Robotics: Kinematics, Dynamics and Control

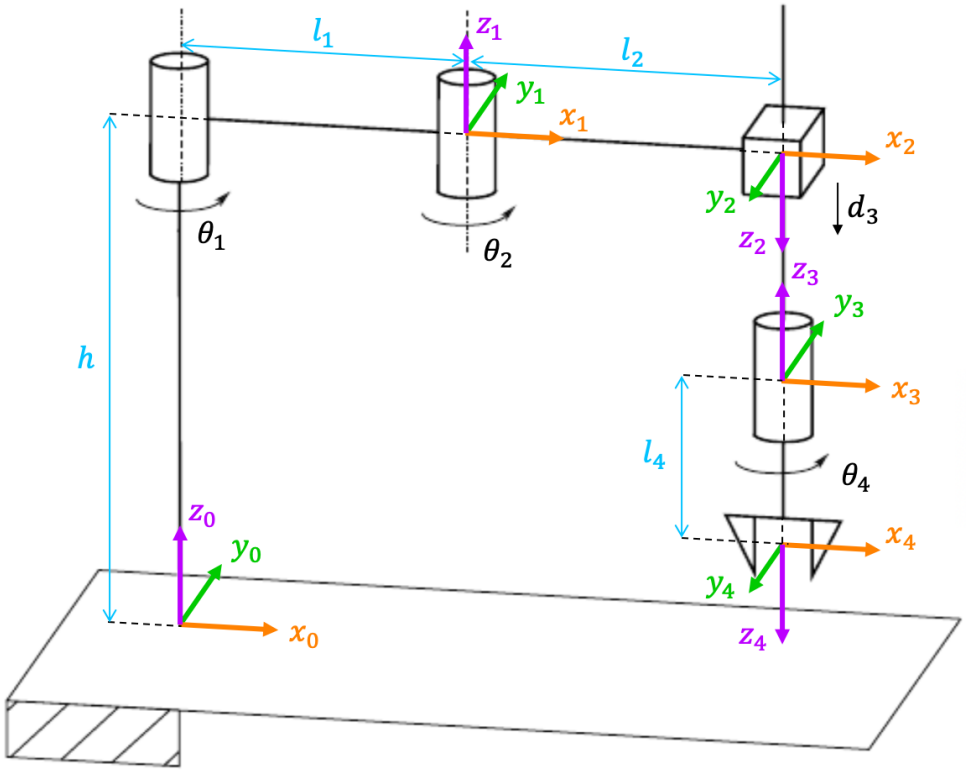
Final Work: SCARA Robot

Adrià Luque Acera
ETSEIB, UPC
January 2020

Exercise 1

Compute the DH parameters. Set numerical values to the link lengths and offsets.

The chosen robot is the SCARA Manipulator, with a total of 4 joints (in order, two revolute ones, a prismatic and another revolute before the End Effector). With the given drawing of the kinematic structure of the robot, we can give names to all the link lengths and offsets, and place the joint axes according to the Denavit-Hartenberg convention. The result of this process can be seen on the following figure:



Where, of course, θ_1, θ_2, d_3 and θ_4 are the joint variables. It has been decided to set the possible offset l_3 to 0, so the distance between frames 2 and 3 is directly the variable d_3. The convention fixes the z axes in the direction of the variables, and x following the common normal between two consecutive z axes. With parallel or collinear axes, the x axis has been chosen so we obtain the easiest solution. The table of DH parameters finally is:

Link i	$\boldsymbol{\theta_i}$	$\boldsymbol{d_i}$	$\boldsymbol{a_i}$	$\boldsymbol{\alpha_i}$
1	q_1	h	l_1	0
2	q_2	0	l_2	π
3	0	q_3	0	$-\pi$
4	q_3	$-l_4$	0	π

First, we have used the names $q_i,\ i \in [1,4]$ for the actual variables, so parameters θ_i and d_i corresponding to these terms are equal to these variables q_i plus an offset. In our case, the offsets for all variables are zero, so on their cell we have the variables alone with no addition or subtraction of a constant term.

On the table, we can also see how defining the axis z_3 in the direction of variable q_4 makes the parameter d_4 negative, and also there are three consecutive changes of sense in the z_i axis, as it goes up for the revolute joints, but down for both the prismatic joint and the End Effector, as it is common to have the final z_{EE} axis pointing outwards in the approach direction.

Now we can set numerical values to these distances to get the final DH table, and introduce it to Matlab. We choose, arbitrarily:

$$h = 0.75\,m, \qquad l_1 = 0.4\,m, \qquad l_2 = 0.3\,m, \qquad l_4 = 0.15\,m$$

So, the table before, now becomes:

Link i	$\boldsymbol{\theta_i}$	$\boldsymbol{d_i}$	$\boldsymbol{a_i}$	$\boldsymbol{\alpha_i}$
1	q_1	0.75	0.4	0
2	q_2	0	0.3	π
3	0	q_3	0	$-\pi$
4	q_3	-0.15	0	π

And we can code into Matlab (written in attached file `Final_Work`):

```
%% 1. DH PARAMETERS

d1 = 0.75;
d2 = 0;
th3 = 0;
d4 = -0.15;
a1 = 0.4;
a2 = 0.3;
a3 = 0;
a4 = 0;
alpha1 = 0;
alpha2 = pi;
alpha3 = -pi;
alpha4 = pi;

DH = [  0  d1  a1  alpha1;
        0  d2  a2  alpha2;
      th3   0  a3  alpha3;
        0  d4  a4  alpha4];
```

These parameters will be used on the computation of the Kinematics of our robot.

Exercise 2

With the aid of the Symbolic Toolbox:

a) Implement a script/function to compute the Forward Kinematics
b) Implement a script/function to compute the Geometric and Analytic Jacobians
c) Implement a script/function to compute the Inverse Kinematics using a closed form
d) Implement a script/function to compute the Inverse Kinematics using an iterative form

---

a) Forward Kinematics:

We know that the general homogeneous transform between a frame and the next in a kinematic link has the following expression:

$$A_i^{i-1} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i \cdot \cos\alpha_i & \sin\theta_i \cdot \sin\alpha_i & a_i \cdot \cos\theta_i \\ \sin\theta_i & \cos\theta_i \cdot \cos\alpha_i & -\cos\theta_i \cdot \sin\alpha_i & a_i \cdot \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Here all terms θ_i and d_i can be substituted by a variable term plus an offset, or just a constant value, depending on the joint. Therefore, every matrix A depends on one joint variable, and the final Forward Kinematics transform between base and End Effector is defined as follows:

$$T_n^0 = A_1^0(q_1) \cdot A_2^1(q_2) \cdot \ldots \cdot A_n^{n-1}(q_n)$$

For the SCARA robot, we have a total of four joints, so the transform is:

$$T_4^0 = A_1^0(\theta_1) \cdot A_2^1(\theta_2) \cdot A_3^2(d_3) \cdot A_4^3(\theta_4)$$

Here we have used θ and d to make emphasis in which type of joint we have for every link, because as we are basically substituting values in the matrices A, this difference is important to know which terms are constant and which are not.

In Matlab, we could just use symbolic variables to define a general matrix A, and then substitute 4 times for our specific cases with the computed DH parameters and joint variables, but it is much more interesting to develop a general function to compute any homogeneous transform from a given DH matrix and the joint types.

This is what custom function `getKinModel` does. The input parameters are the previously defined `DH` matrix, and `sigma`, which can be an array of binary values indicating the joint type (0 for revolute, 1 for prismatic), or directly a string with as many `'r'` or `'p'` as desired (for the SCARA, we would have `[0 0 1 0]` or `'rrpr'`). Sometimes, the joint type is given as an extra DH parameter (like on the Robotics Toolbox), but we have decided to separate them and keep the DH matrix strictly for its parameters. This function returns the transform matrix T_0^n, all symbolic joint variables in a vector q, and a 3D matrix with all joint transforms A_i^{i-1} concatenated in the third dimension.

The function first calculates the number of joints from the length of the second input containing the joint types, and checks that it is coherent with the length of the DH matrix. If not, returns an error:

```
% Number of Joints/Links
n = length(sigma);
if n ~= length(DH(:,1))
    error(['Incoherent number of joints.', ...
           '\nLength of sigma must be the number of rows of DH'],'');
end
```

The empty second parameter of the error call is just so the line break command works, as it doesn't with only one string. If the joint number is correctly defined, then the function proceeds to parse the joint types if they are given as text:

```
% Joint types
jtype = zeros(1,n);
if ischar(sigma)
    sq = replace(sigma,'r','0');
    sq = replace(sq,'p','1');
    sq = num2cell(sq);

    for i=1:n
        jtype(i) = str2double(sq{i});
        if isnan(jtype(i))
            error(['Invalid sigma input.',...
                   '\n\nEnter ''r'' for revolute joints ', ...
                   'and ''p'' for prismatic joints', ...
                   '\nExample: ''rrpr'''],'');
        end
    end

elseif isa(sigma, 'double')
    for i=1:n
        jtype(i) = logical(sigma(i));
    end
else
    error(['Invalid data type for input: sigma', ...
           '\n\nType help getKinModel to see syntax'],'');
end
```

This block also converts any numerical array into a Boolean one, so we are sure to only have 1s and 0s for future operation. If any command fails, an error is risen trying to help the user know what failed and how to fix it.

After this, we have the definition of the needed symbolic variables (joint variables are defined depending on the input length with a row vector):

```
% Symbolic joint variables
q = sym('q',[1 n]);
assume(q,'real');

% Symbolic DH Parameters theta (th) d a alpha (al)
syms thi di ai ali
assume([thi di ai ali],'real')
```

And then we have the definition of the general symbolic matrix A_i^{i-1}, as we would do having this in the main script:

```
% General A matrix
Ai = [cos(thi)  -sin(thi)*cos(ali)   sin(thi)*sin(ali)  ai*cos(thi);
      sin(thi)   cos(thi)*cos(ali)  -cos(thi)*sin(ali)  ai*sin(thi);
         0       sin(ali)            cos(ali)           di;
         0          0                   0                1];
```

Once we have this general expression, we can proceed to the final step, substituting the known values from the DH matrix. This of course depends on the joint type, so we have the two cases explicitly coded:

```
% DH Substitution and FK Transform
T0n = eye(4);
An = 0*sym('An',[4 4 n]);
for i=1:n
    if jtype(i) == 0
        % Revolute joint: q+th d a alpha
        An(:,:,i) = subs(Ai, [thi, di,ai,ali], ...
                             [q(i)+DH(i,1), DH(i,2:4)]);
    else
        % Prismatic joint: th q+d a alpha
        An(:,:,i) = subs(Ai, [thi, di, ai,ali], ...
                             [DH(i,1), q(i)+DH(i,2), DH(i,3:4)]);
    end
    T0n = T0n*An(:,:,i);
end

T0n = simplify(T0n);
An = simplify(An);
```

There are a couple of important remarks here. First, the declaration of '`An`' could seem absurd at first, multiplying by 0, but this is used to tell Matlab the final size of the variable and its class (symbolic), so it can preallocate it in the memory of the computer, and be faster in the iterative computations (we cannot initialize a matrix of numerical zeros either, as later we substitute some values by symbolic expressions and it would not work). This variable '`An`' is used to save all the consecutive frame transforms, and is one of the outputs of the function. The second detail is that depending on the joint, we substitute the variable on the correct position, but not only that, we also add the corresponding offset set of the DH-parameters matrix. For the studied case, the SCARA robot had all offsets at 0, but this is very important to consider in a general situation.

