Matlab Robotics Toolbox Manual

TA: Carlo Cenedese - c.cenedese@rug.nl - office: 5117.0116

Introduction

In this document we will provide a brief introduction to the Robotic Toolbox for Matlab developed by Peter Corke in [1], in particular the commands used to create the model of a robotic arm. A more exhaustive description of the commands that will be presented in this document, together with some more advanced examples, can be found in [2, Part III].

On-line it is very easy to find several tutorials that use this toolbox, since its worth is widely recognize. We encourage the reader to experiment with this powerful tool, even with the subject not introduced in the course, you find official video lectures of all the contents in the toolbox in the site https://robotacademy.net.au/. We want to stress that it is very important to take these tutorials with a grain of salt, since they could adopt different conventions than the ones used during the course.

The document is organized in the following sections.

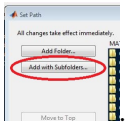
Contents

I. Getting Started

The Toolbox is open source and its latest version (at the moment v. 10.2) is available at Peter Corke's site or by clicking this direct link, that initiates immediately the download.

Once the download is completed, unzip the folder in a `"destination folder"`. To finalize the installation, we add the folder just created to the path of Matlab, there are two ways to do that:

1. Execute the script `startup_ rvc.m` that you find in `"destination folder" \ robot-10.2 \ rvctools` .
2. Manually add the folder, and all its sub-folders, to the Matlab path. Click on 

and then in the windows that appears again on the "Add with Subfolder" button 

. Finally select the folder in which you unzipped the downloaded files, and then save the new path obtained.

In order to check if everything works properly, you can execute the script `rtbdemo.m`, from the Matlab workspace (notice that, since you added the path, you should be able to run this script form any folder). If a window like the one in Figure 1 appears, then it means that the installation was successful.

Figure 1: The window obtained running the script `rtbdemo.m` .

This interface provides several possible applications of the Toolbox in different areas of robotics, the ones associated to robotic arms can be found in the "Robot" column.

A complete description of all the command in the Toolbox can be found in the manual robot.pdf (3.8 MB). It is auto-generated from the comments in the MATLAB code. You can find this in the Toolbox as `rvctools/robot/robot.pdf`. The Toolbox documentation also appears in the MATLAB help browser under Supplemental Software.

II. Create a Robotic Arm from D.H. parameters

I. Problem Introduction

In this section we will construct the model of a robotic arm starting from the Denavit–Hartenberg (DH) parameters. As explanatory example, we use the RPR robotic arm in [3, Example 3.4], made by both revolute and prismatic joints (Figure 2).

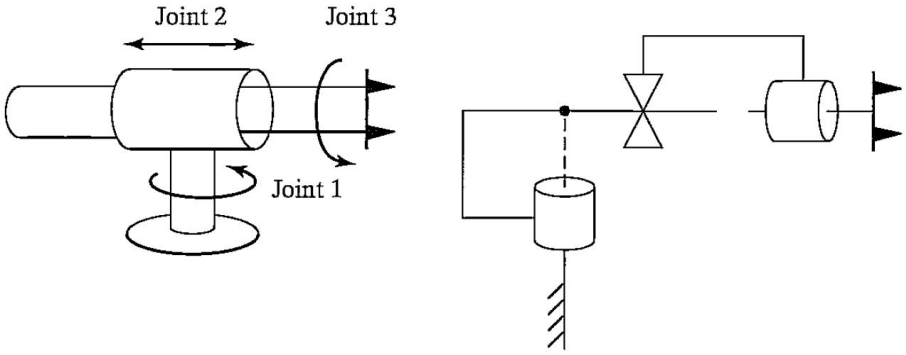


Figure 2: The graphical representation of the robotic arm in [3, Example 3.4]

The DH parameters associated to the arm are shown in Table I.

Table I. DH parameters of the robotic arm in [3, Example 3.4]

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	0	θ_1
2	$\frac{\pi}{2}$	0	d_2	0
3	0	0	L_2	θ_3

II. Step by step model construction

II.1 Main commands

The cornerstone function of the Toolbox, needed to generate a model of a robotic arm, is `Link()`, it creates a Link object in which all the characteristics of the joint are specified. Type `help Link` in the workspace for a complete guide of the function.

```
>> L = Link([0, 0.5, 0.3, pi/2, 0])
L =
Revolute(std): theta=q, d=0.5, a=0.3, alpha=1.5708, offset=0
```

The above code is the most intuitive way to create a Link object, the first four argument are the DH parameters $[\theta, d, a, ,\alpha]$ the fifth, optional, parameter σ stands for the type of joint considered, namely

$$\sigma = \begin{cases} 0 & \text{Revolute} \\ 1 & \text{Prismatic} \end{cases} . \tag{1}$$

Notice that even though θ is imposed equal to a numerical value, it is not displayed, since, in the revolute case, it is the joint variable. Its value can be considered as a place-holder for the function, analogously in the prismatic case the joint variable is d.

