POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE

LAUREA MAGISTRALE IN INGEGNERIA DELL'AUTOMAZIONE

Dynamical Systems Theory

Prof. Eng. Mariagrazia Dotoli
Assist. Prof. Eng. Raffaele Carli
Eng. Fabio Mastromarino

Tesina:

Trajectory Planning Niryo NED 2

Studenti:
Nicola Saltarelli
Francesco Stasi
Davide Tonti

ANNO ACCADEMICO 2023-2024

Abstract

The report presents a trajectory planning methodology for the Niryo Ned 2 robot, based on inverse kinematics in the operating space and the trapezoidal approach. Initially, we conducted simulations using Matlab in combination with the Robotics System Toolbox in order to graphically visualize the planned trajectory. Next, we proceeded with the practical implementation, transferring the waypoints obtained to Python through the help of the pyniryo library. To further improve the accuracy and stability of the system, a tailor-made end-effector has been designed and manufactured to ensure a better grip of the pen during trajectory execution. The results obtained testify to the effectiveness of the strategy adopted, highlighting the robot's ability to successfully execute the planned operations.

Contents

1 Introduction

Robotics is concerned with the study of those machines that can replace human beings in the execution of a task, as regards both physical activity and decision making. The objective of this introductory chapter is to provide an overview of robotic manipulators, focusing specifically on the type of manipulator used in experimental phases. Furthermore, it aims to clarify the concepts of robot trajectory planning.

The mechanical structure of a robot manipulator consists of a sequence of rigid bodies (links) interconnected by means of articulations (joints); a manipulator is characterized by an arm that ensures mobility, a wrist that confers dexterity, and an end-effector that performs the task required of the robot.
A manipulator's mobility is ensured by the presence of joints. The articulation between two consecutive links can be realized by means of either a prismatic or a revolute joint. Each prismatic or revolute joint provides the structure with a single degree of freedom (DOF). A prismatic joint creates a relative translational motion between the two links, whereas a revolute joint creates a relative rotational motion between the two links.
The degrees of freedom should be properly distributed along the mechanical structure in order to have a sufficient number to execute a given task. In the most general case of a task consisting of arbitrarily positioning and orienting an object in three-dimensional (3D) space, six DOFs are required, as in the case of the Niryo Ned 2, three for positioning a point on the object and three for orienting the object with respect to a reference coordinate frame.
The workspace represents that portion of the environment the manipulator's end-effector can access. Its shape and volume depend on the manipulator structure as well as on the presence of mechanical joint limits.
The geometry of the robot used in the experimental phase is anthropomorphic in nature. It consists of three revolute joints; the revolute axis of the first joint is orthogonal to the axes of the other two which are parallel. By virtue of its similarity with the human arm, the second joint is called the shoulder joint and the third joint the elbow joint since it connects the "arm" with the "forearm". The anthropomorphic structure is the most dexterous one, since all the joints are revolute. On the other hand, the correspondence between the DOFs and the Cartesian space variables is lost, and wrist positioning accuracy varies inside the workspace. This is approximately a portion of a sphere and its volume is large compared to manipulator encumbrance.

In all robot applications, completion of a generic task requires the execution of a specific motion prescribed to the robot. The correct execution of such motion is entrusted to the control system which should provide the robot's actuators with the commands consistent with the desired motion. Motion control demands an accurate analysis of the characteristics of the mechanical structure, actuators, and sensors. The goal of such analysis is the derivation of the mathematical models describing the input/output relationship characterizing the robot components. Modelling a robot manipulator is therefore a necessary premise to finding motion control strategies.
Kinematic analysis of the mechanical structure of a robot concerns the description of the motion with respect to a fixed reference Cartesian frame by ignoring the forces and moments that cause motion of the structure. It is meaningful to distinguish between kinematics and differential kinematics. With reference to a robot manipulator, kinematics describes the analytical relationship between the joint positions and the end-effector position and orientation. Differential kinematics describes the analytical relationship between the joint motion and the end-effector motion in terms of velocities, through the manipulator Jacobian. The formulation of the kinematics relationship allows the study of two key problems of robotics, namely, the direct kinematics problem and the inverse kinematics problem, which is the one we are interested in and which we have used for trajectory planning. The former concerns the determination of a systematic, general method to describe the end-effector motion as a function of the joint motion by means of linear algebra tools.

The latter concerns the inverse problem; its solution is of fundamental importance to transform the desired motion, naturally prescribed to the end-effector in the workspace, into the corresponding joint motion.
With reference to the tasks assigned to a manipulator, the issue is whether to specify the motion at the joints or directly at the end-effector. In material handling tasks, it is sufficient to assign only the pick-up and release locations of an object (point-to-point motion), whereas, in machining tasks, the end-effector has to follow a desired trajectory (path motion). The goal of trajectory planning is to generate the timing laws for the relevant variables (joint or end-effector) starting from a concise description of the desired motion.
The following report will outline the procedure for trajectory planning using an empirical approach, employing numerical computing software such as Matlab.

2 Theoretical notions

This section will provide brief theoretical insights into the topics covered in the experimental phase section. This elucidation aims to facilitate a better understanding of the various procedures carried out in Matlab to conduct trajectory planning.

2.1 Pose of a Rigid Body

A rigid body is completely described in space by its position and orientation (in brief pose) with respect to a reference frame. As shown in Fig.1, let O-xyz be the orthonormal reference frame and $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z}$ be the unit vectors of the frame axes.

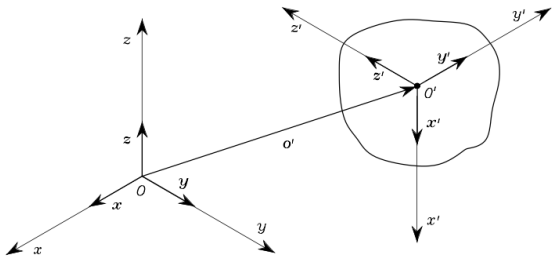


Figure 1: Position and orientation of a rigid body. Source: Bruno Siciliano et al., 'Robotics: Modelling, Planning and Control', published by Springer, 2008, p.40.

The position of a point O' on the rigid body with respect to the coordinate frame O–xyz is expressed by the relation

$$\boldsymbol{o'} = o'_x\boldsymbol{x} + o'_y\boldsymbol{y} + o'_z\boldsymbol{z} \tag{1}$$

where o'_x, o'_y, o'_z denote the components of the vector $\boldsymbol{o'} \in \mathbb{R}^3$ along the frame axes.
In order to describe the rigid body orientation, it is convenient to consider an orthonormal frame attached to the body and express its unit vectors with respect to the reference frame. Let then O'–$x'y'z'$ be such a frame with origin in O' and $\boldsymbol{x'}$, $\boldsymbol{y'}$, $\boldsymbol{z'}$ be the unit vectors of the frame axes. These vectors are expressed with respect to the reference frame O–xyz by the equations:

$$\begin{aligned}
\boldsymbol{x'} &= x'_x\boldsymbol{x} + x'_y\boldsymbol{y} + x'_z\boldsymbol{z} \\
\boldsymbol{y'} &= y'_x\boldsymbol{x} + y'_y\boldsymbol{y} + y'_z\boldsymbol{z} \\
\boldsymbol{z'} &= z'_x\boldsymbol{x} + z'_y\boldsymbol{y} + z'_z\boldsymbol{z}
\end{aligned} \tag{2}$$

The components of each unit vector are the direction cosines of the axes of frame O'-$x'y'z'$ with respect to the reference frame O–xyz.

