Robotic and Human Machine Interfaces

Project Report

Master degree of:

Embedded Computing Systems

Academic year 2015/2016

Submitted by:
Marco Celia

PERCRO lab. Scuola Superiore Sant'Anna, January 20, 2017.

Contents

List of Figures

1 Introduction

In this project the implementation of a robotic arm manipulator task with the Matlab *Robotics Toolbox* is presented. In particular, an Anthropomorphic arm, from now on Aarm, has been used. It has 6 Degree Of Freedom, from now on DOF, then could be considered, at least in the first discussion, as a non redundant manipulator since 6 are also the maximum DOF in a 3D environment. The task that the Aarm has to perform is a laser cutting task, in particular it is conceived with an laser beam emitter placed at the End-Effector, then the robotic holds a predefined pose from which repetitively computes a trajectory in order to cut polygonal objects following in an horizontal plane.

From the above description of the task to be computed it is possible to evince that the Aarm can be considered redundant with respect to such specific task, to prove this it is sufficient to note that it is useless to impose a specific orientation for the approach direction of the End-Effector.

In the first phase of this work will be analysed the Kinematic structure of the Aarm and computed feasible trajectory in the **Operational Space**, from points of such trajectory the respective joint angles are computed with three different type of kinematic inversion:

- **Inverse Kinematic:** performed exploiting the geometric properties of the Aarm links.
- **Inverse Differential Kinematic:** performed exploiting Jacobian's theory and numeric integration
- **Optimized Inv.Diff.Kinematic:** same as previous one but trying to optimize a further objective exploiting arm redundancy with respect to the assigned task

At the end of each procedure all results are evaluated and then compared each other by means of graphical representations.

2 Forward Kinematics

In order to compute the forward kinematcs of the arm all joint axes has been fixed using the Denavit-Hartenberg conventions.

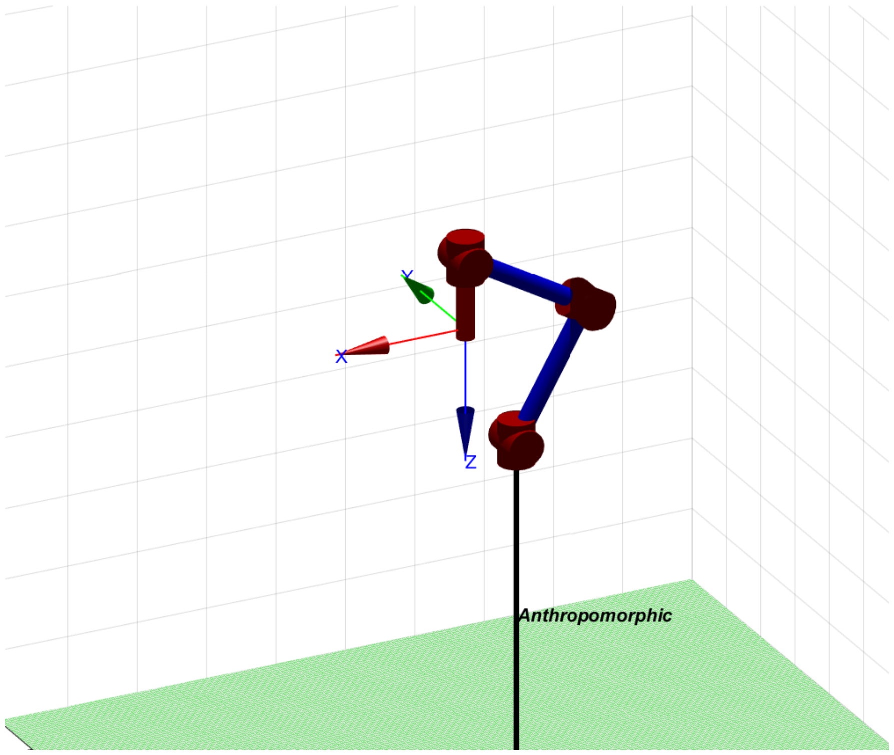
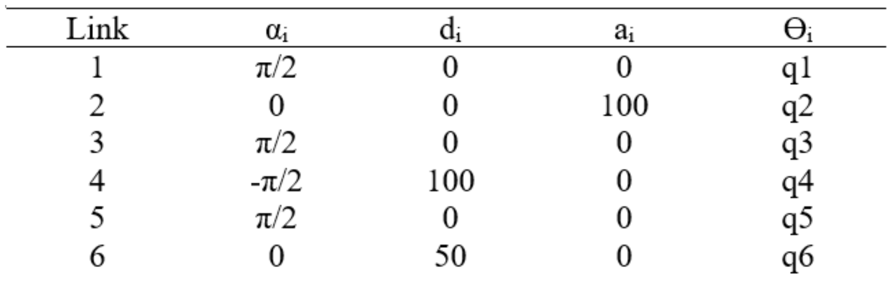


Figure 1: 6 DOF Antrhopomorphic Arm

Figure 1 shows the Aarm in its initial pose and Figure 2 the respective DH-parameters, being q_i the joint angles.



Link	α_i	d_i	a_i	Θ_i
1	$\pi/2$	0	0	q1
2	0	0	100	q2
3	$\pi/2$	0	0	q3
4	$-\pi/2$	100	0	q4
5	$\pi/2$	0	0	q5
6	0	50	0	q6