Back into the main script, we can call the function with our specific case and get the Forward Kinematics for our SCARA robot:

```
%% 2. Symbolic Toolbox Model
%% 2a) Forward Kinematics

% Create model
[T04, q, Ai4] = getKinModel(DH, 'rrpr');
```

The final FK transform matrix from base to end effector results:

```
T04 =
[ cos(q1 + q2 + q4),  sin(q1 + q2 + q4),  0, (3*cos(q1 + q2))/10 + (2*cos(q1))/5]
[ sin(q1 + q2 + q4), -cos(q1 + q2 + q4),  0, (3*sin(q1 + q2))/10 + (2*sin(q1))/5]
[                 0,                  0, -1,                              3/5 - q3]
[                 0,                  0,  0,                                    1]
```

Analytically, in this case it's pretty easy to check that this result is correct, as the robot can only rotate in the z axis of the World frame (the first two links correspond to a conventional two-link planar manipulator), with $z_4 = -z_0$ fixed, the final joint only changes this orientation but not the position, and the height (z-coordinate) is determined by q_3 and the dimensions of the robot. In the end, we can check the obtained result with our calculation:

$$T_4^0(q) = \begin{bmatrix} c_{124} & s_{124} & 0 & a_1c_1 + a_2c_{12} \\ s_{124} & -c_{124} & 0 & a_1s_1 + a_2c_{12} \\ 0 & 0 & -1 & d_1 + d_4 - q_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c_{124} & s_{124} & 0 & 0.4c_1 + 0.3c_{12} \\ s_{124} & -c_{124} & 0 & 0.4s_1 + 0.3c_{12} \\ 0 & 0 & -1 & 0.6 - q_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where $c_{ij} = \cos(q_i + q_j)$ and $s_{ij} = \sin(q_i + q_j)$, changed for simplicity. To check for any given configuration, we can create another function, which will just substitute the value in the matrix, and then extract the position of the end effector (last column) and its orientation. The coded function is specific for our SCARA robot, so it's named `scaraFK`:

```
function [Te, xe] = scaraFK(Tq, qi)
    q = symvar(Tq);
    assume(q,'real')

    Te = double(subs(Tq, q, qi));
    pe = Te(1:3,4);

    % SCARA-Specific
    phie = [0; 0; atan2(Te(2,1), Te(1,1))];
    xe = [pe; phie];
end
```

It is clear that the first step (getting the symbolic variables from T and substituting for the given ones) is valid for any robot, but then we get the orientation in the Z axis knowing that matrix T has the form shown above, and we set the other rotations at zero knowing that the robot cannot rotate in any other axis, so the obtained vector of the End Effector, x_e, is only valid for our SCARA robot. Checking a numerical result for an easy configuration, we get:

```
% Test one configuration
qI = [pi/2 -pi/2 0.4 pi/2];
[TI, xI] = scaraFK(T04, qI);
```

$$T_4^0(q_I) = \begin{bmatrix} 0 & 1 & 0 & 0.3 \\ 1 & 0 & 0 & 0.4 \\ 0 & 0 & -1 & 0.2 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad x_e(q_I) = \begin{bmatrix} 0.3 \\ 0.4 \\ 0.2 \\ 0 \\ 0 \\ \pi/2 \end{bmatrix}$$

With these easy to visualize rotations and the help of the drawing on the first exercise, we can see that in this configuration, the end effector has moved away (from our perspective) in $y_0(+)$ a distance $p_y = a_1 = 0.4$; now is just at a distance $p_x = a_2 = 0.3$ in $x_0(+)$ from the origin, and the final height is just $p_z = 0.6 - 0.4 = 0.2$. From the joint values we can see how the final orientation is $\phi_z = \pi/2 - \pi/2 + \pi/2 = \pi/2$. In the matrix, it is clear how the end effector x axis is now pointing in the direction of $y_0(+)$ and its y axis in the direction of $x_0(+)$, which matches to a final rotation of a quarter of a turn.

b) Jacobians:

I. Geometric Jacobian

As in the last section, here we could proceed applying the formulas to our specific case, but it is more interesting and useful to code a general function that calculates the Geometric Jacobian of any given robot.

The Geometric Jacobian needs all matrices A_i^{i-1}, so they must be an input of the function. If they contain symbolic variables for the joints, then it's easy to know if the joints are revolute or prismatic looking at element (3,4), corresponding to d_i. If it has a symbolic variable, then the joint is prismatic. However, if we want the function to work with numerical inputs too, there is no way of distinguishing the type of joint from a general transform, as d_i could be a constant different from zero in a revolute joint, so that term doesn't prove anything, and θ_i, a_i and α_i could also be non-zero in a prismatic joint producing a rotation and a translation.

In the end, coded function `getJacob` can be called with one input parameter (array of matrices A_i^{i-1} in variable '`An`') or two, adding the `sigma` parameter representing the joint types as before. If input '`An`' has symbolic joint variables, `sigma` can be omitted.

First of all, the function first checks if it was called with just one input. If it was, then looks at the class of the input variable. If it is numerical, it stops with an error, as it wouldn't be able to proceed. If it is symbolic, assumes the symbolic variables are the joints, and finds the joint type looking for $d_i = f(q_i)$. If the type is something else, it stops with another error. The following code snippet shows these first commands of the function:

```
n = length(An(1,1,:));
jtype = zeros(1,n);
if nargin < 2
    if isa(An, 'double')
        error(['Joint types unknown for numerical transforms', ...
            '\n\nsigma can only be omitted with symbolic An'], '');
    elseif isa(An, 'sym')
        % Joint types deduction
        for i=1:n
            jtype(i) = ~isempty(symvar(An(3,4,i)));
        end
    else
        error(['Invalid data type for input: An', ...
            '\n\nAn must be numerical or symbolic An(q)'], '');
    end
```

```
else
    if n ~= length(sigma)
        error(['Incoherent number of joints.', ...
    '\nLength of sigma and 3rd dimension of An must be the same'],'');
    end

    [ ! ] OMITTED: Parse joint types using sigma as in getKinModel

end
```

If there is a second input, then the program checks if the length is coherent (the number of joint matches), and then just performs the same assignment operation as seen in function `getKinModel`, to get a binary array from numbers or text.

After this, the function just proceeds as we would in a script specific for one robot, but generalizing. First, it preallocates symbolic variables `p` and `z`, formed by n+1 vectors, from 0 to n (due to Matlab indexing, they will be from 1 to n+1, so index i actually refers to p_{i-1} and z_{i-1}). After setting the first ones as the World reference origin and z axis, the rest are just computed using a successive multiplication of matrices A_i^{i-1}, as usual. The two arrays of vectors are computed by the following code block:

```
p = 0*sym('p',[4,1,n+1]);
z = 0*sym('z',[3,1,n+1]);
p0 = [0 0 0 1]';
z0 = [0 0 1]';

% Index i corresponds to p and z of i-1
p(:,:,1) = p0;
z(:,:,1) = z0;
Tp = eye(4);
for i=1:n
    Ai = An(:,:,i);
    Tp = Tp*Ai;
    p(:,:,i+1) = Tp*p0;
    z(:,:,i+1) = Tp(1:3,1:3)*z0;
end
```

Finally, the code computes the Jacobian applying directly the formulas:

```
J = 0*sym('J',[6,n]);
for i=1:n
    if jtype(i) == 0
        % Revolute joint
        J(1:3,i) = cross(z(:,:,i), p(1:3,:,n+1)-p(1:3,:,i));
        J(4:6,i) = z(:,:,i);
    else
        % Prismatic joint
        J(1:3,i) = z(:,:,i);
        J(4:6,i) = [0,0,0]';
    end
end
J = simplify(J);

if isa(An,'double'), J = double(J); end
```

Once again, we preallocate symbolic variable `J` to avoid changing its size on each iteration, then after applying the formulas (depending on joint type), we simplify the resulting symbolic expression, and if the input was numerical, then we convert back to numbers.

On the main script, we can call the function with the obtained results from the Forward Kinematics operations, get the general expression of the Jacobian, and also check the results for a given configuration, both calling the function with a numerical array of matrices A, and substituting directly on the symbolic Jacobian. The used code is the following:

```
%% 2b) Jacobians
% 2b I) Geometric Jacobian
J = getJacob(Ai4);

% Test for same configuration
JI = double(subs(J, q, qI));
JI2 = getJacob(double(subs(Ai4,q,qI)),'rrpr');
```

The Jacobian results in the following expression:

$$J(q) = \begin{bmatrix} -0.4s_1 - 0.3s_{12} & -0.3s_{12} & 0 & 0 \\ 0.4c_1 + 0.3c_{12} & 0.3c_{12} & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

With the easy geometry of the SCARA robot, we can see that the results make sense. The Jacobian represents a relation in velocities between the joints (columns) and the end effector pose (rows). All revolute joints can change the orientation of the end effector in the z axis, so the 1s on the last row are easy to check. Rows 4 and 5 are obviously null, as there is no way for the end effector to rotate in these axes. Joint 3 is the only one that changes the z position, and when q_3 increases, the robot goes down, so this -1 is also easy to see. Finally, the first 2x2 minor corresponds to the changes of position (or velocities) of the end effector depending on the first two revolute joints. These match perfectly to those of a two-link planar manipulator, with the movements in x depending on the sines and those in y on the cosines, which makes sense, as a revolute joint will cause velocities perpendicular to the link (radius): when the robot is at $\vec{0}$, the position reaches its maximum in x, but then the manipulator can only move in y (this is also a singularity, as the first row of the Jacobian becomes all 0. We actually know that the Jacobian will be rank-deficient for all configurations where the manipulator is completely stretched or retracted, i.e. $q_2 = 0$, and if physically possible, $q_2 = \pi$). For the configuration chosen before, the Jacobian becomes (both JI and JI2 are equal):

$$J(q_I) = \begin{bmatrix} -0.4 & 0 & 0 & 0 \\ 0.3 & 0.3 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

Which, of course, makes sense with the given explanation. In this case, $q_1 = \pi/2$ and $q_2 = -\pi/2$, so the manipulator is forming an inverted L shape seen from above. Moving only the 2nd joint does not generate a velocity in the x axis, as the second link is aligned with it. With the XY coordinates of the end effector being $(0.3, 0.4)$, it is clear how turning the first joint, which is at the origin, with a positive rotation would result in a velocity perpendicular to the position vector, and indeed $(-0.4, 0.3)$ is perpendicular and in the correct sense.