Using this function and the DH parameters, we can define any type of joint, e.g.

```
>> L = Link([0.1, 0.7, 1, pi/4, 1])
L =
Prismatic(std): theta=0.1, d=q, a=1, alpha=0.785398, offset=0
```

In order to create a Link object with several joints, we use the classical syntax for structures in Matlab. For example if we want to create an object with three joints, we simply write

```
>> L(1) = Link([0, 0.5, 0.3, pi/2, 0]);
>> L(2) = Link([0.1, 0.7, 1, pi/4, 1]);
>> L(3) = Link([0.4, 0.4, 0.5, 0, 0]);
>> L
L =
Revolute(std):  theta=q1   d=0.5         a=0.3         alpha=1.571       offset=0
Prismatic(std): theta=0.1           d=q2 a=1           alpha=0.7854      offset=0
Revolute(std):  theta=q3   d=0.4         a=0.5         alpha=0           offset=0
```

To define the same object we can also use two alternative syntaxes

```
L = Link('revolute', 'd', 1.2, 'a', 0.3, 'alpha', pi/2);
L = Revolute('d', 1.2, 'a', 0.3, 'alpha', pi/2);
```

In these cases the joint variable is omitted.

The outputs of all the previous commands present the notation `(std)`, because it is used the **standard convention** for the DH parameters. During the lectures we always adopted the **modified convention**, therefore, to stay consistent, we have to use the additional parameter `'modified'` during the creation of each Link object. So the possible syntaxes to create a revolute (or prismatic) joint adopting the modified convention are the following:

```
L = Link([0, 1.2, 0.3, pi/2, 0], 'modified');
L = Revolute('d', 1.2, 'a', 0.3, 'alpha', pi/2, 'modified');
L = Link('revolute', 'd', 1.2, 'a', 0.3, 'alpha', pi/2, 'modified')
>> L
Revolute(mod): theta=q, d=1.2, a=0.3, alpha=1.5708, offset=0
```

"The pose of the robot with zero joint angles is frequently some rather unusual (or even mechanically unachievable) pose.[...] This is a consequence of constraints imposed by the Denavit-Hartenberg formalism. A robot control designer may choose the zero-angle configuration to be something more obvious. The joint coordinate offset provides a mechanism to set an arbitrary configuration for the zero joint coordinate case." from [2, Section 7.5.1].
For every Link objectit is possible to specify an offset either during the construction, i.e.

```
L = Link([0, 1.2, 0.3, pi/2, 0, pi/2],'modified');
L = Revolute('d', 1.2, 'a', 0.3, 'alpha', pi/2, 'modified','offset',pi/2);
L = Link('revolute', 'd', 1.2, 'a', 0.3, 'alpha', pi/2, 'modified','offset',pi/2)
>> L
Revolute(mod): theta=q, d=1.2, a=0.3, alpha=1.5708, offset=1.5708
```

or directly accessing the object parameter

```
>> L.offset = pi/2;
```

Once all the joints are added into a Link object, we finalize the creation of the arm passing the created object to the SerailLink constructor. For example for a 2 joints arm we proceed as follows, first we define the Link object `two_joint`

```
>> two_link(1) = Link([0, 0.5, 0.3, pi/2, 0, 0],'modified');
>> two_link(2) = Link([0.1, 0.7, 1, pi/4, 1, 0],'modified');
```

then, we just use it as first argument of the SeralLink constructor. The second input of the constructor is the name of the arm, hence the syntax to be used is

```
>> tony = SerialLink(two_link, 'name', 'tony')

tony =

tony:: 2 axis, RP, modDH, slowRNE
+---+-----------+-----------+-----------+-----------+-----------+
| j |     theta |         d |         a |     alpha |    offset |
+---+-----------+-----------+-----------+-----------+-----------+
|  1|         q1|        0.5|        0.3|     1.5708|          0|
|  2|        0.1|         q2|          1|   0.785398|          0|
+---+-----------+-----------+-----------+-----------+-----------+
```

The output provided by the constructor is a quick description of all the main features of the arm built. The SerialLink class is the founding element used in the Toolbox to model a robotic arm. In the following sections we explore some of the different methods provided by the Toolbox to interact with the model of the arm just created.