Figure 2: Denavit Hattenberg parameters

According to DH-convention, the generic homogeneous transformation converting coordinates of frame $i-1$ in coordinates of frame i is:

$$A_i^{i-1} = \begin{pmatrix} \cos(q_i) & -\sin(q_i)\cos(\alpha_i) & \sin(q_i)\sin(\alpha_i) & a_i\cos(q_i) \\ \sin(q_i) & \cos(q_i)\cos(\alpha_i) & -\cos(q_i)\sin(\alpha_i) & a_i\sin(q_i) \\ 0 & \sin(\alpha_i) & cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

computing the i^{-th} transformation as:

$$p_i = A_i^{i-1} \cdot p_{i-1}$$

The total forward kinematics can be expressed once defined two more homogeneous matrix that represents position and orientation of base and tool frame with respect to frame 0 and frame n respectively:

$$T_e^b = T_0^b T_n^0 T_e^n$$

where $T_n^0 = A_1^0(q_1)A_2^1(q_2)...A_n^{n-1}$.
Referring to Fig.2 it is possible to derive all transformation matrix for the Anthropomorphic manipulator:

$$A_1^0(q_1) = \begin{pmatrix} \cos(q_1) & 0 & \sin(q_1) & 0 \\ \sin(q_1) & 0 & -\cos(q_1) & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_2^1(q_2) = \begin{pmatrix} \cos(q_2) & -\sin(q_2) & 0 & a_2\cos(q_2) \\ \sin(q_2) & \cos(q_2) & 0 & a_2\sin(q_2) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_3^2(q_3) = \begin{pmatrix} \cos(q_3) & 0 & \sin(q_3) & 0 \\ \sin(q_3) & 0 & -\cos(q_3) & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_4^3(q_3) = \begin{pmatrix} \cos(q_4) & 0 & -\sin(q_4) & 0 \\ \sin(q_4) & 0 & \cos(q_4) & 0 \\ 0 & -1 & 0 & d_4 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_5^4(q_5) = \begin{pmatrix} \cos(q_5) & 0 & \sin(q_5) & 0 \\ \sin(q_5) & 0 & -\cos(q_5) & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_6^5(q_6) = \begin{pmatrix} \cos(q_6) & -\sin(q_6) & 0 & 0 \\ \sin(q_6) & \cos(q_6) & 0 & 0 \\ 0 & 0 & 1 & d_6 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3 Task Description

As previously said the robotic arm has to move along a predefined polygonal trajectory with the aim to cut surfaces which follows above it in a horizontal plane. The steps that manipulator performs in order to accomplish its objective are concerned as follows:

i. Every time begins a new instance of the task the robotic arm lies in a initial predefined pose, in such pose its x and y coordinates corresponds to the top-left corner of a rectangle, while its z coordinate distances the cutting plane of a certain amount. Then it starts to moving across down direction.

ii. Once height of the laser tool reaches a reasonable cutting-height it stops to moving down and then follows the rectangular trajectory. When all sides of such rectangle are completed it lies again in the top-left corner, so it starts again to move up to reach the predefined pose and wait a new task instance.

A graphical representation of the trajectory computed in one task instance is represented in Fig.3