II. Analytic Jacobian

For the Analytic Jacobian, we can code another function, `getJacobA`, to compute all the derivatives for us. In this case, inputs are the final FK transform matrix, $T_n^0(q)$, and an additional optional parameter representing the chosen set of Euler angles. If not specified, the function works with the default expressions for ZYZ angles, but these might not get correct results in some specific cases, like, coincidentally, with the SCARA manipulator.

The usual formulas for ZYZ Euler angles are:

$$T_n^0(q) = \begin{bmatrix} T(\Phi_e) & p_e \\ 0 & 1 \end{bmatrix} \rightarrow T(\Phi_e) = R_e = \begin{bmatrix} r_{ij} \end{bmatrix}$$

$$\begin{cases} \varphi = \text{atan2}(r_{23},\ r_{13}) \\ \theta = \text{atan2}\left(\sqrt{r_{13}^2 + r_{23}^2},\ r_{33}\right) \\ \psi = \text{atan2}(r_{32},\ -r_{31}) \end{cases}$$

In our case, we know that there is only one possible rotation, so there is no need for three angles. But computing with our rotation matrix:

$$T_4^0(q) \rightarrow R_e(q) = \begin{bmatrix} c_{124} & s_{124} & 0 \\ s_{124} & -c_{124} & 0 \\ 0 & 0 & -1 \end{bmatrix} \rightarrow \begin{cases} \varphi = \text{atan2}(0,\ 0) = 0 \\ \theta = \text{atan2}\left(\sqrt{0+0},\ -1\right) = \pi \\ \psi = \text{atan2}(0,\ 0) = 0 \end{cases}$$

It is clear that this doesn't work, as one angle should depend on the configuration. This is why the second input of the function is necessary, and specially in our case. To choose the corresponding Euler angles, we have to take a look at the general expression of a Rotation matrix using the ZYZ angles $[\varphi, \theta, \psi]$:

$$R = \begin{bmatrix} c_\varphi c_\theta c_\psi - s_\varphi s_\psi & -c_\varphi c_\theta s_\psi - s_\varphi c_\psi & c_\varphi s_\theta \\ s_\varphi c_\theta c_\psi + c_\varphi s_\psi & -s_\varphi c_\theta s_\psi + c_\varphi c_\psi & s_\varphi s_\theta \\ -s_\theta c_\psi & s_\theta s_\psi & c_\theta \end{bmatrix}$$

Comparing it to the end effector rotation, we can directly see that $\cos\theta = -1$, so there is no other option but $\theta = \pm\pi$, which matches the results seen before, and corresponds to the flip in z: $z_4 = -z_0$. Substituting in the general rotation matrix this angle, we now have:

$$R = \begin{bmatrix} -c_\varphi c_\psi - s_\varphi s_\psi & c_\varphi s_\psi - s_\varphi c_\psi & 0 \\ -s_\varphi c_\psi + c_\varphi s_\psi & s_\varphi s_\psi + c_\varphi c_\psi & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Taking the two non-zero elements of the first column:

$$c_{124} = -c_\varphi c_\psi - s_\varphi s_\psi, \qquad s_{124} = -s_\varphi c_\psi + c_\varphi s_\psi$$
$$\frac{s_{124}}{c_{124}} = \frac{= -\left(s_\varphi c_\psi - c_\varphi s_\psi\right)}{-\left(c_\varphi c_\psi + s_\varphi s_\psi\right)} = \frac{-\sin(\varphi - \psi)}{-\cos(\varphi - \psi)} \rightarrow q_1 + q_2 + q_4 = \varphi - \psi$$

And now we could choose any combination between them. But, of course, the two easy ones are making one or the other zero. Then we would have:

$$\varphi = 0 \rightarrow \psi = -q_1 - q_2 - q_4 \rightarrow R = \begin{bmatrix} -c_\psi & s_\psi & 0 \\ s_\psi & c_\psi & 0 \\ 0 & 0 & -1 \end{bmatrix}$$
$$\psi = 0 \rightarrow \varphi = q_1 + q_2 + q_4 \rightarrow R = \begin{bmatrix} -c_\varphi & -s_\varphi & 0 \\ -s_\varphi & c_\varphi & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Choosing any of the two options, we can apply the formulas as usual, and also code a more general function. First, we handle the optional Euler angles input:

```
TPhie = Tq(1:3, 1:3);
if nargin < 2
    phi = atan2(TPhie(2,3), TPhie(1,3));
    theta = atan2(sqrt(TPhie(1,3)^2 + TPhie(2,3)^2), TPhie(3,3));
    psi = atan2(TPhie(3,2), -TPhie(3,1));
else
    phi = Eul(1);
    theta = Eul(2);
    psi = Eul(3);
end
```

Then we just need to apply the formulas. We could derive the full 6x1 vector at once (by each joint variable), but it is clearer separating the linear and the angular parts. The function ends up being as follows:

```
Pe = Tq(1:3,4);
Phie = [phi;theta;psi];

q = symvar(Tq);
n = length(q);

JA = 0*sym('JA',[6 n]);
for i=1:n
    JA(1:3,i) = diff(Pe,   q(i));
    JA(4:6,i) = diff(Phie, q(i));
end
JA = simplify(JA);
```

This function only works with symbolic inputs, as it needs to derive them, and returns a symbolic Analytic Jacobian. On the main script, we can code, for both mentioned angle options:

```
% 2b II) Analytic Jacobian
theta = pi;

phi1 = q(1) + q(2) + q(4);
psi1 = 0;
JAphi = getJacobA(T04, [phi1; theta; psi1]);

phi2 = 0;
psi2 = -q(1) - q(2) - q(4);
JApsi = getJacobA(T04, [phi2; theta; psi2]);
```

And we get, for $\psi = 0$:

$$J_A(q) = \begin{bmatrix} -0.4s_1 - 0.3s_{12} & -0.3s_{12} & 0 & 0 \\ 0.4c_1 + 0.3c_{12} & 0.3c_{12} & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

This is expected, as both θ and ψ, corresponding to the last two columns, are constant, and do not contribute into any velocity of the end effector. For $\varphi = 0$, the ones on the fourth row here are moved to the last one, with their signs changed. This also makes sense, as this second version of Euler angles has the orientation of the end effector on the third component, and in the negative z direction. Even if just comparing the matrices we could see the relation between both Jacobians (Geometric and Analytic), so that φ or ψ becomes the orientation in the z axis, ϕ_z, the formal equation that relates them is the following:

$$J = T_A(\Phi_e) \cdot J_A(q), \qquad T_A(\Phi_e) = \begin{bmatrix} I_3 & 0_3 \\ 0_3 & T(\Phi_e) \end{bmatrix}, \qquad T(\Phi_e) = \begin{bmatrix} 0 & -s_\varphi & c_\varphi s_\theta \\ 0 & c_\varphi & s_\varphi s_\theta \\ 1 & 0 & c_\theta \end{bmatrix}$$

$$\psi = 0 \rightarrow T(\Phi_e) = \begin{bmatrix} 0 & -s_\varphi & 0 \\ 0 & c_\varphi & 0 \\ 1 & 0 & -1 \end{bmatrix}, \qquad \varphi = 0 \rightarrow T(\Phi_e) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{bmatrix}$$

It is clear how multiplying these matrices by their respective Analytic Jacobians, we would obtain the same result, and it would also be equal to the Geometric Jacobian: in the first case, both terms depending on φ would multiply elements from the 5th row of J_A, but they are all zero. Then only the 1 from the last row selects elements from the 4th row of J_A, putting them in the final row as desired. In the second case, something similar happens, as the 4th and 5th rows of J_A are zero. Only the -1 changes the sign from elements of the last row, and keeps them there. In the code, we can multiply the corresponding matrices to check that they result in J.

We now have three ways of representing the end effector pose, the two derived from these two possible Euler angles $[\varphi,\ \theta = \pi,\ \psi]$ and the one the Geometric Jacobian gives and we used on the Forward Kinematics function: $\phi = [0,0,\phi_z]$.

c) Inverse Kinematics, closed method:

To compute the Inverse Kinematics of our robot in a closed form, we need to express the joint variables as functions of the end effector pose and the DH parameters analytically. For this, it's useful to take a look at the final FK transform matrix, and express it in terms of normal, sliding and approach (n, s, a) and position of the end effector:

$$T_4^0 = \begin{bmatrix} c_{124} & s_{124} & 0 & a_1c_1 + a_2c_{12} \\ s_{124} & -c_{124} & 0 & a_1s_1 + a_2c_{12} \\ 0 & 0 & -1 & d_1 + d_4 - q_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} n_x & s_x & a_x & p_x \\ n_y & s_y & a_y & p_y \\ n_z & s_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

From here we can extract the following useful equalities:

$$p_x = a_1c_1 + a_2c_{12}, \qquad p_y = a_1s_1 + a_2c_{12}, \qquad p_z = d_1 + d_4 - q_3, \qquad \frac{n_y}{n_x} = \frac{s_{124}}{c_{124}}$$

They can be used to express the joint variables depending on known terms of matrix T_4^0 and the also known DH parameters. The easiest one is the third, from the prismatic joint:

$$q_3 = d_1 + d_4 - p_z$$

Which is obtained just rearranging the equation and is easy to check. To get the value of the second joint variable, we use the end effector position in the XY plane:

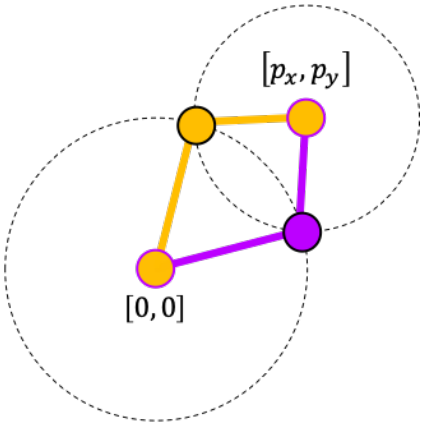
$$\begin{gathered} p_x^2 + p_y^2 = (a_1c_1 + a_2c_{12})^2 + (a_1s_1 + a_2s_{12})^2 \\ = a_1^2c_1^2 + 2a_1a_2c_1c_{12} + a_2^2c_{12}^2 + a_1^2s_1^2 + 2a_1a_2s_1s_{12} + a_2^2s_{12}^2 \\ = a_1^2 + 2a_1a_2(c_1c_{12} + s_1s_{12}) + a_2^2 \\ = a_1^2 + a_2^2 + 2a_1a_2\cos\big(q_1 - (q_1 + q_2)\big) = a_1^2 + a_2^2 + 2a_1a_2\cos(-q_2) \\ \rightarrow p_x^2 + p_y^2 = a_1^2 + a_2^2 + 2a_1a_2c_2 \end{gathered}$$

This means that the distance from the origin to the end effector position in the XY plane squared (the hypotenuse squared of a right triangle with legs p_x and p_y) only depends on constant DH parameters and joint variable 2. Rearranging terms, we get:

$$c_2 = \frac{p_x^2 + p_y^2 - a_1^2 - a_2^2}{2a_1a_2}, \qquad s_2 = \pm\sqrt{1 - c_2^2}, \qquad q_2 = \text{atan2}(s_2, c_2)$$

We use the arctangent instead of directly extracting the joint value from the cosine to be more correct, as cosines don't define a unique angle. In fact, we can see that there are two possible solutions for variable q_2 for the same configuration, depending on the sign of the sine. This is expected, as usually the Inverse Kinematics problem has multiple solutions because an end effector pose can be reached with multiple different joint configurations. We must keep both cases in mind, as they will affect the following calculations.

For our robot, it is geometrically easy to see that if we know the points $[0,0]$ and $[p_x, p_y]$, and also the link lengths, the two possible solutions are the intersections of the circles centered on the points with their radii equal to the lengths. The following figure shows the two possible configurations to obtain the same end effector position (on the XY plane seen from above):



Clearly, each possibility for variable q_2 has a different value for q_1 associated, and they cannot be changed between them. To get the value of q_1 depending on the final pose and DH parameters, we can use the following expressions:

$$p_y = a_1 s_1 + a_2 s_{12} = a_1 s_1 + a_2(s_1 c_2 + c_1 s_2) = (a_1 + a_2 c_2) s_1 + a_2 c_1 s_2$$

$$p_x = a_1 c_1 + a_2 c_{12} = a_1 c_1 + a_2(c_1 c_2 - s_1 s_2) = (a_1 + a_2 c_2) c_1 - a_2 s_1 s_2$$

We now have two equations with two unknowns (as variable q_2 can be obtained from known values, it is not considered as an unknown anymore). We can isolate, for example, the cosine from the second equation, and substitute it on the first to get an expression for the sine:

$$c_1 = \frac{p_x + a_2 s_1 s_2}{a_1 + a_2 c_2}$$

$$p_y = (a_1 + a_2 c_2) s_1 + a_2 \frac{p_x + a_2 s_1 s_2}{a_1 + a_2 c_2} s_2 = (a_1 + a_2 c_2) s_1 + \frac{a_2 p_x s_2 + a_2^2 s_1 s_2^2}{a_1 + a_2 c_2}$$

$$p_y = \left(a_1 + a_2 c_2 + \frac{a_2^2 s_2^2}{a_1 + a_2 c_2}\right) s_1 + \frac{a_2 p_x s_2}{a_1 + a_2 c_2}$$

$$s_1 = \frac{p_y - \frac{a_2 p_x s_2}{a_1 + a_2 c_2}}{a_1 + a_2 c_2 + \frac{a_2^2 s_2^2}{a_1 + a_2 c_2}} = \frac{(a_1 + a_2 c_2) p_y - a_2 s_2 p_x}{(a_1 + a_2 c_2)^2 + a_2^2 s_2^2}$$

$$= \frac{(a_1 + a_2 c_2) p_y - a_2 s_2 p_x}{a_1^2 + 2 a_1 a_2 c_2 + a_2^2 c_2^2 + a_2^2 s_2^2} = \frac{(a_1 + a_2 c_2) p_y - a_2 s_2 p_x}{a_1^2 + 2 a_1 a_2 c_2 + a_2^2}$$

$$s_1 = \frac{(a_1 + a_2 c_2) p_y - a_2 s_2 p_x}{p_x^2 + p_y^2}$$

As we can see, the process is not as straightforward as with q_2, but we can obtain the desired sine expression nonetheless.

Proceeding analogously, isolating the sine on the first equation and substituting it on the second one, we get an expression for the cosine:

$$s_1 = \frac{p_y - a_2 c_1 s_2}{a_1 + a_2 c_2}$$

$$p_x = (a_1 + a_2 c_2)c_1 - a_2 \frac{p_y - a_2 c_1 s_2}{a_1 + a_2 c_2} s_2 = (a_1 + a_2 c_2)c_1 - \frac{a_2 p_y s_2 - a_2^2 c_1 s_2^2}{a_1 + a_2 c_2}$$

$$p_x = \left(a_1 + a_2 c_2 + \frac{a_2^2 s_2^2}{a_1 + a_2 c_2}\right) c_1 - \frac{a_2 p_y s_2}{a_1 + a_2 c_2}$$

$$c_1 = \frac{p_x + \frac{a_2 p_y s_2}{a_1 + a_2 c_2}}{a_1 + a_2 c_2 + \frac{a_2^2 s_2^2}{a_1 + a_2 c_2}} = \frac{(a_1 + a_2 c_2)p_x + a_2 s_2 p_y}{(a_1 + a_2 c_2)^2 + a_2^2 s_2^2}$$

$$c_1 = \frac{(a_1 + a_2 c_2)p_x + a_2 s_2 p_y}{a_1^2 + 2a_1 a_2 c_2 + a_2^2 c_2^2 + a_2^2 s_2^2} = \frac{(a_1 + a_2 c_2)p_x + a_2 s_2 p_y}{a_1^2 + 2a_1 a_2 c_2 + a_2^2}$$

$$c_1 = \frac{(a_1 + a_2 c_2)p_x + a_2 s_2 p_y}{p_x^2 + p_y^2}$$

Finally, with these two final expressions, the joint variable is can be expressed in terms of the tangent, to have the angle completely determined:

$$q_1 = \text{atan2}(s_1, c_1)$$

We can now see how, as both sine and cosine depend on the sine of q_2, we will get two separate values for this first joint depending on the chosen joint 2 value.

Finally, the last joint value can be obtained with the orientation of the end effector and the previously computed joint values:

$$\frac{n_y}{n_x} = \frac{s_{124}}{c_{124}} \rightarrow \text{atan2}(n_y, n_x) = q_1 + q_2 + q_4$$
$$q_4 = \text{atan2}(n_y, n_x) - q_1 - q_2$$

This value also depends on the chosen solution for the first two joints, and with it we finally have all the joint values determined, and the closed form for the IK found analytically.

In Matlab, we can code a function that obtains these joint values from a given numerical matrix T, the DH parameters and a flag to choose between the two possible solutions. As this method is specific for the SCARA manipulator, we have named the function `scaraIK`. To make the flag optional, it assumes a default of +1 if none is entered:

```
if nargin < 3
    flag = 1;
end
```

Then, the function checks that the input matrix T is numerical, extracts the needed data from it and from the DH matrix, and performs the operations from the presented formulas. The code is:

```
if isa(Ti, 'double'), qi = zeros(1,4);
else
    error(['Invalid data type for input: Ti',...
        '\nTi must be numerical'],'');
end

pe = Ti(1:3,4);
ne = Ti(1:3,1);

th = DH(:,1);
d = DH(:,2);
a = DH(:,3);

qi(3) = d(1) + d(4) – d(3) – pe(3);

c2 = (pe(1)^2+pe(2)^2 - a(1)^2 - a(2)^2)/(2*a(1)*a(2));
s2 = sign(flag)*sqrt(1-c2^2);

qi(2) = atan2(s2,c2);

s1 = ((a(1)+a(2)*c2)*pe(2) - a(2)*s2*pe(1))/(pe(1)^2+pe(2)^2);
c1 = ((a(1)+a(2)*c2)*pe(1) + a(2)*s2*pe(2))/(pe(1)^2+pe(2)^2);

qi(1) = atan2(s1,c1);

qi(4) = atan2(ne(2), ne(1)) - qi(1) - qi(2);

qi = qi-th';
qi([1:2,4]) = mod(qi([1:2,4])+pi, pi+pi) - pi;
```

The function includes parameters θ_i from the DH matrix in order to be as general as possible and allow us to use it with any SCARA robot, even one with initial offsets on the revolute joints, and also subtracts a possible offset d_3 of the prismatic joint. Finally, the angular variables are kept in an interval from $-\pi$ to π.

We can also code a variant of this function where only the end effector pose vector is needed, instead of the full T matrix. This comes with a downside, however, because with only the vector the function cannot know which representation has been chosen for the orientation, so we have three possibilities:

- $x_e = [p_x, p_y, p_z, 0, 0, \phi_z]^T$
- $x_e = [p_x, p_y, p_z, \varphi, \pi, 0]^T$
- $x_e = [p_x, p_y, p_z, 0, \pi, \psi]^T$

The previously coded function `scaraFK` only returns vectors in the first form, which matches with the representation given by the geometric Jacobian. To get pose vectors with any of the other two versions, which are the two proposed Euler angle representations, we have modified it, adding an extra input, `Eul`, that can be either `'phi'` or `'psi'`, or omitted.