2.2 Homogeneous Transformations

As illustrated at the previous subsection, the position of a rigid body in space is expressed in terms of the position of a suitable point on the body with respect to a reference frame (translation), while its orientation is expressed in terms of the components of the unit vectors of a frame attached to the body, with origin in the above point, with respect to the same reference frame (rotation).
As shown in Fig.2, consider an arbitrary point P in space. Let $\boldsymbol{p}^0$ be the vector of coordinates of P with respect to the reference frame O_0-$x_0y_0z_0$. Consider then another frame in space O_1-$x_1y_1z_1$.

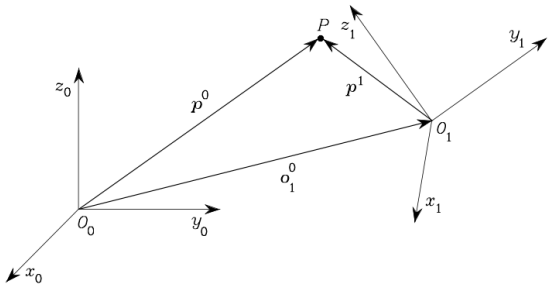


Figure 2: Representation of a point P in different coordinate frames. Source: Bruno Siciliano et al., 'Robotics: Modelling, Planning and Control', published by Springer, 2008, p.56.

Let $\boldsymbol{o}_1^0$ be the vector describing the origin of Frame 1 with respect to Frame 0, and $\boldsymbol{R}_1^0$ be the rotation matrix of Frame 1 with respect to Frame 0. Let also $\boldsymbol{p}^1$ be the vector of coordinates of P with respect to Frame 1. On the basis of simple geometry, the position of point P with respect to the reference frame can be expressed as

$$\boldsymbol{p}^0 = \boldsymbol{o}_1^0 + \boldsymbol{R}_1^0 \boldsymbol{p}^1 \tag{3}$$

Hence, (3) represents the coordinate transformation (translation + rotation) of a bound vector between two frames. The coordinate transformation can be written in terms of the (4×4) matrix

$$\boldsymbol{A}_1^0 = \begin{bmatrix} \boldsymbol{R}_1^0 & \boldsymbol{o}_1^0 \\ \boldsymbol{0}^T & 1 \end{bmatrix} \tag{4}$$

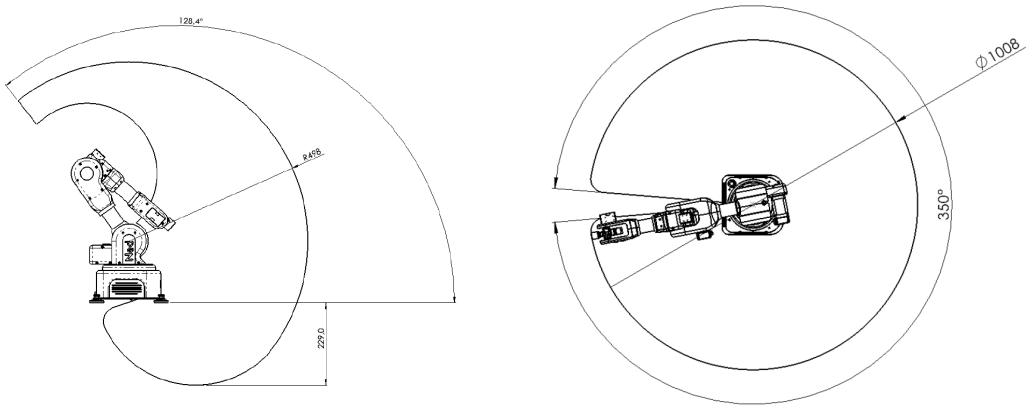
As can be easily seen from (4), the transformation of a vector from Frame 1 to Frame 0 is expressed by a single matrix containing the rotation matrix of Frame 1 with respect to Frame 0 and the translation vector from the origin of Frame 0 to the origin of Frame 1.

2.3 Joint and Operational space

The **Joint Space** refers to the configuration space of a robot, specifically focusing on its individual *joints* previously presented. In joint space, we describe the robot's pose using the angles or positions of its joints in which each joint corresponds to a specific degree of freedom.
The **Operational** (Task) **Space** represents the robot's position and orientation in the *external world*. In operational space, we describe the robot's pose using more intuitive parameters such as position and orientation. The position is described as (x, y, z) coordinates of a specific point in space while roll, pitch, and yaw angles define the robot's orientation.

For this project we worked on the operational space that allows us to define tasks directly related to the robot's functionality, in this case, to reaching specific points. It provides a higher-level perspective, focusing on what the robot needs to achieve rather than the specific joint angles.
The generation of the waypoints is based on the workspace: There is a check in the Matlab code (appendix A) to verify the trajectory points are within the robot's operational space.

The datasheet of Niryo Ned 2 helps us to understand the operational space of the robot:



(a) Ned2 workspace Side View **(b)** Ned2 Workspace Top View

Figure 3: Ned2Workspace

2.4 Trajectory planning

Let's introduce a brief explanation about the trapezoidal approach used for the trajectory planning.

When a robot needs to move from one configuration (start position) to another (target position), it follows a trajectory. The trapezoidal approach is a method used to generate smooth motion profiles for robot joints during trajectory planning; the time sequence of the values attained by the operational space variables is utilized in real time to obtain the corresponding sequence of values of the joint space variables, via an inverse kinematics algorithm.

The trapezoidal approach divides the trajectory into three distinct phases:

- *Acceleration phase*: The robot accelerates from rest to its maximum velocity. During this phase, the acceleration rate is controlled to avoid sudden jerks.
- *Constant velocity phase*: Once the robot reaches its maximum velocity, it maintains a constant speed. This phase ensures smooth motion without abrupt changes.
- *Deceleration phase*: As the robot approaches the target position, it decelerates to come to a stop. Similar to the acceleration phase, controlled deceleration prevents sudden stops.

The advantages of using the Trapezoidal Approach are the following:

- *Smoothness*
- *Predictable Behavior*, by controlling acceleration and deceleration
- *Efficiency*, because the approach balances speed and smoothness, optimizing the trajectory execution time

2.5 Jacobian matrix

The Jacobian is a fundamental matrix that describes the relationship between the velocity of a robot's joints and the velocity of its endpoint (end-effector) with its orientation in space. In simple words, it allows us to understand how the motion of each joint (e.g., the rotation of a robotic arm) affects the position and orientation of the object the robot is manipulating.
The matrix is represented by rows and columns such that the number of rows corresponds to the degrees of freedom of the endpoint (usually 6: 3 translations and 3 rotations), while the number of columns corresponds to the number of joints of the robot. Each element of the matrix represents the sensitivity of the endpoint with respect to the velocity of a particular joint.
In the numerical implementation, computation of joint velocities is obtained by using the inverse of the Jacobian evaluated with the joint variables at the previous instant of time:

$$q(t_{k+1}) = q(t_k) + J^{-1}(q(t_k))v_e(t_k)\Delta t$$

The complexity of the Jacobian, however, lies in computing a closed form analytically for several reasons, including a complexity in derivation, dependence on robot parameters, and sensitivity to modeling errors.
For these reasons, approaches with practical models such as the Broyden-Fletcher-Goldfarb-Shanno method and the Levenberg-Marquardt method are preferred.