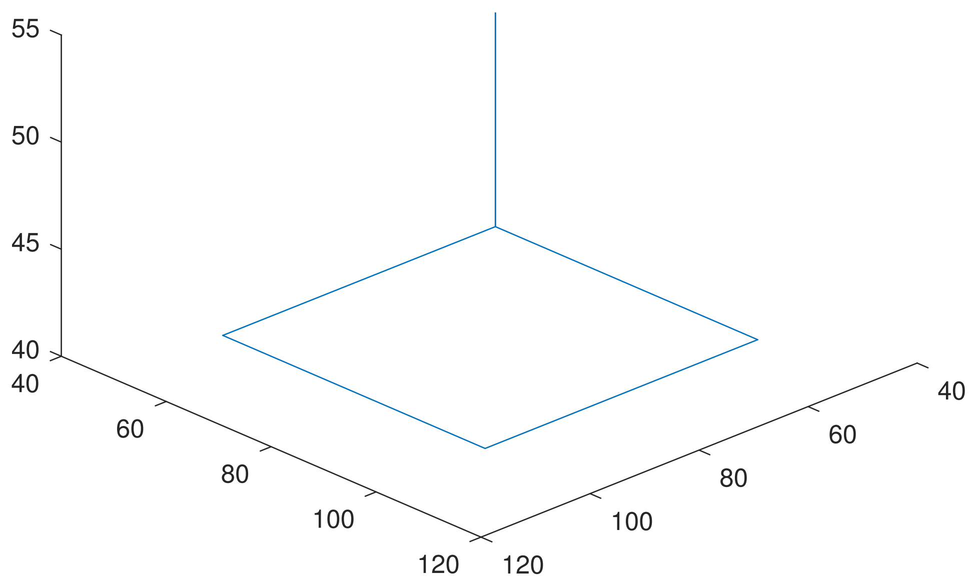


Figure 3: Task trajectory

Which in terms of coordinates is

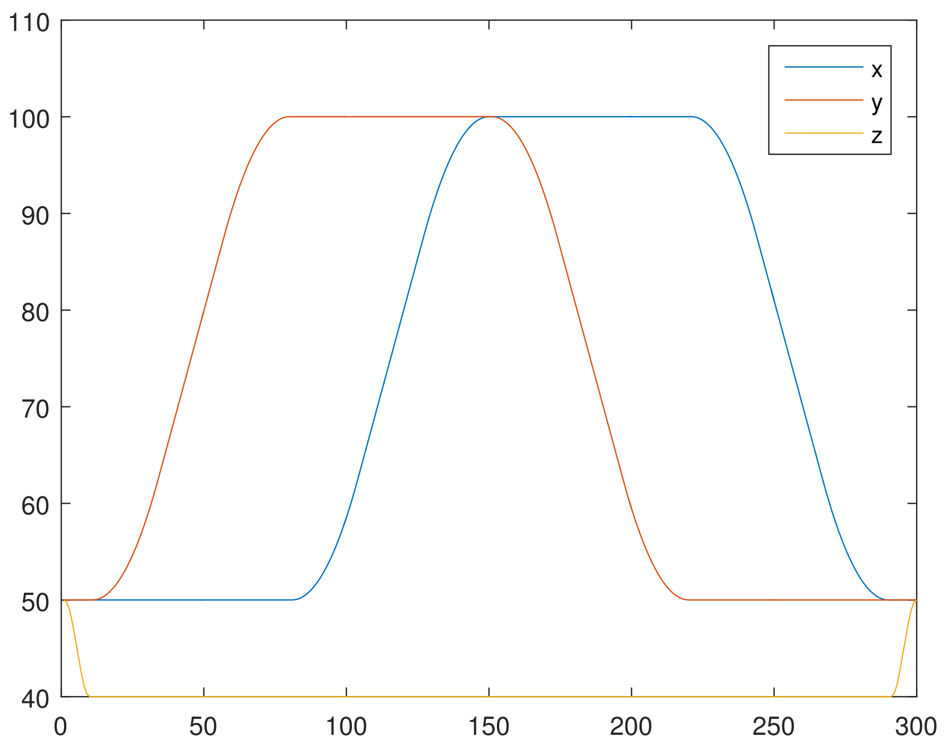


Figure 4: Task trajectory coordinates

Each trajectory side has been generated in the Cartesian space by means of the *ctraj* function of *Robotics Toolbox*, which takes as inputs two homogeneous matrix, representing initial and final pose, and a vector representing the number of point for each segment of the generated trajectory. Follows a stub of trajectory generation procedure:

```
function traj = rectraj(r, sps)
traj = zeros(4,4,sum(sps));
for i=1:7
t(:,:,i)= transl(r(i,:));
for j=1:6
if j==1
traj(:,:,1:sps(j)) = ctraj(t(:,:,j), t(:,:,j+1), sps(j));
else
traj(:,:,sum(sps(1:j-1))+1:sum(sps(1:j-1))+sps(j))=ctraj(t(:,:,j),t(:,:,j+1),sps(j));
end
end
```

Where r is an array representing a collection of rectangle vertices plus the initial point. It is computed through another simple function:

```
  r = rect(tl_corner, width, height);
```

4 Singularity Analysis

Kinematics singularity are special configurations of the manipulator such that Jacobian matrix is *rank deficient.* Regarding the anthropomorphic arm an effective way to determine such points could be decoupling those singularities which are proper of the spherical wrist (last three joints) from those regarding the shoulder (first three joints). A specular techniques will be adopted also in sec.5.1 to determine the inverse kinematics.

In order to understand the procedure let's analyse the Jacobian matrix structure, since it has 6 rotational joints

$$J(q) = \begin{bmatrix} z_0(p_e - p_0) & z_1(p_e - p_1) & z_2(p_e - p_2) & z_3(p_e - p_W) & z43(p_e - p_W) & z_5(p_e - p_W) \\ z_0 & z_1 & z_2 & z_3 & z_4 & z_5 \end{bmatrix}$$

It could be partitioned into four blocks regarding position and orientation of first and last three joints respectively

$$J(q) = \begin{bmatrix} J_{11}(q) & J_{12}(q) \\ J_{21}(q) & J_{22}(q) \end{bmatrix}$$

If we consider the end-effector frame originated in the center of spherical wrist, i.e $p_e = p_W$, blocks $J_{12} = 0_3$ and hence the Jacobian can be rewritten as

$$J(q) = \begin{bmatrix} J_{11}(q) & 0_3 \\ J_{21}(q) & J_{22}(q) \end{bmatrix}$$

which, from linear algebra theory, is a lower-triangular matrix for which is possible to compute determinant as

$$det(J(q)) = det(J_{11})det(J_{22})$$

Hence it is possible to compute singularity zeroing blocks regarding shoulder separately from those regarding wrist. It is important to notice that imposing $p_e = p_W$ does not influence the positions of such points, which are intrinsic of the manipulator's mechanical structure, but rather is an algebraic intuition to make simpler their computation.

4.1 Shoulder Singularities

From block $J(q)_{11}$ is possible to recognize that a singularity condition occurs in two manipulator's configuration, first when

$$q_3 = \frac{\pi}{2} \text{ or } q_3 = \frac{3\pi}{2}$$

That represents the so called *elbow singularity*, were joint 3 is outstretched or retracted. The second situation happens when the wrist center lies in the rotation axis of joint 1, i.e z_0, meaning that

$$p_{Wx} = p_{Wy} = 0$$

That represents the so called *shoulder singularity*, in this case any rotation about z_0 does not produce any motion to end-effector.

4.2 Wrist Singularities

From block $J(q)_{22}$ is possible to recognize that a singularity condition occurs when three vectors z_3, z_4 and z_5 are linearly dependent, hence whenever

$$q_5 = 0 \text{ or } q_5 = \pi$$

Indeed in this situation an opposite rotation on q_4 and q_6 does not produce any end-effector rotation.

From the trajectory proposed in 3 could be simply noted that shoulder singularity could not happen since starting x and y positions are greater than zero and manipulator never comes back. Wrist singularity it's also avoided since end-effector during the task has a fixed orientation which not allow the alignment of link 1 with link 2. Finally elbow singularity has been avoided experimentally verifying that in the farthermost point of the trajectory elbow won't be outstretched.

5 Kinematic Inversion

This section are described all the kinematic inversion procedures used to compute joint angles for the Aarm.

While direct kinematics equations provides a unique representation of the end-effector pose once joint variables are known, the problem of finding the joint variables given an end-effector pose,i.e inverse kinematics , is not any more straightforward. This is due to several reasons like the existence of multiple or infinite solutions for instance in cases of redundant manipulator.

5.1 Linear Kinematics Inversion

In this stage the kinematic inversion problem is faced exploiting the geometric properties of the Aarm, in particular, due to the existence of multiple solutions for a given end-effector pose, will be necessary to impose some constraint on the configuration to be adopted for the manipulator.

Analysing the geometry of the manipulator it's possible to note that the kinematic inversion can be decoupled in two different sub-problems, one relative to the position and one to the orientation. In fact compute the kinematic inversion relative to arm's shoulder and elbow (joint 1,2 and 3) it is independent to the same problem relative to arm's spherical wrist (joint 4,5 and 6). A simple procedure to prove this is the following:

- Given a certain end-effector position p_e, it is possible to compute the centre of the spherical wrist as: $p_W = p_e - d_6 \hat{z}_e$
- Compute (q_1, q_2, q_3) solving the first sub-problem, i.e kinematics inversion for the arm without wrist.
- Compute $R_3^0(q_1, q_2, q_3)$
- Compute $R_6^3(q_4, q_5, q_6) = R_3^{0T} R$, with R rotation matrix extracted from the homogeneous transformation relative to pose p_e.
- Compute (q_4, q_5, q_6) solving the second sub-problem, i.e kinematics inversion for the spherical wrist.

