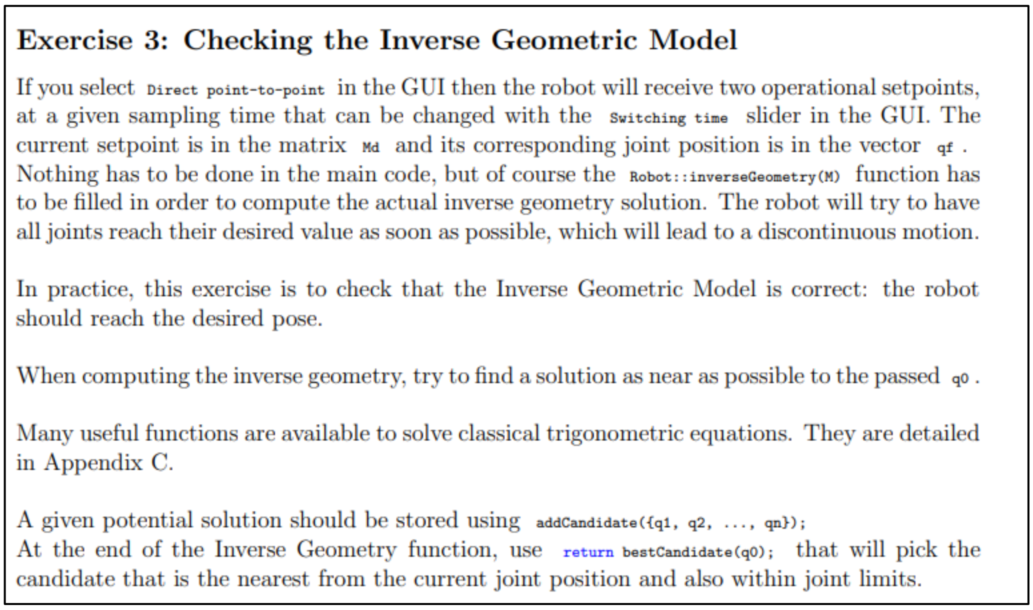
1. Map to the fixed frame: ${}^f\mathbf{v}_e^* = \begin{bmatrix} {}^f\mathbf{R}_e & 0 \\ 0 & {}^f\mathbf{R}_e \end{bmatrix} {}^e\mathbf{v}_e^*$
   Call `M.getRotationMatrix()` to get this rotation.
   The `ecn::putAt` function may be of good use.
2. Map to the joint space: $\dot{\mathbf{q}} = \mathbf{J}^+ . {}^f\mathbf{v}_e^*$
   Call `robot->fJe(q).pseudoInverse()` to get the Jacobian pseudo-inverse

You can check that the end-effector is able to follow straight 3D lines in x, y or z direction in its own frame. Similarly, it should be able to rotate around x, y or z. This means the Jacobian computation is correct.

Here the desired velocity for the end-effector twist is given by *Manual Operational Velocity*. The fixed frame Jacobian of the wrist is obtained from the Robot Jacobian in the function *Robot::fJw(q)*. The velocity is then mapped to the joint velocity from the following steps:

- Mapping it to the fixed frame.
- Later mapping it to the Joint space.

We observe that the robot is able to follow straight line motion in the fixed 3D frame.



Exercise 3: Checking the Inverse Geometric Model

If you select `Direct point-to-point` in the GUI then the robot will receive two operational setpoints, at a given sampling time that can be changed with the `Switching time` slider in the GUI. The current setpoint is in the matrix `Md` and its corresponding joint position is in the vector `qf` .
Nothing has to be done in the main code, but of course the `Robot::inverseGeometry(M)` function has to be filled in order to compute the actual inverse geometry solution. The robot will try to have all joints reach their desired value as soon as possible, which will lead to a discontinuous motion.

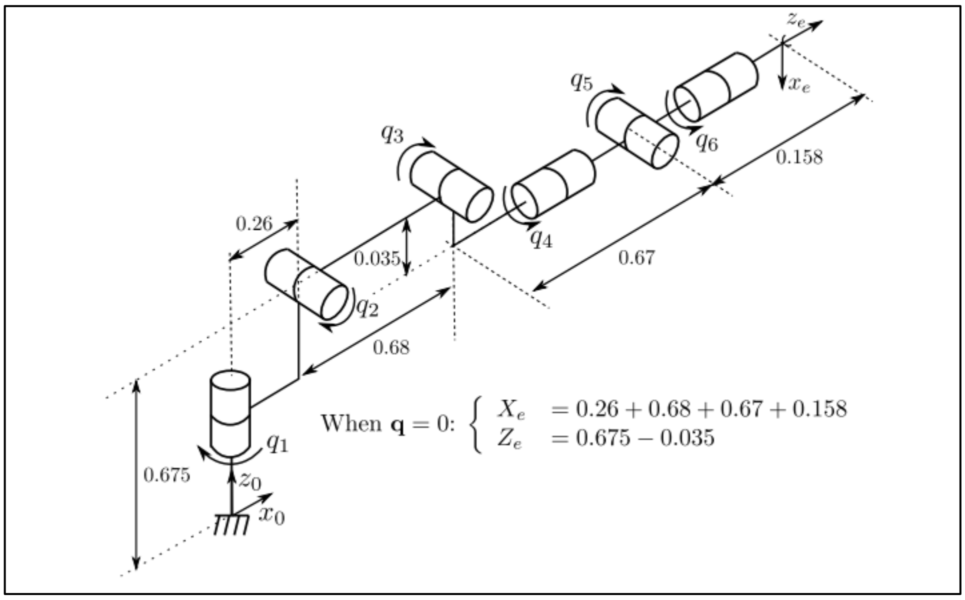
In practice, this exercise is to check that the Inverse Geometric Model is correct: the robot should reach the desired pose.