3 Experimental phase

The experimental phase of our trajectory planning project for the Niryo Ned 2 robot has been characterized by a series of steps aimed at achieving the specified task.
Initially, we developed MATLAB code aimed at generating waypoints within the robot's workspace. This process required the implementation of inverse kinematics techniques to obtain the desired trajectory.
To facilitate the graphical visualization of the planned path by the robot, we utilized the Robotics System Toolbox, which provided us with the necessary tools for a clear and intuitive representation of motion dynamics.
Once we obtained a functional simulation and verified the correctness of the generated data, we proceeded with the transfer of waypoints from MATLAB code to a Python programming environment. This step was crucial for integrating the results obtained in trajectory planning with the robot.
Through the use of the pyNiryo library, we finally achieved our goal of executing the planned trajectory on the robot. This library, specifically designed to interact with the Niryo Ned 2 robot, provided us with the necessary tools to effectively translate the planned data into concrete actions for the robot itself.

3.1 About Niryo Ned 2

The Niryo Ned 2 is a 6-axis robotic arm. It has a payload of 300g, a reach of 490mm, and weighs 7kg. The repeatability is ± 0.5 mm. The power supply is 12 Volts / 7A. The material is made of plastic injection and aluminum. It has WiFi capabilities of 2.4GHz & 5GHz, Ethernet of 1Gb/s, and USB of 2.0 - 3.012.

The Ned 2 takes advantage of the capacities of the Raspberry Pi 4, with a 64-bit ARM V8 high-performance processor, 4 GB of RAM, and improved connectivity345. It is based on Ubuntu 18.04 and ROS (Robot Operating System) Melodic35. The robot can be controlled with Niryo Studio1.

The Ned 2 robot is built from aluminium and also has PLA components that can be reproduced using 3D printing. There are many resources to help you develop new printable accessories. The back panel of Ned 2 is open and accessible, allowing you to connect your accessories but also to interface all your electronic circuits and an infinite number of sensors.

3.2 MATLAB code

The MATLAB section includes the description of the main script used for trajectory planning. The first step involves the use of scripts aimed at representing the robot's geometries.

```
% Rigid Body Tree information
[ned2, numJoints] = ned2_import;
showdetails(ned2)
```

```
--------------------
Robot: (7 bodies)

 Idx            Body Name          Joint Name          Joint Type            Parent Name(Idx)    Children Name(s)
 ---            ---------          ----------          ----------            ----------------    ----------------
   1            base_link          joint_world              fixed                   world(0)    shoulder_link(2)
   2       shoulder_link              joint_1            revolute              base_link(1)    arm_link(3)
   3            arm_link              joint_2            revolute         shoulder_link(2)    elbow_link(4)
   4          elbow_link              joint_3            revolute              arm_link(3)    forearm_link(5)
   5        forearm_link              joint_4            revolute            elbow_link(4)    wrist_link(6)
   6          wrist_link              joint_5            revolute         forearm_link(5)    hand_link(7)
   7           hand_link              joint_6            revolute           wrist_link(6)
--------------------
```

Figure 4: Executing "showdetails(ned2)" command

Through "ned2_import.m" script, it is possible to import the geometry of the Niryo Ned 2 robot and define the physical limits of the joints, in order to obtain a simulation as accurate as possible compared to reality. The use of the Robotics System Toolbox allowed us to explore the various configurations assumed by the robot, enabling us to review our code in case the robot assumed invalid configurations.

The next step involves waypoint generation. In the main script, this process is encapsulated in a single line of code. This instruction calls upon a specific function that, taking as input the string to be composed, the distance between each character, the starting coordinates of the robot, and the text size, generates a series of intermediate points (waypoints) defining the arrangement of the text centered at the starting position, with all the provided specifications as arguments of the function. These waypoints are computed using basic functions that describe each character as a combination of arcs and segments.

Figure 5: Niryo Ned2 robot

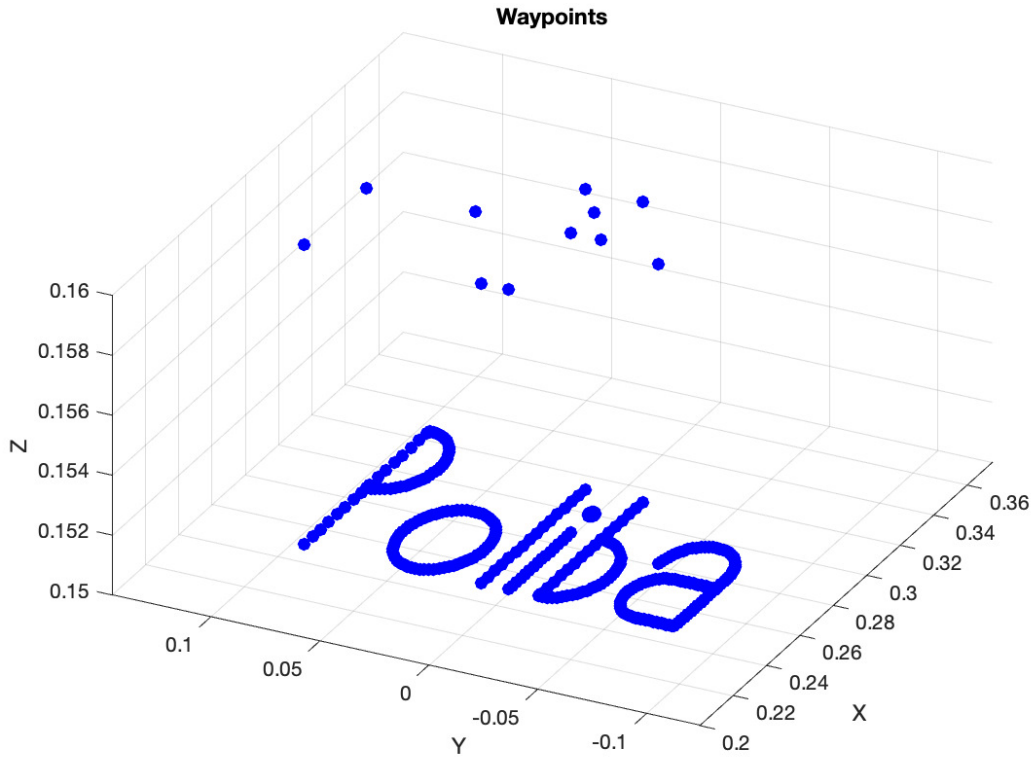


Figure 6: Generated waypoints.

The modular structure of the system allows us to separate the operations performed by each function, thus ensuring greater reusability and maintainability of the source code. It is important to note that, at the moment and for current purposes, the selection of characters available to the user is still limited. However, thanks to the modularity of the code, introducing new characters is extremely straightforward. It simply requires adding the necessary coordinates to define the new letters using specific functions for drawing arcs and segments.

Once the waypoints are acquired, the next phase of the process begins. Initially, the mean velocity that the robot can achieve is specified. Subsequently, the distance between each waypoint is calculated, and the time required to reach each point is determined based on the distance and the previously defined mean velocity. Following this, the sampling time is specified, and the calculation of the so-called "AccelTime" is performed, which represents the duration of the acceleration phase of the velocity profile, and the "EndTime", which represents the duration of each trajectory segment. All these steps are crucial to be able to utilize the Matlab command trapveltraj, which generates a trajectory through a given set of waypoints following a trapezoidal velocity profile.

In Figures 7 and 8, graphs depicting the position, velocity, and acceleration profiles of the end-effector are presented, highlighting a variation in the sampling interval. Decreasing the sampling interval increases the frequency at which data is recorded, leading to greater precision in motion monitoring. However, this can also result in increased sensitivity to instantaneous variations in velocity and acceleration, manifested as more pronounced oscillations in the velocity and acceleration graphs.