II.2 Implementation of Example 3.4

Here we propose the complete code needed to create the arm of [3, Example 3.4], thereafter we refer to this file as `Example_34.m`. In the implementation we give an arbitrary value to the variable L_2 of Table I, namely $L_2 = 10cm$. The code is composed just by two steps:

- construct the Link object by passing to the constructor the DH parameters of Table I;
- create the arm model with the SerialLink constructor, the input is the Link object just obtained.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%   Example 3.4 from Introduction to Robotics        %%
% by: Carlo Cenedese, RUG University - c.cenedese@rug.nl
% created: 8-Aug-2018
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all
close all
%% Create the Link object, with the 3 joints from DH table
L_2 = 10;
L(1) = Link('revolute','d',0 , 'a', 0, 'alpha', 0, 'modified','offset',0);
L(2) = Link('prismatic','theta',0,'a',0,'alpha',pi/2,'modified','offset',0);
```

```
L(3) = Link('revolute','d',L_2 ,'a',0,'alpha',0,'modified','offset',0);
%% Create the SeralLink Object
mArb = SerialLink(L,'name','jarvis');
%% Plot the arm
mArb.plotopt={'workspace', [-20 20 -20 20 -10 20]};
mArb.qlim(2,:) = [0, 10];
figure(1)
mArb.plot([0, 0, 0]) % zero configuration
%mArb.plot([pi/4, 5, pi/4]) % random configuration
figure(2)
mArb.teach
```

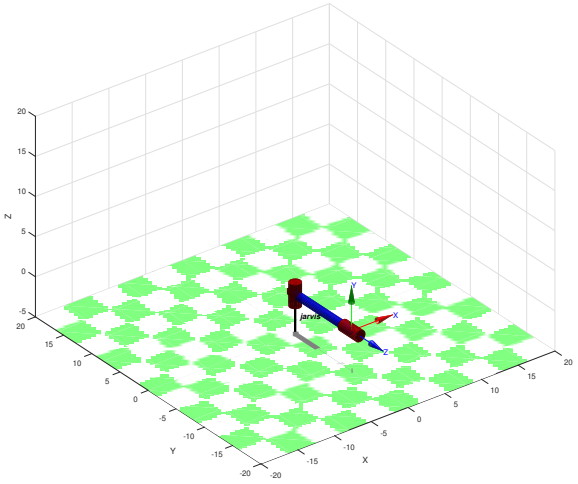
In the last part of the code we introduced the `plot()` function[1]. The function takes as input the pose of the robotic arm in which the latter has to be depicted (Figure 3). Finally the `teach` function creates an interactive model of the arm, indeed on the side of the figure there are some sliders that can be used to control the joint variables singularly. This is a convenient tool to verify if the model reflects the movements of the real arm.

II.3 Model of the end-effector

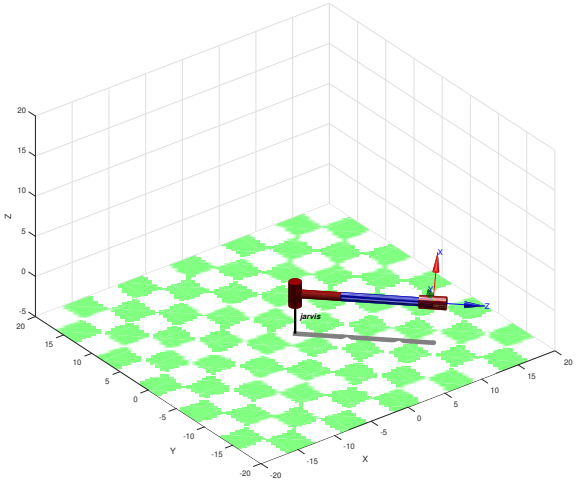
Finally, if we want to model the gripper of a robotic arm, that does not coincide with the last joint, we use the option `tool`, that each SerialLink object has. In fact, we directly specify the transformation matrix ${}^{\text{LJ}}_{E}T$ ("LJ" stands for last joint and "E" for end-effector) as follows

$$\texttt{"SerialLink\_object".tool} = {}^{\text{LJ}}_{E}T \quad .$$

[1] This is not the standard plot function, but one of the many method that every SerialLink object inherit from the class.



(a) The zero configuration without offsets



(b) Random configuration $[\pi/4,\ 5,\ \pi/4]$

Figure 3: Arm in different poses

III. Forward Kinematics

Another useful function, that every SerialLink object is equipped with, is the forward kinematics; in the following we will use the arm named `jarvis`, defined in the previous section.

The forward kinematics can be computed through the command `fkine()`; for a robot with N joints, it takes as input a configuration $q \in \mathbb{R}^N$ and returns the homogeneous transformation matrix ${}_E T$ representing the pose of the last joint of the arm in the base frame.

The syntax used is

$$ {}_E T = \texttt{"SerailLink\_object".fkine}(q)\,. $$

Notice that the object name is, in general, different from the arm name (in the implementation of `Example_34.m` we called the SerialLink object `mArb` and the arm `jarvis`).