When computing the inverse geometry, try to find a solution as near as possible to the passed `q0` .

Many useful functions are available to solve classical trigonometric equations. They are detailed in Appendix C.

A given potential solution should be stored using `addCandidate({q1, q2, ..., qn});`
At the end of the Inverse Geometry function, use `return bestCandidate(q0);` that will pick the candidate that is the nearest from the current joint position and also within joint limits.

KUKA KR-16 – Inverse Geometric Model calculation



z_e
x_e
q_5
q_6
q_3
0.158
q_4
0.26
0.035
0.67
q_2
0.68
When $\mathbf{q} = 0$: $\begin{cases} X_e = 0.26 + 0.68 + 0.67 + 0.158 \\ Z_e = 0.675 - 0.035 \end{cases}$
q_1
0.675
z_0
x_0

Translation matrix (t):

$$t = \begin{array}{c} (a2 + a3c2 - a4s23 + r4c23)c1 \\ -(a2 + a3c2 - a2s23 + r4c23)s1 \\ -a3s2 - a4c23 + r1 \quad - r4s23 \end{array} = \begin{array}{c} tx \\ ty \\ tz \end{array}$$

The above mentioned matrix t is obtained from the Jacobian Matrix from the code.

By observing tx and ty,

If tx or ty is not equal to 0, then q1 can be solved from classic equation no. 3 (with tx and ty)

3	$X_1 s_i + Y_1 c_i = Z_1$ $X_2 s_i + Y_2 c_i = Z_2$	`solveType3(x1, y1, z1, x2, y2, z2)`	q_i

If c1 is not equal to 0, then α = tx/c1

If s1 is not equal to 0, then α = -ty/s1

Consider, α = a3c2-a4s23+r4c23

tz = -a3s2-a4c23-r4s23

From the above conditions we get,

α – a2 ------i

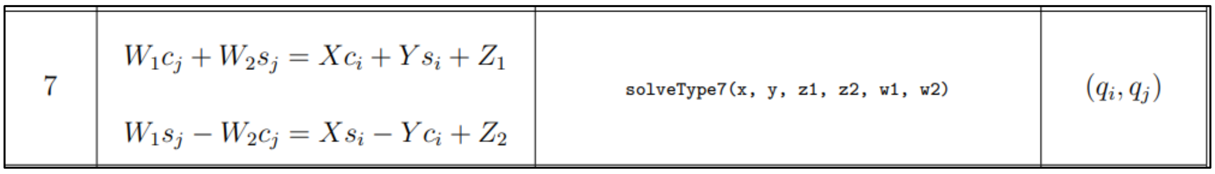
tz – r1 -------ii

From eqn. i and ii,

a4s23 – r4c23 = a3c2 + a2 – α

a4c23 + r4s23 = -a3s2 + r1 – tz

The above solution can be solved by comparing it with classic eqn. no. 7 (q2 and q3)



7	$W_1c_j + W_2s_j = Xc_i + Ys_i + Z_1$ $W_1s_j - W_2c_j = Xs_i - Yc_i + Z_2$	`solveType7(x, y, z1, z2, w1, w2)`	(q_i, q_j)

$$R_6^3 = \begin{matrix} -s4s6 + c4c5c6 & -s4c6 - s6c4c5 & -s5c4 \\ s5c6 & -s5c6 & c5 \\ -s4c5c6 - s6c4 & s4s6c5 - c4c6 & s4s5 \end{matrix} = \begin{matrix} Xx & Yx & Zx \\ Xy & Yy & Zy \\ Xz & Yz & Zz \end{matrix}$$

If Zx = Zz = 0,

q5 = 0 (pick q4)

else calculate q4 from type 3 eqn. (Zx,Zz)

then calculate q5 from type 3 eqn.

If q5 = 0 (singularity case)

Then,

$$R_6^3 = \begin{matrix} -c46 & -s46 & 0 \\ 0 & 0 & 1 \\ -s46 & -c46 & 0 \end{matrix} = \begin{matrix} Xx & Yx & Zx \\ Xy & Yy & Zy \\ Xz & Yz & Zz \end{matrix}$$

q46 can be solved from type 3, as q4 and q6 cannot be solved individually. Here q46 = q4 + q6

Exercise 4: Interpolated point-to-point control

In this exercise the previous pose setpoint `M0` and its corresponding joint position `q0` should be used to interpolate the joint positions.