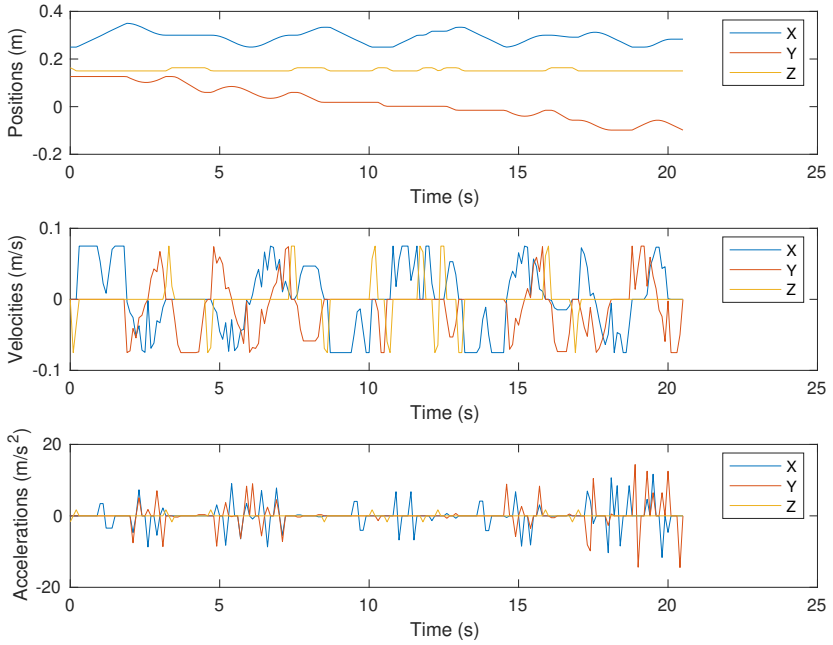


Figure 7: Trajectories positions, trapezoidal velocity profile and acceleration profile sampled every 0.1 seconds, with AccelTime set to one-fifth of EndTime

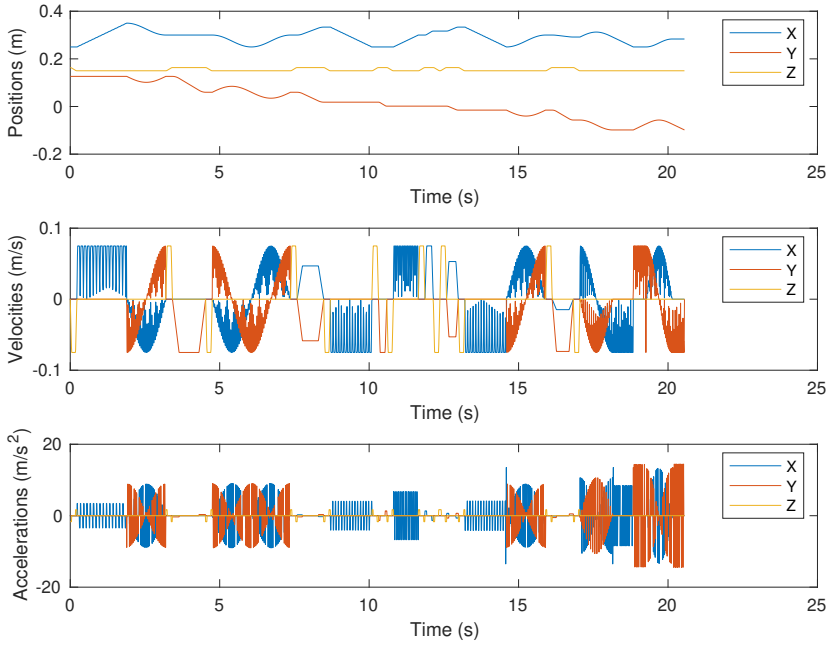


Figure 8: Trajectories positions, trapezoidal velocity profile and acceleration profile sampled every 0.01 seconds, with AccelTime set to one-fifth of EndTime

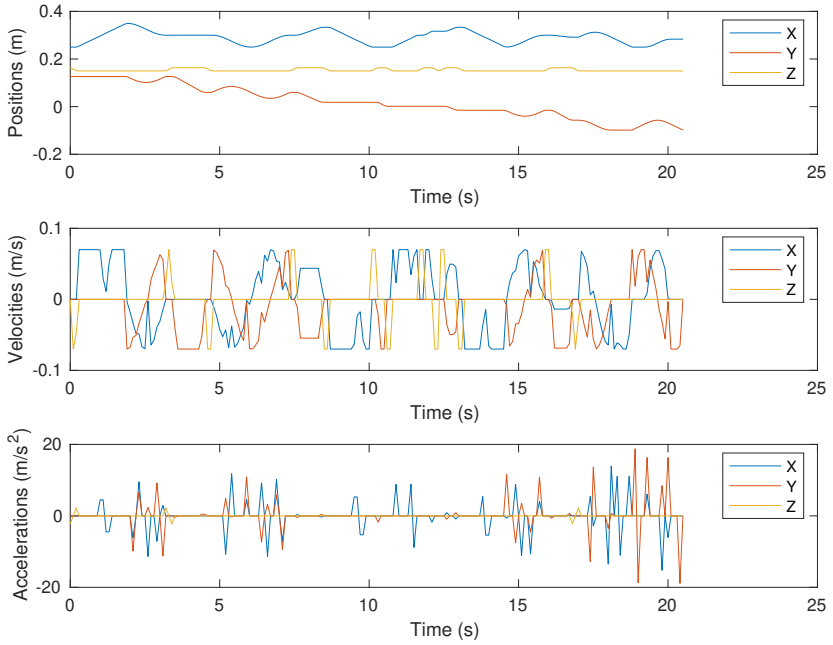


Figure 9: Trajectories positions, trapezoidal velocity profile and acceleration profile sampled every 0.1 seconds, with AccelTime set to one-seventh of EndTime

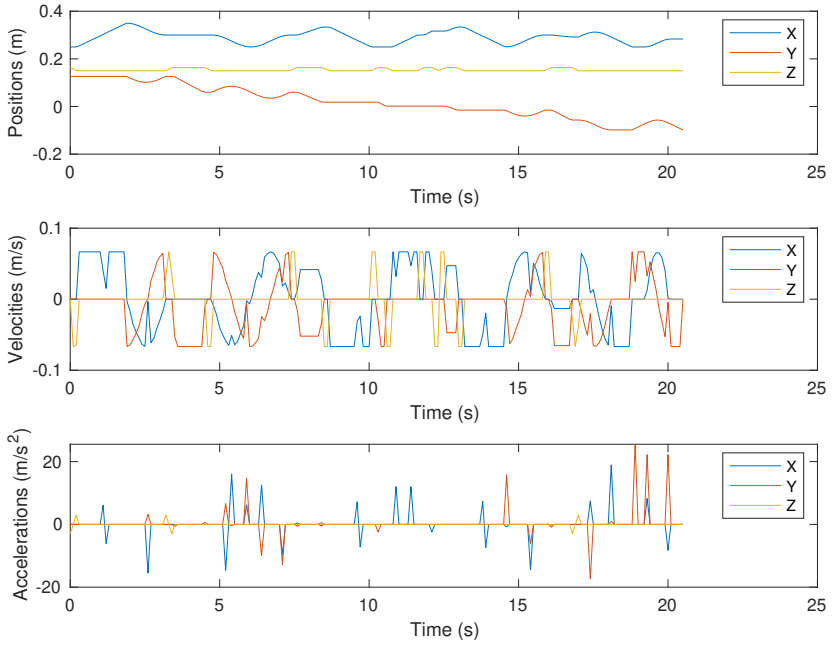


Figure 10: Trajectories positions, trapezoidal velocity profile and acceleration profile sampled every 0.1 seconds, with AccelTime set to one-tenth of EndTime

Similarly, in Figures 9 and 10, corresponding graphs of the position, velocity, and acceleration profiles of the end-effector are shown, highlighting a variation in the duration of the acceleration phase of the velocity profile. By reducing the AccelTime parameter and consequently shortening the duration of the acceleration phase, desired velocities can be reached with a faster acceleration. This may lead to higher acceleration peaks, increasing the effort required by the system's joints to follow the programmed velocity profile.