```
>> mArb.fkine([0, 0, 0])

ans =
         1         0         0         0
         0         0        -1       -10
         0         1         0         0
         0         0         0         1

>> mArb.fkine([pi/4, 5, pi/4])

ans =
    0.5000   -0.5000    0.7071     10.61
    0.5000   -0.5000   -0.7071    -10.61
    0.7071    0.7071         0         0
         0         0         0         1
```

IV. Inverse Kinematics

The inverse kinematics is the converse of the forward kinematics, indeed, starting from a desired position and orientation of the end-effector (expressed as ${}_E T$), it provides a joint configuration $q \in \mathbb{R}^N$ to reach it, if possible.

$$ q = \texttt{"SerailLink\_object".ikine}({}_E T \texttt{,'options'})\,. $$

In general this function is not unique and for some classes of manipulator does not exist a closed form solution, hence, if possible, a numerical solution is computed in those cases.

I. Under-actuated arms

Given a transformation matrix ${}_E T$, the inverse kinematic has to compute a configuration q that leads to either a specified position $[p_{x,E},\ p_{y,E},\ p_{z,E}]$ of the end effector, and also a precise orientation $[r_{x,E},\ r_{y,E},\ r_{z,E}]$. It is easy to understand now why the method cannot find a solution for every transformation matrix, indeed if the robotic arm does not have 6-d.o.f., i.e. it is under-actuated, it cannot match all the constraints.

The Toolbox allows to compute a sub-optimal solution in this cases, using the `'mask'` option; we will illustrate how does it work by mean of the `jarvis` arm. From Figure 3a, we see that the

first two link of the arm provide movements along X_E and Z_E but none along Y_E (these are the axis in the end-effector frame), therefore any position of the form $\hat{\boldsymbol{p}}_{\boldsymbol{E}} = [\hat{p}_{x,E},\ 0,\ \hat{p}_{z,E}]$ is reachable, if $\hat{p}_{z,E} \geq L_2$[2]. Finally the third link allows a rotation around Z_E, hence all the orientations of the type $\hat{\boldsymbol{r}}_{\boldsymbol{E}} = [0,\ 0,\ \hat{r}_{z,E}]$ are attainable. This is an additional information achieved by an analysis of the arm, to pass it to the `ikine` method, we use a 1×6 vector `M`, called mask vector. Each one of its component is a boolean value (0 or 1) associated to $\hat{\boldsymbol{v}}_E := [\hat{\boldsymbol{p}}_{\boldsymbol{E}},\ \hat{\boldsymbol{r}}_{\boldsymbol{E}}]$, in fact if an element of `M` is 0, than the associated constrain generated by $\hat{\boldsymbol{v}}_E$ is not taken into consideration during the computation of the inverse kinematics, and vice versa if the value is 1. In this particular example, we know from the previous reasoning that the mask vector is $[1,\ 0,\ 1,\ 0,\ 0,\ 1]$, indeed we are interested only into the correct value of $p_{x,E}$, $p_{z,E}$ and $r_{z,E}$.

We use, as input of the inverse kinematics, the transformation matrix computed in the previous section through the forward kinematics. As second input, before the mask option, we specify the initial configuration of the arm, in this case $[0,\ 0,\ 0]$.

