

POLITECNICO DI TORINO

MASTER'S DEGREE IN MECHATRONIC ENGINEERING

MASTER'S THESIS

An Off-line Optimized Planner for the Generation of Path and Orientation of Industrial Robots



Supervisor:

Prof.ssa Marina Indri

Candidate:

Gabriele Baldi

Tutor:

Ing. Aldo Bottero
COMAU S.p.A. - RoboLAB

December 2019

Table of contents

Introduction	III
1 Parametric curves	4
1.1 Introduction	4
1.2 Bézier curves	5
1.2.1 De Casteljau's algorithm	6
1.3 Spline curves	7
1.3.1 Continuity of a curve	7
1.3.2 Knot vectors	8
1.4 B-spline curves	9
1.4.1 B-spline basis functions	9
1.5 NURBS	11
1.5.1 Weights modification	11
2 Trajectory planning	13
2.1 Introduction	13
2.2 Path vs. Trajectory	14
2.3 Trajectory in the Joint space	14
2.3.1 Point-to-Point Trajectories	16
2.3.2 Multipoint Trajectories	22
2.4 Trajectory in the task space	24
2.4.1 Multidimensional Trajectories	27
2.5 Orientation of the tool	29
2.5.1 Axis-Angle	31
2.5.2 Planar sliding	32
2.5.3 Euler angles	34
2.5.4 Quaternions	37
3 Off-line trajectory planning for continuous processes	44
3.1 Introduction	44
3.1.1 Case of study	45
3.2 Trajectory planning based on FIR filters	45
3.2.1 Uniform B-splines	46

3.2.2	Interpolation problem solver and computation of the control points	49
3.2.3	Implementation of the trajectory planner	51
3.3	Pre-elaboration of data and Optimization	55
3.3.1	Introduction	55
3.3.2	Nurbs toolbox	55
3.3.3	Elaboration of the geometric path with Nurbs	56
3.3.4	Redistribution and oversampling of the input points over the path	58
3.3.5	Elaboration of the set of orientations with Nurbs	61
3.4	Matlab simulations	63
3.4.1	Test 1	64
3.4.2	Test2	67
4	Tests on Robot	72
4.1	Description of the NJ130	72
4.2	The teach pendant and the moni	73
4.3	Experimental tests	74
5	Conclusions	82
5.1	Future works	82
A	Matlab codes	83
A.1	angles_calc.m	83
A.2	arc_length.m	84
A.3	samp_calc.m	84
A.4	interpolation.m	85
A.5	sequencer.m	86
A.6	FIR.m	86
A.7	elab_data.m	87
A.8	curve_rad.m	88
A.9	curve_weights.m	89
A.10	weights_calc.m	90
A.11	weights_orientation.m	91
A.12	test.m	92
	Bibliography	95

Introduction

From the introduction of the first robot in the industry, in the late '60s, many things have changed. Nowadays, industrial robotics' market is growing rapidly. According to the International Federation of Robotics(IFR),this kind of market showed in 2018 an annual global sales value of 16.5 billion USD in 2018. 422.000 units where shipped globally in those year, with a 6% increment with respect to the previous year. Industrial robots become every year lighter, tough, faster and easier to reprogram and, due to their major efficiency and versatility are spreading in many applications of the industrial domain.

Continuous processes

Most of the current robotics' applications are mainly continuous processes, in which robots are fundamental. There are applications, for example welding or spray painting, which in some kind of industry, like automotive, are made exclusively by them. A brief description of these operations is made in the following [1]:

- **Welding:** involves the use of a robotic manipulator to fully automatize the process of welding an object, handling both the weld itself and the handling of the tool.
- **Spray painting:** requires covering a surface with an even coat of paint. This is typically done by pre-specifying the trajectory along which the arm should move for both position and orientation.
- **Machining** of mechanical parts is a growing field in industrial robotics. Operations like grinding, milling, polishing etc. . . are challenging techniques which require both the ability to follow the surface to work on and to maintain the force required to perform the operation.

All these operations have the common ground to be tasks in which it is necessary to look for a good trade-off between the robot capacity to show a high accuracy in following the surface and its speed, to reduce the cycle time of the processing.

Objective of the thesis

The algorithm proposed in this thesis gives to the user the possibility to plan a trajectory in the task space in which it can handle the trade-off mentioned above,

in order to obtain a result as close as possible to its expectations. To reach this objective I started from the work in [2], in which a combined use of B-Spline and a cascade of FIR(Finite Impulse Response) filters is made in order to obtain a smooth curve which interpolates the given via-points, and I tried to optimize it, through a combined use of NURBS curves and a pre-elaboration on the points. The study and the relative simulations and tests are related to an example of sealing of a hood. In Figure 1 an operation of sealing made by an industrial robot is shown.

COMAU, RoboLAB and the NJ130

COMAU(Consorzio macchine utensili) is an Italian multinational company which has his headquarter in Grugliasco, Turin. Founded in 1973, it is an integrated company that works mainly in the automotive field. The company has a well established cooperation with the Politecnico di Torino. As a matter of fact, in November 2013, from this synergy RoboLAB was born, a joint research laboratory between these two realities. The work of thesis was carried out mainly in this laboratory, where I also had the possibility to test my algorithm on a real robot. The robot used for the tests is an NJ130 - 2.6 (Figure 1), an anthropomorphic industrial robot, with six revolute joints and a spherical wrist, capable to do the continuous processes previously described with a very good accuracy and speed of execution.

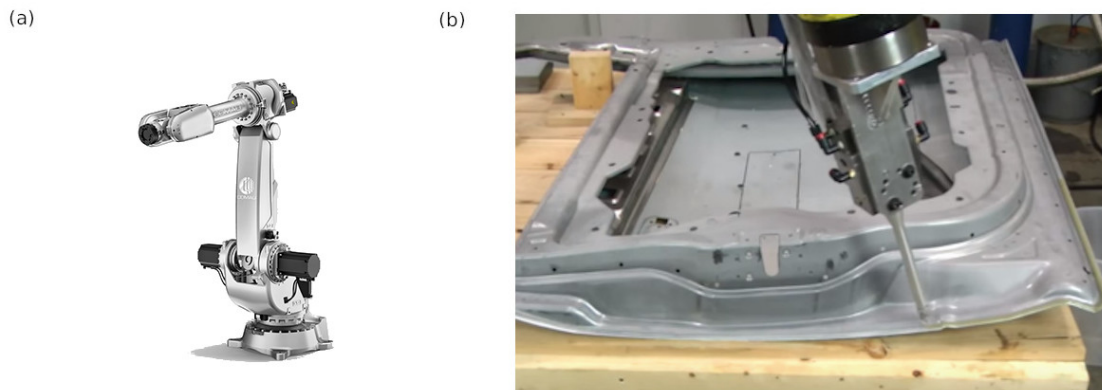


Figure 1. In (a) a model of the NJ130 - 2.6 [3]. In (b) an industrial robot during the operation of sealing of a car door. [4]

Thesis Outline

The thesis is structured in the following chapters:

1. **Chapter 1 - Parametric Curves:** a state of the art on parametric curves is given, focusing on the most important issues for this thesis.
2. **Chapter 2 - Trajectory Planning:** state of the art on planning of the motion law for path and orientation.
3. **Chapter 3 - Off-line trajectory planning for continuous processes:** In this chapter the main contribution of the thesis is provided. It is divided into three parts. In the first part the planner is discussed and implemented, then an optimization process is carried out, and finally the overall structure is tested in Matlab and the results are analysed.
4. **Chapter 4 - Tests on robot:** Discussion and analysis of the results obtained by testing the algorithm on a real robot.
5. **Chapter 5 - Conclusion and future works:** The results of the thesis are examined and future developments for this kind of work are proposed.

Chapter 1

Parametric curves

In this chapter, an overview of different parametric curves will be given. A brief definition of parametric curve will be given in the introduction, then different kind of curves will be examined from a pure theoretical point of view, in order to clarify the choices that will be made in the thesis.

1.1 Introduction

In robot trajectory planning a parametric representation of the curve is often adopted, to represent a free curve in the space. A parametric curve $\mathbf{P}(u)$ is a curve that returns a point \mathbf{P} in the space for a particular value of the parameter $u \in [u_{min}, u_{max}]$. An example of parametric curves is given in Figure 1.1, meanwhile, in the next section some important kinds of parametric curves are described.

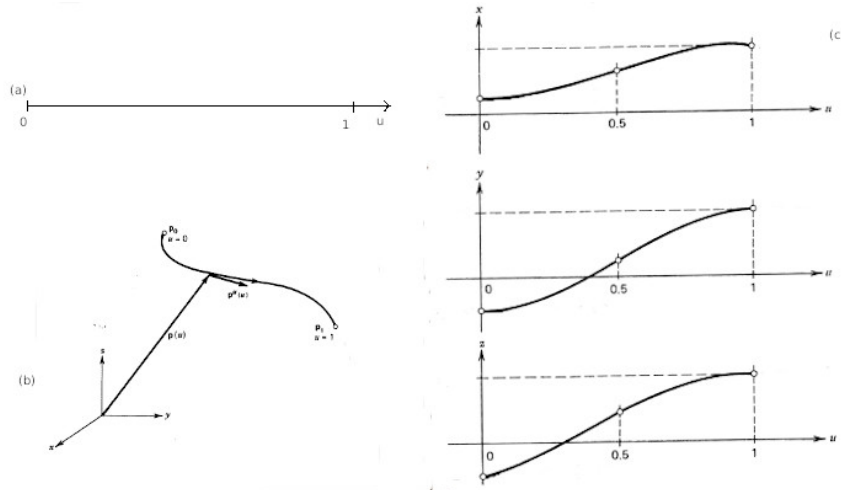


Figure 1.1. Representation of a parametric curve. (a) Parametric vector, (b) Parametric curve in 3D space, (c) Parametric curve decomposed into the three axis over the parametric space [5]

1.2 Bézier curves

Bézier curves are a simple kind of curves, used in computer graphics to draw shapes. A n -th degree Bézier curve is defined by [6]:

$$\mathbf{C}(u) = \sum_{i=0}^n B_{i,n}(u) \mathbf{P}_i \quad 0 \leq u \leq 1 \quad (1.1)$$

The basis functions, $\{B_{i,n}(u)\}$, are called *Bernstein polynomials* (Figure 1.2) of degree n , and are expressed by:

$$B_{i,n}(u) = \frac{n!}{i!(n-i)!} u^i (1-u)^{n-i} \quad (1.2)$$

The coefficients \mathbf{P}_i are called *control points* and will form the control polygon which defines the shape for the curve (Figure 1.3).

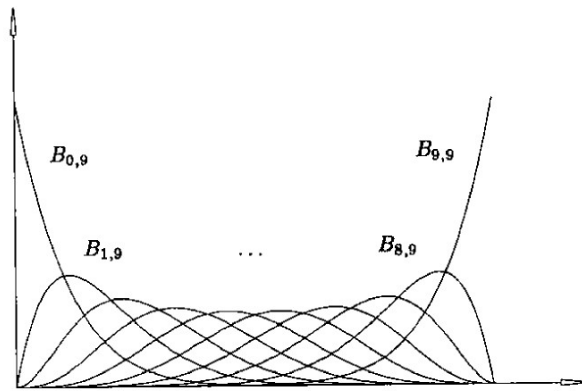


Figure 1.2. Bernstein polynomials [6]

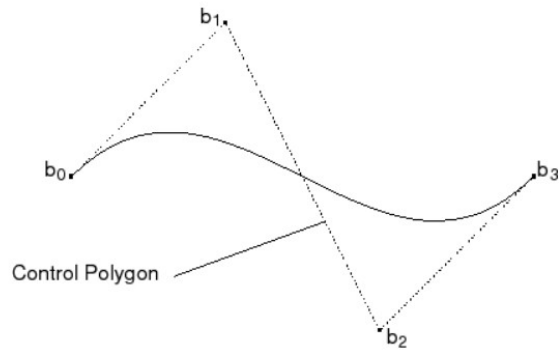


Figure 1.3. Bezier curve and control Polygon [6]

1.2.1 De Casteljau's algorithm

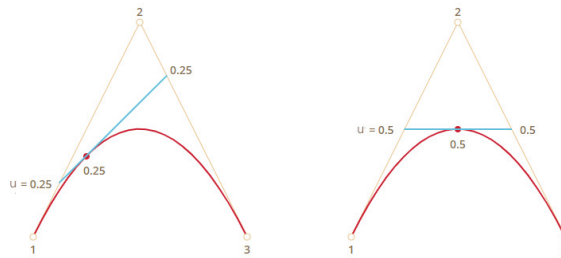


Figure 1.4. Graphical representation of De Casteljau's algorithm [7]

Besides the maths notions, a basic understand on how the control points define the shape of the curve is given by the following algorithm. Given a parameter curve with 3 control points and parameter u , and referring to Figure 1.4, from [7]:

1. Draw the control points (1,2, 3 in the example)
2. Connect the control points through segments (brown segments in figure)
3. The parameter u moves from 0 to 1 with a certain step. For each step:
 - take a point on each segment connecting the control points, located at a distance from the brown segments proportional to the distance of u from 0
 - connect the points found on the segments, obtaining a new segment, i.e., the one drawn in blue

4. Now, on this segment take a further point(in red) again at a distance proportional to the displacement of u from 0
5. Connecting all these points for u that goes from 0 to 1 the curve is obtained

This is a recursive algorithm, which means that can be done with some more iteration in case of curves with more than 3 control points.

From the algorithm it is also possible to imagine that Bézier curves are really simple to use for few control points, but their difficulty grows as the number of control points grows. Furthermore, only the first and the last control points are reached by the curve, as shown in Figure 1.3 the intermediate points are only *approximated* from it, this could be an unwanted behaviour in some cases, in which an exact *interpolation* could be desired.

1.3 Spline curves

To overcome the problem of exact interpolation, spline curves are introduced. Spline curves are connected functions, ensuring certain condition of continuity in the connection. Of particular interest in the field of robotics are the cubic splines, which guarantee a C^2 continuity in the junctions. [5] This particular kind of splines is examined more in details in Chapter 3, in the field of trajectory planning. In this section only some details about spline curves and other tools related to them will be given.

1.3.1 Continuity of a curve

We said that a cubic spline has C^2 continuity. From [8], the *continuity* of a curve is, practically, the description of how smooth is the considered curve. If a break or a speed change appears along a curve, this is called *discontinuity*, that is, in maths terms, a jump in the n -th derivative of the C^n curve. There are several kinds of continuity, as shown in Figure 1.5. The most useful for our analysis are:

- C^0 - change in position, the curve has literally a break in it
- C^1 - change in tangent, the curve has a critical change in direction(e.g., 90° angle)
- C^2 - change in acceleration of the curve

It is important to differentiate this kind of continuity (called *geometric continuity*) to the *parametric continuity*, which refers to discontinuity of the derivative in the parametric space.

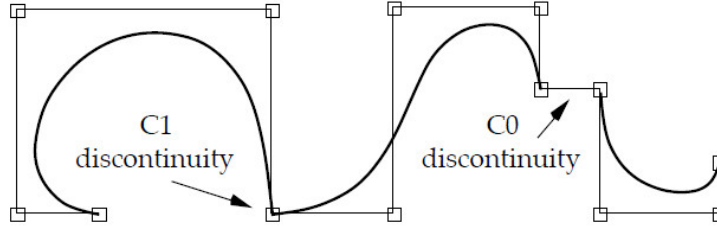


Figure 1.5. Curve discontinuities [8]

1.3.2 Knot vectors

Being a spline curve a union of several polynomials, differently from the Bézier curves, it cannot be defined on a single vector $u \in [0, 1]$ but on a union of vectors $u \in [U_0, U_1, \dots, U_n]$, which define the junctions of the various polynomials over the curve. This is called *knot vectors* (Figure 1.6) [5].

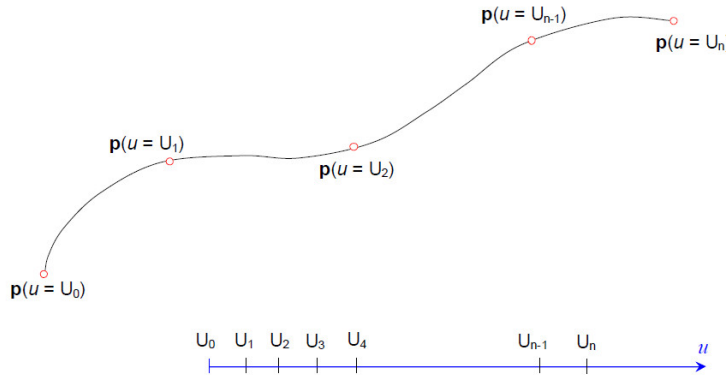


Figure 1.6. Knot vector [5]

The ideal way to choose the knots is to assign between each couple of them the arc length of the corresponding curve, but it is impossible to compute the arc length of the curve before assigning it the knots vector. So, the most simple and utilized way to define a knots vector is to impose a uniform distance among them. Although very simple, this method has the drawback to produce unwanted oscillations in some cases

if distance among the knots does not correspond to the distance among the points on the curve. Then, a more practical way to assign them is to choose an intermediate distance equal to the length of the segment connecting the correspondent arc on the curve.

1.4 B-spline curves

Even though with spline curves the problem of the exact interpolation is solved, there is another problem in common between Bézier curves and splines, which is the lack of local controllability. In fact, in both kind of curves, if we move one control point the shape of the overall curve changes. So, the question is:

Does it exists a kind of curve which has the advantages of both the previous curves, i.e., simplicity in control, like Bézier, and exact interpolation of the given points, like splines, and, moreover, supports local controllability?

B-splines are the answer to this question. A p -th degree B-spline curve is defined as: [6]:

$$\mathbf{C}(u) = \sum_{i=0}^n N_{i,p}(u) \mathbf{P}_i \quad u_0 \leq u \leq u_n \quad (1.3)$$

where \mathbf{P}_i are the control points, and $N_{i,p}(u)$ are the p -th degree B-spline basis functions defined on the knot vector:

$$U = \left\{ \underbrace{u_0, \dots, u_0}_{p+1}, u_{p+1}, \dots, u_{m-p}, \underbrace{u_n, \dots, u_n}_{p+1} \right\} \quad (1.4)$$

where, in general, $u_0 = 0$ and $u_n = 1$. The number of knots m is equal to number of the control points plus the order of the curve, so:

$$m = n + p + 1 \quad (1.5)$$

1.4.1 B-spline basis functions

Considering the knot vector in (1.4), the i -th B-spline basis function of p -degree is defined by: [6]

$$N_{i,0}(u) = \begin{cases} 1 & u_i \leq u \leq u_{i+1} \\ 0 & \text{otherwise} \end{cases} \quad (1.6)$$

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u)$$

As we can see from (1.6), B-splines have a basis function which only depends on a restricted span of the knot vector. In this way the desired local controllability has been reached. Every B-spline of order p is composed by polynomial pieces of degree $p - 1$ which are connected with continuity C^{p-2} . So, every control point has an influence only on p segments of the curve and, at the reverse, each segment of the B-Spline is influenced by only p control points. Figure 1.7 shows an example of the basis functions over a certain knot vector.

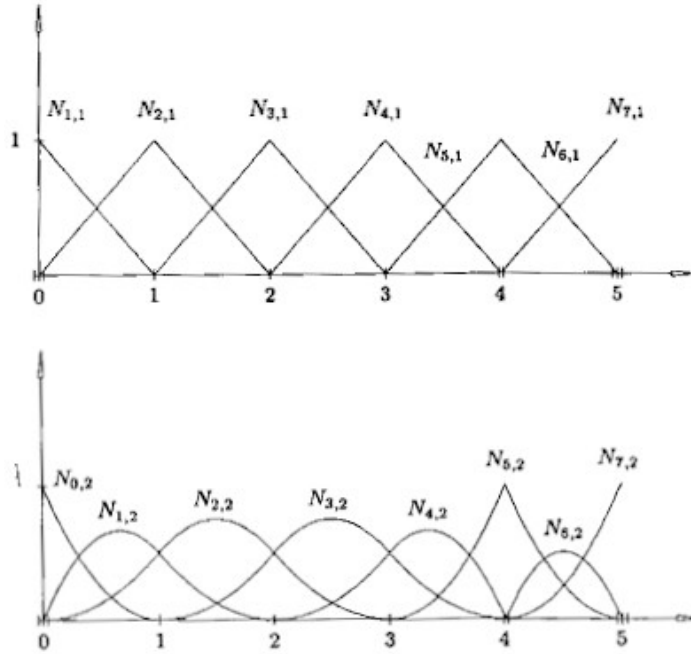


Figure 1.7. Basis functions defined over the knots vector $U = \{0, 0, 0, 1, 2, 3, 4, 5, 5, 5\}$ [6]

1.5 NURBS

An important improvement to the properties of local controllability of the B-spline is given by Non-Uniform Rational B-Splines(NURBS). A p -th degree NURBS is defined by [6]:

$$\mathbf{C}(u) = \frac{\sum_{i=0}^n N_{i,p}(u)w_i\mathbf{P}_i}{\sum_{i=0}^n N_{i,p}(u)w_i}, \quad u_0 \leq u \leq u_n \quad (1.7)$$

where \mathbf{P}_i are the control points, $N_{i,p}(u)$ are the same p -th degree basis functions of the B-splines and w_i are the weights. Setting:

$$\mathbf{R}_{i,p}(u) = \frac{\sum_{i=0}^n N_{i,p}(u)w_i}{\sum_{i=0}^n N_{i,p}(u)w_i} \quad (1.8)$$

it permits us to rewrite (1.7) as:

$$\mathbf{C}(u) = \sum_{i=0}^n \mathbf{R}_{i,p}(u)\mathbf{P}_i \quad (1.9)$$

where $\mathbf{R}_{i,p}(u)$ are called *rational basis functions* and have the same properties of the classical basis functions, but differ from them for the presence of the weights w_i .

1.5.1 Weights modification

The presence of the weights in the NURBS basis functions allows another degree of freedom for what concerns the modification of the shape of the curve. In fact, in the B-spline there are mainly two ways to reshape the curve: [5]

- Modification of the knot vector
- Modification of the control points

but, de facto, due to the difficulty in modifying the knots vector for what we said in Section 2.3.2, the only possibility was to modify the control points. This new object gives instead a new parameter through which the curve can be reshaped. Increasing a specific weight, the portion of the curve associated to it is pushed against the relative control point, meanwhile, decreasing its value, the same portion of curve is pulled away from it. All of this is possible while keeping the local controllability property of the B-splines, since the two objects share the same structure. Figure 1.8 shows a curve modification for different weights while in Figure 1.9 the rational basis functions for the different weights are reported.