Let's now analyse the two sub-kinematic inversion separately beginning from the shoulder-elbow part.

5.1.1 Shoulder-Elbow angles

The wrist position is computed as follows:

$$\begin{cases} p_{Wx} = \cos(q_1)\left(a_2 \cos(q_2) + d_4 \cos(q_1 + q_3)\right) \\ p_{Wy} = \sin(q_1)\left(a_2 \cos(q_2) + d_4 \cos(q_1 + q_3)\right) \\ p_{Wy} = a_2 \sin(q_2) + a_2 \sin(q_2 + q_3) \end{cases}$$

After squaring and summing above equations it's possible to solve the resulting one for $\cos(q_3)$ whose expression is not reported for sake of briefness. Then imposing $-1 < \cos(q_3) < 1$ we can obtain also

$$\sin(q_3) = \pm\sqrt{1-\cos(q_3)^2}$$

Through $\sin(q_3)$ and $\cos(q_3)$ is possible to find

$$q_{3_{1,2}} = \arctan 2(\pm \sin{(q_3)}, \cos{(q_3)})$$

By means q_3, after squaring ad summing firs two equations of p_W it's possible to compute $\sin(q_2)$ and $\cos(q_2)$, each can assuming two different values. Moving straightforward to the solution it's possible to prove that q_2 can assume four different values $q_{2,1}, q_{2,2}, q_{2,3}, q_{2,4}$ corresponding to the sings chosen for q_3 in both resulting formulas.
Finally q_1 can be computed from first 2 equations of p_W as function of q_2 and q_3

$$q_{1_{1,2}} = \arctan 2(\pm p_{Wy}, \pm p_{Wx})$$

As result the combination of above three angles lead to the specification of four possible configuration for the manipulator, Fig.5 shows two of those namely *Elbow-Up*, $\left(q_{1_{1,2}}, q_{2_3}, q_{3_2}\right)$, and *Elbow-Down* $(q_{1_{1,2}}, q_{2_1}, q_{3_1})$.

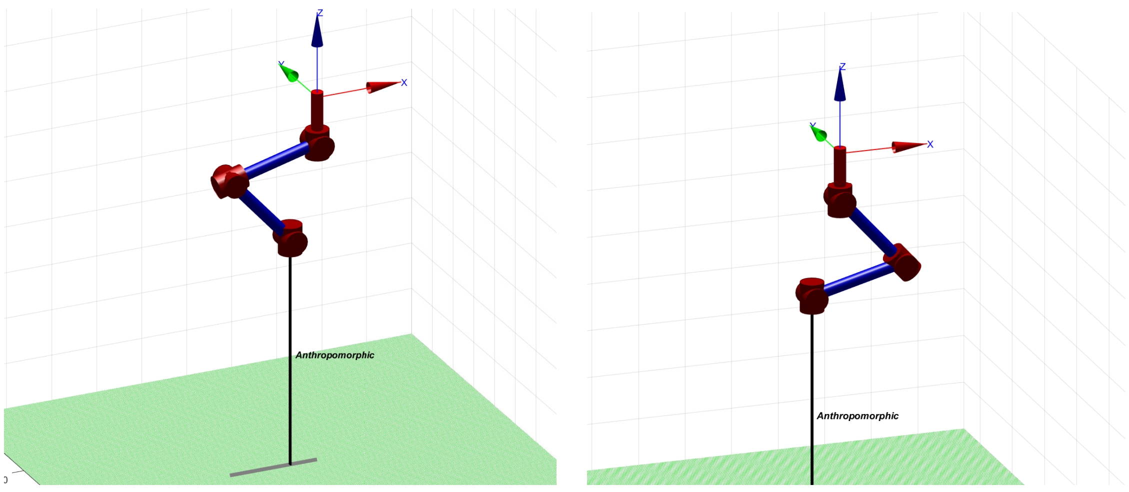


Figure 5: Elbow-Up and Elbow-Down Configurations

The other two possible configuration are same as above with angle q_1 rotated by π so the difference is not clearly visible.

5.1.2 Wrist angles

Spherical-Wrist joint angles can be computed from the intermediate rotation matrix

$$R_6^3 = \begin{pmatrix} n_x^3 & s_x^3 & a_x^3 \\ n_y^3 & s_y^3 & a_y^3 \\ n_z^3 & s_z^3 & a_z^3 \end{pmatrix}$$

Which represents the Euler angles with respect to frame 3. Even in this cases joint angles can assume different values based on the equations solving, in particular

$$q_{4_{1,2}} = \arctan 2(\pm a_y^3, \pm a_x^3)$$

$$q_{5_{1,2}} = \arctan 2(\pm\sqrt{(a_y^3)^2 + (a_x^3)^2}, a_z^3)$$

$$q_{6_{1,2}} = \arctan 2(\pm s_z^3, \mp n_z^3)$$

From above joint angles it is possible to determine two different wrist configurations, one having $q_5 \in (0, \pi)$, i.e $(q_{1_1}, q_{2_1}, q_{3_1})$, and the other having $q_5 \in (-\pi, 0)$, i.e $(q_{1_2}, q_{2_2}, q_{3_2})$.

5.1.3 Implementation

The implementation of the linear kinematics inversion described above starts from the choice on what configuration impose to the robotic arm during the task execution, being the shape to be cut in the horizontal plane the *Elbow-Up* plus *Shoulder-Front* configuration was preferred, while for what concern the wrist no specific advantages are due to the choice of one configuration in place of the other.

The function that performs the kinematic inversion is the following

```
function q = ikineAarmSW(robot, T)
pe = T2Coords(T);
pw = pe(1:3) + [0 0 robot.d(6)];
q(1:3) = ikineAarm(pw, [0 robot.a(2) robot.d(4)], 'sbeu');
q(3) = q(3) + pi/2;
R = t2r(T);
R03 = t2r(SerialLink(robot.links(1:3)).fkine(q(1:3)));
R36 = R03'*R;
q(4:6) = ikineSW(R36, '-pi0');
end
```

where function

```
pe = T2Coords(T);
```

simply extracts position coordinates from an homogeneous matrix and function

```
q = ikineAarm(pw, [0 robot.a(2) robot.d(4)], 'sbeu');
```

exploits the computations explained in 5.1.1 to retrieve first three joint angles taking as input the wrist position, links dimension and manipulator configuration. The second sub-kinematics inversion for a specific wrist configuration is performed through function

```
q = ikineSW(R36, '-pi0');
```

by means of 5.1.2 formulas.