```
>> qn = [pi/4, 2 , pi/4];
   T =  mArb.fkine(qn);
   qi = mArb.ikine(T,[0 0 0],'mask',[1 0 1 0 0 1]);
>> qn

qn =

    0.7854    2.0000    0.7854

>> qi

qi =

    0.7854    2.0000    0.7854
```

As expected the pose `qi` obtained by the inverse kinematics is equal to desired one `qn`. Notice that, if we feed to the inverse kinematics a point that cannot be reached, the method still provides a solution that matches in the best way the three relevant constraints defined by the mask vector.

Finally, if we don't provide a mask vector for an under-actuated arm, the `ikine` command throws an error

```
>> qi = mArb.ikine(T);
Error using  SerialLink/ikine (line 164)
Number of robot DOF must be >= the same number of 1s in the mask matrix
```

A general rule, to make the function compile normally, is that the number of 1s in the mask vector has to be less or equal the number of d.o.f of the arm (for `jarvis` that is 3).

II. Fully actuated arms

"A necessary condition for a closed-form solution of a 6-axis robot is that the three wrist axes intersect at a single point. This means that motion of the wrist joints only changes the end-effector orientation, not its translation. Such a mechanism is known as a spherical wrist and almost all industrial robots have such wrists." from [2, Section 7.3.1].

[2]Because the state of the prismatic joint cannot have negative values.

An example of this type of arm is the Puma robot, a model of this arm is built in the Toolbox (notice that the implementation adopts the standard convention for the DH parameters). In this case a closed form of for the inverse kinematics exists and can be computed through the method `ikine6s`. The following code shows how to proceed to obtain the model and compute the inverse kinematics.

```
>> mdl_puma560
>> qn = [0 0 pi/2 pi pi/4 pi]

qn =

         0         0    1.5708    3.1416    0.7854    3.1416

>> T = p560.fkine(qn)


T =
    0.7071         0   -0.7071         0
         0         1         0   -0.1501
    0.7071         0    0.7071    0.0203
         0         0         0         1
>> qi = p560.ikine6s(T)

qi =

    0.0000    0.0000    1.5708   -3.1416    0.7854    3.1416
```

Notice that, if `qn` $\neq$ `qi` it means, in this case, that more than one configuration lead to the target position (try with `qn` $= [0, 0, 0.7854, 3.1416, 0, 0.7854]$).

V. Trajectory Planning

A trajectory can be planned in two different spaces, the joint space and the Cartesian one. In the first case the construction of the trajectory is easier but the actual movement of the end effector can be difficult to predict a priori (especially with complex robotic arms), in the second we plan exactly the path that the end-effector has to follow, but the trajectory of each joint can be complex and therefore challenging for the actuators.

I. Joint space

In this framework, we want the arm to change its configuration from `q1` to `q2`[3]. The function of the Toolbox used is `jtraj` and its syntax is

$$[q, \dot{q}, \ddot{q}] = \texttt{jtraj(q1, q2, t)}, \tag{2}$$

where `t` is a $1 \times T$ vector of the time instants and T is the number of time instants. The output are: the positions, velocities and accelerations that each joint has to adopt at each moment in order to

[3]Notice that to obtain a desired final position in the Cartesian space we can compute `q2` via the inverse kinematics

perform the desired movement, therefore q, $\dot{q}$, $\ddot{q}$ are $T \times N$ vectors where N is the number of links of the arm.

Once again, we use `jarvis` as test bench. The starting configuration chosen is `q1` $= [0, 8, 0]$ and the ending one `q2` $= [\frac{\pi}{2}, 3, 2\pi]$. The trajectory and the according animation of the arm are obtained as follows,

```
>> t = 0:0.05:2;
   [q, qd, qdd] = jtraj( q1, q2, t);
   mArb.plot(q)
```

The position in the Cartesian space of the end-effector is obtained using the forward kinematics, feeding as input of `fkine` the complete vector q; the output is a structure of T transformation matrices that define the pose of the end-effector at each moment. Finally the position 3-D position of the end-effector, i.e. the first 3 element of the last column of each transformation matrix obtained by `fkine`, is obtained using the command `transl`.