Here the goal is to compute the current setpoint $\mathbf{q}_c = \mathbf{q}_0 + p(t)(\mathbf{q}_f - \mathbf{q}_0)$. 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 $\mathbf{q}_f$ from $\mathbf{q}_0$. This computation should be done in the `if(robot->newRef())` and then used to compute the current `qCommand` .

Here we focus on the point-to-point control of the robots. To achieve that we use the polynomial point-to-point control methodology.

The time taken to achieve the final pose of the end-effector (tf) is calculated based on 2 parameters: the maximum velocity and the maximum acceleration.

$$\text{tf} \geq \frac{3dq}{2Vmax}$$

$$\text{tf} \geq \text{sqrt}(\frac{6dq}{Amax})$$

From the above equations we find the smallest tf value. This is in turn applied to the interpolation function p(t).

$$p(t) = 3(t/tf)^3 - 2(\frac{t}{tf})^3$$

The obtained p(t) is substituted to compute the current setpoint.

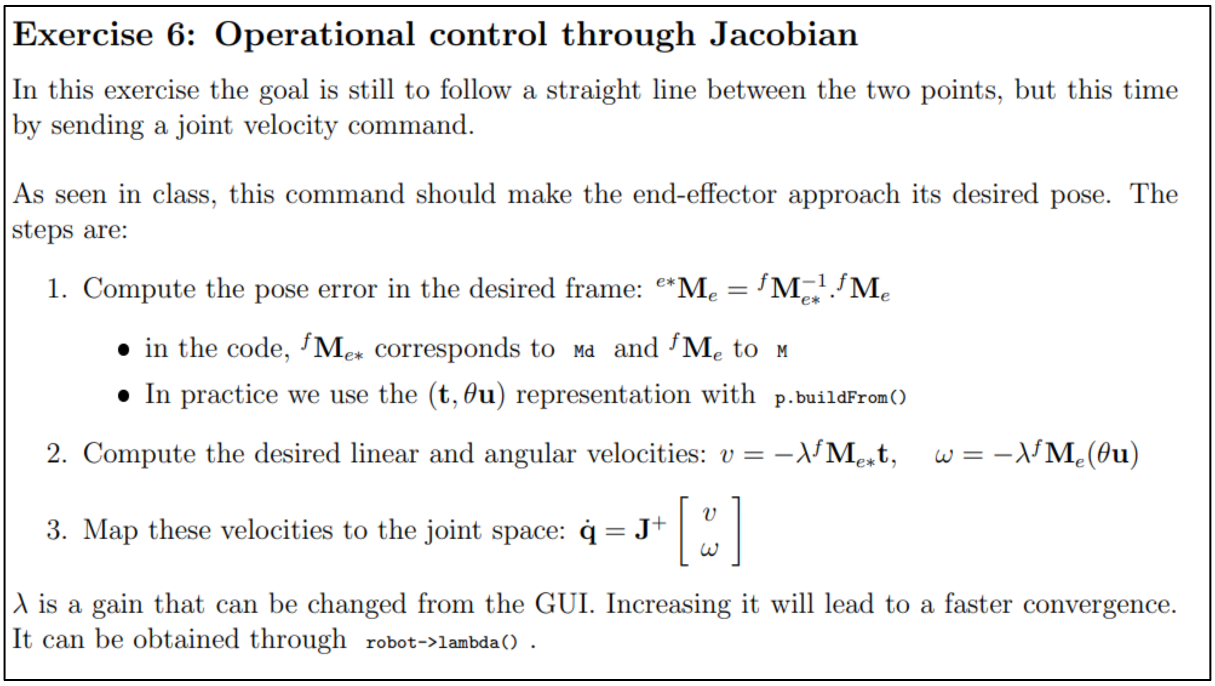
$$q_c = q_0 + p(t)(\,q_f - q_0)$$

Exercise 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 one from 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, t_0 and $t_f = 1$ s.
Then, the inverse geometry of this pose should be sent to the robot as joint position setpoint.

In this case we shift the focus from Joint space to Cartesian space. Here the value of tf is given as 1. Here we calculate the intermediate poses in cartesian space between M_0 *and* M_d using the predefined function *robot->intermediaryPose(*$M_0, M_d, alpha$*)*.

The point of focus is controlled by *alpha* which varies from 0 to 1. If *alpha* =0, the point of focus is in M_0. If *alpha* = 1, then the point of focus is in M_d. The obtained poses are transformed back to the joint space with the help of Inverse Geometric Model and then pass it to the joint commands.



Exercise 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()` .

Here we find the transformation between the current and desired pose. This is then changed to a (t,θu) representation. Further, the desired linear and angular velocity is calculated from the following equations:

$$v = -\lambda M_{e*}^{f} t \text{ (Linear velocity)}$$

$$w = -\lambda M_{e}^{f} (\theta u) \text{ (Angular velocity)}$$

These parameters are then mapped to joint space. The performance of the system is controlled by the variation of the gain λ.