The final step to execute the desired trajectory by the robot involves utilizing the Matlab solver "inverseKinematics". This necessitates, firstly, converting the Cartesian representation of the translation vector, obtained from the preceding command "trapveltraj", into the corresponding homogeneous transformation. This conversion can be performed using the Matlab command "trvec2tform". Subsequently, the "inverseKinematics" solver is employed to compute the joint configurations required to achieve a desired pose of the end-effector.
It should be noted that the "inverseKinematics" solver in Matlab utilizes two types of algorithms. Both algorithms are iterative, gradient-based optimization methods that initiate from an initial estimation of the solution and aim to minimize a specified cost function.
The Broyden-Fletcher-Goldfarb-Shanno (BFGS) gradient projection algorithm is a quasi-Newton method that utilizes the gradients of the cost function from previous iterations to approximate second-derivative information. This method serves as the default algorithm and demonstrates greater robustness in finding solutions compared to the Levenberg-Marquardt method. It proves more effective for configurations near joint limits or when the initial guess is distant from the solution.
The variant of the Levenberg-Marquardt (LM) algorithm employed in the InverseKinematics class is an error-damped least-squares method. The error-damped factor helps to prevent the algorithm from escaping a local minimum. The LM algorithm is particularly optimized to converge significantly faster if the initial guess is proximate to the solution.
Given that in our case the waypoints are closely spaced, the LM algorithm demonstrates faster convergence, a fact empirically validated through testing.

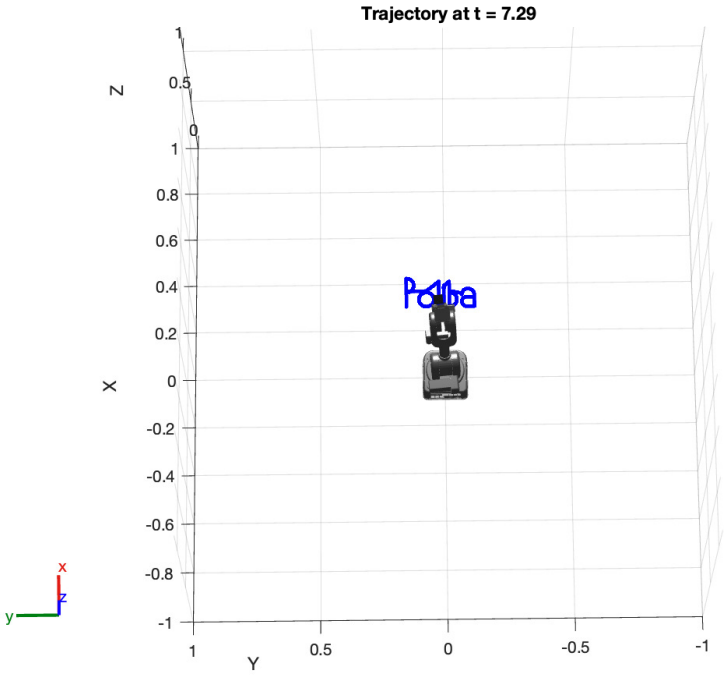


Figure 11: Robotics System Toolbox simulation

4 Results

4.1 Issues encountered

Once the code was finished and the waypoints dermined, and moved on to the practical test, several problems arose in executing the required trajectory:

- **Robot operating space**: working outside the operating space, or at the limit, makes it impossible to achieve the trajectory. It is therefore necessary to define an operating space in which to plan our trajectory to avoid not only unreachable waypoints but also unpermitted joint configurations. To do this, a check will be defined in the code that makes sure that the trajectory is included in the operational space, as told by the robot datasheet.
- **Number of waypoints**: The task of determining the optimal number of waypoints to incorporate is complex. Insufficient waypoints may result in a crude approximation of the desired path, while an excess of waypoints can overwhelm the system, leading to a slowdown in operations. This task will require a cycle of iterations to find the right tradeoff according to the purpose of the task.
- **Speed Between Waypoints**: The speed at which the robot moves between waypoints is of paramount importance. If the robot traverses too swiftly, it risks overshooting the waypoints, thereby compromising the accuracy of the drawing. Conversely, a slow pace can render the process inefficient. Achieving the right speed necessitates careful planning and precise execution.
- **End-effector**: The lack of a suitable end-effector, specifically designed to hold a marker pen among the options provided by the company, poses a significant challenge. Without an appropriate tool to securely grip and manipulate the pen, the robot is unable to effectively perform the drawing task. This highlights the need for custom-designed end effectors tailored to specific tasks.
- **Joint Tolerances**: The trajectory precision of the robot is heavily dependent on the tolerances of its joints. These tolerances, even when they deviate slightly, can lead to substantial discrepancies in the robot's path. This makes it exceedingly difficult to achieve the desired level of accuracy in the drawing process.

In order to make trajectory execution possible, different problems need to be solved, intervening where possible also at the hardware level and then mechanically.

4.2 Working Environment

When transitioning from simulating trajectories in Matlab to dealing with the practical problem, several critical aspects arise. First and foremost, the actual offset of the drawing plane is defined differently from what was done in Matlab. Consequently, we proceed to scale the z-axis position for all waypoints. Specifically, the Nyrio Ned 2 robot uses reference points located at the edges of a conveyor belt that comes with the robot. By replacing this conveyor belt, we position ourselves optimally to operate within the operational space and joint space. Additionally, we physically elevate the work surface using a support and then scale the waypoint coordinates to manage their dimensions.

Initially, the custom gripper end effector is chosen because it is the most practical for grasping a pen or marker. However, once mounted, two practical problems related to the end effector's structure become apparent, making it difficult to execute pen writing on paper and evaluate the calculated trajectory.

- **Insufficient Force:** The force exerted by the servo motor on the end effector is not sufficient to securely hold the pen. The pen has more degrees of freedom, allowing it to rotate and translate along an axis. Initially, the pen is constrained to the grip using a spring.
- **Mechanical Joints and Tolerance:** The second problem concerns mechanical joints, which introduce excessive tolerance. As a result, there is a noticeable decrease in trajectory precision. Consider, for instance, the connection of a magnetic end effector.

Understanding these challenges is crucial for assessing the results obtained during trajectory calculations. It helps distinguish between inaccuracies due to incorrect calculations and limitations inherent in the physical structure and hardware. To achieve practical success beyond the Matlab simulation, targeted interventions are necessary.