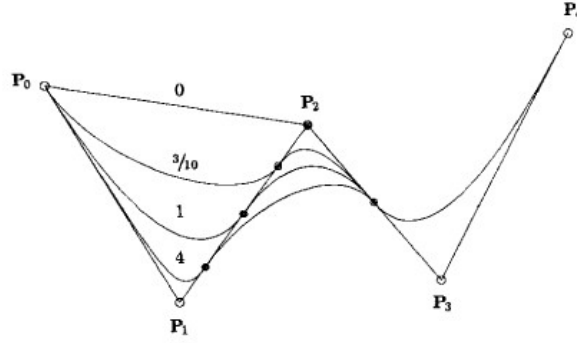


Figure 1.8. Curve modification with $w_1 = \frac{3}{10}, 1, 4$ [6]

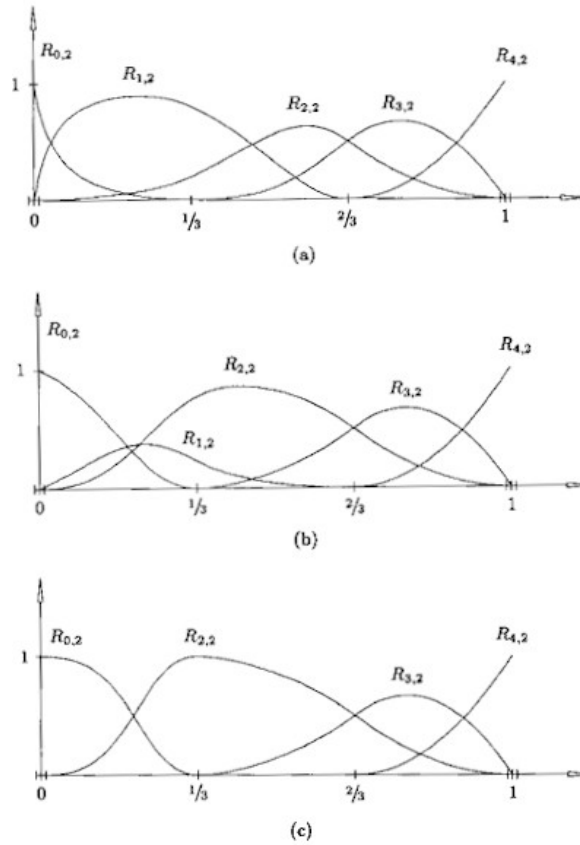


Figure 1.9. Rational basis function of the curve in Figure (1.8). (a) $w_1 = 4$, (b) $w_1 = \frac{3}{10}$, (c) $w_1 = 0$ [6]

Chapter 2

Trajectory planning

This chapter deals with the problem of planning motion laws and trajectories. In the introduction the definition of path and trajectory is given, together with a brief general overview of the problem to plan a trajectory in the task or in the joint space. Then, several kinds of trajectories are illustrated. Finally, an overview on some techniques used to represent orientations is given, with particular attention on quaternions, which are utilized in the thesis.

2.1 Introduction

Objective of trajectory planning is to generate the input for the robot control system in a way to ensure the execution of the planned trajectory by the manipulator. As a matter of fact, the trajectory cannot be totally planned from the user due to its complexity. For this reason the user usually gives a limited number of parameters in the task space, like the extreme points or also some intermediate points, the time of execution of the motion law, the maximum velocity of the robot etc. . . and, then, the task of the engineer is to create a geometric path and a time law $s(t)$ which respect those specifications. Due to the fact that the control action on the manipulator is made at the joint level (see Figure 2.1), the set of variables in the task space $(p_x(t), p_y(t), p_z(t), \phi(t), \theta(t), \psi(t))$ need to be translated in variables in the joint space $(q_1(t), q_2(t), q_3(t), q_4(t), q_5(t), q_6(t))$ through some inverse kinematics algorithm [9].

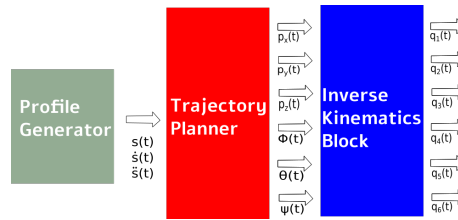


Figure 2.1. Cascade sequence of trajectory planner and Inverse Kinematics block [10]

2.2 Path vs. Trajectory

To avoid confusion it is useful, as a starting point, to clarify the difference between two terms that are used very often in this field and at a first sight may appear as synonyms: 'Path' and 'Trajectory' [10], graphically represented in Figure 2.2.

PATH: is the geometrical description of the desired set of points in the task space.

TRAJECTORY: is a path to which is assigned the time law required to follow that path.

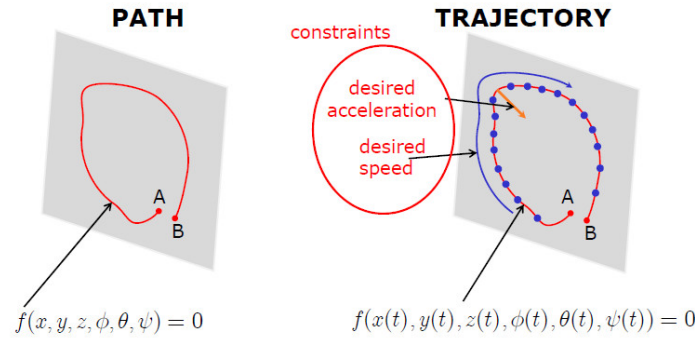


Figure 2.2. Path and Trajectory [10]

The planning of the trajectory can be set either in the joint space or in the task space, depending on what are the specifications given by the user and which objective the manipulator has to accomplish.

2.3 Trajectory in the Joint space

Even though the trajectory is usually specified in the task space, sometimes it could be useful and simpler to plan it directly in the joint space. The first step is to take the user variables and translate it to the equivalent variables in the joint space. To do this, an inverse kinematics algorithm can be employed if the programming is made offline, otherwise the acquisition of these variables can be done on-line if the programming of the robot is made by teaching techniques. [9] Then different techniques are employed to generate the path and the related time law, some of

them are discussed in the next chapter. In general, a joint space trajectory planning is required to have the following features:

- The trajectory can't be too demanding from the computational point of view
- The joints positions and velocities have to be continuous function of the time
- Every undesirable effect should be minimized

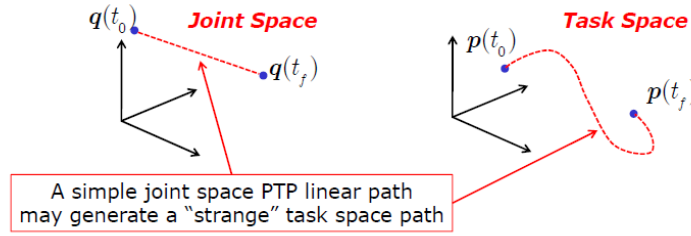


Figure 2.3. Joint space-to-Task space [10]

At first sight one can think to simply connect the desired points in the joint space through linear paths, but due to direct kinematics needed to convert the joint trajectory in the task trajectory this is not possible, because this algorithm is not linear. So a linear path in the joint space would be converted in a non-linear one in the task space, as shown in Figures 2.2 - 2.3. Other kinds of solution need to be implemented, and some of them are explained in the following chapters.

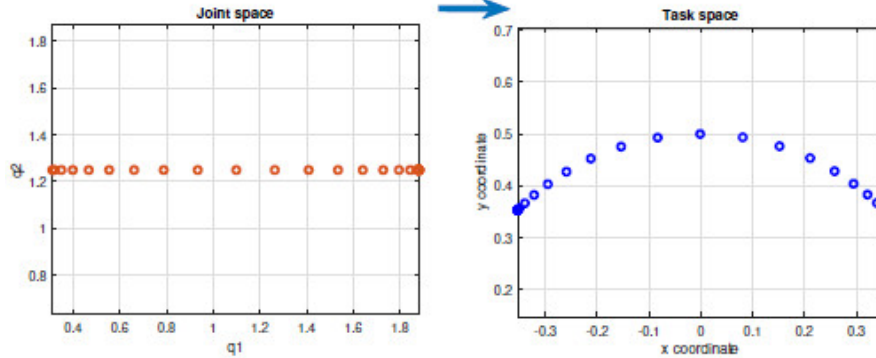


Figure 2.4. Joint space-to-Task space [11]

2.3.1 Point-to-Point Trajectories

Introduction

In the simplest case a motion law can be obtained assigning the initial and final time instants, respectively t_0 and t_1 , and some conditions on position, velocity and acceleration at these times. Then, the problem is to find a function:

$$q = q(t), \quad t \in [t_0, t_1] \quad (2.1)$$

such that those conditions are satisfied. It can be easily seen that a solution to this problem is to consider a polynomial function, i.e., a function like:

$$q(t) = a_0 + a_1 t + a_2 t^2 + \dots + a_n t^n \quad (2.2)$$

where the degree n of the polynomial depends on how many conditions there are to be satisfied and how much the motion should be smooth [12].

Linear trajectory

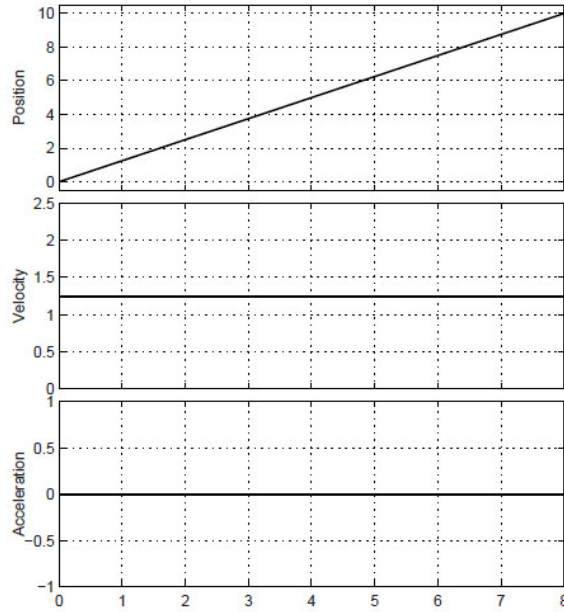


Figure 2.5. Linear Trajectory [12]

The most simple trajectory to determine a motion from a point q_0 to a point q_1 is a linear trajectory, in Figure (2.5) defined as [12] :

$$q(t) = a_0 + a_1(t - t_0) \quad (2.3)$$

Once initial and final time instants and positions are specified, we can determine a_0 and a_1 with the following system of equations:

$$\begin{cases} q_0 = a_0 \\ q_1 = a_0 + a_0(t_1 - t_0) \end{cases} \implies \begin{cases} a_0 = q_0 \\ a_1 = \frac{q_1 - q_0}{t_1 - t_0} = \frac{h}{T} \end{cases} \quad (2.4)$$

with $T = t_1 - t_0$ the time duration and $h = q_1 - q_0$ the displacement. The velocity is constant over this T and its value is:

$$\dot{q}(t) = \frac{h}{T} \quad (2.5)$$

Obviously, the acceleration present a null value all over the interval minus at the extremities whereas present an impulsive behaviour.

Parabolic Trajectory

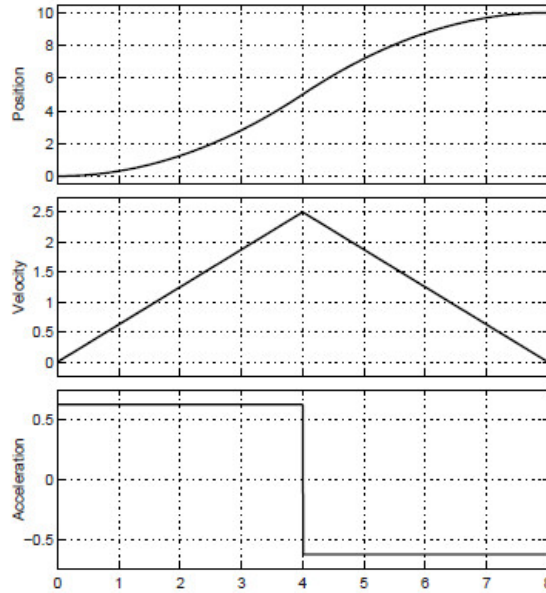


Figure 2.6. Parabolic Trajectory [12]

This trajectory in Figure 2.6 is characterized by an acceleration whose absolute value is constant along all the path, meanwhile the sign changes depending if the motion is in the acceleration or in the deceleration period. From an analytical point

of view, this is the composition of two second degree polynomials with a flex point in the middle, so, the motion can be expressed by the two functions [12]:

$$\begin{aligned} q_a(t) &= a_0 + a_1(t - t_0) + a_2(t - t_0)^2, & t \in [t_0, t_f] \\ q_b(t) &= a_3 + a_4(t - t_f) + a_5(t - t_f)^2, & t \in [t_f, t_1] \end{aligned} \quad (2.6)$$

where t_0 and t_1 are respectively the initial and final time instants and t_f is the time instants related to the flex point. Doing the maths we can find the velocity at the flex point as:

$$v_{max} = \dot{q}_a(t_f) = 2\frac{h}{T} - v_0 \quad (2.7)$$

It is important to notice that, in case $v_0 = 0$, v_{max} results to be double with respect to the constant velocity in the linear case. The jerk will be always null at except of the flex point, when the acceleration changes sign.

Cubic Trajectory

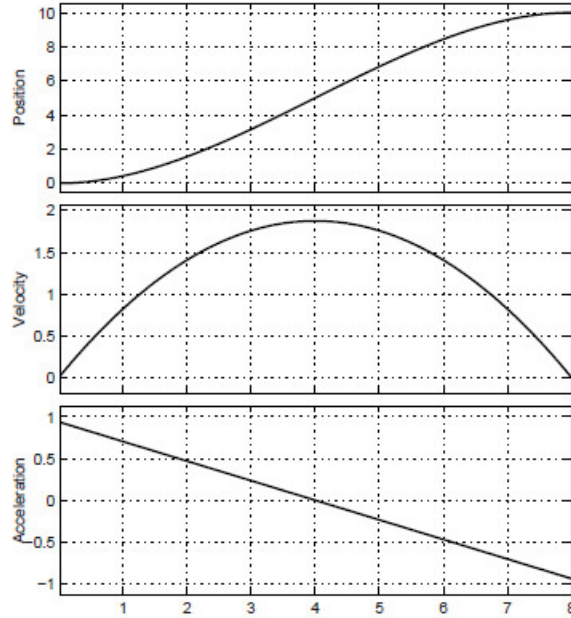


Figure 2.7. Cubic trajectory [12]

A good criterion to choose the primitive function to interpolate the points is to impose a trajectory that minimizes the energy dissipated from the joint motors [9].

The solution of this problem is a third degree equation like this:

$$q(t) = a_3 t^3 + a_2 t^2 + a_1 t + a_0 \quad (2.8)$$

The following velocity and acceleration are associated to it:

$$\dot{q}(t) = 3a_3 t^2 + 2a_2 t + a_1 \quad (2.9)$$

$$\ddot{q}(t) = 6a_3 t + 2a_2 \quad (2.10)$$

Having four coefficients to choose, we can impose, in addition to the initial and the final values q_i and q_f , also the initial and final velocities \dot{q}_i and \dot{q}_f . The trajectory, depicted in Figure 2.7 is therefore given by the following system of equations:

$$\begin{aligned} a_0 &= q_i \\ a_1 &= \dot{q}_i \\ a_3 t^3 + a_2 t^2 + a_1 t + a_0 &= q_f \\ 3a_3 t^2 + 2a_2 t + a_1 &= \dot{q}_f \end{aligned} \quad (2.11)$$

Trapezoidal profile(or 2-1-2)

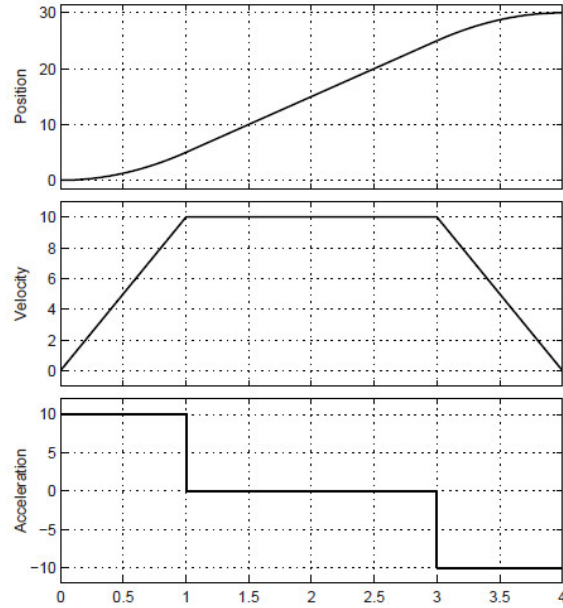


Figure 2.8. Trapezoidal trajectory [12]

Another approach often used in the industry is the 2-1-2 trajectory, in which a trapezoidal profile, shown in Figure 2.8, is assigned to the velocity. This permits to impose a constant acceleration at the beginning of the motion, a constant cruising speed, and finally a constant deceleration in the final phase of the motion [9]. The name of this trajectory is given by the resultant position profile, which is a composition of two parabolic segments, at the beginning and at the end of the trajectory, and a linear segment in the middle of them.

Once q_i , q_f and t_f are assigned, we obtain the motion law referred in (2.12) [12], and plotted in Figure 2.8:

$$q(t) = \begin{cases} q_i + \frac{1}{2}\ddot{q}_c t^2 & 0 \leq t \leq t_c \\ q_i + \ddot{q}_c t_c (t - t_c/2) & t_c < t \leq t_f - t_c \\ q_f - \frac{1}{2}\ddot{q}_c (t_f - t)^2 & t_f - t_c < t \leq t_f \end{cases} \quad (2.12)$$

The advantage of this method is that it allows to verify immediately if the velocity and acceleration imposed by these laws are consistent with the physical characteristics of the manipulator. The drawback is a poorer quality with respect of the third order polynomial examined previously.

Double-S Trajectory

The main drawback of the trapezoidal trajectory is to present a discontinuous acceleration, which could generate stresses and undesired vibrational effects on the mechanical system. Therefore, the Double-S profile, in Figure 2.9 can resolve this problem, being a smoother motion profile in which the jerk is characterized by a step profile, instead of an impulsive profile. This trajectory is made by the composition of several linear segments connected each others by parabolic blends.

Online computation of Double-S trajectory

From [12] also another elegant and simple way to implement a Double-S trajectory is shown, that is, an on-line computation of this kind of trajectory. This kind of computation is appropriate for CNC machines, where the trajectory profiles are computed in discrete time, Being T_s the sampling period, let us define the values of

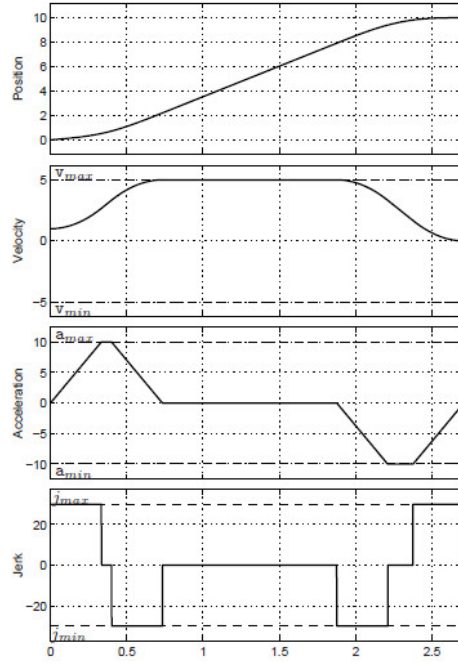


Figure 2.9. Double-S Trajectory [12]

position, velocity, acceleration and jerk at a certain time instant k as:

$$\begin{cases} q(t = kT_s) = q_k \\ \dot{q}(t = kT_s) = \dot{q}_k \\ \ddot{q}(t = kT_s) = \ddot{q}_k \\ q^{(3)}(t = kT_s) = q_k^{(3)} \end{cases} \quad (2.13)$$

Then, given the initial and final values of position, velocity and acceleration and also the constraints $(v_{max}, v_{min}, a_{max}, a_{min}, j_{max}, j_{min})$, the profile is computed by setting the desired jerk profile and then by integrating it three times, obtaining something like this:

$$\begin{aligned} \ddot{q}_k &= \ddot{q}_{k-1} + \frac{T_s}{2}(q_{k-1}^3 + q_k^3) \\ \dot{q}_k &= \dot{q}_{k-1} + \frac{T_s}{2}(\ddot{q}_{k-1} + \ddot{q}_k) \\ q_k &= q_{k-1} + \frac{T_s}{2}(\dot{q}_{k-1} + \dot{q}_k) \end{aligned} \quad (2.14)$$

The basic idea is to perform the acceleration phase and then the constant velocity phase segment until it is necessary to decelerate in order to reach the final position q_i

with the desired values of velocity and acceleration and keeping jerk and acceleration within the desired constraints. Figure 2.10 reports an example of structure for this kind of trajectory planner.

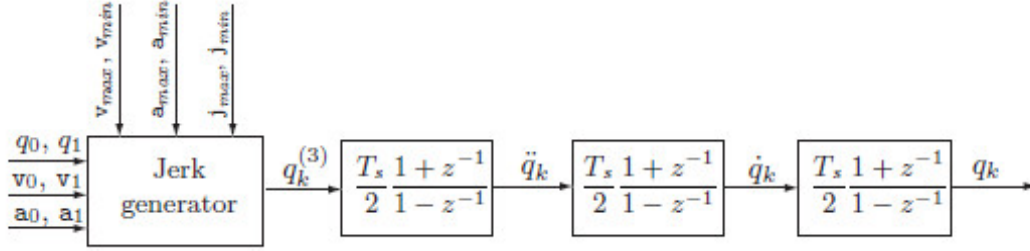


Figure 2.10. Online double-S generator [12]

2.3.2 Multipoint Trajectories

Introduction

Although point-to-point techniques are simple to implement, in most cases the path specified by the user is more complex and requires a number of point greater than two. At a first sight, a reasoning like the one of the third order polynomial could be made, i.e., if n point are specified to be reached by the robot, that's means that each one of the joints of the manipulator has n constraints and, so, a polynomial of degree $n - 1$ should be used. But, in the practice, the result is not so obvious and this approach carries many drawbacks:

- No possibility to assign initial and final velocity
- Higher is the degree, higher is the oscillatory nature of the manipulator. This can lead to not so natural trajectories
- The system of equation derived by the constraints becomes more complex and tougher from a numerical point of view
- All the coefficients are related. If you want to change a part of the trajectory you need to re-calculate all the coefficients
- If the degree of the polynomial increases, the numerical accuracy to calculate it decreases

A way to avoid these problems is to consider the polynomial not as a unique complex polynomial of high degree but as a sum of simpler polynomials of lower degree, which are connected in the n desired points of the path [9], as shown in Figure 2.11. The overall function $s(t)$ in this way is called *spline* of degree p , and it is examined in detail in Chapter 2.3.

Cubic splines

The lowest degree polynomial that can be taken is the cubic polynomial [9]:

$$q(t) = a_0 + a_1t + a_2t^2 + a_3t^3 \quad (2.15)$$

since it is the lowest degree polynomial that preserves the continuity of the velocity in the junction points of the path. Then, the overall function is:

$$\begin{aligned} s(t) &= \{q_k t, \quad t \in [t_k, t_{k+1}], \quad k = 0, \dots, n-1\}, \\ q_k(t) &= a_{k0} + a_{k1}(t - t_k) + a_{k2}(t - t_k)^2 + a_{k3}(t - t_k)^3 \end{aligned} \quad (2.16)$$

Since n cubic polynomials are necessary to define a trajectory passing through $n + 1$ points, the number of coefficients to be determined is $4n$. Then, to solve this problem, there are the following conditions to be satisfied [12]:

- $2n$ conditions for the interpolation of the points given by the user.
- $2(n - 1)$ conditions for the continuity of velocity and acceleration at the transition points.

Doing the maths we can check that, once the upper conditions are satisfied there are two degrees of freedom left. Then, two constraints must be imposed to compute the spline. There are possibilities to impose such constraints:

- The initial and final velocity $\dot{s}(t_0) = v_0, \quad \dot{s}(t_n) = v_n$.
- The initial and final acceleration $\ddot{s}(t_0), \quad \ddot{s}(t_n)$.
- The conditions $\dot{s}(t_0) = \dot{s}(t_n), \quad \ddot{s}(t_0) = \ddot{s}(t_n)$, useful when it is necessary to define a periodic spline.
- The Jerk's continuity at time instants t_1, t_{n-1} .