```
>> T_traj = mArb.fkine(q);
   p_E = transl(T_traj);
```

We show now in Figure 4[4] the plot of q, $\dot{q}$ and p_E versus the time, for the above example. In

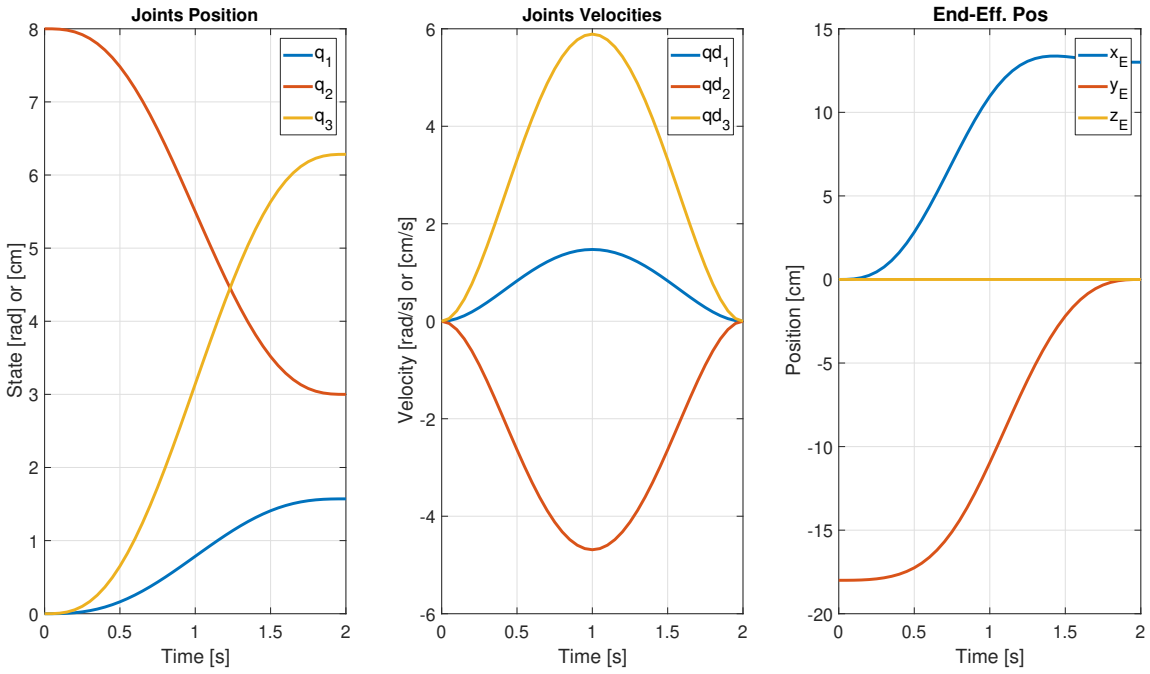


Figure 4: Trajectories planned in the joint space for the RPR manipulator `jarvis`. The plots represents respectively the joint position, velocity and the end effector position w.r.t. the time.

the XY-plane the actual path followed by the end-effector is the one depicted in Figure 5.

Wrapping Up (trajectory planning in the joint space)

[4]Notice that in the first two plot, the position (velocity) is in radians or centimetres (radians over second or centimetres over seconds), if respectively the joint is revolute or prismatic.

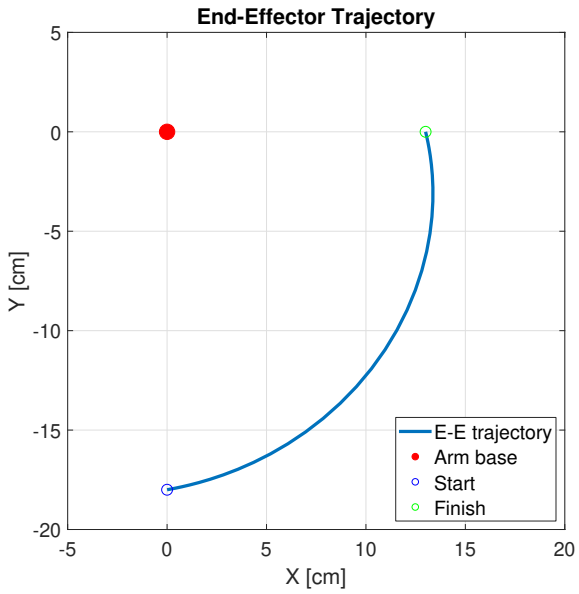


Figure 5: End-effector trajectory, obtained from a planning in the joint space, in the XY-plane for the RPR arm `jarvis`.

- Pros:
  - + given any initial and final pose, the trajectory planned is always implementable;
  - + the trajectories to be implemented by the joints are smooth;
  - + the velocities and accelerations required to the joints are lower.
- Cons:
  - – the path followed by the end-effector can be not clear a priori;
  - – it is difficult to deal with external objects, e.g. avoid obstacles or perform precise movement with the end effector.

II. Cartesian space

In this case we plan directly the path of the end-effector using the function `ctraj`, it takes as input the initial and final pose, defined through two transformation matrices `T1` and `T2`, and the number of time instants that the movement has to last. The output is instead a sequence of T transformation matrix `T_ctraj`. The syntax that has to be used is

$$\texttt{T\_ctraj} = \texttt{ctraj(T1, T2, } T\texttt{)}\,, \quad \text{with } T := \texttt{length(t)}\,.$$

The position of the end-effector is obtained directly from `T_ctraj`, on the other hand to retrive the joints states we need to use the inverse kinematics.