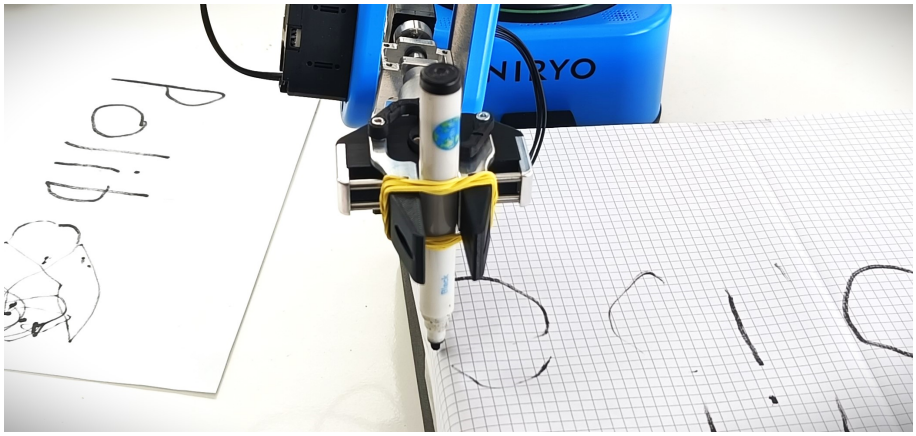
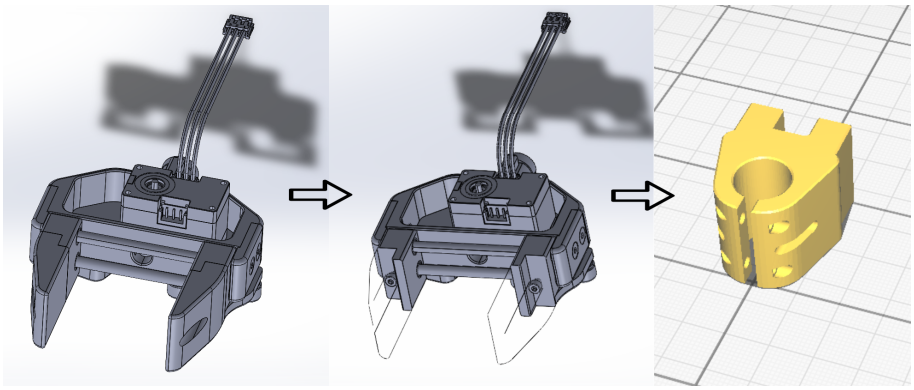


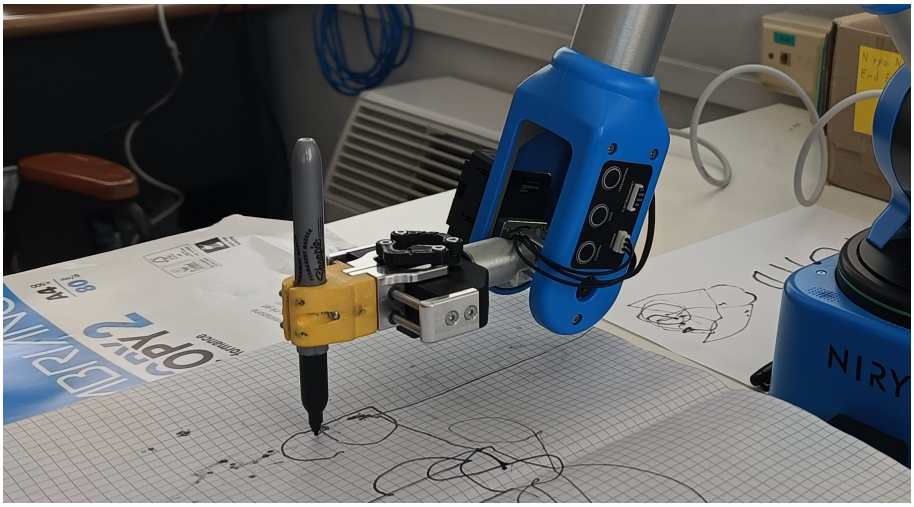
Figure 12: Test with Nyrio Ned 2 custom gripper

4.3 3D model End-Effector

To address the structural issue, a redesign of the Nyrio Ned 2's custom gripper is being considered. This end effector comprises an assembly of metallic mechanical pieces and 3D-printed contact parts. Notably, all the provided module designs are open source, allowing users to download the projects by cloning the repository.

The 3D model responsible for gripping the pen will be designed using SolidWorks software. It will incorporate modifications to the final contact portion, ensuring a secure grip on the pen.





We observe an improvement in execution that highlights the calculated trajectory.

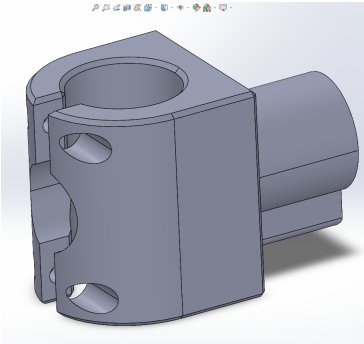


Figure 13: 3D model of the final end-Effector

However, the structural issues are not entirely resolved. Each joint introduces a degradation in precision. To further enhance the project, we consider replacing the custom gripper with a specially designed end effector. The new end effector in figure 13 would require a magnetic attachment, potentially mitigating some of the stresses. Notably, the trajectory is calculated near the attachment point; the farther it is from this point, the more challenging it becomes to reach the designated waypoints.

The new 3D model is printed in resin for higher precision. Once attached to the robot through the magnet, a more stable behavior on the joints and generally reduced tolerances are noticed, given by the disappearance of a strong lever (distance between the tip of the marker and the junction of the end effector).

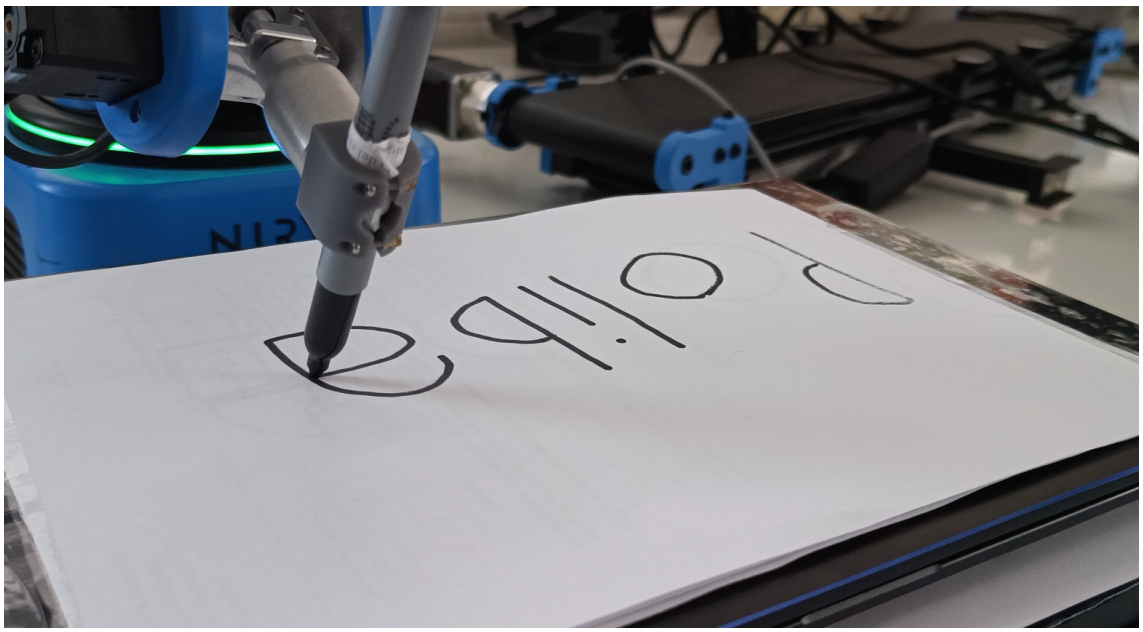
5 Conclusions

Resolving the errors found leads to a significant improvement in trajectory execution, which results in more accurate writing than in all previous tests performed. Restarting the program shows a complete re-tracing of the lettering, which makes us think of a small if any positional error (the thickness of the marker makes it difficult to deduce).