After adding the modifications, the function is now:

```
function [Te, xe] = scaraFK(Tq, qi, Eul)
[OMITTED: Help text]

    if nargin < 3 || isempty(Eul)
        Eul = 0;
    elseif strcmp(Eul,'phi')
        Eul = 1;
    elseif strcmp(Eul, 'psi')
        Eul = 3;
    else
        error(['Invalid Eul input.\n\nPlease specify either',...
               ' ''phi'' or ''psi'' on Euler angle input Eul.', ...
               '\nLeave input blank for default (z rotation)'],'');
    end

    q = symvar(Tq);
    assume(q,'real')

    Te = double(subs(Tq, q, qi));
    pe = Te(1:3,4);

    % SCARA-Specific, geometric rotation:
    phiz = atan2(Te(2,1), Te(1,1));

    % Assigned to requested angle:
    if Eul == 0
        phie = [0;0;phiz];
    elseif Eul == 1
        phie = [phiz;pi;0];
    else
        phie = [0;pi;-phiz];
    end

    % Final pose vector:
    xe = [pe; phie];

end
```

With this modified function we can now easily get the pose vectors in all mentioned formulations, and it can be handy to check the Inverse Kinematics functions.

After this parenthesis to improve the FK function, we can go back to our objective with the IK problem. A different IK function (named `scaraIKx`) has been coded to get the joint variables just with the pose vector, as we wanted. Inside the body of the function, the only difference with respect to the previous `scaraIK` function is that a new variable `phiz` is assigned depending on the input `Eul` at the beginning:

```
if nargin < 4
    phiz = xi(6);
elseif strcmp(Eul,'phi'), phiz = xi(4);
elseif strcmp(Eul, 'psi'), phiz = -xi(6);
else, [ ! ] OMITTED: error message
end
```

And then instead of obtaining the orientation from elements of the matrix as before, we use this new variable on the computation of q_4:

```
qi(4) = phiz - qi(1) - qi(2);
```

This function also uses the general expressions of the DH parameters, so even if there are non-zero offsets on the joint variables, they are considered. The angular variables are also kept in a $[-\pi, \pi]$ interval.

With these two new functions coded, and the improved FK one, we can write, in our main script, to check our results:

```
% Test same configuration
qI_IK = scaraIK(TI, DH);
TI_IK = scaraFK(T04, qI_IK);

eIK = max(max(abs(TI-TI_IK)));

% Test new functions with Euler angles
[~, xIphi] = scaraFK(T04, qI, 'phi');
[~, xIpsi] = scaraFK(T04, qI, 'psi');
qIgeo_IK = scaraIKx(xI,DH,-1);
qIphi_IK = scaraIKx(xIphi,DH,-1,'phi');
qIpsi_IK = scaraIKx(xIpsi,DH,-1,'psi');

eIK_geo = max(abs(qIgeo_IK-qI));
eIK_phi = max(abs(qIphi_IK-qI));
eIK_psi = max(abs(qIpsi_IK-qI));
```

First, we call the first IK function with the previously obtained `TI` matrix (using FK for configuration `qI`), without specifying the sign of the sine (it will take the default). As there are two possible solutions, instead of comparing the obtained configuration vector, we apply the FK once again and compare the transform matrices. The error obtained is of the order of 1e-17, which is actually smaller than epsilon, the tolerance of Matlab, so both vectors are equal.

Then, we compute the pose vectors for the other two formulations (using Euler angles and the new FK function) which are:

$$\varphi \rightarrow x_e = [0.3 \quad 0.4 \quad 0.2 \quad \pi/2 \quad \pi \quad 0]^T$$
$$\psi \rightarrow x_e = [0.3 \quad 0.4 \quad 0.2 \quad 0 \quad \pi \quad -\pi/2]^T$$

With these and the previous `xI`, we can get the joint vectors for all three options using `scaraIKx`. Testing these functions in the command window, we have seen that the flag that gives the same solution as the original q_I is -1, so using that we can check the joint vector errors directly. Finally, they are all of the order of 1e-15, so all solutions are correct.

d) Inverse Kinematics, iterative method:

We have two main options for implementing this section: writing a function that performs an iterative computation, or creating a Simulink model with the corresponding scheme to simulate the iterations. And for each implementation, we have two different iterative methods available: the Jacobian (pseudo)inverse and the Jacobian transpose methods.

The idea of these methods is to find the configuration q_d corresponding to a desired pose x_d iteratively, from a given starting point q_0. Using FK, we obtain the initial pose of the end effector, $x_e = FK(q_0)$, and calculate the error to the desired pose: $e = x_d - x_e$. We want this error to tend to zero, so we can derivate this expression and find:

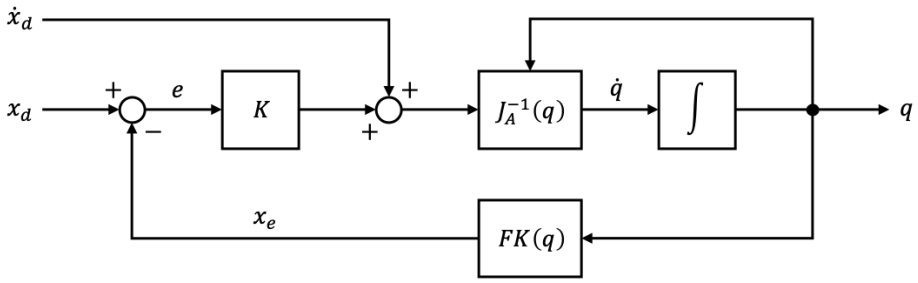
$$\dot{e} = \dot{x}_d - \dot{x}_e = 0 - J_A(q) \cdot \dot{q}$$

Where we have set the derivative of the desired position at zero, as it will not change over time, we only want the corresponding joint values of x_d. Here we find two options depending on the chosen method.

If we use the pseudoinverse, then we have:

$$\dot{q} = J_A^{-1}(q) \cdot (\dot{x}_d + K \cdot e) \rightarrow \dot{e} = \dot{x}_d - \dot{x}_e = -J_A(q) \cdot J_A^{-1}(q) \cdot (0 + K \cdot e) = -K \cdot e$$
$$\dot{e} + K \cdot e = 0$$

The scheme derived from this expression can be seen on the following figure:



This method is precise, but it requires the computation of an inverse at every iteration, making it very time consuming on the long term. The second method tries to solve this problem with a different approach, using the transposed Analytic Jacobian. The idea is that we have a nonlinear control system, so we can apply Lyapunov:

$$V = \frac{1}{2} e^T \cdot K \cdot e \rightarrow \dot{V} < 0$$
$$\dot{V} = e^T \cdot K \cdot \dot{x}_d - e^T \cdot K \cdot J_A(q) \cdot \dot{q} = -e^T \cdot K \cdot J_A(q) \cdot \dot{q}$$

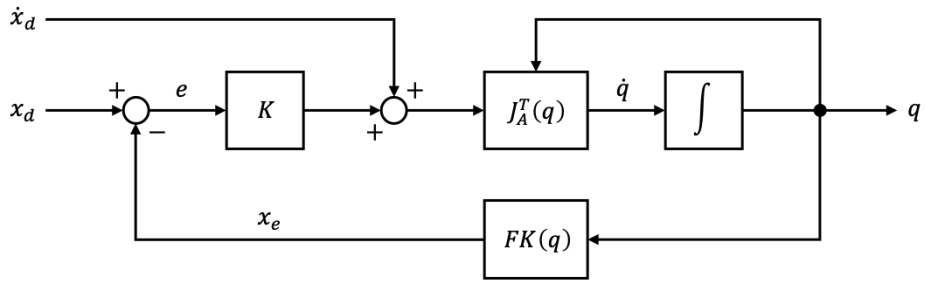
To make this derivative always negative, we try to find a quadratic form without using the inverse of the Jacobian. If we choose:

$$\dot{q} = J_A^T(q) \cdot K \cdot e$$

Then we also end up with a negative quadratic form:

$$\dot{V} = -e^T \cdot K \cdot J_A(q) \cdot J_A^T(q) \cdot K \cdot e < 0$$

The scheme derived from this second method can be seen on the following figure:



So, in the end, the only difference is whether we invert or transpose the Jacobian, which is a small change in the whole algorithm, so we can implement both without a lot of extra effort. In fact, also both mentioned implementations (function and Simulink file) have been coded, so now we will proceed to explain each one.

I. Matlab Function

This implementation is based on the following formula, defined iteratively:

$$q(k+1) = q(k) + \dot{q}(k) \cdot T_s$$

This means that for each iteration of our method, we will compute the joint variables depending on the previous ones plus the obtained velocity vector (using one of the two methods) multiplied by a fixed sample time T_s. When this expression converges, either because $q(k+1) = q(k)$ or the computed error $e = x_d - x_e$ is (almost) zero, we have obtained the desired joint variable vector.

In Matlab, we have coded function `scaraIKiter` to implement this algorithm. It has a lot of input parameters (a total of 10), but only the first 4 are mandatory. In order, it needs:

- The desired end effector pose vector, x_d.
- The FK transform matrix (depending on q), $T(q)$.
- The Analytic Jacobian (depending on q), $J_A(q)$.
- The sample time, T_s.
- The initial configuration seed, q_0
- The algorithm, `Alg`, a binary variable with 0 representing the inverse Jacobian algorithm, and 1 the transpose Jacobian one.
- The error gain, K.
- The error when the iterations can stop (tolerance), `eStop`.
- The Euler angle representation, which has to match the one used to compute x_d and J_A. If left blank, the "Geometric" ϕ_z approach is considered. If included, it must be either `'psi'` or `'phi'`. If not coherent with the rest, the algorithm doesn't work.
- A maximum time to find a solution, `Timeout`.

The iterative IK function internally calls `scaraFK` with parameter `Eul` in order to get the end effector pose x_e, which will be compared to the desired x_d to obtain the error, so this is why it is important to have coherent formulations, so the error makes sense and it can tend progressively to zero.

Also, the pseudoinverse computed in advance with the symbolic variable, so the computation time for the first algorithm is reduced drastically. There is only one problem with this, as we get the term $\sin(q_2)$ as a denominator in the resulting matrix, and it can be equal to zero. This has an easy workaround, substituting first on the original $J_A(q)$ and computing the pseudoinverse only when $q_2 = \{0, \pi\}$, so the computation time is still lower than doing an inverse on each iteration.

The main loop of the function can be seen on the following chunk of code:

```
JAqinv = simplify(pinv(JAq));
while eTh >= eStop
    if Alg == 0
        if qe(2) == 0 || qe(2) == pi
            JAinv = pinv(double(subs(JAq,q,qe)));
        else
            JAinv = double(subs(JAqinv, q, qe));
        end

        qdot = JAinv*K*e;
    else
        JA = double(subs(JAq, q, qe));
        qdot = JA'*K*e;
    end

    qe = qe + qdot'*Ts;
    qe([1:2,4]) = mod(qe([1:2,4])+pi,pi+pi)-pi;

    [~, xe] = scaraFK(Tq,qe,Eul);
    e = xd-xe;
    eTh = max(abs(e));

    [ ! ] OMITTED: Timeout handling

end
```

Here we can see how the error compared to the tolerance is actually the maximum across its components, and also the obtained angular joint variables are kept in a $[-\pi, \pi]$ interval to avoid convergence on other bigger multiples.