Generally, a spline is characterized by the following properties [12]:

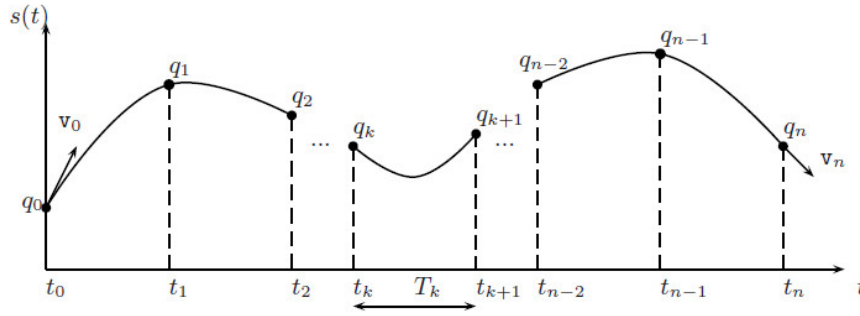


Figure 2.11. Cubic spline curve [12]

1. $[n(p+1)]$ parameters are sufficient to define a trajectory $s(t)$ of degree p , which interpolates the given points (t_k, q_k) , $k = 0, \dots, n$.
2. Once given the $n + 1$ points and the boundary conditions, the spline of degree p interpolating those points is univocally determined.
3. The degree p of the polynomials composing the spline does not depend on the number of points.
4. $s(t)$ has continuous derivatives up to the order $(p - 1)$.
5. If the condition $\ddot{s}(t_0) = \ddot{s}(t_n) = 0$ is assumed, the cubic spline is, among all the functions which interpolate the given entry points with continuous first and second derivatives, the one which minimizes the functional

$$J = \int_{t_0}^{t_n} \left(\frac{d^2 f(t)}{dt^2} \right) dt$$

that is a sort of deformation energy, proportional to the curvature of the function. The last condition is particularly important because it shows that the function which minimizes the oscillatory motion of the robot is the cubic spline with zero condition on the initial and final velocities. This kind of spline is called '*natural spline*'.

2.4 Trajectory in the task space

Quite different is the situation when the manipulator has to follow a specific path in the task space. In this case it is necessary to plan the trajectory directly in the operational space. In this case the planning can be done either by interpolating a sequence of prescribed path points or by creating the analytical motion primitive

and its relative trajectory in the space in a punctual way. In both cases, it is important to take in consideration that a part of the time used by the planner is devolved to the computation of the values of the joint equivalent variables through an inverse kinematics algorithm, so the computational complexity induced by the union of the blocks Planner-Inverse kinematics in Figure 2.1 sets an upper limit on the maximum sampling rate required to generate the sequence of variables [9]. Here, the same techniques used for the generation of the joint trajectories can be followed but, due to the fact that the motion is made in a three-dimensional space, it is necessary to express it analytically, i.e, it is necessary to refer to some one-dimensional motion primitives which describe the geometric path and the time law in the 3D space.

Parametric representation in 3D space

Referring to Chapter 2, and, from [12], let's consider a vector $\mathbf{p} \in \mathbb{R}^{3 \times 1}$ and a continuous function $\mathbf{f}(u)$, with $u \in [u_i, u_n]$, then:

$$\mathbf{p} = \mathbf{f}(u) \quad (2.17)$$

The sequence of values of \mathbf{p} with u varying in $[u_i, u_f]$, with reference to its geometrical description, is the total path Γ in the space, which means that, increasing u in the interval, the point \mathbf{p} moves along the path in a certain direction. So, the equation (2.17) is the *parametric representation* of the path Γ . Now, let \mathbf{p}_i be a fixed point and \mathbf{p} a generic point on the path Γ , on which a certain direction has already been fixed, and, according to the direction of the path, let's \mathbf{p}_i precede \mathbf{p} . Then, the *arc length* s of \mathbf{p} is the length of the arc on the path which goes from \mathbf{p}_i to \mathbf{p} , and the point \mathbf{p}_i is the origin of the arc length ($s = 0$). So, it is easy to understand that, since s represents each point on the path, it can be used as a parameter in the parametric representation of Γ . So, the equation (2.17) becomes:

$$\mathbf{p} = \mathbf{f}(s) \quad (2.18)$$

Then, let's consider a path Γ represented by the parametric representation (2.18) and let's take a generic point \mathbf{p} on this path. The point \mathbf{p} allows the definition of three unit vectors characterizing the path. Their orientation will depend only on the path geometry, meanwhile their direction will depend on the direction induced by the parametric representation on the path. The three vectors are:

- The *tangent unit vector* denoted by \mathbf{t} , oriented along the direction induced by s on the path

- The *normal unit vector* denoted by \mathbf{n} , oriented along the line containing \mathbf{t} at right angle with it and standing on the same plane
- The *binormal unit vector* denoted by \mathbf{b} , oriented in a way such that the frame of the three vectors is right-handed

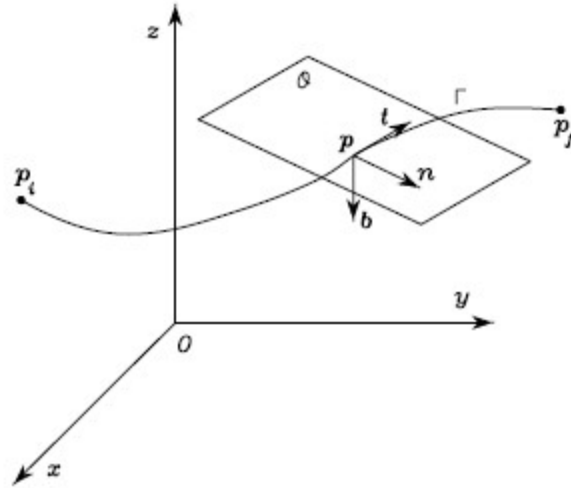


Figure 2.12. Path representation in 3D space [12]

Figure 2.12 shows a graphical representation of the path and of the three vectors.

2.4.1 Multidimensional Trajectories

Global Interpolation

As already discussed, one of the easiest and most efficient way to interpolate a set of points q_k , $k = 0, \dots, n$, is to use p degree B-Spline curves.

A first step to interpolate the above points with the B-splines is to consider the parametric representation of the curve and, so, to choose the parameter \bar{u}_k for each point q_k . The parameter \bar{u}_k is usually assumed within the range $[0,1]$. So, the initial and final values are $\bar{u}_0 = 0$ and $\bar{u}_n = 1$. For what concerns the other values, there are several ways to determine them, but a simple and efficient method is to take them equally spaced in the interval $[0,1]$, so:

$$\bar{u}_k = \frac{k}{n} \quad (2.19)$$

Then, it is necessary to choose a suitable knot vector $u = [u_0, \dots, u_{nknot}]$, (a more detailed discussion about this is available in Chapter 2.3.2). The global interpolation consists in setting up a system of $m+1$ linear equations in the unknown p_j obtained by imposing that the curve crosses all the points q_k in the instants \bar{u}_k [12]:

$$q_k = s(\bar{u}_k) = \sum_{j=0}^m p_j B_j^p(\bar{u}_k) \quad (2.20)$$

or, in matrix form:

$$q_k^T = [B_0^p(\bar{u}_k), B_1^p(\bar{u}_k), \dots, B_{m-1}^p(\bar{u}_k), B_m^p(\bar{u}_k)] \begin{bmatrix} p_0^T \\ p_1^T \\ \vdots \\ p_{m-1}^T \\ p_m^T \end{bmatrix} \quad (2.21)$$

Cubic B-spline Interpolation

Such problem is often solved assuming $p = 3$, that produces the classic cubic splines. In the following there are some general details of this implementation [12]. In this case the parameters \bar{u}_k are used to determine the knot vector as:

$$\begin{aligned} u_0 = u_1 = u_2 = \bar{u}_0 & \quad u_{n+4} = u_{n+5} = u_{n+6} = \bar{u}_n \\ u_{j+3} = \bar{u}_j & \quad j = 0, \dots, n \end{aligned} \quad (2.22)$$

This choice makes the interpolation occur at the knots. Since $n_{knot} = m + 4$ and there are $m + 1$ control points and $n_{knot} + 1$ knots, with $n_{knot} = n + 6$, it descends that the control points p_j are $n + 3$. So, it is necessary to impose two additional constraints. Usually these two constraints are fixed on the first derivatives at the endpoints, t_0 and t_n . As a consequence, the first two equations and also the last two of the linear system are:

$$\begin{aligned} p_0 &= q_0 \\ -p_0 + p_1 &= \frac{u_4}{3}t_0 \\ -p_{n+1} + p_{n+2} &= \frac{1 - u_{n+3}}{3}t_n \\ p_{n+2} &= q_n \end{aligned} \tag{2.23}$$

The remaining control points are then computed imposing:

$$s(\bar{u}_k) = q_k, \quad k = 1, \dots, n - 1 \tag{2.24}$$

Since in an interior knot of a cubic spline there are only three basis functions which are not null, the $n - 1$ equations have the expression:

$$q_k = B_k^3(\bar{u}_k)p_k + B_{k+1}^3(\bar{u}_k)p_{k+1} + B_{k+2}^3(\bar{u}_k)p_{k+2} \tag{2.25}$$

Therefore we can write them in form of a tridiagonal system:

$$\mathbf{B}\mathbf{P} = \mathbf{R} \tag{2.26}$$

where

$$\mathbf{B} = \begin{bmatrix} B_2^3(\bar{u}_1) & B_3^3(\bar{u}_1) & 0 & \dots & 0 \\ B_2^3(\bar{u}_2) & B_3^3(\bar{u}_2) & B_4^3(\bar{u}_2) & & \vdots \\ 0 & & \ddots & & 0 \\ \vdots & & & B_{n-2}^3(\bar{u}_n - 2) & B_{n-1}^3(\bar{u}_n - 2) & B_n^3(\bar{u}_n - 2) \\ 0 & \dots & & 0 & B_{n-1}^3(\bar{u}_n - 1) & B_n^3(\bar{u}_n - 1) \end{bmatrix}$$

$$\mathbf{P} = \begin{bmatrix} p_2^T \\ p_3^T \\ p_4^T \\ \vdots \\ p_{n-1}^T \\ p_n^T \end{bmatrix} \quad \mathbf{R} = \begin{bmatrix} q_1^T - B_1^3(\bar{u}_1)p_1^T \\ q_2^T \\ q_3^T \\ \vdots \\ q_{n-2}^T \\ q_{n-1}^T - B_{n+1}^3(\bar{u}_{n-1})p_{n+1}^T \end{bmatrix}$$

Finally, the control points necessary to obtain the curve can be computed by a simple inversion problem:

$$\mathbf{P} = \mathbf{B}^{-1}\mathbf{R} \quad (2.27)$$

Once the control points are computed, given the knot vector \mathbf{u} , the B-spline is completely defined and we can compute $\mathbf{s}(u)$ for any value of u .

Use of NURBS for trajectory generation

NURBS(Non-Uniform Rational B-Splines) are standard curves, used in many CNC machines. They are described by the following expression [12]:

$$n(u) = \frac{\sum_{j=0}^m p_j w_j \mathbf{B}_j^p(u)}{w_j \sum_{j=0}^m \mathbf{B}_j^p(u)}, \quad u_{min} \leq u \leq u_{max} \quad (2.28)$$

and represented in Figure 2.13. From (2.28) it can be seen that they are a generalization of non-rational B-Splines with the difference of the weights w_j associated to them, which change the shape of the curve (More details about NURBS are available in Chapter 2.5).

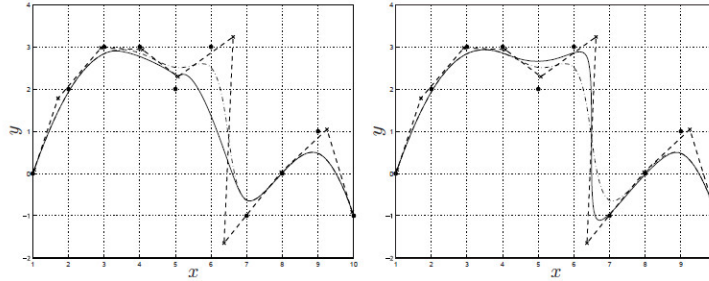


Figure 2.13. NURBS with different weights [12]

The techniques previously explained, based on B-Spline, can be adopted with minor modifications in the case of Nurbs. An immediate and easy approach is to use the classic techniques with B-Spline and afterwards change the shape of the obtained trajectory through the modification of the weights associated to each B-Spline.

2.5 Orientation of the tool

The orientation of the end-effector at each time instant may be expressed in terms of a rotation matrix composed by three orthogonal vectors (called *normal*, *slide* and

approach) [12]:

$$\mathbf{R} = [\mathbf{n}, \mathbf{s}, \mathbf{a}]$$

which describe the orientation of the tool with respect of the orientation of the base world frame. Then, to specify the orientation of the tool for each point of the curve, we have to specify a rotation matrix \mathbf{R}_k at each point p_k and interpolate them using one of the techniques detailed in Chapter 2.3 and 2.4. In fact, the planning of the trajectory in this case can be made as in the case of the path planning, with the difference that in this case we are not interpolating positions, but matrices or some parameters, as will be explained later. It is also important to notice that the orientation planning is coupled with the path, so the planning has to be coherent to the trajectory planning made previously in order to avoid discrepancies between the movement of the arm of the robot and the movement of the tool. An important drawback of using rotation matrices to describe the evolution of the orientation of the end effector is the impossibility to guarantee the orthonormality of these matrices along the path. So, different criteria are taken in consideration to describe the orientation path, with the objective of change the parameter which will be interpolated, making easier to use the techniques previously examined. A very important tool in orientation planning, which permits to overtake the rotation matrices' representation and permits to introduce other techniques of representation is the *Euler's rotation Theorem*:

Theorem 1 (Euler's rotation theorem) *Let $O, O' \in \mathbf{R}^3$ be two orientations. Then there exists an axis $l \in \mathbf{R}^3$ and an angle of rotation $\theta \in]-\pi, \pi]$ such that O yields O' when rotated θ about l [13]*

Plainly, as we can see in Figur 2.14, given two orientations O and O' we can always express them trough a rotation of an angle θ about an axis l .

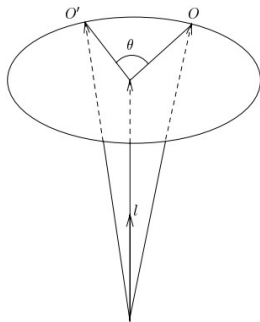


Figure 2.14. Rotation expressed by the Euler's theorem [13]

This is a powerful tool which permits us to describe the change of orientation in several way, the most used of them are detailed in the following, and are [10] [14] [15]:

- **Axis-Angle** : A rotation of an angle θ about an axis \mathbf{u} .
- **Planar sliding** : A composition of an axis sliding along a plane and an angle rotating about an other axis.
- **Euler angles** : Three angles which describe the orientation of a rigid body with respect to a fixed coordinate system.
- **Quaternions** : A 4D object used to describe rotation in the 3D space.

It is important to notice that, no matter if the initial and final orientation frames of the motion are the same, each parametrization gives a different kind of motion between these two frames, as can be seen in Figure 2.15.

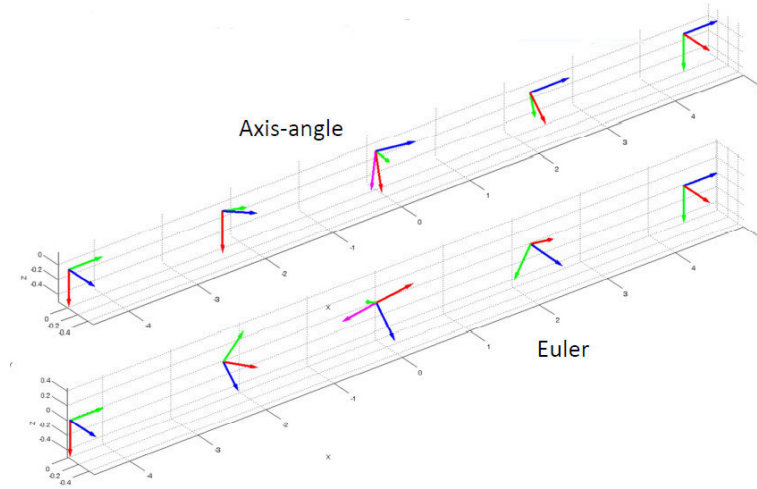


Figure 2.15. Different orientations [10]

2.5.1 Axis-Angle

From the Euler's theorem we can conclude that, given two different frames in the Cartesian space with same origin and different orientations, it is always possible to find a unit vector through which it is possible to pass from the first to the second frame by a rotation of a certain angle about this unit vector. Given the two rotation matrices \mathbf{R}_i and \mathbf{R}_f , representing respectively the initial and final orientation, to

plan the rotation between the two it is necessary to compute the *incremental* rotation \mathbf{R}_k such that:

$$\mathbf{R}_f = \mathbf{R}_i \mathbf{R}_k \quad (2.29)$$

This expression leads to the following statement:

$$\mathbf{R}_k = \mathbf{R}_i^T \mathbf{R}_f = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \quad (2.30)$$

If we define the matrix $\mathbf{R}^k(t)$ to define a rotation from \mathbf{R}_i to \mathbf{R}_f , it is $\mathbf{R}^k(0) = \mathbf{I}$ and $\mathbf{R}^k(t_f) = \mathbf{R}_k$. So, the rotation matrix can effectively be expressed as a rotation of a given angle about a fixed axis in the space (see Figure 2.14). The angle and the axis can be computed as:

$$\begin{aligned} \theta &= \cos^{-1} \left(\frac{r_{11} + r_{22} + r_{33} - 1}{2} \right) \\ \mathbf{u} &= \frac{1}{2\sin(\theta)} = \begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \end{aligned} \quad (2.31)$$

with $\sin(\theta) \neq 0$. So, the matrix $\mathbf{R}_k(t)$ can be also expressed as $\mathbf{R}(\theta(t), \mathbf{u})$. At this point, keeping \mathbf{u} it is possible to express the rotation matrix at each time instant through the planning the variable θ [10] [9].

2.5.2 Planar sliding

The term "planar sliding" derive from the fact that with this kind of technique the rotation is made by the composition of two different motions. In fact, there is both a motion of one of the reference axis (usually z) sliding on a plane and a rotation about this sliding axis. Summarizing, these are the two motions are [10]:

- A first rotation about a fixed axis \mathbf{u}
- A second rotation about a moving local axis \mathbf{k}

Composing the two rotations we have:

$$\mathbf{R}_k = \mathbf{R}(\mathbf{u}, \beta_k) \mathbf{R}(\mathbf{k}, \alpha_k) \quad (2.32)$$

The plane on which \mathbf{k} (the unit vector which defines the axis z) slides is usually defined by the two unit vectors \mathbf{k}_i and \mathbf{k}_f , given by the last columns of the rotation matrices \mathbf{R}_i and \mathbf{R}_f . From these we can compute the vector \mathbf{u} as:

$$\mathbf{u} = \frac{\mathbf{k}_i \times \mathbf{k}_f}{\|\mathbf{k}_i \times \mathbf{k}_f\|}, \quad \|\mathbf{u}\| = 1 \quad (2.33)$$

And, since $\|\mathbf{k}_i\| = \|\mathbf{k}_f\| = 1$, it follows that $\|\mathbf{k}_i \times \mathbf{k}_f\| = \sin(\beta)$. From this we can compute the angle β which provides the first rotation, that around \mathbf{u} in order to move from \mathbf{k}_i to \mathbf{k}_f :

$$\beta = \arcsin(\|\mathbf{k}_i \times \mathbf{k}_f\|) \quad (2.34)$$

This rotation, which produces the sliding depicted in Figure 2.16, is represented by

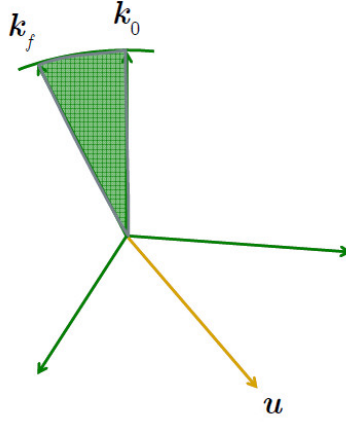


Figure 2.16. Sliding of \mathbf{k} on the plane defined by \mathbf{k}_i and \mathbf{k}_f [10]

the matrix $\mathbf{R}(\mathbf{u}, \beta)$ which can be computed in the following way:

$$\mathbf{R}(\mathbf{u}, \beta) = \mathbf{I} + \sin(\beta) * \mathbf{S} + (1 - \cos(\beta))\mathbf{S}^2 \quad (2.35)$$

with

$$\mathbf{S} = \begin{bmatrix} 0 & -u_3 & -u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{bmatrix} \quad (2.36)$$

The second rotation is an elementary one about the local axis \mathbf{k} . So:

$$\mathbf{R}(\mathbf{k}, \alpha) = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.37)$$

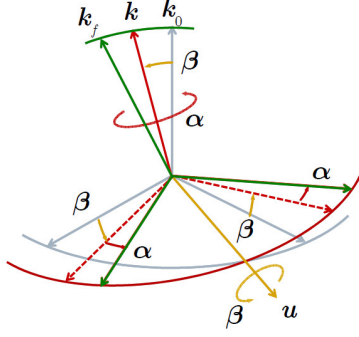


Figure 2.17. Overall movement through planar sliding technique [10]

In order to compute α , we can observe that, after the rotation $\mathbf{R}(\mathbf{u}, \beta)$, the unit vector representing the y axis in the initial frame, \mathbf{j}_i , changes and becomes $\tilde{\mathbf{j}}_i$. So, we can compute α from:

$$\sin(\alpha) = \|\tilde{\mathbf{j}}_i \times \mathbf{j}_f\| \quad (2.38)$$

observing that:

$$\mathbf{j}_i = \mathbf{R}(\mathbf{u}, \beta) \tilde{\mathbf{j}}_i \implies \tilde{\mathbf{j}}_i = \mathbf{R}^T(\mathbf{u}, \beta) \mathbf{j}_i \quad (2.39)$$

we can compute α :

$$\alpha = \arcsin(\|(\mathbf{R}^T(\mathbf{u}, \beta) \mathbf{j}_i) \times \mathbf{j}_f\|) \quad (2.40)$$

Finally, once computed α and β , from (2.40) and (2.34), we can plan the trajectory through these two parameters. The total movement is represented in Figure 2.17.

2.5.3 Euler angles

Another choice to parametrize the space of rotations is given by Euler's angles. When this method is used, a general orientation is written as a series of elementary rotations about three mutually orthogonal axes in the space. This gives us three parameters to express the rotation of the object. Several choices can be made to decide the series of axes about which these rotations should happen. Usually, the ZYZ representation is used for this purpose, that is, given the three elementary rotation matrices, representing a rotation about a single axis in the 3D space, as

[14], [12]:

$$\begin{aligned}\mathbf{R}(\mathbf{i}, \phi) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix} \\ \mathbf{R}(\mathbf{j}, \theta) &= \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \\ \mathbf{R}(\mathbf{k}, \psi) &= \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}\end{aligned}\tag{2.41}$$

The ZYZ rotation is made by:

$$\mathbf{R}_k = \mathbf{R}(\mathbf{k}, \phi) \mathbf{R}(\mathbf{j}, \theta) \mathbf{R}(\mathbf{k}, \psi)\tag{2.42}$$

that is, an elementary rotation about z in the first frame F_0 followed by an elementary rotation about y in the second frame F_1 and, finally, another rotation about the z axis in the last frame obtained, F_2 . Figure 2.18 shows a representation of these movements.