5.1 Further development

Adding the missing characters would make possible the correct writing of any text (uppercase, lowercase and special characters), this would allow the use of the software for different purposes, from writing to photoengraving and all those applications in which the use of a Cartesian machine is most used (CNC) but with the vandalism of being able to be adapted to all the most awkward surfaces, going to change pose and orientation of the various waypoints.

Another possible improvement can be made on the direct control of the robot using systems such as ROS, these would allow the use of velocity and acceleration profiles on the end-effector that have been calculated on Matlab. One would then have the advantage of complete and direct control of the robot, avoiding reliance on python's inverse kinematics solver to which we pass the waypoints and then the use of the PyNiryo library.



References

[1] Bruno Siciliano et al. “Robotics: Modelling, Planning and Control”. In: 2008. URL: https://api.semanticscholar.org/CorpusID:109962001.

[2] MathWorks. *Inverse Kinematics*. URL: https://it.mathworks.com/help/robotics/ref/inversekinematics-system-object.html (visited on 04/27/2024).

[3] MathWorks. *Trajectory Generation*. URL: https://it.mathworks.com/help/robotics/ref/trapveltraj.html (visited on 04/27/2024).

A Matlab Code

```
% NIRYO NED2 TASK SPACE TRAJECTORIES
% This script is designed for trajectory planning of the Niryo NED2 robot,
% utilizing the trapezoidal approach within its operational space.
% -------------------------------------------------------------------------
% AUTHORS:
% - Nicola Saltarelli
% - Francesco Stasi
% - Davide Tonti
% -------------------------------------------------------------------------
% Date modified 27/04/2024

%% Setup
clear, clc, close all

% Import Niryo Ned2 robot features
[ned2, numJoints] = ned2_import;

% Uncomment the lines below to verify the robot's construction
% and avoid showing this information every time the code is launched.
% "showdetails" lists all the bodies in the MATLAB  command window.
% showdetails(ned2)

% Define the starting position and orientation
toolPositionHome = [0.25, 0, 0.15, 0, 0, 0];
% toolPositionHome = [0.25, 0, 0.15, 0, 0.5, 0];

% Define the end-effector name
eeName = 'hand_link';

% Define the string to be followed by the robot
inputString = 'poliba';

% Define the distance between each letter
distance_between_letters = 0.1;

% Set the scale of the string (scale down to achieve a higher dimension)
scale = 6;

% Generate waypoints
waypoints = generateWaypoints(inputString, distance_between_letters, ...
    toolPositionHome, scale);

% Specify the desired mean velocity (e.g., in meters per second)
meanVelocity = 0.06; % 0.468 m/s is the max speed of Niryo Ned 2

% Calculate the distance between trajectory points
distances = sqrt(sum(diff(waypoints,[],2).^2));

```

```
% Calculate time required to reach each point based on distance and mean velocity
waypointTimes = [0, cumsum(distances / meanVelocity)];

% Trajectory sample time
ts = 0.1; % s
% Generate a vector of sample times with interval 'ts'
trajTimes = 0:ts:waypointTimes(end);

% Duration of acceleration phase of velocity profile
waypointAccelTimes = diff(waypointTimes)/5;

% Duration of each trajectory segment
endTimes = diff(waypointTimes);

% Define inverse kinematics (IK) solver
ik = inverseKinematics('RigidBodyTree',ned2, 'SolverAlgorithm','LevenbergMarquardt');

% Define weights for pose tolerances
ikWeights = [0 0 0 1 1 1]; % [orientation_weight, position_weight]

% Initial guess of robot configuration
ikInitGuess = ned2.homeConfiguration;

% Plot setup for trajectory simulation
figure(1);
show(ned2,ikInitGuess,'Frames','off','PreservePlot',false);
xlim([-1 1]), ylim([-1 1]), zlim([0 1.2])
hold on
hTraj = plot3(waypoints(1,1),waypoints(2,1),waypoints(3,1),'b.-');

%% Generate trajectory
[q,qd,qdd] = ...
    trapveltraj(waypoints,numel(trajTimes),'AccelTime',repmat(waypointAccelTimes,[3 ...
    1]), ...
    'EndTime',repmat(endTimes,[3 1]));

% Show the full trajectory with the rigid body tree
set(hTraj,'xdata',q(1,:),'ydata',q(2,:),'zdata',q(3,:));

%% Trajectory following loop
for idx = 1:numel(trajTimes)
    % Solve IK
    tgtPose = trvec2tform(q(:,idx)');
    [config, info] = ik(eeName, tgtPose, ikWeights, ikInitGuess);
    ikInitGuess = config;

    % Show the robot
    figure(1)
    show(ned2,ikInitGuess,'Frames','off','PreservePlot',false);
    title(['Trajectory at t = ' num2str(trajTimes(idx))])
    drawnow
end

%% Generation of plots

% Waypoints plot
figure(2);
scatter3(waypoints(1,:), waypoints(2,:), waypoints(3,:), 'filled', ...
    'MarkerEdgeColor', 'b', 'MarkerFaceColor', 'b');
xlim([min(waypoints(1,:))-0.05 max(waypoints(1,:))+0.05]), ...
    ylim([min(waypoints(2,:))-0.05 max(waypoints(2,:))+0.05]), ...
    zlim([min(waypoints(3,:)) max(waypoints(3,:))])
xlabel('X');
ylabel('Y');
zlabel('Z');
```

```
title('Waypoints');

% Trajectories positions, trapezoidal velocity profile and acceleration
% profile.
figure(3)
subplot(3,1,1)
plot(trajTimes, q)
xlabel('Time (s)')
ylabel('Positions (m)')
legend('X','Y', 'Z')
subplot(3,1,2)
plot(trajTimes, qd)
xlabel('Time (s)')
ylabel('Velocities (m/s)')
legend('X','Y', 'Z')
subplot(3,1,3)
plot(trajTimes, qdd)
xlabel('Time (s)')
ylabel('Accelerations (m/s^2)')
legend('X','Y', 'Z')
```

```
function waypoints = generateWaypoints(inputString, offset, toolPositionHome, scale)
    % GENERATEWAYPOINTS - Generate waypoints for a trajectory to be executed by ...
        the Niryo Ned 2 robot.
    %
    %   This function takes as input a string to be traced, the offset between ...
        each letter,
    %   the starting position of the robotic arm, and the size of the letters. It ...
        generates
    %   a set of points constituting waypoints for computing a trajectory to be ...
        executed by
    %   the Niryo Ned 2 robot. The waypoints are positioned at the center of the ...
        starting
    %   position, and a check is performed to determine whether the trajectory points
    %   are within the robot's operational space and thus reachable.
    %
    %   Input:
    %       inputString - String to be traced (e.g., 'poliba')
    %       offset - Offset between each letter
    %       toolPositionHome - Starting position of the robotic arm
    %       scale - Size of the letters
    %
    %   Output:
    %       waypoints - Matrix containing the coordinates of the generated waypoints

    % Define the mapping of letters to their respective sets of coordinates
    coordinates = createCoordinates();
    coordinateMap = containers.Map('KeyType', 'char', 'ValueType', 'any');
    coordinateMap('p') = [coordinates.pCoordinates];
    coordinateMap('o') = [coordinates.oCoordinates];
    coordinateMap('l') = [coordinates.lCoordinates];
    coordinateMap('i') = [coordinates.iCoordinates];
    coordinateMap('b') = [coordinates.bCoordinates];
    coordinateMap('a') = [coordinates.aCoordinates];

    % Initialize the coordinates of the waypoints
    waypoints = [];

    % Calculate the coordinates for each letter in the input string
    for i = 1:length(inputString)
        letter = inputString(i);
        if isKey(coordinateMap, letter)
            coordinates = coordinateMap(letter);
            if isempty(waypoints)
```

```
                waypoints = coordinates;
            else
                coordinates(:,2) = coordinates(:,2) + min(waypoints(:,2)) - offset;
                waypoints = [waypoints; coordinates];
            end
        end
    end

    % Calculate the center of the generated waypoints
    center_y = mean(waypoints(:, 2));

    % Calculate the offset to center the waypoints around toolPositionHome on the ...
        y-axis
    offset_center_y = toolPositionHome(2) - center_y;

    % Adjust the waypoints to be centered around toolPositionHome only on the y-axis
    waypoints(:,2) = waypoints(:,2) + offset_center_y;

    % Calculate the waypoints
    waypoints = toolPositionHome(1:3)' + waypoints' ./ scale;

    % Ensure that the waypoints are within the operational space

    % Define the radius of the robot's workspace
    radius_workspace = 1.008/2; % m

    % Boolean variable for control
    checkbool = 0;

    % Check if waypoints are within the operational space
    for i = 1:size(waypoints, 2)
        x = waypoints(1, i);
        y = waypoints(2, i);

        % Calculate the upper limits allowed within the robot's operational space
        % based on the specified workspace radius
        x_limit = sqrt(radius_workspace^2 - y^2);
        y_limit = sqrt(radius_workspace^2 - x^2);

        % Limit verification
        if x > -x_limit && x < x_limit && y > -y_limit && y < y_limit
        else
            checkbool = 1;
        end
    end

    % Display a warning if waypoints might exceed the workspace
    if checkbool == 0
        disp('The waypoints are inside the workspace.');
    else
        disp('Attention: The waypoints might exceed the workspace.');
    end

end
```

```
function [coordinateStruct] = createCoordinates()
    % CREATECOORDINATES - Generate a structure containing the coordinates of the ...
        letters.
    % This function generates a structure containing the coordinates of the
    % letters used. Each letter is represented by a set of coordinates,
    % which are calculated using various trace functions.
    %
    % Output: coordinateStruct - Structure containing coordinates of each letter.