5.2 Differential Kinematics Inversion

Another technique used to solve the inverse kinematics could be used in cases in which the above solution it's difficult to perform, e.g no easy closed-form could be found, or one wants to exploit some manipulator features in order to optimize secondary objective functions. Such technique comes directly from the *Fundamental Differential Kinematics* formula

$$v_e = J(q)\dot{q}$$

which ties end-effector and joint velocities by means the so called Jacobian matrix. Hence joint velocity could be obtained simply inverting the above relation

$$\dot{q} = J^{-1}(q)v_e$$

A first problem is to determine whether the Jacobian matrix is invertible, a necessary condition is that it's a square matrix. When a robot is intrinsically redundant it never happens, since J has at most 6 rows and n columns, with $n > 6$ number of joints. On the other side if the robot is not intrinsically redundant, i.e $n \leqslant 6$, but redundant with respect to an assigned task, i.e $n > r$ with r number of DOF required from that task, the J matrix could be square but one, or more, of its rows is useless. That's exactly the case of the laser-cutting anthropomorphic arm, it is sufficient to note that in order to perform the task the information provided by the 6^{th} coordinate of the end-effector velocity v_e, i.e ω_z , is not needed any more. Hence it is possible to remove the 6^{th} row of J thus making it a 5×6 matrix.

Now J is not square any more, so there exist more than one solution of the differential kinematics formula; a common way to face with this kind of problem is to find solutions $\dot{q}$ that minimize the quadratic cost functional of joint velocities

$$g(\dot{q}) = \frac{1}{2}\dot{q}^T W \dot{q}$$

where W is a weight matrix, positive definite. Hence the solution of the problem

$$\begin{cases} min\,(g(\dot{q})) \\ v_e = J(q)\dot{q} \end{cases}$$

becomes

$$\dot{q} = W^{-1}J^T(JW^{-1}J^T)v_e$$

and in the case in which $W = I$

$$\dot{q} = J^T(JJ^T)v_e$$

and said $J^\dagger = J^T(JJ^T)^{-1}$ *right pseudo-inverse of Jacobian*, the final result is

$$\dot{q} = J^\dagger(q)v_e$$

Which allow to compute joint velocities from end-effector velocities. Once obtained $\dot{q}$ it is possible to perform a numerical integration to retrieve q as follow

$$q(t_{i+1}) = q(t_i) + \dot{q}(t_i)dt$$

with dt time interval between two consecutive points of the trajectory.

5.2.1 Open-Loop Implementation

The above procedure has been implemented in Matlab using also some utilities of the *Robotics Toolbox*. The main function is

```
function [q, qend, man] = iDiffKine(robot, Ttg, dt, q)
```

Which taking as input *robot*, *Ttg* that is the set of homogeneous matrix corresponding to each point of the trajectory, *dt* and an initial joint pose $q0$, returns the set of joint angles q for each point and the final angle q_e. At every step the function estimate the end-effector velocity through a numeric derivation

$$v_{e,i} = \frac{p_i - p_{i-1}}{dt}$$

Follows the whole implementation of the *iDiffKine* function

```
function [q, qend] = iDiffKine(robot, Ttg, dt, q, opt)
   steps = size(Ttg,3);
   qcurr = q;
   Tprev = Ttg(:,:,1);
   for i=1:steps
       Tcurr = Ttg(:,:,i);
       prev = T2Coords(Tprev);
       new = T2Coords(Tcurr);
       dVe = (new - prev)/dt;
       dVe = dVe(1:5);
       J = robot.jacob0(qcurr);
       J = J(1:5,:);
       Jpi = J'*((J*J')^-1);
       dqcurr = (Jpi*dVe')';
       qcurr = qcurr + dqcurr*dt;
       q(:,i) = qcurr;
       Tprev = Tcurr;
   end
   qend = qcurr;
end
```

The function *jacob0(q)* is provided by the *Robotics Toolbox* and given a joint angles q returns the Jacobian matrix with respect to frame 0.

5.2.2 Improvements

The implementation provided in 5.2.1 provide solutions according to an open-loop scheme, at every step a new q is computed, but of course, due to the numerical integration, it could drift from the desired one. A possible solution to mitigate such phenomenon is taking into account the error made at every step and then try to minimize it, there is a built-in function in *Robotics Toolbox* that exactly performs what said

```
    q = robot.ikcon(T);
```

Taking as input the homogeneous transformation relative to the end-effector pose.

5.3 Null-Space optimization

As far the arm redundancy with respect to the assigned task had the only effect to change the way to compute the inverse matrix that multiplies end-effector velocity in order to get joint velocities, the kinematic redundancy explication make sense when becomes possible exploit it to optimize secondary objective functions. Being $n - r$ the dimension of Jacobian *null space*, if $\tilde{\dot{q}}$ solves $v_e = J(q)\dot{q}$ then also

$$\dot{q} = \tilde{\dot{q}} + P\dot{q_0}$$

solves it, being P a matrix which project an arbitrary joint velocity vector $\dot{q_0}$ in the null space ($P\dot{q_0} = 0$)

$$J\dot{q} = J\tilde{\dot{q}} + JP\dot{q_0} = v_e$$

Now problem of 5.2 can be rewritten, using the additional vector q_0, as

$$\begin{cases} \min \frac{1}{2}(\dot{q} - \tilde{\dot{q_0}})^T(\dot{q} - \tilde{\dot{q_0}}) \\ v_e = J(q)\dot{q} \end{cases}$$