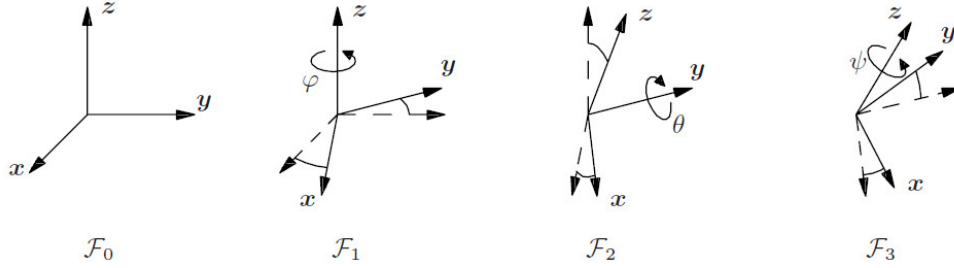


Figure 2.18. Set of elementary rotations expressed in Euler angles terms [12]

The rotation matrix corresponding to the three rotations in (2.42) is:

$$\mathbf{R}_k(\phi, \theta, \psi) = \begin{bmatrix} c_\phi c_\theta c_\psi - s_\phi s_\psi & -c_\phi c_\theta s_\psi - s_\phi c_\psi & c_\phi s_\theta \\ s_\phi c_\theta c_\psi + c_\phi s_\psi & -s_\phi c_\theta s_\psi + c_\phi c_\psi & s_\phi s_\theta \\ -s_\theta c_\psi & s_\theta s_\psi & c_\theta \end{bmatrix}\tag{2.43}$$

with $c = \cos()$ and $s = \sin()$.

Given \mathbf{R}_k it is possible to find the three parameters which characterize the rotation through the following equations:

1. If $r_{13}^2 + r_{23}^2 \neq 0$, then $\sin(\theta) \neq 0$, there are two different sets of solutions, depending on the sign of the angle θ . If $0 < \theta < \pi$, from the matrix in (2.43):

$$\begin{cases} \phi = \text{atan2}(r_{23}, r_{13}) \\ \theta = \text{atan2}(\sqrt{r_{13}^2 + r_{23}^2}, r_{33}) \\ \psi = \text{atan2}(r_{32}, -r_{31}) \end{cases} \quad (2.44)$$

meanwhile, if $-\pi < \theta < 0$:

$$\begin{cases} \phi = \text{atan2}(-r_{23}, -r_{13}) \\ \theta = \text{atan2}(-\sqrt{r_{13}^2 + r_{23}^2}, r_{33}) \\ \psi = \text{atan2}(-r_{32}, -r_{31}) \end{cases} \quad (2.45)$$

2. If $r_{13}^2 + r_{23}^2 = 0$, which implies $\sin(\theta) = 0, \pi$ and $\cos(\theta) = \pm 1$, there are other two different solutions. If $\theta = 0$, i.e., $\cos(\theta) = 1$:

$$\begin{cases} \theta = 0 \\ \phi + \psi = \text{atan2}(r_{21}, r_{11}) = \text{atan2}(-r_{12}, r_{11}) \end{cases} \quad (2.46)$$

In the other case, if $\theta = \pi$, that means $\cos(\theta) = -1$:

$$\begin{cases} \theta = \pi \\ \phi - \psi = \text{atan2}(-r_{21}, -r_{11}) = \text{atan2}(-r_{12}, -r_{11}) \end{cases} \quad (2.47)$$

Through (2.46) and (2.47) we see the principal drawback of the parametrization with Euler's angles. In both systems of equations there are infinite solutions, since ϕ and ψ cannot be determined separately, but only the sum(or difference) of the two can be computed. Physically, what it is happening is that two axes of rotations are driven on parallel planes by previous rotations. In this case, known also as *gimbal lock*, a loss of one of the three degrees of freedom given by the configuration occurs, driving the rotation in a two dimensional space. Figure 2.19 shows a graphical representation of this phenomenon.

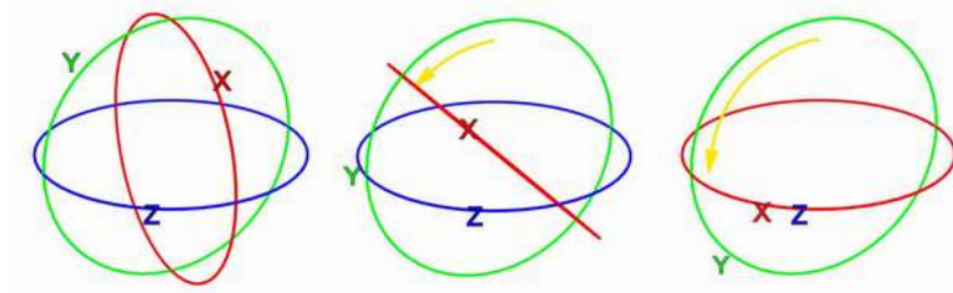


Figure 2.19. Gimbal lock representation [16]

Although the presence of this problem, this method it is widely used in the industry, due to the fact that it is really simple to use, even though it suffers also of a poor geometrical insight.

Having a set of orientations described by Euler angles it is also really simple to plan the trajectory along this set. In fact, it is only necessary to take these three parameters and interpolates them using one of the techniques detailed in Chapter 2.4 for the positions.

Anyway, in those applications where it is strongly needed to avoid singularities, a more robust technique is preferred, like the one described in the next section.

2.5.4 Quaternions

Another rotation modality derived from the Euler's Theorem is based on the definition of a new mathematical object, the *quaternion*. Invented in 1843 by Sir William Rowan Hamilton, whose aim was to generalize complex numbers in three dimensions, quaternions are elements of a 4D linear space $\mathbb{H}(\mathbb{R})$ and defined on the real number field $\mathfrak{F} = \mathbb{R}$, with base $\{1 \ i \ j \ k\}$, which nowadays are really useful in such fields like computer graphics, computer vision and, for sure, also robotics.

Historical background

Hamilton's aim was to generalize complex numbers to the third dimension, i.e., to obtain an object of the form $a + ib + jc$, with $a, b, c \in \mathbb{R}$ and $i^2 = j^2 = -1$. One of the principal motivations of Hamilton to look for this generalization was to find a description of a rotation in 3D space analogue to the one in 2D space, which was possible with complex numbers, where a multiplication correspond to a rotation and also a scaling in the plane. Unfortunately he never succeeded to obtain this kind of

generalization because, as it was proved later, the set of three-dimensional numbers is not closed under multiplication.



Figure 2.20. Hamilton portrait and the plaque on the Broom Bridge [15] [17]

One day, in October 1843, while walking along the Broom bridge on the Royal Canal in Dublin (Figure 2.20), he realized that four numbers instead of three were necessary to describe a rotation followed by a scaling in the three-dimensional space. In fact, one number describes the size of the scaling, another one the angle of rotation and the last two the plane on which the rotation takes place. Then, he found a closed multiplication for 4D complex numbers of the form $ix + jy + kz$, where $i^2 = j^2 = k^2 = ijk = -1$, and gave to this particular object the name of *quaternion*.

Maths of quaternions

As said, quaternions are elements of a 4D linear space $\mathbb{H}(\mathbb{R})$ with base $\{1 i j k\}$. i , j and k have *hypercomplex* numbers which satisfy the following multiplication rules [14] [13]:

$$\begin{aligned} i^2 &= j^2 = k^2 = ijk = -1 \\ ij &= -ji = k \\ jk &= -kj = i \\ ki &= -ik = j \end{aligned} \tag{2.48}$$

Then, a quaternion $q \in \mathbb{H}$ can be defined in several ways. For example, remembering the analogy with complex numbers, it can be defined as a linear combination of the base $\{1 \mathbf{i} \mathbf{j} \mathbf{k}\}$:

$$q = q_0 + q_1 \mathbf{i} + q_2 \mathbf{j} + q_3 \mathbf{k} \quad (2.49)$$

where q_i , $i = 0, \dots, 3$ are real.

It is also possible to represent it as a quadruple of real numbers:

$$q = (q_0, q_1, q_2, q_3) \quad (2.50)$$

in analogy with complex numbers which can be defined as a couple of real numbers. At the same way in which complex numbers are a sum of a real and an imaginary part, quaternions are a sum of a real part and a **vectorial** one. The real part q_r is defined as $q_r = q_0$, meanwhile the vectorial part is defined as $\mathbf{q}_v = q_1 \mathbf{i} + q_2 \mathbf{j} + q_3 \mathbf{k}$. Then, the quaternion can be written as $q = (q_r, \mathbf{q}_v)$ or $q = q_r + \mathbf{q}_v$. Another way to see quaternions is to see them as mathematical objects which, in turn, include three different objects:

- Real numbers:

$$r = (r, 0, 0, 0), \quad r \in \mathbb{R} \quad (2.51)$$

- Complex numbers:

$$a + \mathbf{i}b = (a, b, 0, 0), \quad a, b \in \mathbb{R} \quad (2.52)$$

- Real vector in \mathbb{R}^3 :

$$\mathbf{v} = (0, v_1, v_2, v_3), \quad v_i \in \mathbb{R}, \quad i = 1, 2, 3 \quad (2.53)$$

In the last case the elements $\{1 \mathbf{i} \mathbf{j} \mathbf{k}\}$ can easily be seen as unit vectors $\{\mathbf{1} \mathbf{i} \mathbf{j} \mathbf{k}\}$ forming an orthonormal base in the cartesian reference frame. In fact, it can be checked that multiplication rules among the elements $\mathbf{i} \mathbf{j} \mathbf{k}$ have the same properties of the cross product among the unit vectors $\mathbf{i} \mathbf{j} \mathbf{k}$:

$$\begin{aligned} \mathbf{i} \mathbf{j} &= \mathbf{k} \Leftrightarrow \mathbf{i} \times \mathbf{j} = \mathbf{k} \\ \mathbf{j} \mathbf{i} &= -\mathbf{k} \Leftrightarrow -\mathbf{j} \times \mathbf{i} = -\mathbf{k} \end{aligned} \quad (2.54)$$

and so on. . .

Another interesting way to write a quaternion is in matrix form. A quaternion can

be seen, from this point of view, as four 2×2 complex matrices. So, starting from the classical form:

$$\begin{aligned}
 q &= q_0 \mathbf{1} + q_1 \mathbf{i} + q_2 \mathbf{j} + q_3 \mathbf{k} \\
 &\Downarrow \\
 \mathbf{1} &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \mathbf{i} = \begin{bmatrix} \mathbf{i} & 0 \\ 0 & -\mathbf{i} \end{bmatrix} \mathbf{j} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \mathbf{k} = \begin{bmatrix} 0 & \mathbf{i} \\ \mathbf{i} & 0 \end{bmatrix}
 \end{aligned} \tag{2.55}$$

with $\mathbf{i}^2 = -1$.

Each one of these matrix is in the form:

$$\begin{bmatrix} c & d \\ -d^* & c^* \end{bmatrix} \tag{2.56}$$

which are called **Cayley matrices**.

Algebraic properties

Taking into account a generic quaternion $q = q_0 \mathbf{1} + q_1 \mathbf{i} + q_2 \mathbf{j} + q_3 \mathbf{k} = (q_r, \mathbf{q}_v)$ the following properties hold [14] [13]:

- null or zero quaternion:

$$0 = 0\mathbf{1} + 0\mathbf{i} + 0\mathbf{j} + 0\mathbf{k} = (0, \mathbf{0}) \tag{2.57}$$

- conjugate quaternion q^* :

$$q^* = q_0 - (q_1 \mathbf{i} + q_2 \mathbf{j} + q_3 \mathbf{k}) = (q_r, -\mathbf{q}_v) \tag{2.58}$$

which satisfies the property $(q^*)^* = q$.

- quaternion norm $\|q\|$ is defined as:

$$\|q\| = qq^* = q^*q = \sum_{i=0}^3 q_i^2 = q_0^2 + \mathbf{q}_v^T \mathbf{q}_v \tag{2.59}$$

Given two generic quaternions $q = q_0 \mathbf{1} + q_1 \mathbf{i} + q_2 \mathbf{j} + q_3 \mathbf{k} = (q_r, \mathbf{q}_v)$ and $p = p_0 \mathbf{1} + p_1 \mathbf{i} + p_2 \mathbf{j} + p_3 \mathbf{k} = (p_r, \mathbf{p}_v)$ the following operations are defined:

- sum:

$$q + p = (q_0 + p_0, q_1 + p_1, q_2 + p_2, q_3 + p_3) \tag{2.60}$$

- difference:

$$q - p = (q_0 - p_0, q_1 - p_1, q_2 - p_2, q_3 - p_3) \quad (2.61)$$

- product:

$$qp = (q_r p_r - \mathbf{q}_v \cdot \mathbf{p}_v, q_r \mathbf{p}_v + p_r \mathbf{q}_v + \mathbf{q}_v \times \mathbf{p}_v) \quad (2.62)$$

where:

$$\mathbf{q}_v \cdot \mathbf{p}_v = \sum_i q_{vi} p_{vi} = \mathbf{q}_v^T \mathbf{p}_v = \mathbf{q}_v \mathbf{p}_v^T$$

and

$$\mathbf{q}_v \times \mathbf{p}_v = \begin{bmatrix} q_2 p_3 - p_3 q_2 \\ q_3 p_1 - p_1 q_3 \\ q_1 p_2 - p_2 q_1 \end{bmatrix}$$

The quaternion product has the following properties:

- anti-commutative:

$$qp \neq pq \quad (2.63)$$

- associative, with $q, p, l \in \mathbb{H}$:

$$(qp)l = q(pl) \quad (2.64)$$

- multiplication by unit scalar:

$$1q = q1 = (1, \mathbf{0})(q_r, \mathbf{q}_v) = (1q_r, 1\mathbf{q}_v) = (q_r, \mathbf{q}_v) \quad (2.65)$$

- multiplication by $\lambda \in \mathbb{R}$:

$$\lambda q = q\lambda = (\lambda, \mathbf{0})(q_r, \mathbf{q}_v) = (\lambda q_r, \lambda \mathbf{q}_v) \quad (2.66)$$

- bilinear, with $\lambda_1, \lambda_2 \in \mathbb{R}$:

$$\begin{aligned} q(\lambda_1 p_1 + \lambda_2 p_2) &= \lambda_1 qp_1 + \lambda_2 qp_2 \\ (\lambda_1 q_1 + \lambda_2 q_2)p &= \lambda_1 q_1 p + \lambda_2 q_2 p \end{aligned} \quad (2.67)$$

- inverse:

$$q^{-1} = \frac{q^*}{\|q\|^2} \quad (2.68)$$

Unit quaternion and rotations

After the necessary introduction on the quaternion and his maths, the main question is:

How can this be related with rotations? How can we, through quaternions, interpolate a set of key-frames like in Figure 2.21?

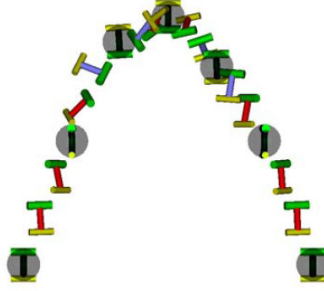


Figure 2.21. Set of orientations interpolated with unit quaternions [18]

To answer this question is necessary to introduce the *unit quaternion* [14]:

Definition 1 (Unit quaternion) Let $q \in \mathbb{H}$. If $\|q\| = 1$, then q is called a *unit quaternion*. We will use \mathbb{H}_1 to denote the set of unit quaternions [13].

The set of unit quaternions \mathbb{H}_1 forms a unit-sphere in four-dimensional space and plays a main role in rotations in 3D space. We assume a general quaternion $u \in \mathbb{H}_1$ as a sum of two trigonometric functions:

$$u = \cos(\theta) + \mathbf{u} \sin(\theta) \quad (2.69)$$

where \mathbf{u} is a vector with unit norm and θ a generic angle. It can be shown that the unit quaternion u represents the rotation of an angle 2θ about a unit vector $\mathbf{u} = [u_1, u_2, u_3]^T$. That's means:

$$u = \left(\cos \frac{\theta}{2}, u_1 \sin \frac{\theta}{2}, u_2 \sin \frac{\theta}{2}, u_3 \sin \frac{\theta}{2} \right) = \left(\cos \frac{\theta}{2}, \mathbf{u} \sin \frac{\theta}{2} \right) = \mathbf{R}(\mathbf{u}, \theta) \quad (2.70)$$

where \mathbf{R} is a rotation matrix.

So, we can associate to any rotation matrix a unit quaternion and vice-versa. Given a quaternion $u = (u_0, \mathbf{u})$, the equivalent rotation matrix $\mathbf{R}(u)$ is given by:

$$\mathbf{R}(u) = \begin{bmatrix} u_0^2 + u_1^2 - u_2^2 - u_3^2 & 2(u_1 u_2 - u_3 u_0) & 2(u_1 u_3 + u_2 u_0) \\ 2(u_1 u_2 + u_3 u_0) & u_0^2 - u_1^2 + u_2^2 - u_3^2 & 2(u_2 u_3 - u_1 u_0) \\ 2(u_1 u_3 - u_2 u_0) & 2(u_2 u_3 + u_1 u_0) & u_0^2 - u_1^2 - u_2^2 + u_3^2 \end{bmatrix} \quad (2.71)$$

Conversely, given a rotation matrix $\mathbf{R}(u)$ the unit quaternion $u = (u_0, \mathbf{u})$ is given by:

$$\begin{aligned} u_0 &= \pm \frac{1}{2} \sqrt{1 + r_{11} + r_{22} + r_{33}} \\ u_1 &= \frac{1}{4u_0} (r_{32} - r_{23}) \\ u_2 &= \frac{1}{4u_0} (r_{13} - r_{31}) \\ u_3 &= \frac{1}{4u_0} (r_{21} - r_{12}) \end{aligned} \tag{2.72}$$

Then, also operations between rotation matrices can be defined for unit quaternions:

- Rotation product:

$$\mathbf{R}(u) = \mathbf{R}(u_1)\mathbf{R}(u_2)\dots\mathbf{R}(u_n) \quad \Leftrightarrow \quad u = u_1u_2\dots u_n \tag{2.73}$$

- Transpose matrix:

$$\begin{aligned} \mathbf{R} &\Leftrightarrow u \\ \mathbf{R}^T &\Leftrightarrow u^* \end{aligned} \tag{2.74}$$

- Vector rotation: Given a generic vector \mathbf{x} , and the pure quaternion equivalent to it: $q_x = (0, \mathbf{x}) = (0, x_1, x_2, x_3)$, and, given a rotation matrix $\mathbf{R}(u)$ equivalent to the quaternion u , the rotated vector $\mathbf{y} = \mathbf{R}(u)\mathbf{x}$ is given by:

$$q_y = (0, \mathbf{y}) = uq_xu^* \tag{2.75}$$

where $\|q_x\| = \|q_y\|$.

At this point the utility of quaternions in describing rotations is evident. With respect to the previous techniques, especially Euler angles, its main advantage is the robustness. In fact, in this case the problem of singularities disappears and, once understood the not easy maths of this object, it is possible to plan the trajectory through a set of orientation with the planning of its four components q_0, q_1, q_2, q_3 .

The blending of a set of orientations described by quaternions can be made by interpolating the set of quaternions using the techniques detailed in Chapter 2.4. In this case, being rotation expressed by the unit quaternion, it is important to verify that the norm of the set of quaternions is kept constantly equal to one. Although more robust than Euler angles the planning can be quite more complicated to treat with respect to the last one, due to complex maths properties of this object.

Chapter 3

Off-line trajectory planning for continuous processes

This chapter illustrates the main contribution of this thesis. First, the case of study will be introduced in details, then the trajectory planner implemented by L.Biagiotti and C.Melchiorri in [12] will be explained and a personal implementation of it on Matlab will be presented. Then, another chapter is dedicated to the optimization of the data in input to the trajectory planner, in order to guarantee a fair trade-off between accuracy and velocity on the path, both for the geometric path and for the set of orientations. Finally, some tests on Matlab are carried out and the results are shown.

3.1 Introduction

My work of thesis is mainly based on the approach developed by Luigi Biagiotti and Claudio Melchiorri in [2]. An analysis of this kind of implementation has already been carried out in [19], and a trajectory planner was obtained. My contribute to this work is to try to optimize the trajectory planner in order to obtain a choosable trade-off between accuracy on the path and a constant scalar velocity of the manipulator. In addition, I have extended the implementation to the orientation planning of the robot's end effector, analysing the results.

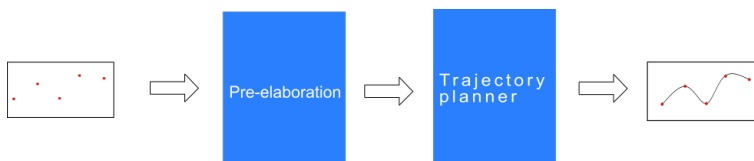


Figure 3.1. Overall structure of the trajectory planner

3.1.1 Case of study

As a case of study an operation of sealing of a hood has been taken in consideration. The entry points are in Figure 3.2, while in Figure 3.3 a clear distinction has been made between technologic and re-orientation path. This distinction is fundamental to point out the areas where a compromise on the accuracy can be assumed in order to have a constant scalar velocity of the manipulator on the path and to 'distribute' the variation of the tool's orientation. For what concerns the rest of the geometric path, with respect to the original points it is assumed acceptable an error within ± 5 mm. The complete structure of the trajectory planner, shown in Figure 3.1, will be roughly composed of two main blocks:

- A first block for the elaboration of the input points. Two kinds of elaboration will be performed on them, a re-sampling and a re-shaping made by using Nurbs
- A second block which, given the points in output from the first block, will build the trajectory according to [2].

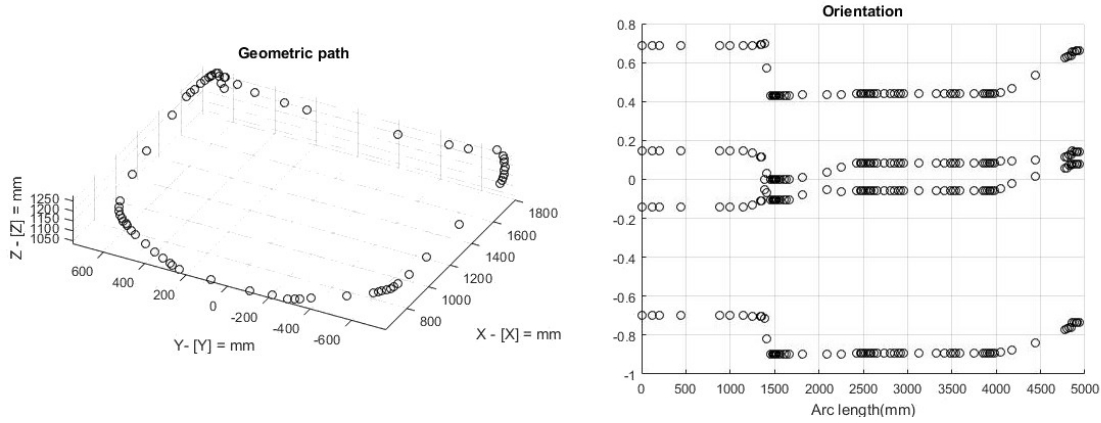


Figure 3.2. Entry points for the building of the hood

3.2 Trajectory planning based on FIR filters

In this section a necessary resume of the work in [2] will be made, before my implementation and upgrades of it. The work is divided into three parts:

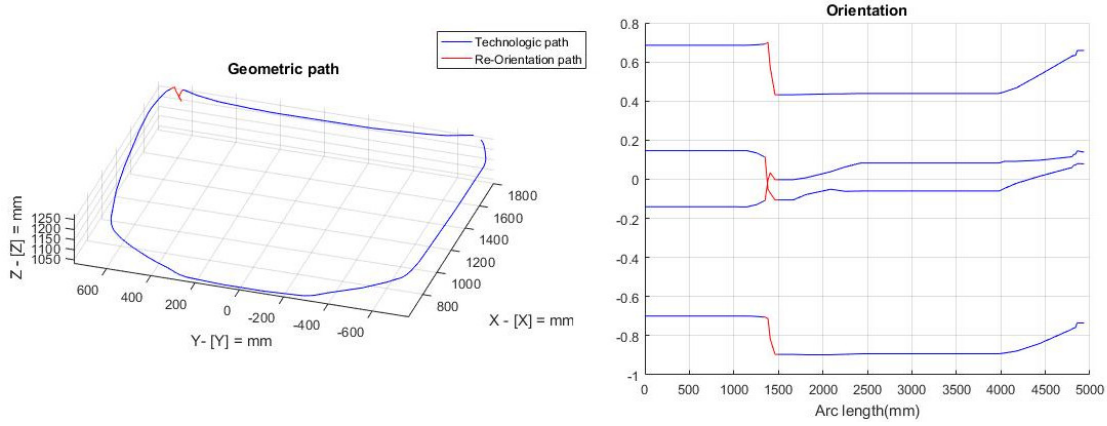


Figure 3.3. Entry points of the hood with differentiation between technologic and re-orientation path

1. Given the via-points from the user, an algorithm to obtain the related control points is built
2. Building of a piecewise constant function from the control points
3. Sending this function into a cascade of moving average filters

This sequence of operations gives life to the structure in Figure 3.4. In the following these procedures, as thought by Biagiotti and Melchiorri, will be explained and then a personal implementation of this algorithm is presented and discussed.