To test this function, we need a reachable end effector pose vector, with the orientation described in one of the three admitted possibilities. To get it, on the main script, we can code:

```
%% 2d) Inverse Kinematics (iterative)

% Desired position obtention
qd = [pi/2 -pi/3 0.3 -pi];
[Td, xd] = scaraFK(T04, qd, 'phi');
```

We also need to fix the input parameters of the function:

```
% Main parameters
q0 = [0 0 0 0];
K = 100;
eStop = 5e-4;
JA = JAphi;
Ts = 1e-3;
TfMax = 10;
Eul = 'phi';
```

These values will actually be used for both implementations of the iterative IK. Finally, we can call the function:

```
% 2d I) scaraIKiter function
qd_IKiterf = scaraIKiter(xd, T04, JA, Ts, q0, 0, ...
                         K, eStop, Eul, TfMax);
clc;
Td_IKiterf = scaraFK(T04, qd_IKiterf);
eIKiterf = max(max(abs(Td-Td_IKiterf)));
```

The function includes a display to see how the error evolves over time, and so we clear the command window after the function finishes. With the inverse Jacobian method and the shown parameters, in a few seconds we get the desired results. The error checked on the transform matrix is lower than 5e-4. We can try with errors of 1e-5 and the function also returns a correct solution in a few seconds (less than the 10 s timeout).

However, calling the function with the Jacobian transpose method doesn't converge as fast, and even with timeouts of 60 seconds we get errors around 5e-3. Changing K and T_s has not led to better results, as they are related, for a bigger gain K we need smaller sample time T_s for the algorithm to actually converge and not oscillate around the solution. Changing the initial seed q_0 helps the algorithm a lot, but it is still slower (converging) than the inverse Jacobian algorithm without the computational load of computing an inverse on every iteration.

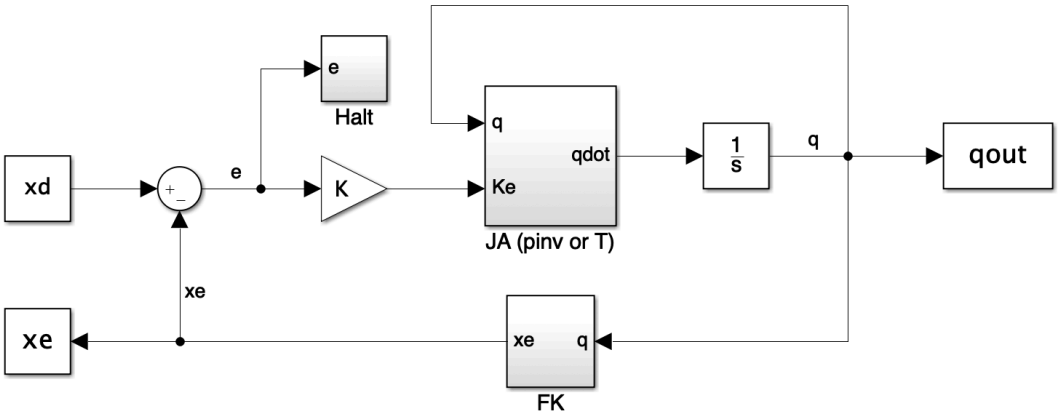
On the final script file, this step has been made optional, so the function is only called if the user wants to, known evaluating an input from the command line.

II. Simulink Model

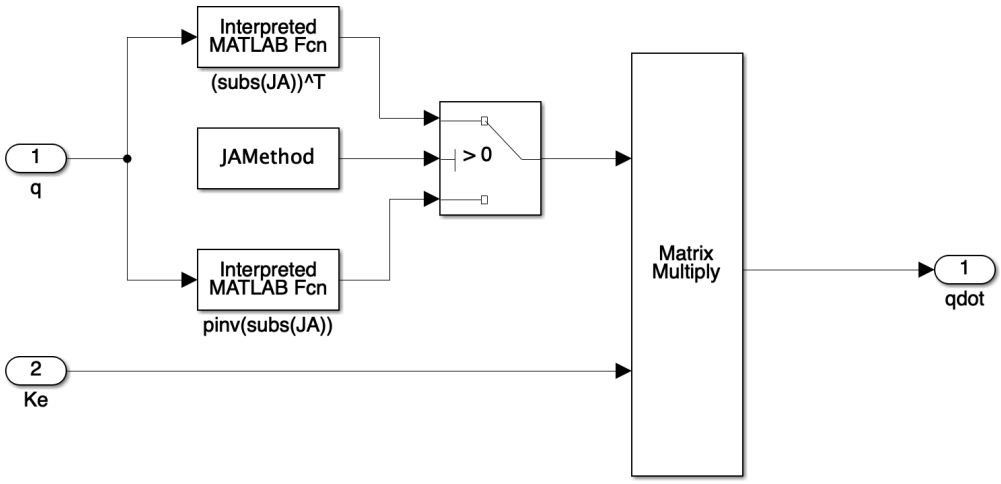
For this second implementation, we need to convert the seen block diagrams into a Simulink model, complete with all the parameters needed. Before getting into the model, in the main script we need to convert the chosen Euler angle description to a number, as Simulink cannot load strings as constants. We also initialize some variables used for plotting the results:

```
% Eul conversion
if strcmp(Eul,'phi'), EulSim = 1; Thz = 4; Thtxt = '\varphi';
elseif strcmp(Eul, 'psi'), EulSim = 3; Thz = 6; Thtxt = '\psi';
else, EulSim = 0; Thz = 6; Thtxt = '\phi';
end
```

The main Simulink model has the following structure:

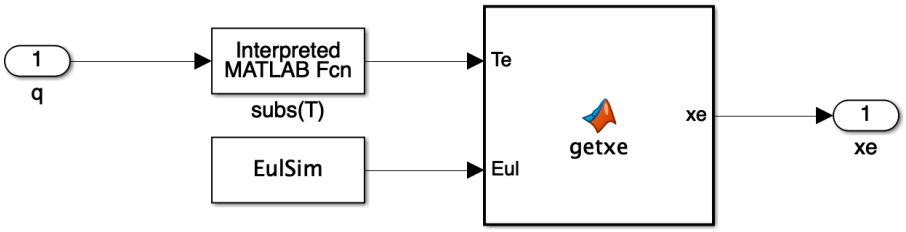


Basically, it follows the presented diagrams, getting the desired end effector pose from the workspace and saving the evolutions of both the joint variables and the current end effector pose. In the "JA" subsystem we have the following block diagram:



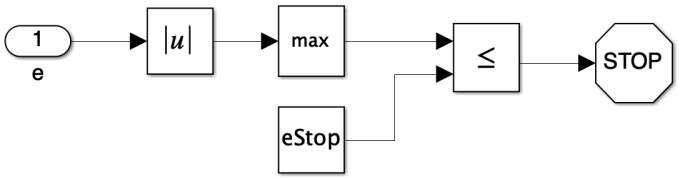
We have added a switch that allows us to choose the method before simulating, by setting a constant on the Matlab script (`JAMethod`, 0 or 1), so we can change without having to open the model or have more than one. In this case, even for the inverse Jacobian method, the program substitutes the symbolic variables for the current values in the Analytic Jacobian at every step, so the computational load is greater than in the previous function.

Going to the "FK" subsystem, we have the following blocks:



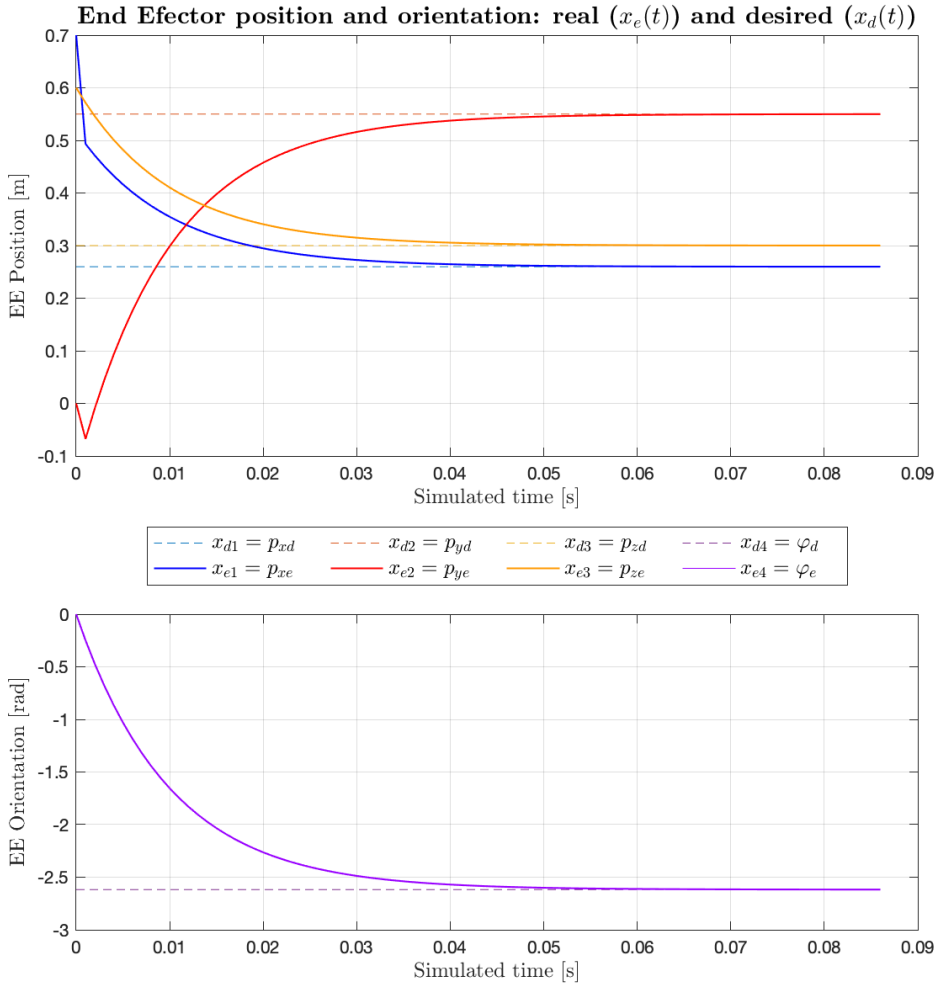
Here we substitute into the transform matrix $T_4^0(q)$ the current configuration, and then with the given Euler angle representation, we extract x_e on every step with a function.