```
>> T_ctraj = ctraj(T_1, T_2, length(t));
   p_E_ctraj = transl(T_ctraj);
   q_ctraj = mArb.ikine(T_ctraj,'mask',[1 0 1 0 0 1]);
   mArb.plot(q_ctraj)
```

The method `ctraj` create a straight path, in the Cartesian space, between the initial position and the final one. Notice that it is not guarantee by any means that all the point of the path are reachable by the end-effector, hence it is important to analyse a priori the feasibility of the movement.

Using the `jarvis` arm and the following two transformation matrices (obtained from the forward kinematics of `q1` and `q2`),

$$\texttt{T1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & -18 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \texttt{T2} = \begin{bmatrix} 0 & 0 & 1 & 13 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

we obtain the evolution of the joints' state and the position of the end-effector shown in Figure 6. The trajectories of the joints are more complex than the ones obtained in the previous section, because of the straight motion of the end-effector in the Cartesian space. These differences become more evident the more complex the arm is. Finally the movement of the end-effector in the

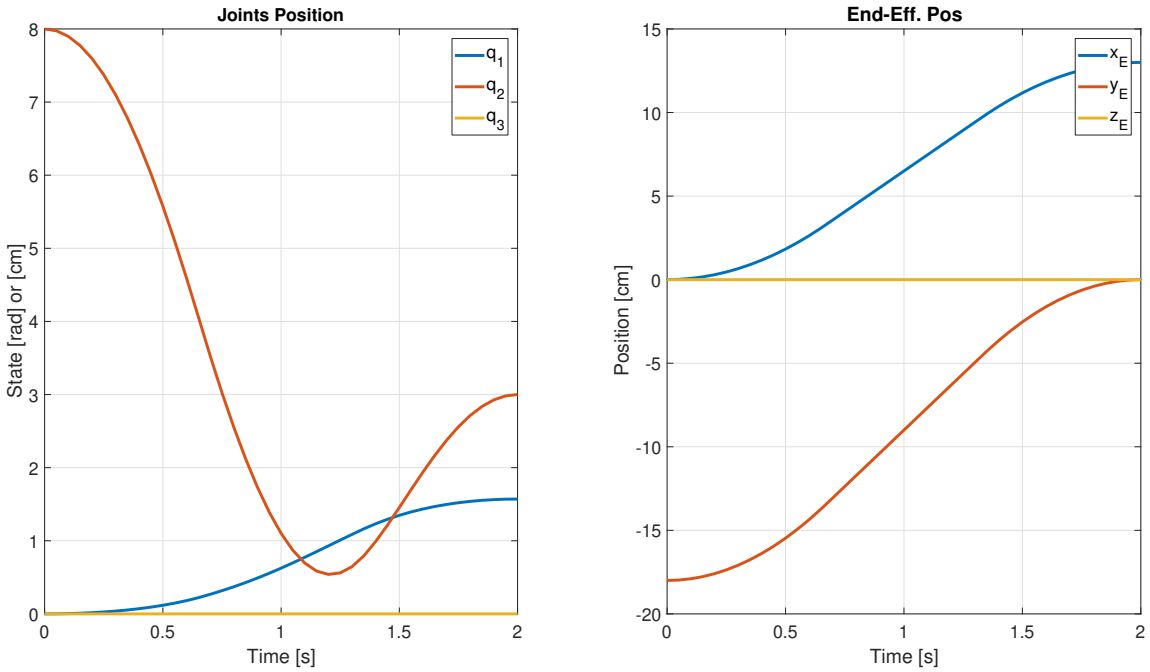


Figure 6: The joint and end effector positions w.r.t. the time, planned in the Cartesian space for the RPR arm `jarvix`.

XY-plane is reported in Figure 7, as sought it is a straight line between the starting and the ending point.

Wrapping up, trajectory planning in Cartesian space

- Pros:
  - + direct knowledge of the full trajectory of the end-effector;
  - + smooth movement of the end-effector.

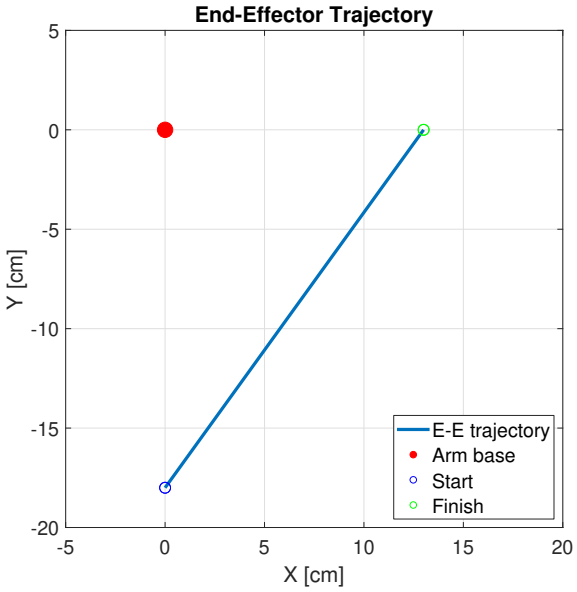


Figure 7: The end-effector trajectory in the XY-plane, planned in the Cartesian space for the RPR arm `jarvix`.

- Cons:
  - the trajectory that the joints have to follow is complex;
  - usually leads to a more complex practical implementation, since complex trajectories imply stress on the arm motors;
  - given a reachable initial and final point it is not guaranteed that path obtained is composed only by reachable points, i.e. it is implementable by the arm. Therefore a knowledge about the operational space of the arm is required.

VI. Jacobian

The final function of the Toolbox that we discuss is `jacob0`, it calculates the Jacobian of a given arm for a specific pose q in the global framework. The syntax, that is used to call it, is

$$J = \texttt{"SerailLink\_object".jacob0}(q)\,.$$

Exploiting once last time our exemplifying arm, we can calculate the Jacobian for the `q2` pose as follows