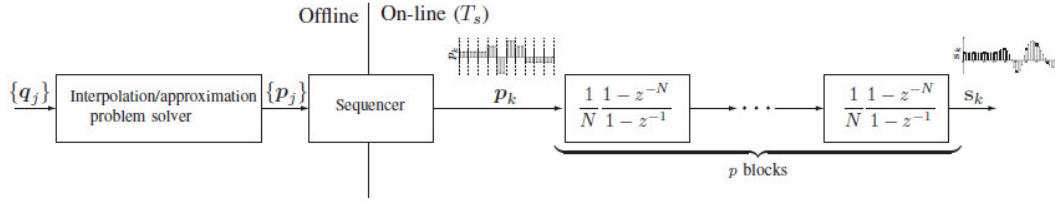


Figure 3.4. Overall structure for the generation of B-splines trajectories [2]

3.2.1 Uniform B-splines

Analytical B-splines

As detailed in Chapter 2.3, a p -th degree B-spline curve is defined by:

$$s(t) = \sum_{j=0}^n p_j B_j^p(t), \quad t_{min} \leq t \leq t_{max} \quad (3.1)$$

where $[t_{min}, t_{max}] \in \mathbb{R}$ is the parametric vector where the curve is defined and p_j are control points, which determine the shape of it. Taking the knot vector $\mathbf{t} = [t_0, \dots, t_{m-1}]$ (discussed in Chapter 2.3.2), with $t_j \leq t_{j+1}$, the B-spline basis functions of degree p is defined by:

$$B_j^p(t) = \frac{t - t_j}{t_{j+p} - t_j} B_j^{p-1}(t) + \frac{t_{j+p+1} - t}{t_{j+p+1} - t_{j+1}} B_{j+1}^{p-1}(t) \quad (3.2)$$

where:

$$B_j^0(t) = \begin{cases} 1, & \text{if } t_j \leq t \leq t_{j+1} \\ 0, & \text{otherwise} \end{cases} \quad (3.3)$$

An interesting case of B-splines is the *Uniform B-spline*. In this case the B-spline is defined over an *equally spaced* knot vector. So, we have that:

$$t_{j+1} - t_j = T, \quad j = 0, \dots, m-2 \quad (3.4)$$

It is then possible to rewrite the $(j+1)$ -th basis function in terms of the first basis function B_0^p through a simple shift operation:

$$B_j^p(t) = B_0^p(t - jT), \quad j = 0, \dots, m-1 \quad (3.5)$$

and the B-spline in (3.1) can be rewritten as:

$$s(t) = \sum_{j=0}^n p_j B_0^p(t - jT), \quad 0 \leq t \leq (m-1)T \quad (3.6)$$

Moreover, rewriting B_0^p as B^p for brevity, (3.2) can be now expressed as:

$$B^p(t) = \frac{1}{T} B^{p-1} * B^0 = \underbrace{\frac{1}{T} B^0 * \frac{1}{T} B^0 * \dots * \frac{1}{T} B^0}_{p \text{ times}} * B^0 \quad (3.7)$$

with

$$B^0(t) = \begin{cases} 1, & \text{if } 0 \leq t \leq T \\ 0, & \text{otherwise} \end{cases} \quad (3.8)$$

Therefore, by applying the Laplace transform to the uniform B-spline equation in (3.6) and replacing in (3.7), we have the following equation:

$$S_u(s) = \sum_{j=0}^n \mathcal{L} \left\{ p_j B^0 * \frac{1}{T} B^0 * \frac{1}{T} B^0 * \dots * \frac{1}{T} B^0 \right\} e^{-jsT} \quad (3.9)$$

From (3.9) we can notice that $\mathcal{L} \left\{ \frac{1}{T} B^0 \right\}$ performs the mean of the input function over an interval of width T , so we can rewrite it as:

$$M(s) = \mathcal{L} \left\{ \frac{1}{T} B^0 \right\} = \frac{1}{T} \frac{1 - e^{-sT}}{s} \quad (3.10)$$

This equation suggests that a uniform B-spline can be rewritten as a constant piecewise function, obtained from the control points p_j :

$$p(t) = \sum_{j=0}^n p_j B^0(t - jT) \quad (3.11)$$

and sending it in a cascade of p (as the desired degree of the curve) moving average filters $M(s)$. In case of a curve of degree $p = 3$ we obtain $S_u(s)$ of the following form:

$$S_u(s) = p(t) \cdot M(s) \cdot M(s) \cdot M(s) \quad (3.12)$$

It is important to notice that, if the B-spline is multidimensional the control points are multi-dimensional as well. So, it is necessary to iterate this procedure for each component of the B-spline. In our case it is necessary to add four channels more to take into account the additional B-splines that represent the planning of the orientation. So, on the whole procedure there will be seven different chains of filters.

Discrete B-splines

The first step to discretize a B-spline is to discretize its basis functions. Setting a sampling time T_s , the value of a B-spline basis function at the discrete time instant $t = KT_s$ is equal to:

$$\begin{aligned} B^p(t) &= \frac{1}{T} B^{p-1}(t) * B^0(t) \\ &= \frac{1}{T} \int_{-\infty}^{\infty} B^{p-1}(t - \tau) B^0(\tau) d\tau \\ &\Downarrow \\ B^p(kT_s) &\approx \frac{1}{T} \sum_{-\infty}^{\infty} B^{p-1}(kT_s - nT_s) B^0(nT_s) T_s \\ &= \frac{1}{N} \sum_{-\infty}^{\infty} B_{k-n}^{p-1} B_n^0 = \frac{1}{N} B_k^{p-1} * B_k^0 \end{aligned} \quad (3.13)$$

Writing $B_k^p = B^p(kT_s)$, from (3.13) it follows that:

$$B_k^p = \underbrace{\frac{1}{N}B_k^0 * \frac{1}{N}B_k^0 * \dots * \frac{1}{N}B_k^0}_{p \text{ times}} * B_k^0 \quad (3.14)$$

where $N = \frac{T}{T_s}$ is the number of samples present in each span of the knot vector, and

$$B_k^0 = \begin{cases} 1, & \text{if } 0 \leq k \leq N-1 \\ 0, & \text{otherwise} \end{cases} \quad (3.15)$$

Once found the basis function, through some manipulation the curve in (3.12) can be expressed in the discrete time domain as follows:

$$\begin{aligned} S(z) &= \sum_{j=0}^n \mathcal{Z} \{ p_j B_k^0 \} z^{-jN} \cdot M(z) \cdot M(z) \cdot \dots \cdot M(z) \\ &= \mathcal{Z} \left\{ \sum_{j=0}^n p_j B_{k-jN}^0 \right\} \cdot M(z) \cdot M(z) \cdot \dots \cdot M(z) \end{aligned} \quad (3.16)$$

with piecewise constant function:

$$p_k = \sum_{j=0}^n p_j B_{k-jN}^0 \quad (3.17)$$

and $M(z)$, which is a mean filter, defined as:

$$\begin{aligned} M(z) &= \frac{1}{N} \frac{1 - z^{-N}}{1 - z^{-1}} \\ &= \frac{1}{N} (1 + z^{-1} + z^{-2} + \dots + z^{-(N-1)}) \end{aligned} \quad (3.18)$$

It is important to notice that the discrete formulation only gives an *approximation* of the analytical B-splines, so there is not an exact interpolation of the entry points. Even though, it is possible to demonstrate that the staircase function obtained from this formulation tends to the analytical one in a sense of the root mean square if $T_s \rightarrow 0$.

3.2.2 Interpolation problem solver and computation of the control points

To find the control points of the curve the cubic spline interpolation algorithm discussed in Chapter 2.5 will be exploited, with some modification in order to adapt

it to the structure of the trajectory planner. As a quick recap, given a set of $l + 1$ entry points $\{q_0, q_1, \dots, q_{l-1}, q_l\}$ to impose the interpolation of the curve for those points at certain time instants t_i , it is necessary to set the system of equations:

$$s(t_i) = q_i, \quad i = 0, \dots, l \quad (3.19)$$

To find the control points the first step is to decide the degree of the curve. In the implementation the degree will be $p = 3$ since it ensures a C^2 continuity of the curve (see Chapter 2.3.1 for the definition of *continuity*). At this point the relation between p and t_i generates the knot vector according to the following rules:

- if p is odd $\Rightarrow t_i = iT$
- if p is even $\Rightarrow t_i = \frac{2i+1}{2}T$

In our case it will be $t_i = iT$.

Then, it is possible to build up the system. In order to have a system of equations well conditioned there is the need to consider symmetrical B-splines $s_s(t)$, which are uniform B-splines with basis functions symmetric with respect to the origin. To obtain the symmetric basis function $\beta^p(t)$ from the uniform basis function, a delay can be simply introduced, obtaining $\beta^p(t) = B^p(t + \frac{p+1}{2}T)$, and so the relation between the uniform and symmetrical B-splines can be naturally obtained as:

$$\begin{aligned} s_s(t) &= \sum_{j=0}^n p_j \beta^p(t - jT) \\ &= \sum_{j=0}^n p_j B^p(t + \frac{p+1}{2}T - jT) = s_u(t + \frac{p+1}{2}T) \end{aligned} \quad (3.20)$$

Interpolating the symmetrical B-splines with the points we have:

$$s_s(t_i) = \sum_{j=0}^n p_j B^p(t - jT + \frac{p+1}{2}T) = q_i \quad (3.21)$$

where the unknowns to be interpolated are the control points p_j . Since all the filters have unitary static gain, the output will reach in a stable way the first and the last via-points if and only if the same value is applied pT seconds before. That means that, in order to smoothly start and to end the curve, the first p control points must be equal to q_0 , as well as the last p control points must be equal to q_l . This will cause an unavoidable delay on the curve of length $\frac{p+1}{2}T$. Once made this, it is possible to compute the others $l - 1$ control points by solving the system of equations in (3.21).

3.2.3 Implementation of the trajectory planner

Cubic B-splines interpolator

The first step to build the interpolator is to choose the degree of the curve. The p degree of the curve will be $p = 3$ in order to obtain the continuity of velocity and acceleration, necessary in order to avoid undesired vibration on the robot moving on the path. From [2], substituting the values from the Tables 3.1 and 3.2 in the equation (3.21), the following system is obtained:

$$s_s(T) = \frac{1}{6}p_{i-1} + \frac{4}{6}p_i + \frac{1}{6}p_{i+1} = q_i, \quad i = 1, \dots, l-1 \quad (3.22)$$

Adding the conditions on the first and last points it can be rewritten in the matrix form:

$$\begin{bmatrix} 4 & 1 & 0 & \dots & \dots & 0 \\ 1 & 4 & 1 & 0 & & 0 \\ \vdots & & \ddots & & & \vdots \\ \vdots & & 1 & 4 & 1 & 0 \\ & & & 1 & 4 & 1 \\ 0 & \dots & \dots & 0 & 1 & 4 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_{l-3} \\ p_{l-2} \\ p_{l-1} \end{bmatrix} = \begin{bmatrix} 6q_1 - q_0 \\ 6q_2 \\ 6q_3 \\ \vdots \\ 6q_{l-3} \\ 6q_{l-2} \\ 6q_{l-1} - q_l \end{bmatrix} \quad (3.23)$$

\Downarrow

$$\mathbf{Ax} = \mathbf{B} \quad \longrightarrow \quad \mathbf{x} = \mathbf{A}^{-1}\mathbf{B}$$

Due to the tridiagonal structure of the matrix, the control points can be found by a simple inversion of the tridiagonal matrix, as detailed in Chapter 2.4.5. A Matlab implementation of the interpolator is available in Appendix A.4

	0	T	$2T$	$3T$	$4T$	$5T$	$6T$
$p = 1$	0	1	0				
$p = 3$	0	$\frac{1}{6}$	$\frac{4}{6}$	$\frac{4}{6}$	0		
$p = 5$	0	$\frac{1}{120}$	$\frac{26}{120}$	$\frac{66}{120}$	$\frac{26}{120}$	$\frac{1}{120}$	0

Table 3.1. Matrix coefficients for odd p

	$\frac{1}{2}T$	$\frac{1}{2}T$	$\frac{3}{2}T$	$\frac{5}{2}T$	$\frac{7}{2}T$	$\frac{9}{2}T$	$\frac{11}{2}T$
$p = 2$	$\frac{1}{8}$	$\frac{6}{8}$	$\frac{1}{8}$				
$p = 4$	$\frac{1}{384}$	$\frac{76}{384}$	$\frac{230}{384}$	$\frac{76}{384}$	$\frac{1}{384}$	0	

Table 3.2. Matrix coefficients for even p

Sequencer

```
function pk = sequencer(P,N,n)

% Take the control points in input and transforms them
% in a piecewise constant function 'pk'

pk = zeros(1,n*N);
for i = 1:n
    for k = 1:N
        pk(1,(i-1)*N+k) = P(i);
    end
end
end
```

Listing 3.1. Sequencer algorithm

In 3.1 a Matlab implementation of the second part of the structure, the sequencer. It simply takes as input the control points and keeps them constant for N samples. Taking into account the sampling time of the robot's controller, $T_s = 0.002$ second, the number of samples between two points is given by:

$$N = \text{round}\left(\frac{T}{T_s}\right) \quad (3.24)$$

where $\text{round}()$ [20] is a Matlab function which approximates the result to the nearest positive integer and T is the desired time's distance between two points.

Cascade of FIR filters

As a last point, the cascade of filters like the one in Figure 3.6 has to be implemented. To build the transfer function of the filter in (3.18) the Matlab function 'filter' [20] has been used, which takes as inputs the numerator and denominator of the desired

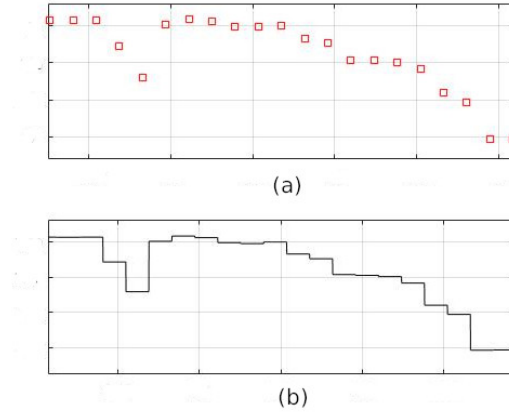


Figure 3.5. An example of piecewise constant function in (b). In (a) the control points from which the function is derived

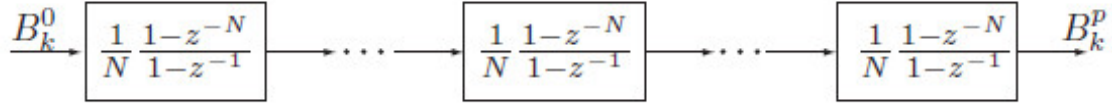


Figure 3.6. A cascade of p moving average filters [2]

filter and gives the transfer function as output. Then, the control points have been sent p times into this filter through a 'for' cycle. After another elaboration, made to delete the transitory from the output of the filters, which code is shown in appendix A.7, the result coming out from the curve in (3.5) is the one reported in Figure 3.7.

It is relevant to notice the delay introduced by the sequence of filters. As said in chapter 4.2, the curve has been delayed with respect to the piecewise constant function of $\frac{p+1}{2}T = 2T$.

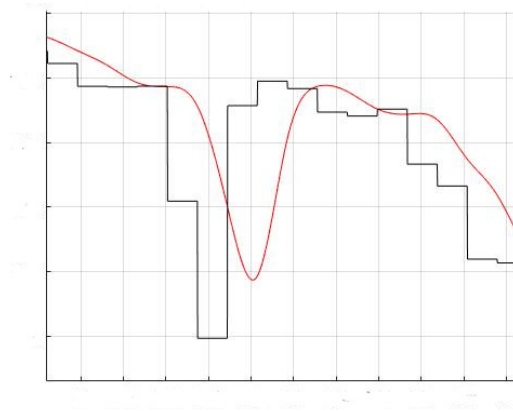


Figure 3.7. B-spline coming out from the filtered piecewise constant function

```
function spline = FIR(Numb_samples, piecewise_func...  
    InitStates, degree)  
  
% In this function a cascade of moving average filters  
% has been implemented. The number of filters is  
% chosen by the input parameter 'degree'  
  
% Building of the filter  
b = (1/(Numb_samples))*ones(1, Numb_samples);  
a = 1;  
initStates = initStates*ones(1, Numb_samples-1);  
  
% Implementation of the cascade of filters  
spline(1,:) = filter(b, a, piecewise_func, initStates);  
for k=2:degree  
    spline(k,:) = filter(b, a, spline(k-1,:), initStates);  
end  
spline = spline(3,:);
```

Listing 3.2. Cascade of filters' implementation

3.3 Pre-elaboration of data and Optimization

3.3.1 Introduction

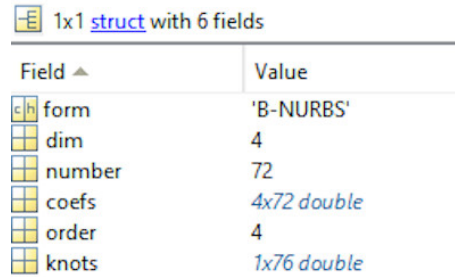
In order to pursue the constant velocity of the robot a first step is to manipulate the points by using Nurbs. A brief description of Nurbs is given below, while a more detailed characterization is given in chapter 2.1.5. A p -th degree Nurbs curve is defined as follows:

$$\mathbf{C}(u) = \frac{\sum_{i=0}^n N_{i,p}(u)w_i\mathbf{P}_i}{\sum_{i=0}^n N_{i,p}(u)w_i}, \quad u_0 \leq u \leq u_n \quad (3.25)$$

As explained in Chapter 1, tuning the weights of the Nurbs it is possible to take a point and bring it closer or push it away the associated control point. So, an algorithm able to automatically understand whether and where to move a certain point in order to flatten the curve if required has been implemented. The algorithm has been built so to have a local influence on the curve, so to avoid any modification of the curve when no action of reshaping is required, and to give to the user the possibility to decide how much heavy the modification should be. Two different algorithms have been implemented, in order to cope with the differences between trajectory planning on the geometric path and on the orientations. In the following the two algorithms are examined in details.

3.3.2 Nurbs toolbox

The *Nurbs Toolbox* by D.M.Spink has been used to build the nurbs. A nurbs is intended as a Matlab structure composed as depicted in Figure 3.8.



Field	Value
form	'B-NURBS'
dim	4
number	72
coefs	4x72 double
order	4
knots	1x76 double

Figure 3.8. Nurbs object by Nurbs Toolbox [20]

The functions mainly used from the toolbox were:

- *nrbmak* - It builds the nurbs structure starting from a knot vector and the control points
- *nrbeval* - Given a parametric vector $u \in [0,1]$, it evaluates the nurbs at each parametric point

The knot vector is defined by the piece of code in 3.3 as a uniform vector. The knot vector is not so useful for the final trajectory but in this phase, to use the toolbox, has to be implemented, so the choice has been a uniform configuration for sake of simplicity.

```
function u = DefineKnots(degree,n)

% Building of a uniform knot vector

nknots = n+2*degree;
C = nknots-2*(degree+1);
u = [ zeros(1,degree+1) (1:C)/(C+1) ones(1,degree+1) ]
```

Listing 3.3. Knot vector definition

Once the nurbs are built the field *crv.coefs* is of main interest for our purposes. That field is a four row-vectors field in which the first three rows contain the control points of the curve, while the last one indicates the weight associated to the related control point. To change the shape of the curve it is necessary to multiply the weight for the column indicating the desired point of the curve to be changed, as shown by the line of code in (3.4) .

```
crv.coefs = weights.*crv.coefs;
```

Listing 3.4. Weights modification

3.3.3 Elaboration of the geometric path with Nurbs

The process to compute the weights of the nurbs for the geometric path is made of several points:

1. The nurbs obtained with *nrbmak* were evaluated with *nrbeval* on a parametric vector made by $n*multi$ points, where n is the length of the buffer containing the input points ($n = 70$ in the case of the hood) and *multi* is a multiplier designed to improve the precision during the result of the successive analysis. The value of *multi* was set to 100 in the case of the hood, after having made several trials with different values.

2. The curvatures along the curve must be examined, in order to understand how much effort has to be put in the reshaping of each section. To do this, the function *curvature* by Are Mjaavatten [21] was used. This function takes as input the two or three coordinates (depending if the study is made in 2D or in 3D) and computes the radius of curvature for each point along the curve.

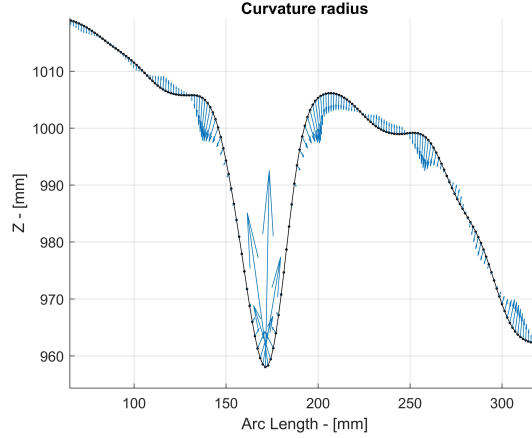


Figure 3.9. Radius of curvature of the re-orientation area of the hood along the axis Z, with *multi* = 10

To do this, the algorithm computes the circumscribed circle of the triangle formed by the three corners represented by the input point p_i and its neighbours p_{i-1} and p_{i+1} . Once obtained the radius R_i of this circle, the radius of curvature for each point will be obtained as:

$$c_i = \frac{1}{R_i} \quad (3.26)$$

An example of the radius of curvature is in Figure 3.9, where a part of the hood is examined and an arrow is associated to each point indicating the bending in that point.

3. Once computed the radii of curvatures, these are re-associated to the original points, selecting the maximum radius from each buffer of '*multi*' elements and associating it to the corresponding point from the original buffer, i.e., if in the original buffer there are $n = 70$ elements and *multi* = 100, a new buffer of $m = n \cdot \text{multi} = 7000$ elements is created. From this new buffer each array $\in [i \cdot \text{multi}, (i + 1) \cdot \text{multi}]$ with $i = 0, \dots, n$ corresponds to the i -th element in the old vector of points.