Finally, the subsystem "Halt" is used to compute the maximum error across its components, compare it to the tolerance set as the maximum error to accept the solution, and when this threshold is reached, stop the simulation. This way of halting the program is better than a fixed simulation time, as it only consumes the time until convergence, and we know that if the (quite large) total simulation time is reached, probably the algorithm didn't converge.

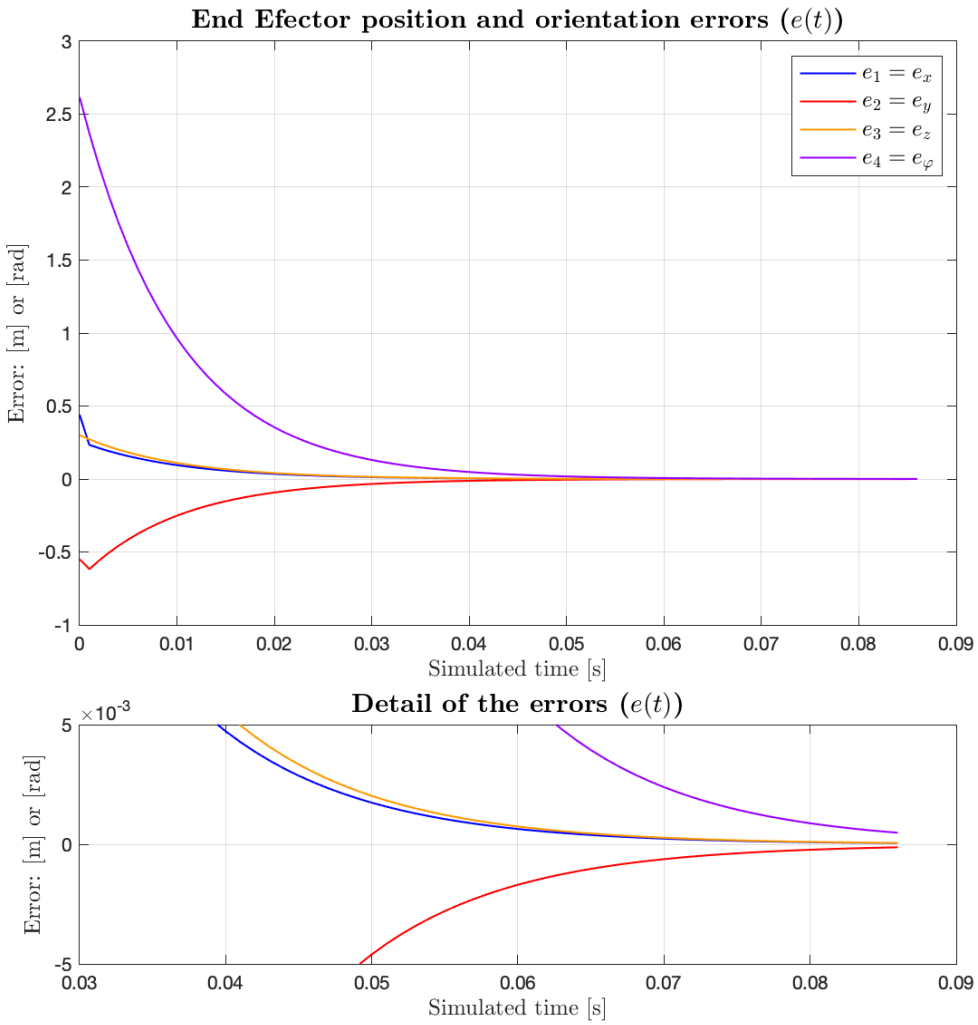


After adding scopes to visualize data when simulating, we have our final Simulink model ready to use. On the main script, we let the user decide the method to be used (and also if they want to perform the simulation or not), with a command window input evaluation, and then we simulate automatically. Two scopes are set to be opened on simulation start even with the model closed, so that the user can see how the pose and its error evolve over time.

Simulating with the previous parameters, setting a fixed time step simulation on Simulink, for the first method (inverse) we obtain a solution very quickly, in about 10 real time seconds, computing until less than 0.1 seconds in simulation time. The evolution of the end effector pose can be seen on the following figure:



The main script includes a block of code that treats the selected Euler angle description and the obtained data in order to plot the correct orientation signal, and to name it correctly on the legend too. For this inverse Jacobian method, and selecting φ as the variable angle, we can see how all the pose vector components tend to their desired values quite fast. The x axis corresponds to the Simulink internal simulated time, which ends up being very small, but as each step takes some time to compute, the final real time counting loading the model and the online plotting of the scopes has been of 10.8 seconds (got using `tic` and `toc` on the script). Subtracting the changing computed values from the desired ones, we get the error plots shown in the following figure:

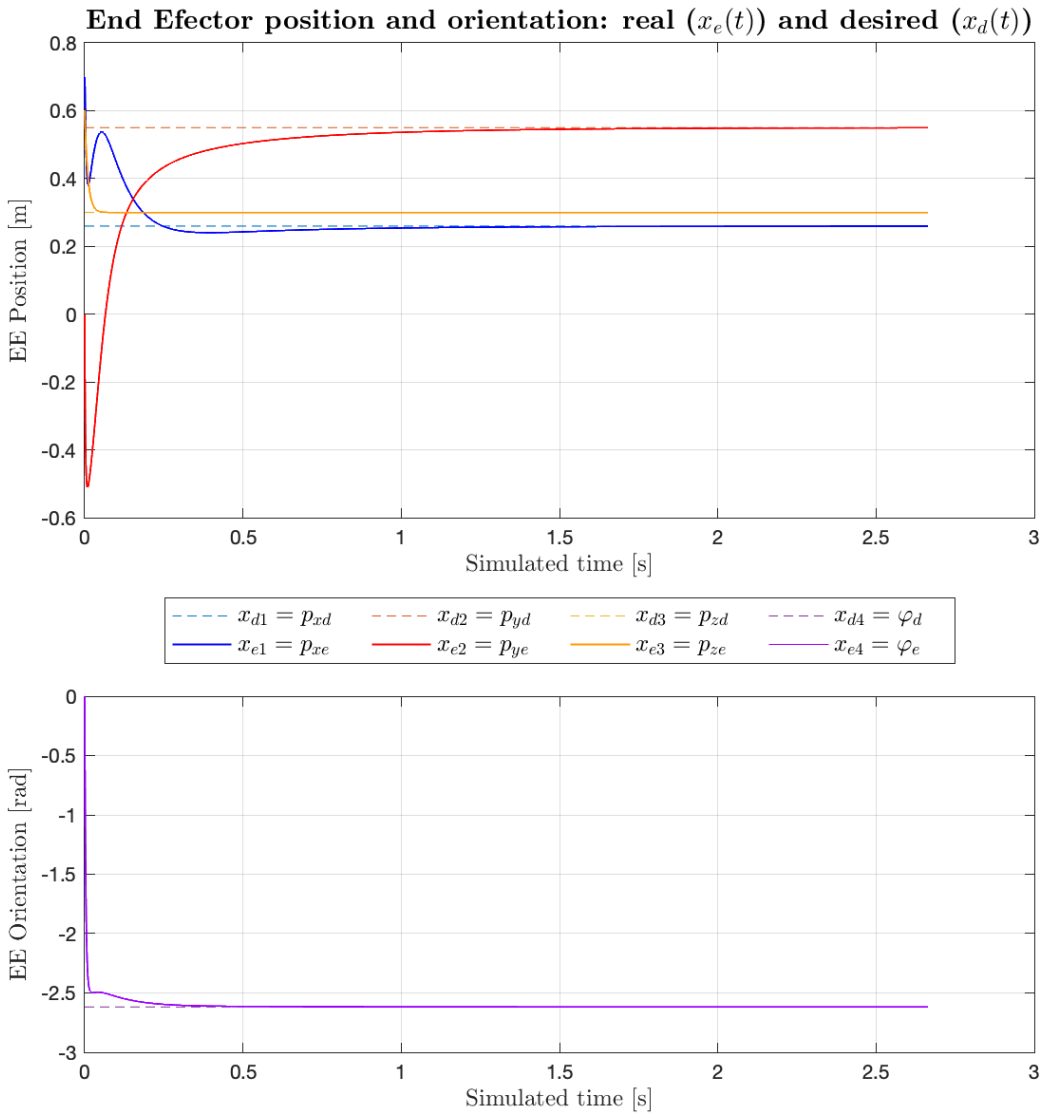


Here we can see how the orientation error is the maximum one for the whole simulation, so when it reaches the accepted tolerance, the program stops. The detail of the ending is very useful, as it gives more visual information about the evolution of the errors.

Simulating with the transpose method with the same parameters, we actually get a longer execution time, both real (slightly more than a minute) and simulated (about 2.7 seconds). Even increasing the sample time to 5 ms, the simulation time stays at a bit under 50 seconds. This might be due to the processing capabilities of the computer, but it is surprising, as the inverse method is way faster.