```

```
    % Define constants
    zOffset = 0.08;
    numPointsCircle = 32;
    numPointsSegment = 16;

    % Define coordinates for each letter

    % Letter 'P'
    pCircle = trace_circumference(numPointsCircle,[0.45 0 0],0.15, 360, 180);
    pSegment = trace_segment(numPointsSegment, [0 0 0], [0.59 0 0]);
    pCoordinates = [pSegment(1,1) pSegment(1,2) zOffset; pSegment; pCircle; ...
        pCircle(end,1) pCircle(end,2) zOffset];

    % Letter 'O'
    oCircle = trace_circumference(numPointsCircle*2,[0.15 -0.15 0], 0.15, 0, 360);
    oCoordinates = [oCircle(1,1) oCircle(1,2) zOffset; oCircle; oCircle(end,1) ...
        oCircle(end,2) zOffset];

    % Letter 'L'
    lSegment = trace_segment(numPointsSegment, [0.5 0 0], [0 0 0]);
    lCoordinates = [lSegment(1,1) lSegment(1,2) zOffset; lSegment; ...
        lSegment(end,1) lSegment(end,2) zOffset];

    % Letter 'I'
    iSegment = trace_segment(numPointsSegment, [0 0 0], [0.3 0 0]);
    iCoordinates = [iSegment(1,1) iSegment(1,2) zOffset; iSegment; ...
        iSegment(end,1) iSegment(end,2) zOffset; 0.4 0 zOffset; 0.4 0 0; 0.4 0 ...
        zOffset];

    % Letter 'B'
    bCircle = trace_circumference(numPointsCircle,[0.15 0 0],0.15, 180, 360);
    bSegment = trace_segment(numPointsSegment, [0.5 0 0], [0.01 0 0]);
    bCoordinates = [bSegment(1,1) bSegment(1,2) zOffset; bSegment; bCircle; ...
        bCircle(end,1) bCircle(end,2) zOffset];

    % Letter 'A'
    aCircle1 = trace_circumference(numPointsCircle,[0.25 -0.125 0],0.125, 90, -90);
    aCircle2 = trace_circumference(numPointsCircle,[0.1 -0.1 0],0.1, 180, 0);
    aSegment1 = trace_segment(numPointsSegment, [0.24 -0.25 0], [0 -0.25 0]);
    aSegment2 = trace_segment(numPointsSegment, [0 -0.24 0], [0 -0.099 0]);
    aSegment3 = trace_segment(numPointsSegment, [0.2 -0.11 0], [0.2 -0.25 0]);
    aCoordinates = [aCircle1(1,1) aCircle1(1,2) zOffset; aCircle1; aSegment1; ...
        aSegment2; aCircle2; aSegment3];

    % Organize coordinates into a cell array
    coordinateCell = {
        pCoordinates;
        oCoordinates;
        lCoordinates;
        iCoordinates;
        bCoordinates;
        aCoordinates
    };

    % Create the structure
    coordinateStruct.pCoordinates = coordinateCell{1};
    coordinateStruct.oCoordinates = coordinateCell{2};
    coordinateStruct.lCoordinates = coordinateCell{3};
    coordinateStruct.iCoordinates = coordinateCell{4};
    coordinateStruct.bCoordinates = coordinateCell{5};
    coordinateStruct.aCoordinates = coordinateCell{6};
end
```

```
function points = trace_segment(points_number, start_point, end_point)
```

```
    % TRACE_SEGMENT - Generate points along a line segment between two 3D points.
    %   points = trace_segment(points_number, start_point, end_point) generates
    %   a set of points along the line segment between start_point and end_point.
    %   points_number is the number of points to generate.
    %   start_point and end_point are 3-dimensional vectors defining the start
    %   and end of the line segment, respectively.

    % Check if start_point and end_point are 3-dimensional vectors
    if numel(start_point) ~= 3 || numel(end_point) ~= 3
        error('Start and end points must be 3-dimensional vectors.');
    end

    % Calculate the direction vector from start_point to end_point
    direction_vector = (end_point - start_point) / norm(end_point - start_point);

    % Calculate the distance between start_point and end_point
    distance = norm(end_point - start_point);

    % Calculate the coordinates of the points along the segment
    points = zeros(points_number, 3);
    for i = 1:points_number
        % Calculate the position of each point along the segment
        points(i, :) = start_point + direction_vector * ((i - 1) / (points_number ...
            - 1)) * distance;
    end
end
```

```
function points = trace_circumference(points_number, center, radius, ...
    start_angle, end_angle)
    % TRACE_CIRCUMFERENCE - Generate points along the circumference of a circle.
    %   points = trace_circumference(points_number, center, radius, ...
        start_angle, end_angle)
    %   generates a set of points along the circumference of a circle.
    %   points_number is the number of points to generate.
    %   center is a 3-dimensional vector defining the center of the circle.
    %   radius is the radius of the circle.
    %   start_angle and end_angle define the range of angles (in degrees)
    %   to generate points on the circumference.

    % Check if center is a 3-dimensional vector
    if numel(center) ~= 3
        error('Center point must be a 3-dimensional vector.');
    end

    % Convert angles from degrees to radians
    start_angle = deg2rad(start_angle);
    end_angle = deg2rad(end_angle);

    % Angles to generate points on the circumference
    theta = linspace(start_angle, end_angle, points_number);

    % Calculate the coordinates of the points
    points = [center(1) + radius*cos(theta);
              center(2) + radius*sin(theta);
              repmat(center(3), 1, points_number)]';
end
```