4. Then, a vector of n radii of curvature is obtained:

$$\mathbf{c} = [c_1, c_2, \dots, c_{n-1}, c_n] \quad (3.27)$$

and the vector of weights is finally computed through the simple algorithm listed in 3.5:

```
% Computation of the vector of weights 'w'

c = c / accuracy;
k = 1 / c;
for i = 1 : n
    if (c(i,1) < 1)
        w(i,1) = 1;
    else
        w(i,1) = k(i);
    end
end
```

Listing 3.5. Creation of the vector of weights

There a simple inversely proportional relation of the kind $x = \frac{1}{y}$ has been implemented between the vector of the curvatures and the weights' vector. A threshold of 1 has been set up on the values of the curvature radius in order to modify the weights. This has been decided in order to avoid to modify too much the curve globally and to restrain the weight in a range of values between 0 and 1, that is necessary to have an action of smoothing on the curve. A value greater than 1 on some control point would attract the point of the curve increasing the curvature. Finally, a variable named *accuracy* can be set. This is the parameter which can be modified by the user (in a range of values between 0.1 and 1) to set a manual threshold on the heaviness of the shaping on the curve.

3.3.4 Redistribution and oversampling of the input points over the path

It is important to remember that the use of nurbs is only made in order to take a new distribution of points more suitable to the purpose of the trade-off in the processing. Another elaboration is needed in order to make the cascade of filters works properly. In fact, the filter works with uniform B-splines and so, if the objective is a constant

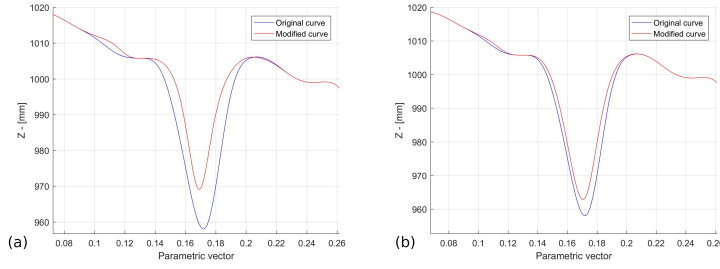


Figure 3.10. In (a) modification of the curve for *accuracy* = 0.1. In (b) *accuracy* is set to 0.3

scalar velocity of the manipulator, it is of crucial importance the distribution of the points along the path. The time duration between two consecutive points cannot change and must be equal to a certain time period T , so, being impossible to change the time, the only possible action is to redistribute the points so to have the same distance in terms of arc length between each couple of them. To fulfil this task, the function *interparc* by John D’Errico [22] was used.

```
new_points = interparc(parametric_vector, x, y, z, 'method');
```

Listing 3.6. Interparc function example

In the example in 3.6 it can be seen that *interparc* is a function which, given as input a parametric vector $\in [0, 1]$, the three buffers of points of the 3D curve and a method of interpolation, interpolates the given points at equal distances on the arc length, following the method of interpolation indicated as last argument. So, a redistribution of the initial $n = 70$ points along the path has been made, as shown in Figures 3.11 and 3.12. But this redistribution is not enough to obtain a constant velocity. So, a second redistribution is made, in which the original points are oversampled. The point of the oversampling is that *interparc* doesn’t give an exact equal space interpolation over the arc length, so, oversampling the points reduces the distance between them and consequently the error on the distance, guaranteeing a more constant velocity, especially along the re-orientation area, moreover, this operation introduces the second parameter which the user can manipulate in order to decide the trade-off between accuracy and velocity.

```
new_points = interparc(linspace(0,1,round(n_samp/velocity))...
    x, y, z, 'spline');
```

Listing 3.7. Resampling of the points

In 3.7 the parameter *velocity* is the user input which, reducing the number of new samples, whose ceil is indicated by n_samp , which is computed through the Matlab function in appendix A.3 , increases the velocity of the manipulator. Increasing the parameter *velocity*, the number of samples is decreased and, consequently, the total time of the processing is reduced. The new motion laws are then:

$$d = \frac{Arc\ Length}{\frac{n_samp}{velocity} \times N}$$

$$V_{tot} = \frac{d}{T_s}$$

$$T_{tot} = \frac{Arc\ Length}{V_{tot}}$$
(3.28)

where N is the number of samples already computed in (3.24), T_s is the sampling time of the robot's controller and V_{tot} is the total velocity over the arc length.

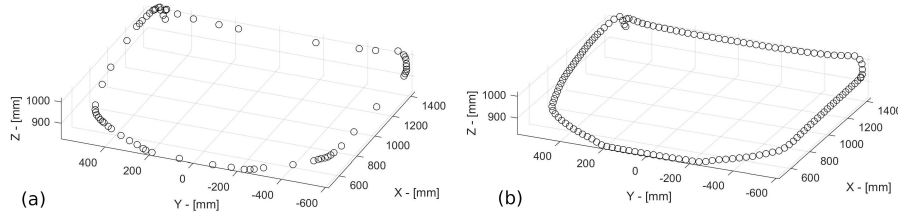


Figure 3.11. In (a) the original points, in (b) the points redistributed with *interparc*

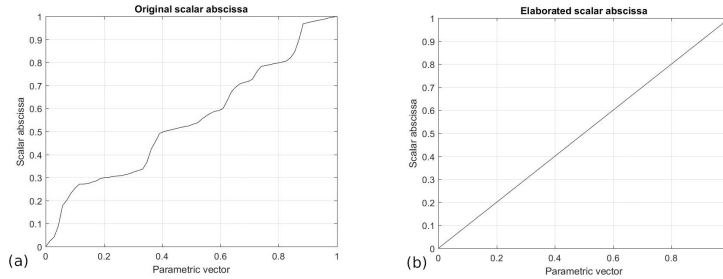


Figure 3.12. In (a) the original scalar abscissa over the parametric vector. In (b) the new scalar abscissa obtained redistributing the points

3.3.5 Elaboration of the set of orientations with Nurbs

In order to plan the trajectory for the set of orientations a quaternion approach was implemented.

So, four additional buffers are added to the three already mentioned in the previous chapter. The quaternions path on the hood, and in many of these kinds of processes, shows a pretty constant behaviour interrupted by an abrupt change of value in the area where the manipulator changes its orientation (Figure 3.2 and 3.3). So, something that smooths this change of orientation is needed. On a first sight, this problem would seem to be solvable through a low pass filter, but the solution is not so immediate since each quaternion framework is connected to the geometric path and must be coherent with it. Since a model designed to reach a constant scalar velocity of the robot was implemented, the frameworks of the manipulator must be consistent with this model. The procedure in Chapter 4.4.3 has shown some limitation when applied on the blending of orientations due to the problem described above, and then, a slightly different and more specific nurbs' algorithm has been made in order to deal with the complexity introduced by these kinds of trajectories.

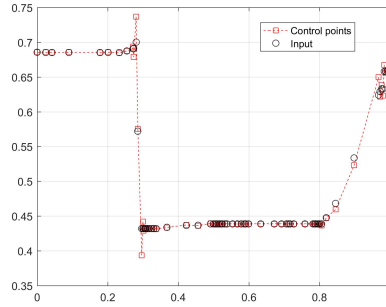


Figure 3.13. One of the four buffer for the orientation in the hood example

Looking at Figure 3.13, an appropriate and simple approach in order to obtain a smooth and less oscillatory behaviour is to compute the lines passing through two consecutive pairs of points and to assign the weight of the point on the base of the angle which the two lines form. So, the following steps are done:

1. A buffer of three consecutive control points $[P_{i-1}, P_i, P_{i+1}]$ is collected for each $i = 0, \dots, n$.
2. The vector p_1 passing trough P_{i-1} and P_i as well as the vector p_2 passing

through P_i and P_{i+1} and the angle between them are calculated through:

$$\theta = \arctan\left(\frac{p_1 \times p_2}{p_1 \cdot p_2}\right) \quad (3.29)$$

3. The angle is useful to set a threshold on the region of the modifications. Imposing a high θ , the region of modification is restrained to the zone where the change of orientation happens, being this zone characterized by high oscillations and, so, high curvatures. Also in this case a parameter of accuracy, which can be modified by the user, is inserted in order to make possible to decide how appreciable should be the reshaping.

Moreover, these orientation curves must be connected to the new curvilinear abscissa generated during the manipulation of the geometric path, in Figure 3.12. To do this, the function *interp1()* [20] is used.

```
new_quaternions = interp1(original_abscissa ,old_quaternions...
    new_abscissa);
```

Listing 3.8. Interpolation of the quaternions on the new abscissa

Given as arguments to the Matlab function in 3.8 the original and the new scalar abscissa, it interpolates the quaternions on the new one. The scalar abscissa and the arc length were computed through polygonal in the function *arc_length()*, shown in appendix A.2. In Figure 3.14 the result of the above procedure is shown. It can be seen that a discrete smoothing is achieved in the critical zone thanks to the action of the nurbs. .

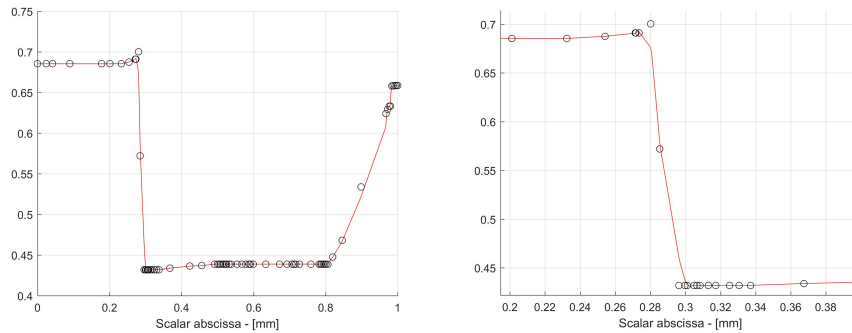


Figure 3.14. Interpolation and smoothing of the first set of quaternions with *accuracy* parameter equals to 0.4

3.4 Matlab simulations

In order to show the results of the implemented structure, some Matlab simulations have been carried out. In particular, two different tests have been performed, showing the changes in the trajectories through different values of the three user's parameters. The tests have been performed in the following way:

- Test 1 - In this simulation the values of the accuracy both on the position and orientation is set to the default value of '1' in order to show the performance of the robot without the nurbs algorithm
- Test 2 - A fair trade-off between velocity and accuracy is set and the result are showed

Table 3.3 reports the values of the parameters used over the three simulations, where F and T are the frequency and the period of the FIR cascade, which can be computed by (3.24).

N	F	T	T_s	n_{samp}
5	100Hz	0.01s	0.002s	3460

Table 3.3. Simulations' parameters

In the following tests the parameters of accuracy for position and orientation will be labelled respectively as ' Acc_{pos} ' and ' Acc_{or} ', while the parameter for velocity will be ' $Velocity$ '.

$$\begin{aligned}
 Acc_{pos} &\in [0.1, 1] \\
 Acc_{or} &\in [0.1, 1] \\
 Velocity &\in [1, 5]
 \end{aligned}
 \tag{3.30}$$

3.4.1 Test 1

<i>Velocity</i>	<i>Acc_{pos}</i>	<i>Acc_{or}</i>
1.2	1	1

Table 3.4.
Test 1 - User's parameters

In Table 3.4 the parameters used in this test are reported, while Table 3.5 and from Figures 3.15 - 3.19 report the results of the simulation.

Total time	28.86 <i>s</i>
Max error on the path	2.8305 <i>mm</i>
Cruise velocity	0.1366 <i>m/s</i>
Max scalar velocity	0.141 <i>m/s</i>
Max scalar acceleration	13.66 <i>m/s²</i>
Max jerk	1367 <i>m/s³</i>

Table 3.5.
Test 1 - Results

It can be noticed from Figure 3.17 the coherency of the robot in following the path in this case, in which no modifications on path and orientation have been made, with a maximum error with respect to the desired path of 2.8305 mm. Being '*velocity* = 1.2', the number of samples of the curve is still high and, consequently, the velocity is enough low to have an almost constant scalar velocity value. Nevertheless, a change in velocity happens in the re-orientation phase, as it can be noticed in Figure 3.16. Furthermore, Figure 3.15 shows a sudden change of the tool's orientation, which can be seen also in Figure 3.18, where the velocity of the tool is not well distributed but it is concentrated in the re-orientation area. It is immediate to think that, smoothing the change in orientation and the path geometry in the critical area, a

smoother movement of the robot would be obtained, which is the tentative made in the second test. Finally, in Figure 3.19 it is shown the error with respect to the unit quaternion to demonstrate that the quaternions remain unitary along all the path, that is a fundamental constraint for the quaternions to effectively represent a rotation, as already explained in Chapter 2.5.4.

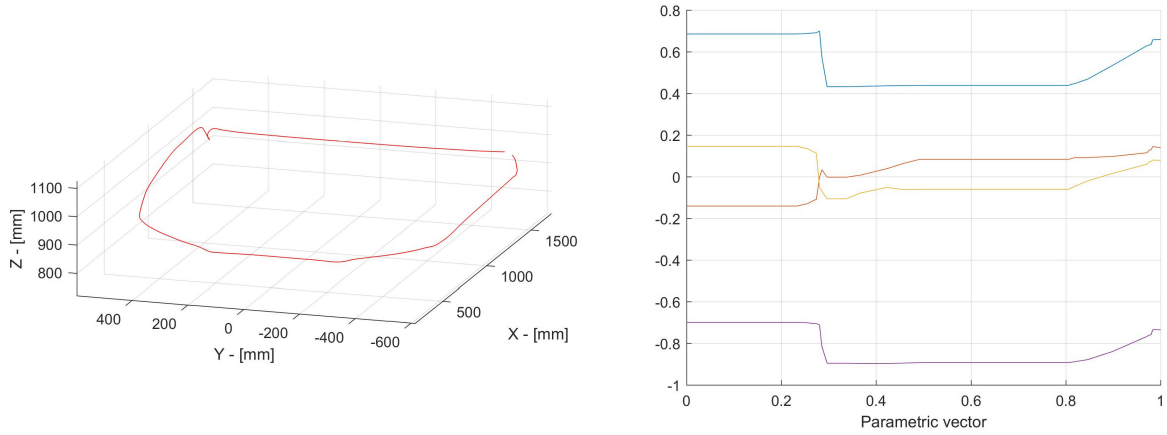


Figure 3.15. Test 1 - Geometric path and quaternions

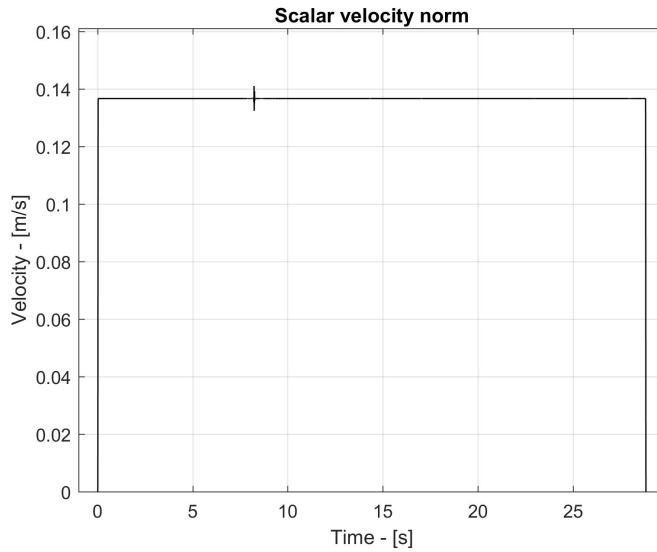


Figure 3.16. Test 1 - Scalar velocity

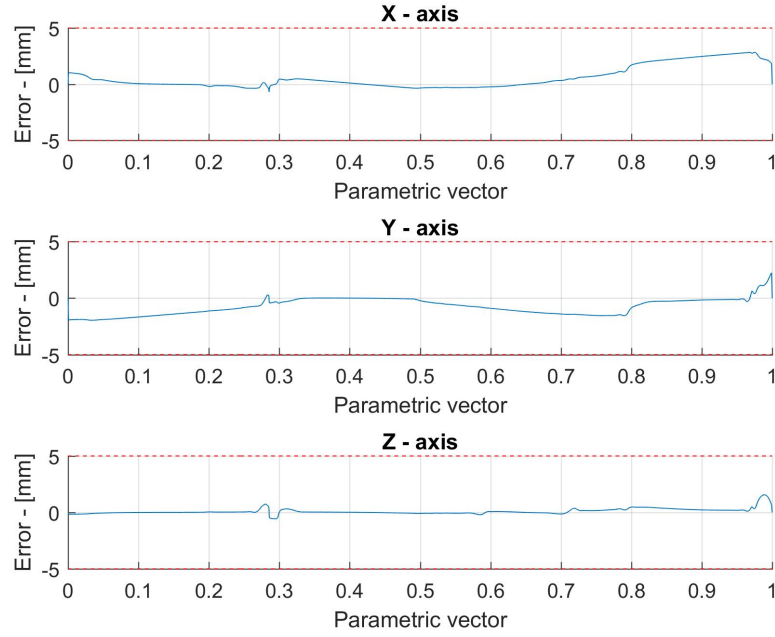


Figure 3.17. Test 1 - Error along the path

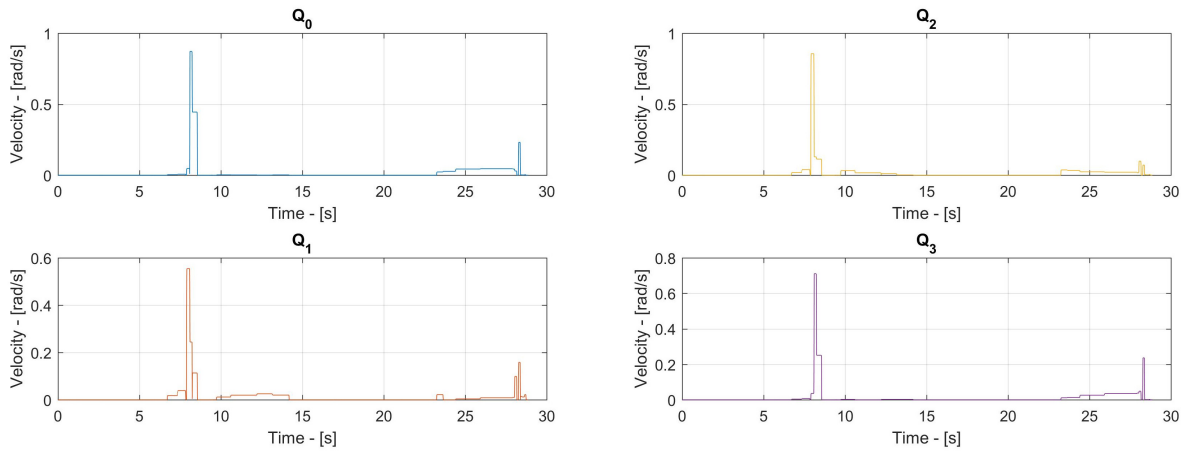


Figure 3.18. Test 1 - Analytical quaternion velocities

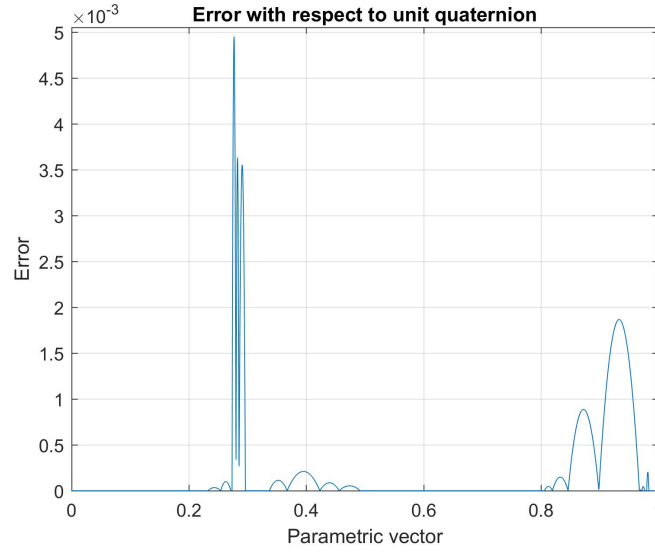


Figure 3.19. Test 1 - Error with respect to the unit quaternion

3.4.2 Test2

<i>Velocity</i>	<i>Acc_{pos}</i>	<i>Acc_{or}</i>
1.2	0.4	0.6

Table 3.6.
Test 2 - User's parameters

In Table 3.6 the parameters used in this test are reported, while in Table 3.7 and Figures 3.20 - 3.25 report the results of the simulation.

It can be noticed that, with this second choice of the parameters, an improvement has been reached for what concerns the scalar velocity, that, as shown in Figure 3.21, reaches a cruise value of 0.136 m/s with no remarkable changes along the path. This result is reached at the cost of a greater inaccuracy along the path, as shown in Figure 3.22, where it can be seen that the robot shows a greater error with respect to the previous test, with a peak of 10.34 mm. Regarding the orientation, Figure 3.24 shows the effect of the smoothing on the orientations interpolation. Unfortunately, the smoothing action is not so consistent as can be seen also in

Total time	28.86 s
Max error on the path	10.3337 mm
Cruise velocity	0.136 m/s
Max scalar velocity	0.136 m/s
Max scalar acceleration	13.6 m/s ²
Max jerk	1360 m/s ³

Table 3.7.
Test 2 - Results

Figure 3.23, where the analytical quaternion velocities are still pretty concentrated around the re-orientation are. This is due to the fact that the orientation is strongly connected to the trajectory and, since a trajectory that pursues a constant scalar velocity has been built, the cost to pay is a less smooth re-orientation of the tool. A solution could be to decrease the ' Acc_{pos} ' parameter, but we need to be careful to not decrease it too much, because it could compromise the unity of the quaternions. For what concerns the consistency of the rotation, in Figure 3.25 is shown that also in this case the error with respect to the unit quaternion is negligible.