Which means to find those solutions subjected to the constraint $v_e = J(q)\dot{q}$ which minimizes the norm of vector $(\dot{q} - \tilde{\dot{q_0}})$ and hence are as close as possible to vector $\tilde{\dot{q_0}}$. In this way the additional objective functions can be specified by means $\tilde{\dot{q_0}}$. Solving the above problem leads to

$$\dot{q} = J^\dagger v_e + (I - J^\dagger J)\tilde{\dot{q_0}}$$

where $(I - J^\dagger J)$ is a possible P matrix. Now let

$$\dot{q_0} = k_0 \left(\frac{\partial w(q)}{\partial q} \right)$$

$w(q)$ corresponds the secondary objective function, that in this work has been set to a typical measure namely *manipulability.* Such measure is usually adopted as distance of manipulator from singular configurations.

$$w(q) = \sqrt{det(J(q)J(q)^T)}$$

5.3.1 Implementation

First problem to face with is that by means the *Robotic Toolbox* functions is possible to obtain numeric evaluation of the manipulator quantities, but not their analytical expressions, thus creating the need to compute the gradient of $w(q)$ numerically. Let's try to explain such concept by considering the *manipulability* expression $\sqrt{det(J(q)J(q)^T)}$, in order to compute its gradient it is necessary to express it in an analytical form, but as long J is calculated through the *Robotics Toolbox* function *robot.jacob0(q)* only a numerical evaluation is provided. A possible way is to derive the analytical expression of J and then obtain the corresponding analytical expression of $w(q)$ thus allowing also the computation of its gradient expression, this is in practice very difficult since even for simple manipulator the analytical expression can be very complicated. Another possibility is to directly compute it with a numerical procedure. Referring to this second case the way adopted in this work is to use the *Matlab Optimzation Toolbox.* The problem is to find a value $\dot{q}_0^*$ which maximizes the function $w(q)$ could be formulated as an unconstrained optimization problem to be solved by means *fminunc* function

```
optFunction = @(x) manipulabilityOF(x, robot, k0, dt);
opt = optimoptions('fminunc','Algorithm','quasi-newton','Display','off');
[dq0] = fminunc(optFunction, dq0, opt);
```

The objective functions is defined as follow

```
function m = manipulabilityOF(dq0, robot, k0, dt)
J = robot.jacob0(q);
J = J(1:5,:);
nj = size(robot.qlim, 1);
I = eye(nj);
J_pinv = J'*((J*J')^-1);
dq0 = J_pinv*dVe' + (I-J_pinv*J)*dq0'*k0;
q = q + dq0'*dt;
J = robot.jacob0(q);
J = J(1:5,:);
m = -sqrt(det(J*J'));
end
```

Once the value of $\dot{q_0}$ has been determined the new joint angle can be simply computed as

```
[dq0] = fminunc(optFunction, dq0, opt);
dqcurr = dqcurr + (I-Jpi*J)*dq0'*k0;
q = qcurr + dqcurr'*dt;
```

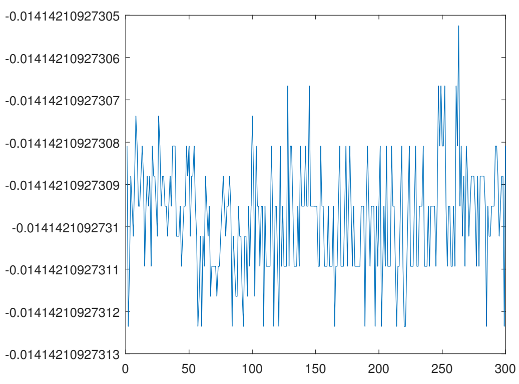
It's important to notice how such procedure slows down the performance of the algorithm due to computational heaviness, a deeper discussion on this will be make in 6. Another things to take into account is the value of constant k_0, which represents the optimization contribute to the displacement, it should be tuned in base on the iterations number n, in particular for high values of it k_0 should be smaller to spread its contribute among more iterations and vice-versa.

6 Results and Performance Evaluation

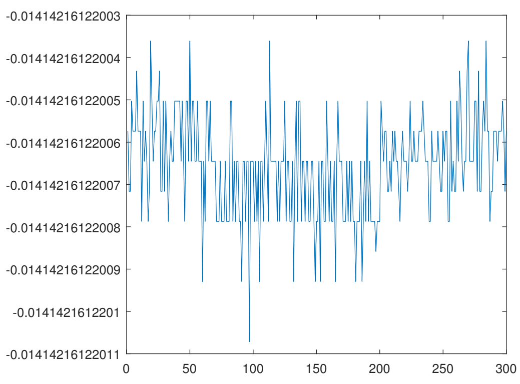
Let's now analyse the results achieved from every kinematic inversion, then a comparison among these will be performed. The computation time of every kinematics inversion has been recorded with the Matlab *tic-toc* system. The computed trajectory has 10 points per segment for falling and rising phases, since distance travelled is lower, and 70 point for each segment of the polygon, so total dimension is 300 points.

6.1 Linear Inverse Kinematics Results

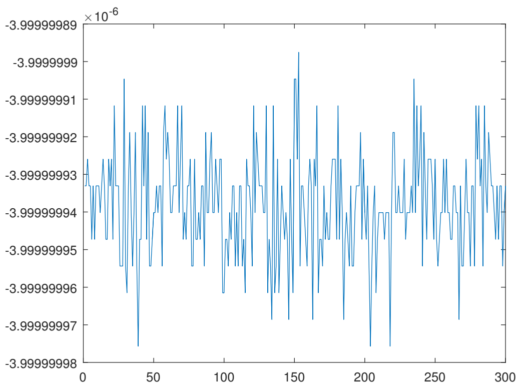
As one could expect errors are very low since all values are computed directly and only numerical approximations could create some drift from desired values. The computation time is 0.79 seconds, the magnitude order of such value would be clearer once compared with others.