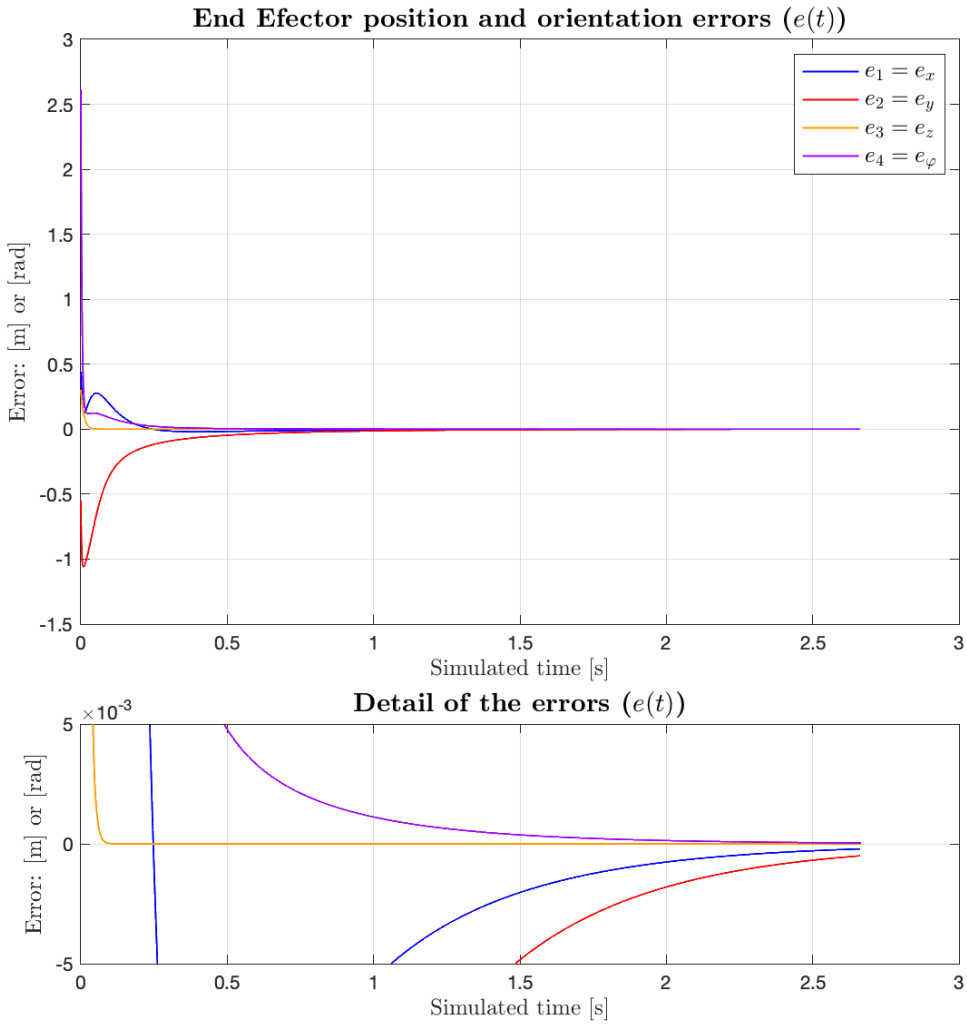
To be fair, the transpose method is a simplification, so it does not give the optimal and fastest convergence to the desired point. We know it will get there, and that in theory it will do more iterations per second than computing an inverse per iteration, but if the total iterations needed are too many, the inverse method can still be faster. Also, in a real application, we would probably call the IK iterative function to compute a trajectory, so the initial solution q_0 would be pretty close to q_d, and the transpose method gets better in this case.

Plotting the results obtained with the same parameters as the ones used for the first method, to compare fairly, we get the following figure:



The difference in the plots is quite noticeable. The total simulated time is much longer, but we can see how after 0.5 seconds of this time, the configuration seems to be almost the desired one. After this point, the algorithm needs a lot more time to get the error under the tolerance threshold, converging really slowly towards the final configuration. It seems clear that this convergence speed is determined by $\dot{q}$, and this is what changes between both methods. While the inverse Jacobian method kept a considerable velocity even when reaching the desired point, this transpose method slows down near the solution.

The corresponding error plot can be seen on the following figure:



Even if the errors seem to converge at the mentioned 0.5 simulated seconds mark, it is not until triple that time that the errors are contained within ±0.005, which for the position corresponds to 5mm of error, not a negligible amount for a robot with the used dimensions.

Changing the final desired configuration, the time step size, the gain or the initial configuration can modify the results considerably, but in all the tested combinations, the first method wins in both the function and the simulation implementations. This makes us think that for the SCARA manipulator, obtaining the pseudoinverse does not take so much computational time, so with the only drawback reduced, the results are obviously better. And if we can compute it only when strictly necessary like on the improved function, then this first method is the best to implement in all ways.

Exercise 3

Using the Robotics Toolbox, model the manipulator as a Serial Link object, and create a script file to verify the correctness of the functions implemented in the previous exercise.

Creating a Serial Link object that models the robot with this Toolbox is very easy once we have obtained the DH parameters. The commands are:

```
% Create SCARA manipulator
L(1) = Link([DH(1,:), 0]);
L(2) = Link([DH(2,:), 0]);
L(3) = Link([DH(3,:), 1]);
L(4) = Link([DH(4,:), 0]);
scara = SerialLink(L, 'name', 'SCARA [ALA]');
scara.links(3).qlim = [0 d1-d4];
```

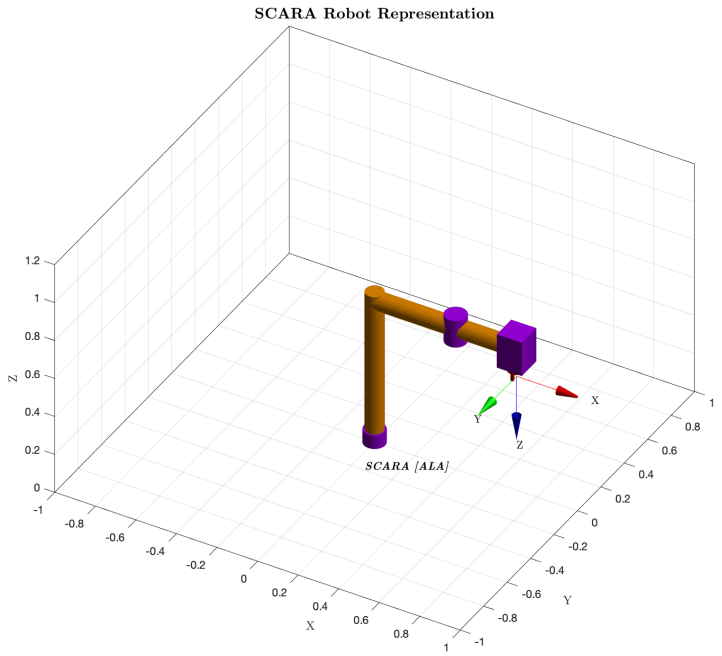
Where the additional parameter of the `Link` function corresponds to what `sigma` was in our custom functions, the joint type, 0 for revolute and 1 for prismatic. To verify the previously obtained results, we can set a configuration and use the available commands. The following code snipped shows how:

```
% Test and compare a configuration
qr = qd; %qd = [pi/2 -pi/3 0.3 -pi];

[Tr, xr] = scaraFK(T04,qr);
Tr_rtb = scara.fkine(qr);
eTr = max(max(abs(Tr-double(Tr_rtb))));

Jr = double(subs(J, q, qr));
Jr_rtb  = scara.jacob0(qr);
eJr = max(max(abs(Jr-Jr_rtb)));

TPhi1r = double(subs(TPhi1,q,qr));
TPhi2r = double(subs(TPhi2,q,qr));
TPhi_rtb = eul2jac(Tr_rtb.toeul);
eTPhi = max(max(min(abs(TPhi_rtb - TPhi1r), abs(TPhi_rtb - TPhi2r))));

qrIK1 = scaraIK(Tr, DH, +1);
qrIK2 = scaraIK(Tr, DH, -1);
qrIK_rtb = scara.ikine(Tr_rtb, 'mask', [1 1 1 0 0 1], ...
                               'q0', [1 -1 0 -3]);
eIKr = max(min(abs(qrIK1-qrIK_rtb), abs(qrIK2-qrIK_rtb)));
```

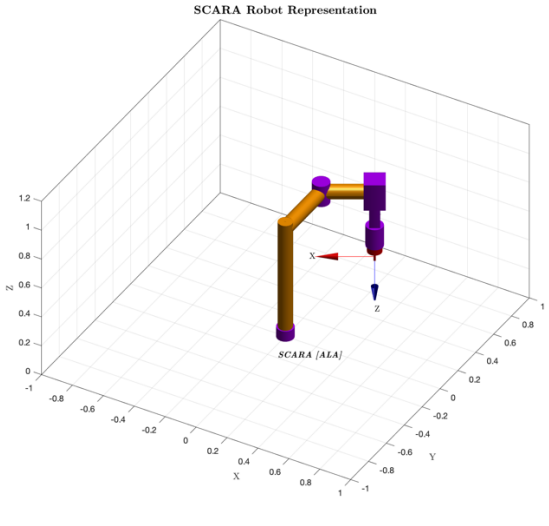
Finally, we obtain errors in the order of 1e-16 for the FK transform matrix, for the geometric Jacobian, and for matrix $T(\Phi_e)$ that relates both Jacobians. We actually get the minimum error between the two used Euler angle formulations, because at first, we don't know which will the Toolbox use. For the IK, we also get both solutions with our function, and compare them to the obtained using the Toolbox. Here `ikine` actually needs a starting point much closer to the final pose than our iterative function in order to converge (if it fails to converge it raises a warning that seems to imply that it only iterates 100 times before stopping). The maximum error between the two nearest joint vectors finally is around 1.7e-11, with flag -1 being the one that gives the same result as the Toolbox function. It is clear that all results are equivalent.

Finally, the main script finishes plotting the robot on the initial configuration q_0 for a couple of seconds, followed by the animation of the convergence to q_d (we can see how for the Jacobian inverse method this animation is really short and the robot goes quite fast, almost moving instantly on the first frames, and also how with the Jacobian transpose the robot reaches the final location coarsely quite fast, but then moves very slowly until completely reaching it). After this animation, the robot shows the animation of a trajectory from q_0 to q_r, created with `jtraj`, just to see the robot move normally and not representing the convergence of an algorithm. The last plot corresponds to the other IK solution for the obtained pose x_r using FK.

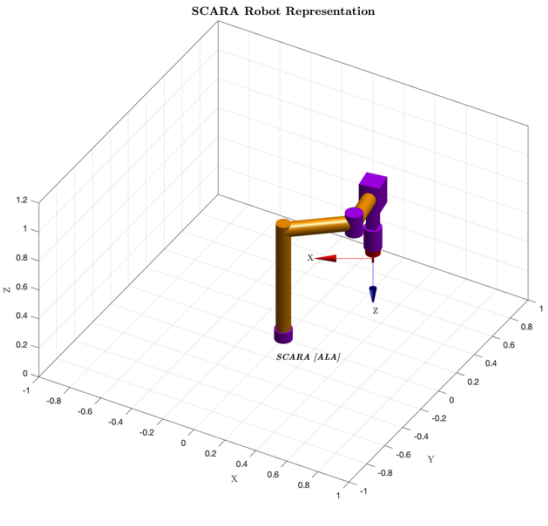
The following final figures represent the SCARA manipulator in the initial and final configurations (both alternatives for the latter). They have been made with a modified version of the Serial Link plot function, which makes links and arrows smaller, the text black, and most importantly, draws a bigger fixed box on prismatic joints, and a simple gripper at the tool point. These changes can be seen better on the videos in the attached "Animations" folder.



Initial configuration



Final configuration



Final configuration - Alternative