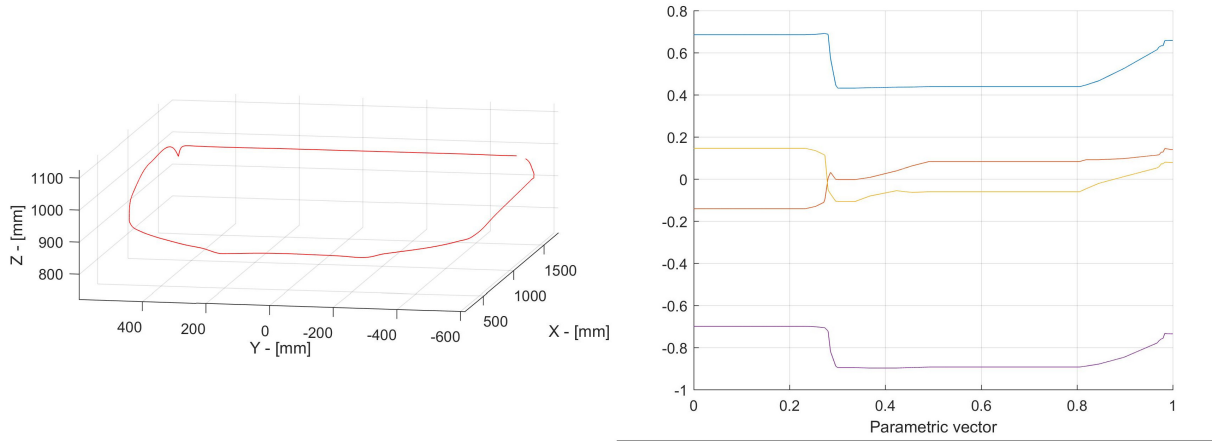


Figure 3.20. Test 2 - Geometric path and quaternions

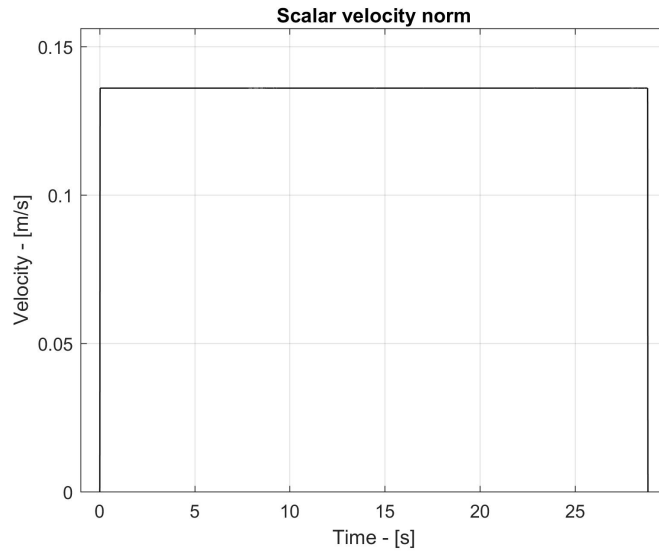


Figure 3.21. Test 2 - Scalar velocity

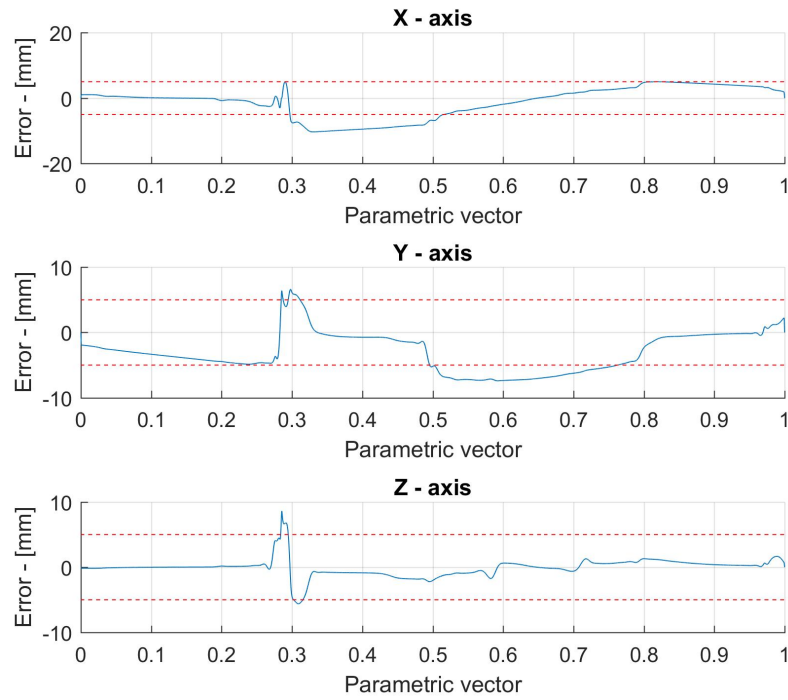


Figure 3.22. Test 2 - Error on the path

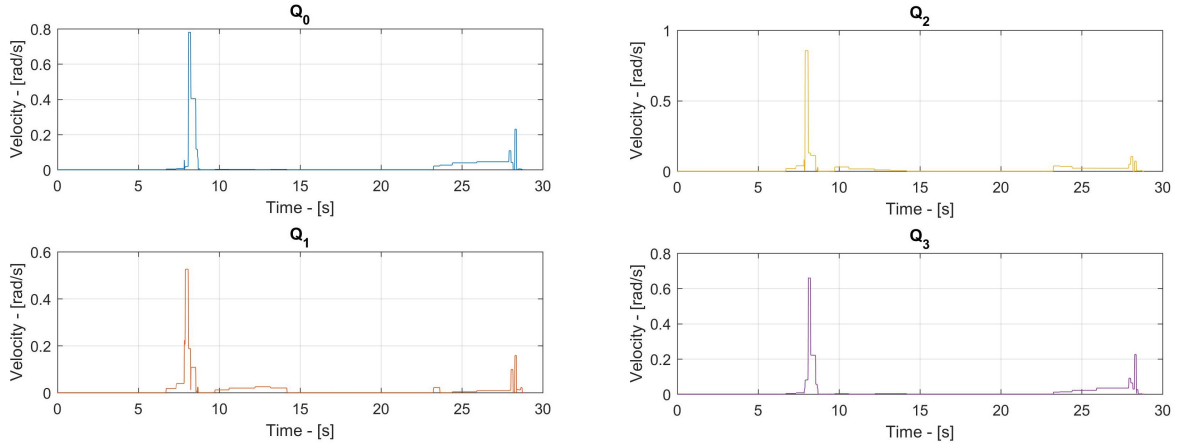


Figure 3.23. Test 2 - Analytical quaternion velocities

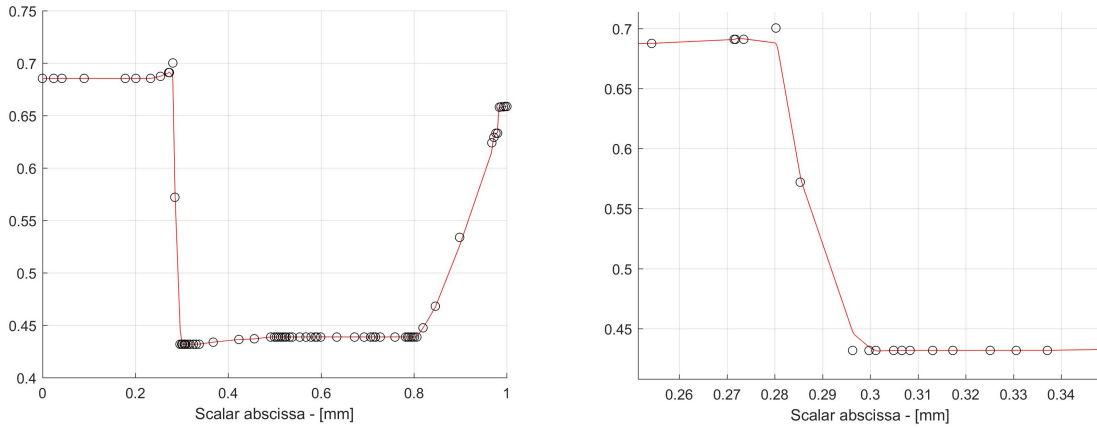


Figure 3.24. Test 2 - A detail of the quaternion Q_0 in the re-orientation area

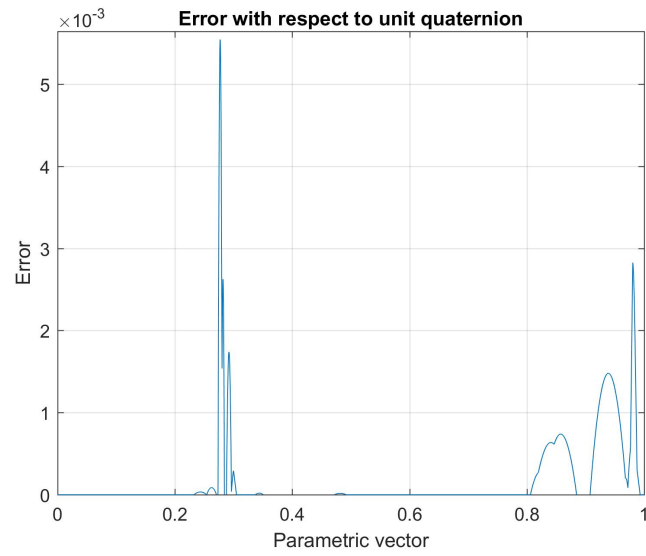


Figure 3.25. Test 2 - Error with respect to the unit quaternion

Chapter 4

Tests on Robot

In this chapter the results of the experimental tests carried out in Comau are shown. First, a brief description of the utilized robot is given, then the procedure to move the robot, both manually and automatically, is explained, and, finally, the results of the tests are shown and compared with the simulations in Chapter 3.

4.1 Description of the NJ130

For the purpose of the tests a NJ130-2.6, shown in Figure 4.1, has been used. This manipulator is a six axes anthropomorphic robot which is capable to carry up to 130 Kg of payload on the tool with the possibility of 50 Kg extra payload on the forearm. During the test no payload was applied on it. The controller, shown in Figure 4.2, is a C5G control unit by Comau, a modular controller which uses inside the industrial PC's APC820 with Core2 Duo technology CPU which is capable of obtaining high performances with low energetic consumption.



Figure 4.1. Model of NJ130-2.6 by Comau [3]

4.2 The teach pendant and the moni

In Figure 4.2 it is also shown the used teach pendant, which is the interface between the user and the controller of the robot. Through this it is possible to move the robot both manually and automatically. To move automatically the robot a binary file, called 'moni', must be created. In this file is contained the curve which the manipulator will follow, sampled at a rate of 2 ms between two consecutive points. To read the binary file it is also necessary to create a program which is capable to read the set of positions and velocities contained in the moni. This kind of file is the PDL2 program, written in a Comau proprietary language of which an example is shown in 4.1.



Figure 4.2. TP5 and control unit C5G [3]

```

PROGRAM Test_2
VAR p1 : POSITION
BEGIN
$RPL_DIR_PATH := 'UD:\\usr\\Prova_1'
$RPL_SPD_OVR := 100
$BASE      := POS(0)
$TOOL      := POS(0)
$UFRAME    := POS(0)
p1 := POS(0,500,0,0,-90,0, '' )

  MOVE TO {0, 0, -90, 0, 90, 0}
  delay 1000
CYCLE
MOVE TO p1
delay 1000
MOVE FROM p1 REPLAY 'moni_2.log'
delay 1000
END Test_2

```

Listing 4.1. Example of PDL script

4.3 Experimental tests

Several tests have been made on the robot. For all the tests the parameters in Table 3.6 are chosen, in order to compare the real performance to the simulation in Chapter 3, and all the manipulation on velocity have been made changing the frequency of the filters and, consequently the parameter N in (3.24). A first test has been made reducing the band frequency of the cascade of filters, in order to have a minor velocity of the robot and to analyse and foresee eventual criticisms at nominal frequency. Choosing a frequency of $F = 40$ Hz, and consequently, through (3.24) a number of samples between the points equals to $N = 13$, the result in terms of scalar velocity is the one shown in Figure 4.3. It can be noticed that the result is really different to what is obtained in simulation. This is due to the connection between path and orientation, that makes unavoidable an acceleration of the arm in the re-orientation phase, although this is reduced through the nurbs optimization.

So, increasing the velocity by setting the frequency at $F = 100$ Hz $\rightarrow N = 5$, the result in term of scalar velocity is the one in Figure 4.4, where it can be noticed that an increased velocity produces also an increased acceleration in the critical area of re-orientation. It can also be noticed a high angular acceleration in Figure 4.5, where the original acceleration is partially attenuated but the measured one remains quite high. For what concerns the precision on the path, Figures 4.6 - 4.9 show a good accuracy in following the target path, both for the geometric path, where the maximum error is 14.5 mm coherently with the simulation (where the maximum error was about 10 mm), and for the orientation, which apart for a negligible delay with respect to the target, shows a really good accuracy. In Figure 4.6 the curve apparently unrelated to the hood represents the approaching phase of the manipulator to the path, which has been performed upside down in the experimental test.

From the previous tests it is clear that a constant scalar velocity is impossible to reach in this case, due to the strong variations of the tool orientation. So, another kind of approach has been tested. Taking inspiration from [23] and tuning the frequency of the filters' cascade, two different velocities are obtained, a faster one along the technological path and a decelerated one during the re-orientation, in order to avoid sudden changes in the velocities and to have a smoother movement of the robot. Setting $F = 100$ Hz in the technological path and $F = 10$ Hz in the re-orientation area, a scalar velocity profile like the one in Figure 4.12 is obtained.

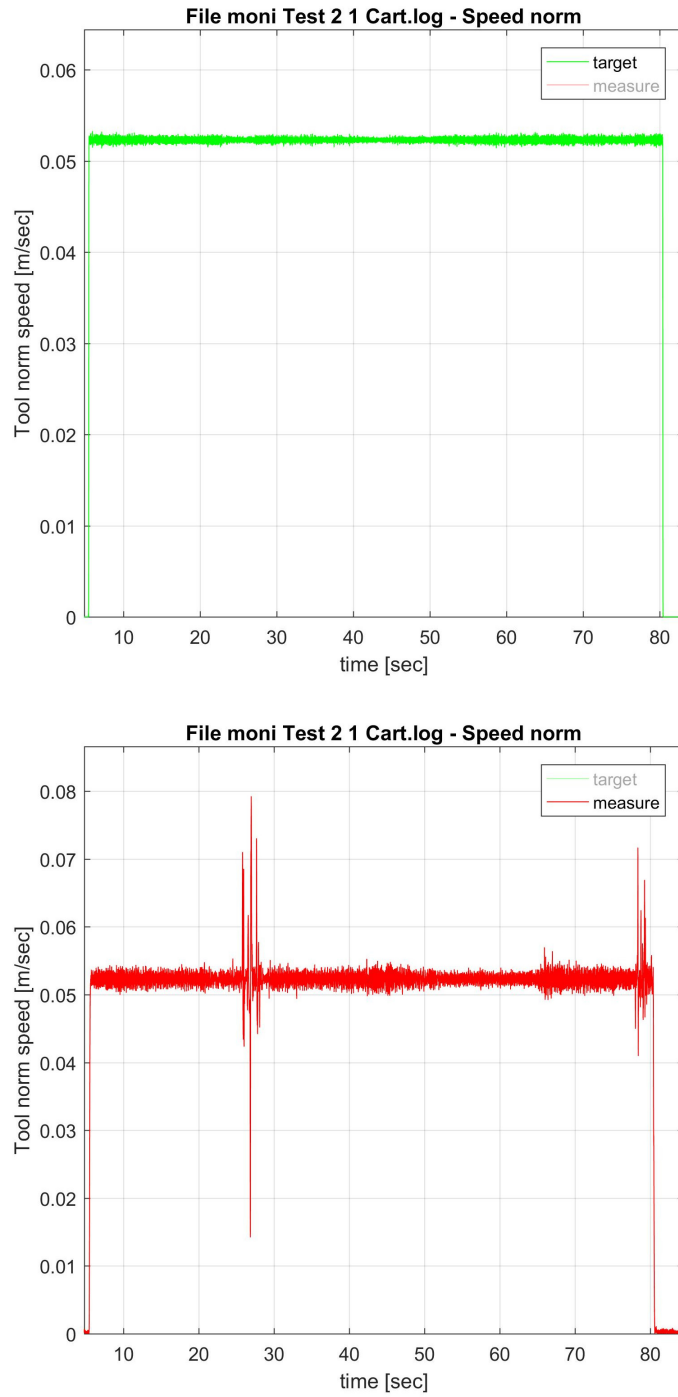


Figure 4.3. Scalar velocity of the robot during the first test, at $F = 40\text{Hz}$

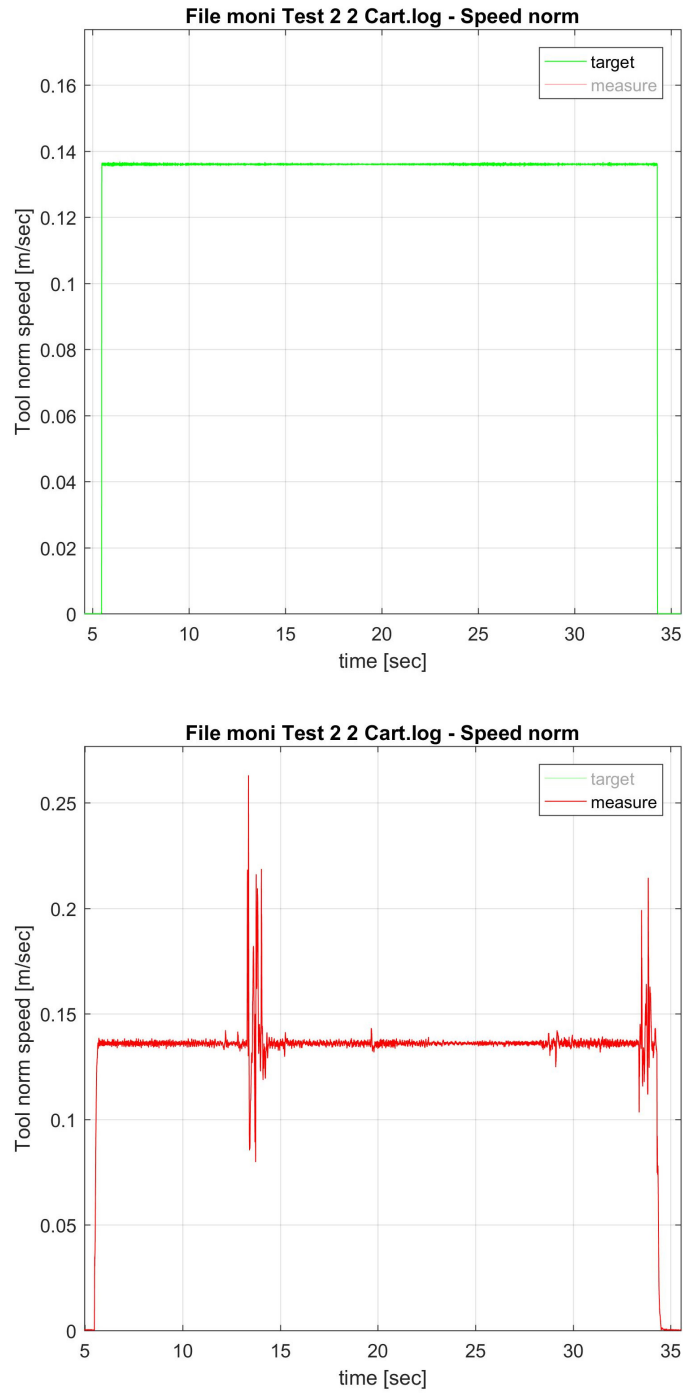


Figure 4.4. Scalar velocity of the robot during the second test, at $F = 100\text{Hz}$

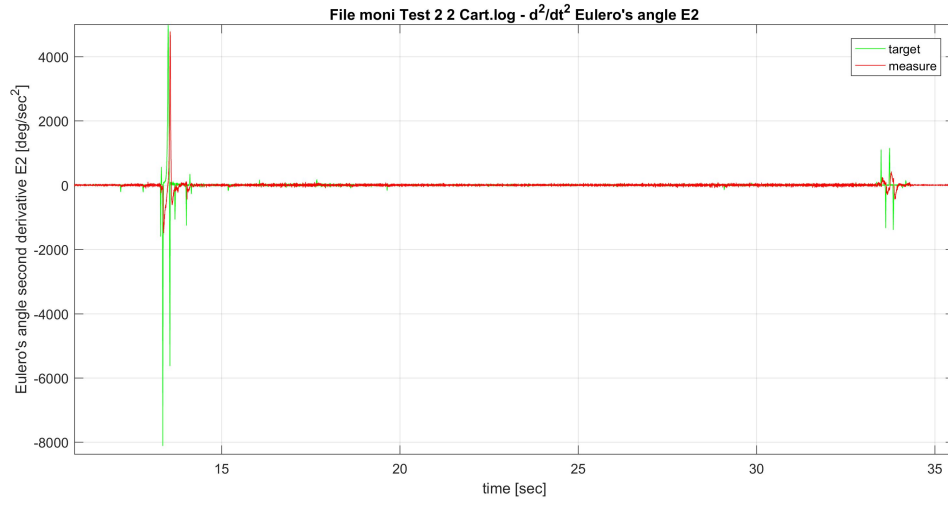


Figure 4.5. Test 2 - Angular acceleration of E2



Figure 4.6. Geometric path of the hood on test 2

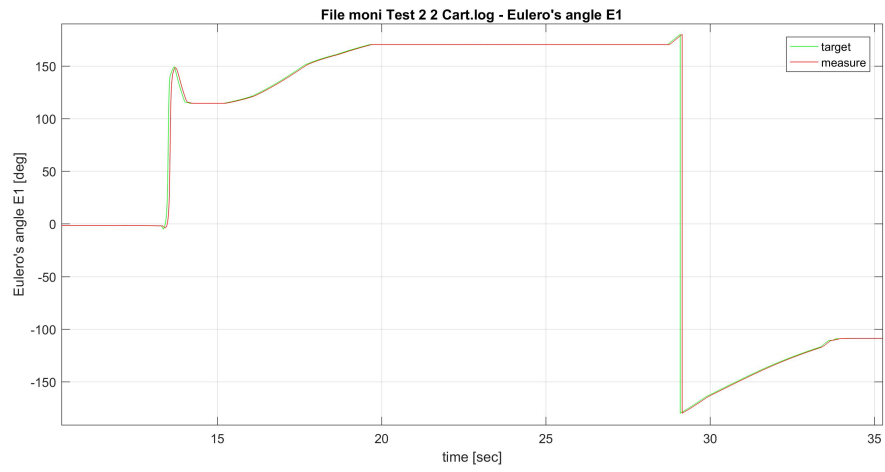


Figure 4.7. Test 2 - Euler angle E1

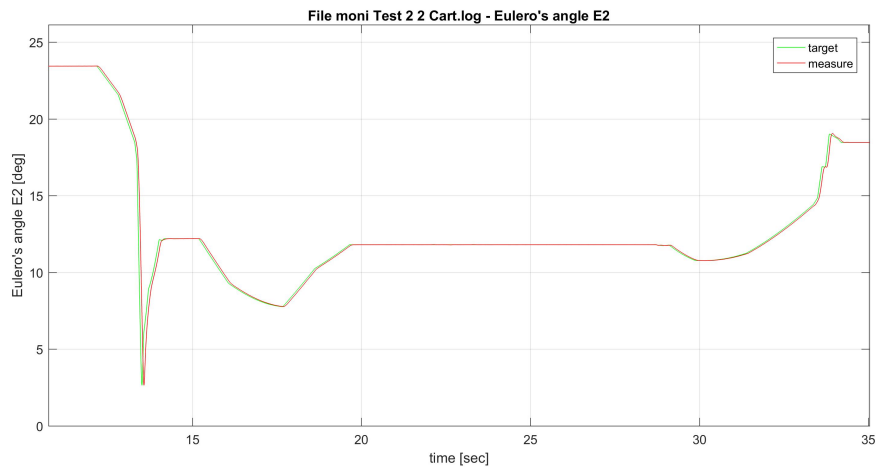


Figure 4.8. Test 2 - Euler angle E2

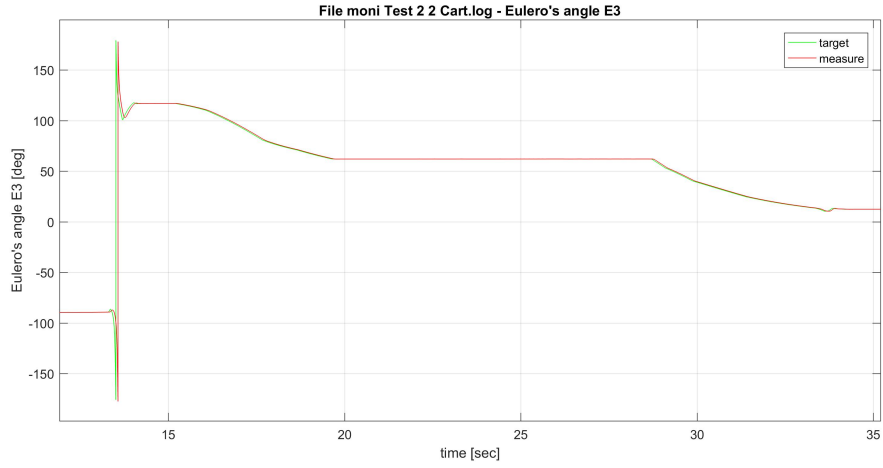


Figure 4.9. Test 2 - Euler angle E3

Moreover, also the motion law was filtered with a non-causal filter to obtain a smoother transition among the different velocities. The motion law, shown in Figure 4.10, is obtained applying the low pass filter, whose bode diagram is shown in Figure 4.11, on the scalar abscissa in Figure 3.12. The experimental results obtained with this approach are shown in Figures 4.13 and 4.14.