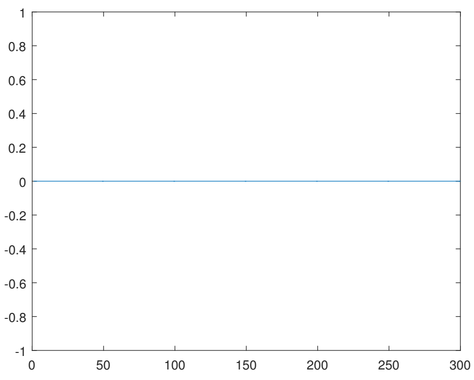
(a) x-axes error



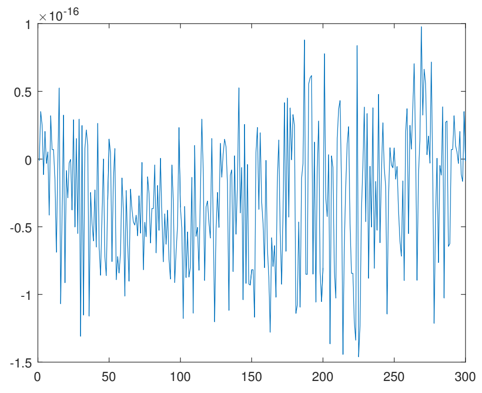
(b) y-axes error



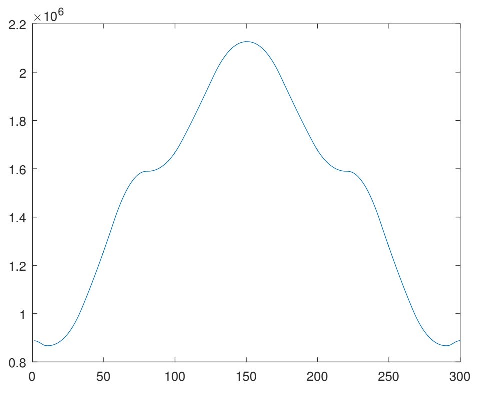
(c) z-axes error



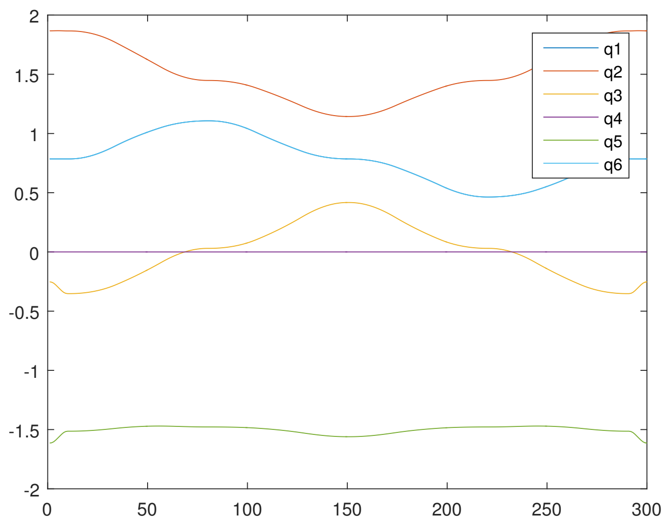
(d) Roll-Angle error



(a) Pitch-Angle error



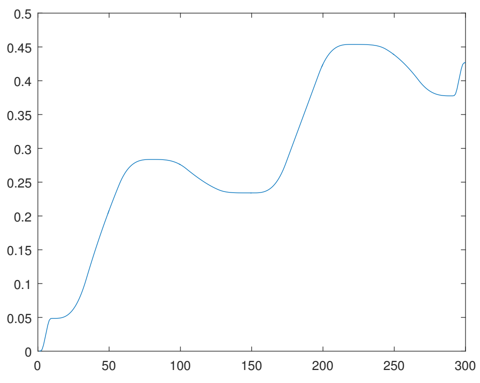
(b) Manipulability



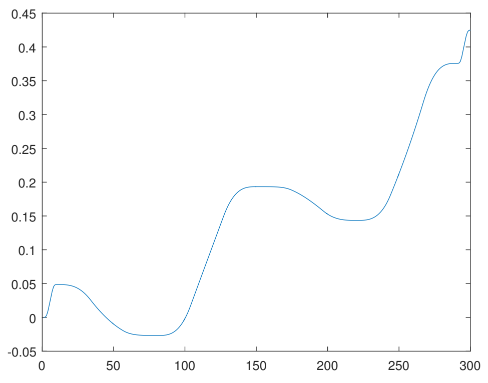
(c) Joint Trajectory

6.2 Differential Kinematics Inversion

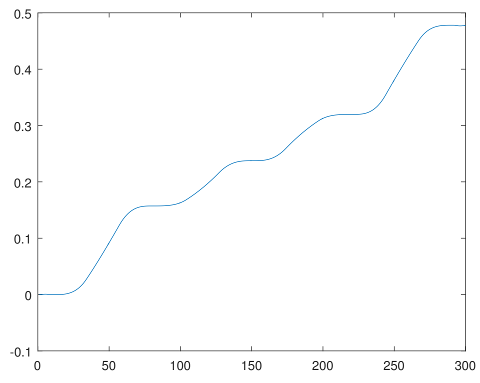
At this stage the pose-error significantly increases since the kinematics inversion is performed in open-loop and numerical integration error is involved. It is worth to notice that the orientation error, even if it increases about 12 order magnitude with respect to the previous, remain reasonably low, indeed such error could be only visible in manipulators with resolution less or equal to 10^{-4}, i.e tenths of millimetres. Computation time is 0.5242 seconds, quite similar to linear inversion.