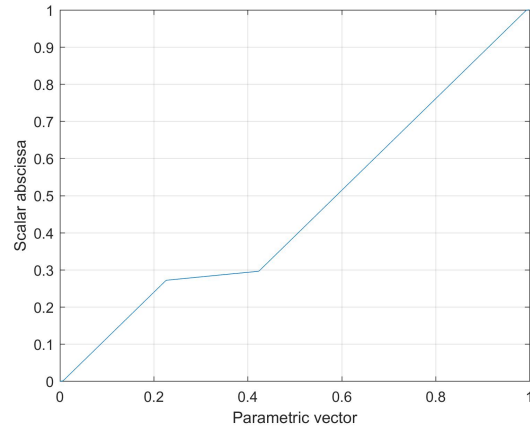


Figure 4.10. Test 3 - Motion law of the robot

Apart from the change in velocity during the transition, due to a non-optimal transition's algorithm, it is evident the difference with the previous approach, since the scalar velocity remains constant into the two areas. The price to pay is a huge local deceleration and a dilatation of the total time of processing, which is generally an acceptable cost in this kind of work. In Figure 4.14 it can also be appreciated the attenuation of the angular acceleration for the Euler angle E2, which remains almost null during all the movement. In appendix A.12 the code used to implement this last test is shown.

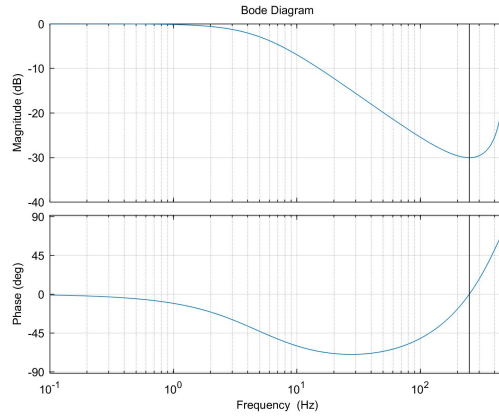


Figure 4.11. Test 3 - Non-causal filter with cut frequency at 5 Hz

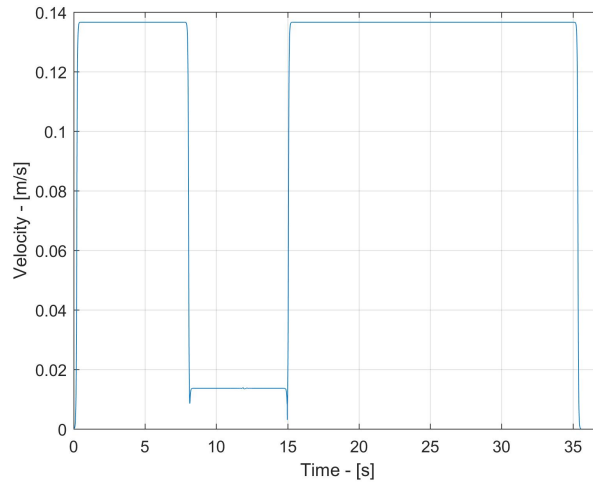


Figure 4.12. Test 3 - Scalar velocity in simulation

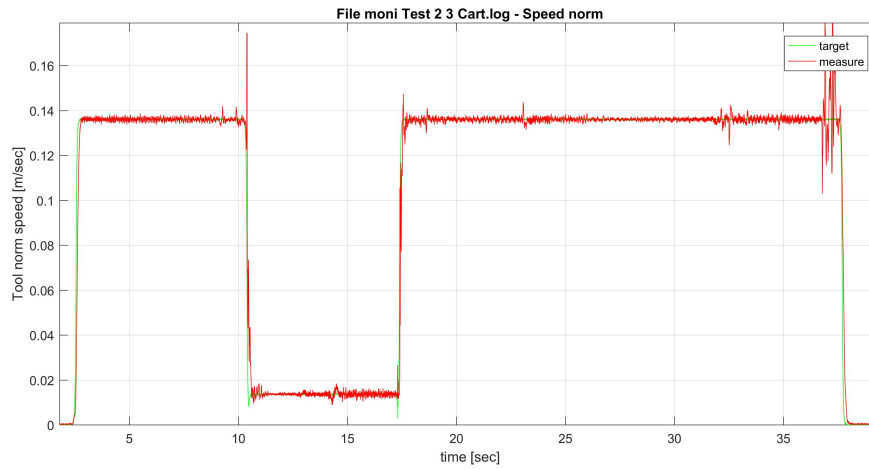


Figure 4.13. Test 3 - Scalar velocity during the test

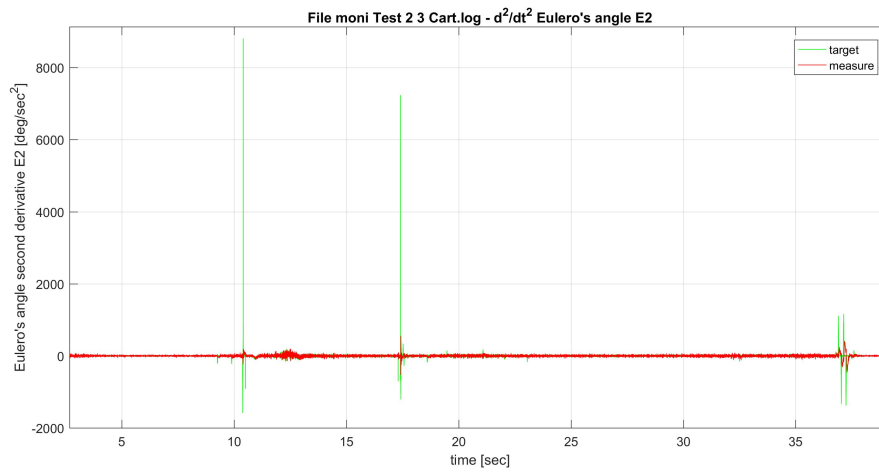


Figure 4.14. Test 3 - Angular acceleration of E2

Chapter 5

Conclusions

In this thesis the problem of finding a solution for the coupled planning of path and orientation for an industrial robot has been dealt with. This is a problem which doesn't have a simple solution, since it requires the search for a trade-off between the movement of the arm and the variable orientation of the tool, which 'drags' the rest of the arm during his movements, changing the scalar velocity. From the experimental tests it is evident that the robot is unable to follow the constant velocity along all the path. This is due to the strong and sudden variation of the orientation of the tool in the re-orientation phase, which causes an increment of the scalar velocity of the arm, being arm and tool coupled. So, another arrangement has been made reducing the scalar velocity in the re-orientation area through a tuning of the frequency of the filters' cascade of the planner.

5.1 Future works

The algorithm developed shows a good behaviour in case of planning of the path only, while it suffers the coupling of path and orientation and it needs some further arrangement in case of strong variation of the tool's attitude. Taking into account that in this specific work the critical area is the re-orientation phase of the path, in which the robot has no particular constraints, a solution could be to simply reshape that area in order to smooth the movement of the manipulator. On the other hand, a good idea could be the one explained in Chapter 4, in which the properties of the filters' cascade are used to selectively decrease the scalar velocity in the areas in which the robot suddenly changes the tool's attitude. So, an optimization of the algorithm which handles the motion law in that case, especially for the handling of the transition between different velocities, should be done. In addition to this, an optimization of the nurbs' algorithm, making it more robust and making its effect more local on the curve, is suggested, since the two techniques could work together to obtain a smoother and nicer movement of the robot.

Appendix A

Matlab codes

A.1 angles_calc.m

```
function [theta] = angles_calc(curve)

% 'angles_calc' - It computes the angles between each couple of
% consecutive points on the curve

% INPUT
%   curve - Input curve

% OUTPUT
%   theta - buffer of output angles

x = linspace(0,1,length(curve));
y = curve;
n = size(curve,1);

for i = 2:n-1
    x1(i-1,1) = x(i)-x(i-1);
    y1(i-1,1) = y(i)-y(i-1);
end
for i = 2:n-1
    x2(i-1,1) = x(i+1)-x(i);
    y2(i-1,1) = y(i+1)-y(i);
end
p1 = [x1 y1 zeros(n-2,1)];
p2 = [x2 y2 zeros(n-2,1)];
theta(1,1) = 0;
for i = 2:n-1
    theta(i,1) = atan2d(norm(cross(p1(i-1,:),p2(i-1,:))),dot(p1(i-1,:),
        p2(i-1,:)));
end
theta(end+1,1) = 0;
```

A.2 arc_length.m

```
function [arc, sc_abs] = arc_length(curve)

% 'arc_length' - Given the curve, it compute its arc length
% through the polygonal

% INPUT
%   curve - Input curve

% OUTPUT
%   arc - arc length
%   sc_abs - scalar abscissa

% spline = spline;
s = 0;
arc = zeros(length(curve),1);
for i = 2:length(curve)
    temp = sqrt((curve(i,1)-curve(i-1,1))^2+(curve(i,2)-curve(i-1,2))^2+...
                (curve(i,3)-curve(i-1,3))^2); % Poligonale
    s = s + temp;
    arc(i) = s;
end
sc_abs = arc/max(arc);
```

A.3 samp_calc.m

```
function [n_samp] = samp_calc(curve)

% 'samp_calc' - It computes the maximum possible number of samples
% to which the given curve could be oversampled by the 'interparc'
% function by computing minimum distance over the arc length

% INPUT
%   curve - Curve to be oversampled

% OUTPUT
%   n_samp - Number of samples

Arc = arc_length(curve);
```

```
for i = 2:length(curve)
    dist(i-1) = (Arc(i)-Arc(i-1))/Arc(end);
end
minimum = min(dist);
n_samp = round(1/minimum);
```

A.4 interpolation.m

```
function P = interpolation(q)

% 'interpolation' - Given the input points representing the
% desired curve it computes the control points computing the
% tridiagonal system : Ax = b

% INPUT
% q - Input points

% OUTPUT
% P - Control points

degree = 3;
n = length(q);

%% Tridiagonal matrix
A = zeros(n-2,n-2);
row = [1 4 1 zeros(1,n-5)];
for i = 1:n-3
    A(i+1,:) = row;
    row = circshift(row,1);
end
A(1,:) = [4 1 zeros(1,n-4)];
A(end,:) = [zeros(1,n-4) 1 4];

%% Computation of the control points
B = zeros(n-2,1);
B(1) = 6*q(2)-q(1);
for k = 2:length(B)-1
    B(k) = 6*q(k+1);
end
B(end) = 6*q(end-1)-q(end);
```



```
P(1:degree-1) = q(1);  
P(degree:n) = A\B;  
P(n+1:n+2) = q(end);
```

A.5 sequencer.m

```
function pk = sequencer(P,N,n)  
  
% 'sequencer' - Given the control points 'P' computes the  
% piecewise constant function associated to them, keeping  
% constant each control point for N samples  
  
% INPUT  
%   P - Control points  
%   N - Number of samples  
%   n - length of input point buffer  
  
% OUTPUT  
%   pk - piecewise constant function  
  
pk = zeros(1,n*N);  
for i = 1:n  
    for k = 1:N  
        pk(1,(i-1)*N+k) = P(i);  
    end  
end
```

A.6 FIR.m

```
function curve = FIR(N ,pk, initState, degree)  
  
% 'FIR' - Build the moving average filters and filter three times  
% the piecewise constant function through them  
  
% INPUT  
%   N - Number of samples  
%   pk - Piecewise constant function  
%   initState -  
%   degree - Desired degree of the spline
```

```
% OUTPUT
%   curve - Output curve

% Building of the filter
b = (1/(N))*ones(1,N);
a = 1;
initStates = initStates*ones(1,N-1);

% Implementation of the cascade of filters
curve(1,:) = filter(b,a,pk,initStates);
for k=2:degree
    curve(k,:) = filter(b,a,curve(k-1,:),initStates);
end

curve = curve(3,:);
```

A.7 elab_data.m

```
function [pk_new,curve_new] = elab_data(pk,curve,N)

% 'elab_data' - Impose a delay of 3N to the curve obtained by the
% FIR filters to remove the transitory

% INPUT
%   pk - Piecewise constant function
%   curve - Curve obtained by the FIR filters
%   N - Number of samples

% OUTPUT
%   pk_new - Elaborated piecewise constant function
%   curve_new - Elaborated curve

curve(1:3*N) = curve(3*N);
post_curve = curve(2*N+1:end);
clear curve;
curve_new = zeros(1,length(post_spline));
curve_new = post_curve;
temp = curve_new(end);
curve_new(end+1:end+N) = temp;

clear temp;
pk = pk(2*N+1:end);
```

```
temp = pk(end);  
pk(end+1:end+N) = temp;
```

A.8 curve_rad.m

```
function [rad_norm] = curve_rad(n_original, points, multiplicator)  
  
% 'curve_rad' - Compute the radius of curvature along the points of  
% the curve, temporally oversampled through the parameter '  
%     multiplicator'  
% to get a better resolution on the curvatures  
  
% INPUT  
%     n_original - Original number of points  
%     points - Input points  
%     multiplicator - Multiplicator of the points  
  
% OUTPUT  
%     rad_norm - Norm of the curvature's radius associated to the  
%     original  
%     buffer of points  
  
%% Curvature calculation and plot  
  
[~,~,curv] = curvature(points');  
figure, hold on, plot(points(1,:),points(2:,:), 'k', 'Marker', '.')  
quiver(points(1,:)', points(2,:) ', curv(:,1), curv(:,2))  
title('Curvature radius'), xlabel('Arc Length - [mm]'), ylabel('Z -  
    [mm]'), grid on  
  
%% Norm  
  
n = length(points);  
points = points';  
for i = 1:n  
    rr_norm(1,i) = sqrt(curv(i,1)^2+curv(i,2)^2+curv(i,3)^2);  
end  
  
%% Assignment of norm and coordinates to the original points  
  
temp = zeros(n_original,2);  
for i = 1:n_original
```

```
for j = 1:multiplicator
    temp(j,1) = rr_norm(:,j+(i-1)*multiplicator);
end
rad_norm(i,:) = max(temp(:,1));
index = find(temp(:,1) == max(temp(:,1)));
index = index(1);
end
```

A.9 curve_weights.m

```
function w = curve_weights(P, points, rad_norm, accuracy)

% 'curve_weighths' - Given the norm of the curvature's radius
% computes
% the buffer of weights to be associated to the Nurbs

% INPUT
% P - Control points
% points - Input points of the curve
% rad_norm - Buffer of radius of curvature
% accuracy - Regulable parameter which tell how strong should
% the shaping action on the nurbs

% OUTPUT
% w - Buffer of weights

%% Pre-elaboration of the datas

P = P(:,2:end-1)';
rad_norm = rad_norm*1/accuracy;
k = 1./rad_norm;

%% Weight's calculation

for i = 1 : length(points)
    if (rad_norm(i,1) < 1)
        w(i,1) = 1;
    else
        w(i,1) = k(i);
    end
end
```

```
w = ceil(w*1e1)/1e1;  
w(1,1) = 1;  
w(end,1) = 1;
```

A.10 weights_calc.m

```
function [crv,w] = weights_calc(crv, accuracy, P, p)  
  
% 'weights_calc' - Oversample the curve in input and assign to  
% that the weights in output from 'curve_weights'  
  
% INPUT  
%   crv - Input nurbs  
%   accuracy - Regulable parameter which tell how strong should  
%   the shaping action on the nurbs  
%   P - Control points of the nurbs  
%   p - Points of the nurbs  
  
% OUTPUT  
%   crv - Modified nurbs  
%   w - Buffer of weights  
  
%% Oversampling of the curve  
  
multi = 10;  
n = length(p);  
neval = n*multi;  
ut = linspace(0,1,neval);  
nrb = nrbeval(crv,ut);  
  
%% Computation of the angles between points  
  
for i = 1:3  
    theta(:,i) = angle_calc(nrb(i,:))';  
end  
  
%% Computation and assignment of the weights  
  
for i = 1:3  
    [Norm, Coord] = curve_rad(n,[linspace(0,nrb(i,end),length(nrb));  
        nrb(i,:); zeros(1,length(nrb))],multi);  
    w(:,i) = curve_weights(P(i,:),p(:,i),Norm,Coord,accuracy);
```

```
end
for i = 1:3
    if (max(theta(:,i)) > 160)
        crv.coefs(:,2:end-1) = w(:,i)'.*crv.coefs(:,2:end-1);
    end
end
```

A.11 weights_orientation.m

```
function[q] = weights_orientation(points, accuracy_or)

% 'weights_orientation' - It compute the angle between the two
% segments
% connecting three consecutive points and, depending on this,
% assign
% the weights to the nurbs

% INPUT
% points - Points of the nurbs
% accuracy_or -

% OUTPUT
% q - Points of the modified nurbs

%% Computation of the angle between the segment connecting the
% points

for j = 1:4
P = interpolation(points(:,j));
for i = 1:length(P)
    p(i,:) = [ut(1,i), P(1,i)];
end
distance(1) = 0;
theta(1) = 1;
for i = 2 : length(P)-1
    tg1 = (p(i,:)-p(i-1,:))/norm(p(i,:)-p(i-1,:));
    tg2 = (p(i+1,:)-p(i,:))/norm(p(i+1,:)-p(i,:));
    tg1 = [tg1 0];
    tg2 = [tg2 0];
    theta(i) = atan2d(norm(cross(tg1,tg2)),dot(tg1,tg2))
    distance(i,1) = norm(crv.coefs(1,i)-points(i-1));
end
```

```
theta(end+1) = 1;
distance(end+1) = 0;

%% Assignment of the weights

if (accuracy_or == 1)
    v=1;
    w=1;
else
    v = 2 - accuracy_or;
    w = accuracy_or;
end
for i = 2:length(P)-1
    if (theta(i) > 0 && theta(i)<120 && distance(i)<0.003)
        crv.coefs(:,i) = v.*crv.coefs(:,i);
    end
    if (theta(i) > 120 && distance(i) > 0.003)
        crv.coefs(:,i) = w.*crv.coefs(:,i);
    end
end
nrb1 = nrbeval(crv,linspace(0,1,length(points))) ;
q(:,j) = nrb1(1,:);
end
```

A.12 test.m

```
%% BM FILTER

% In this Matlab code the use of the cascade of filters to
% selectively
% reduce the velocity in the re-orientation area and the filtering
% of
% the overall trajectory to smoothen the transitions among
% different
% velocities

clc, clear all, close all

load points.mat
load Cntrl_Point.mat

%% Implementation of the filter
```

```

[ num, den ] = get_IIR2( 5, 0.99 , 5, 1, 0.002 );
LP = tf( num, den, 0.002, 'variable', 'z^-1' );
my_bode( LP, 0.1, 1/0.002 )

%% Use of the moving average filters

[arc, par] = ArcLength(points);
%BM FILTER
for i = 1:7
    F = 100;
    [T1,N1,pk1(:,i),spline1(:,i)] = B_M_Filter(P(:,i),F);
end
for i = 1:7
    F = 10;
    [T2,N2,pk2(:,i),spline2(:,i)] = B_M_Filter(P(786:856,i),F);
end
temp = spline1(1:3929,1:7);
temp1 = spline1(4269:end,1:7);
curve = [temp; spline2; temp1];
[arc, par] = ArcLength(curve);
clear spline;

%% Filtering of the scalar abscissa

n = 100;
par1 = filtfilt( num, den, [zeros(n,1); par; ones(n,1)]);
figure, plot(linspace(0,1,length(par1)),par1)
xlabel('Parametric vector'), ylabel('Scalar abscissa'), grid on
spline = interparc(par1,curve(:,1),curve(:,2),curve(:,3),'linear');
quat_in = curve(1,4:7);
quat_fin = curve(end,4:7);
spline(:,4:7) = [quat_in.*ones(n,1); curve(:,4:7); quat_fin.*ones(n,1)];

%% PLOT

figure,
plot3(spline(:,1),spline(:,2),spline(:,3),'r'), grid on, axis equal
figure,
plot(par1,[spline(:,4) spline(:,5) spline(:,6) spline(:,7)]), grid
on

```



```
Ts = 0.002;
t = [0:Ts:Ts*(length(spline)-1)];

% VELOCITY
dx = zeros(1,length(spline));
dy = zeros(1,length(spline));
dz = zeros(1,length(spline));
dx(1) = 0;
dy(1) = 0;
dz(1) = 0;
dx(2:end) = diff(spline(:,1))/Ts;
dy(2:end) = diff(spline(:,2))/Ts;
dz(2:end) = diff(spline(:,3))/Ts;
Vel = sqrt(dx.^2+dy.^2+dz.^2)';
figure, plot(t,Vel./1000), grid on, xlabel('Time - [s]'), ylabel('
    Velocity - [m/s]')
```

Bibliography

- [1] A. Grau, M. Indri, L. Lo Bello, T. Sauter, “Industrial Robotics in Factory Automation: from the Early Stage to the Internet of Things” in *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, 2017.
- [2] L. Biagiotti, C. Melchiorri, “B-Spline Based Filters for Multi-Point Trajectories Planning” in *IEEE International Conference on Robotics and Automation*, May 3-8, 2010.
- [3] [Online]: <https://www.comau.com/en>
- [4] [Online]: <https://www.lord.com>
- [5] G. D. Gironimo, *Modellazione di curve e superfici a forma libera*, Università degli studi di Napoli Federico II.
- [6] L. Piegl, W. Tiller, *The NURBS book* Springer, July 1996.
- [7] [Online]: <https://javascript.info/bezier-curve>
- [8] J. Peterson, *How to use Knot Vectors* Apple, June 1990.
- [9] B. Siciliano, L. Sciavicco, L. Villani, G. Oriolo, *Robotics: Modelling, planning and control* Springer, 2009.
- [10] B. Bona, *Slides from lessons of 'Robotics'*, Politecnico di Torino, A.Y. 2018/2019.
- [11] L. Anderlucci, *Smooth trajectory planning for anthropomorphic industrial robots employed in continuous processes*, Politecnico di Torino, April 2019.
- [12] L. Biagiotti, C. Melchiorri, *Trajectory Planning for Automatic Machines and Robots* Springer, 2008.
- [13] E. B. Dam, M. Koch, M. Lillholm, “Quaternions, Interpolation and Animation” in *Technical report DIKU-TR-98/5*, July 17, 1998.
- [14] B. Bona, *Slides from lessons of 'Modelling and Simulation of Mechatronic Systems'*, Politecnico di Torino, A.Y. 2018/2019.
- [15] [Online]: <https://en.wikipedia.org/wiki/Quaternion>
- [16] L. Niesink, “Human-Media Interaction”
- [17] [Online]: https://en.wikipedia.org/wiki/William_Rowan_Hamilton
- [18] R. Ramamoorthi, A. H. Barr, “Fast Construction of Accurate Quaternion Splines” California Institute of Technology, 1997.
- [19] G. Bianchi, R. Rastegarian, *Trajectory planning for continuous processes using anthropomorphic industrial robots*, Politecnico di Torino, July 2017.
- [20] [Online]: <https://it.mathworks.com>

- [21] [Online]: <https://it.mathworks.com/matlabcentral/fileexchange/69452-curvature-of-a-2d-or-3d-curve>
- [22] [Online]: <https://it.mathworks.com/matlabcentral/fileexchange/34874-interparc>
- [23] L. Biagiotti, C. Melchiorri, "Input Shaping via B-spline Filters for 3-D Trajectory Planning" in *IEEE International Conference on Robotics and Automation*, September 25-30, 2011.