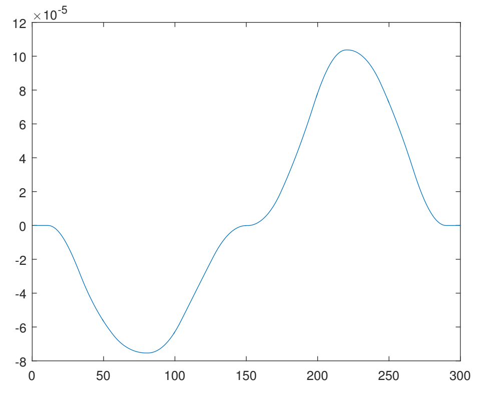
(a) x-axes error



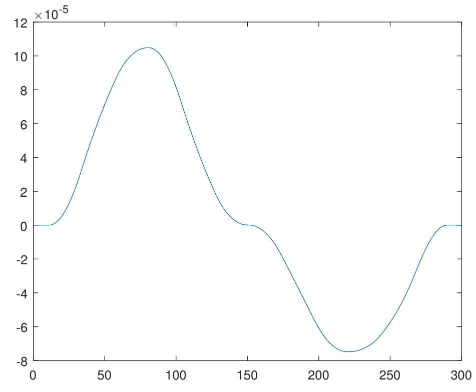
(b) y-axes error



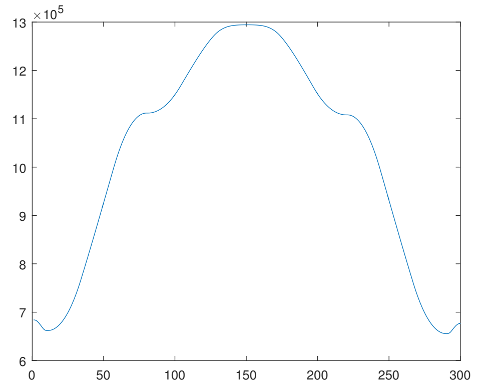
(c) z-axes error



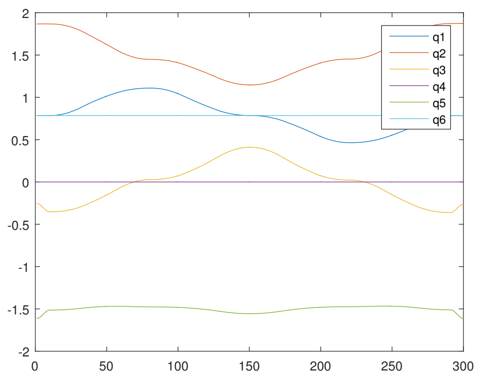
(d) Roll-Angle error



(e) Pitch-Angle error

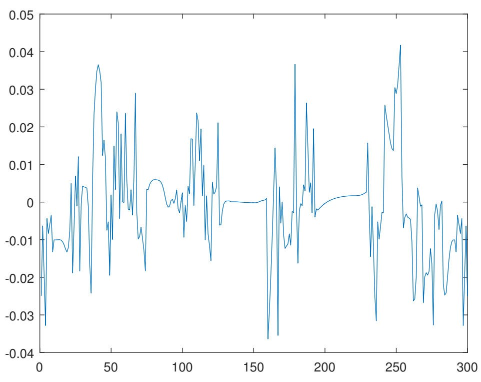


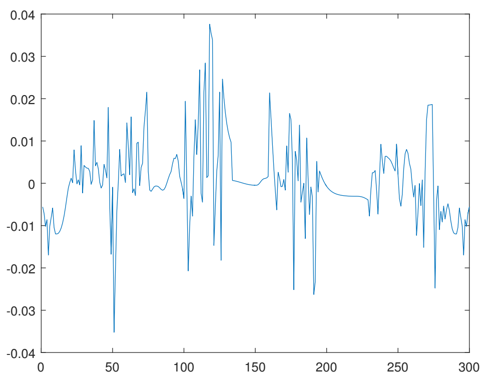
(f) Manipulability

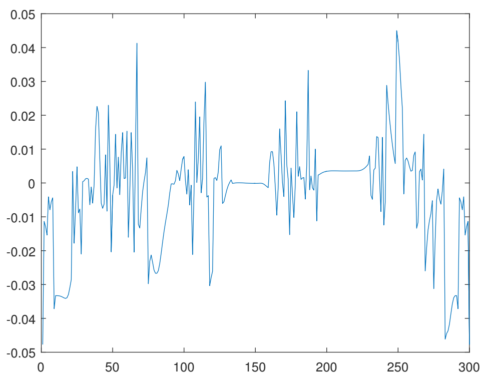


(g) Joint Trajectory

The pose error could be further reduced if the kinematics inversion is performed through the *ikcon* function as described in 5.2.2. In this case there are significant improvement only in the pose error while the other quantities remains almost the same and hence are not reported.


(a) x-axes error


(b) y-axes error


(c) z-axes error

It is possible to see that error it's ten times less then before, such improvements has obviously a cost in terms of performance degradation due to the fact that the algorithm solves an optimization problem at each trajectory step, indeed now the whole procedure requires 8.62 seconds, i.e more than ten times the previous computation time. In this case a trade-off is necessary in order to achieve good precision in a reasonable computation time.

6.3 Optimized Inverse Differential Kinematics

The null-space optimization attempt to improve the manipulability measure. Fig6 shows how in average there is an improvement of 10^{-9} on the achieved manipulability, but in order to get this minimum results the cost paid is multiply by 100 times the computation time without it, hence in this case the optimization does not provide benefits that sufficiently counter balances the performance degradation.

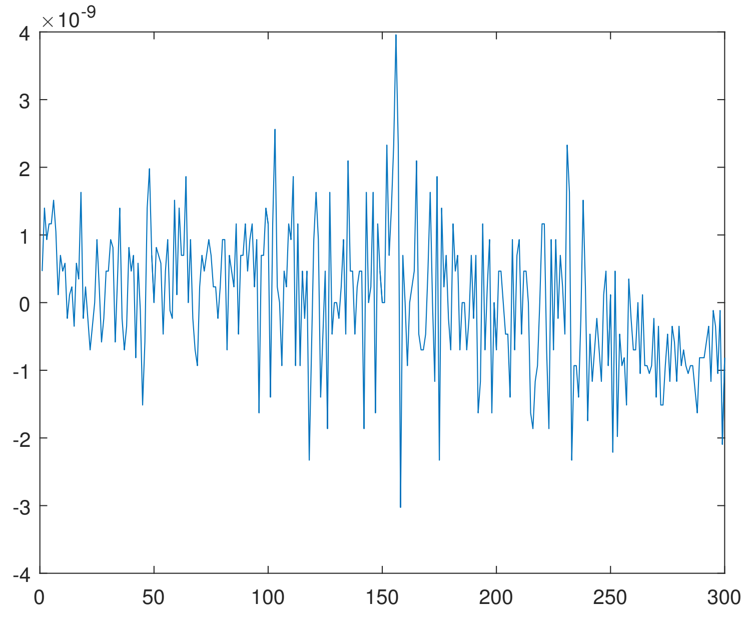


Figure 6: Manipulability improvement