```
>> q2 = [pi/2, 3, 2*pi ];
   J2 = mArb.jacob0(q2)

J2 =

    0.0000    1.0000         0
   13.0000   -0.0000         0
         0    0.0000         0
```

```
     0         0    1.0000
     0         0   -0.0000
1.0000         0    0.0000
```

This command can be very useful in the implementation of a numerical approximation of the inverse kinematics for under-actuated arms.

VII. Conclusion

This concludes the brief overview of part of the Matlab Robotics Toolbox regarding the robotic arms. We strongly suggest to experiment with the powerful tools provided by this software, especially to gain a better grasp of why and how certain calculation are performed. For more advanced examples you can directly go to the site of the author of the Toolbox Peter Corke.

References

[1] P. Corke, "A robotics toolbox for MATLAB," *IEEE Robotics and Automation Magazine*, vol. 3, no. 1, pp. 24–32, Mar. 1996.

[2] ——, *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, 1st ed. Springer Publishing Company, Incorporated, 2013.

[3] J. J. Craig, *Introduction to Robotics: Mechanics and Control*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

Appendix: Complete Matlab code

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%    Example 3.4 from Introduction to Robotics                %%
% by: Carlo Cenedese, RUG University - c.cenedese@rug.nl
% created: 8-Aug-2018
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all
close all
% Create the Link object, with the 3 joints from DH table
% dh = [THETA D A ALPHA SIGMA OFFSET]
L_2 = 10;

L(1) = Link('revolute','d',0 , 'a', 0, 'alpha', 0, 'modified','offset',0);
L(2) = Link('prismatic','theta',0,'a',0,'alpha',pi/2, 'modified','offset',0);
L(3) = Link('revolute','d',L_2 ,'a',0,'alpha',0, 'modified','offset',0);
% Create the SeralLink Object
mArb = SerialLink(L,'name','jarvis');
% Plot the arm
mArb.plotopt={'workspace', [-20 20 -20 20 -10 20]};
mArb.qlim(2,:) = [0, 10];
figure(1)
%mArb.plot([0, 0, 0]) % zero configuration
mArb.plot([0, 5, pi/4]) % random configuration
% figure(2)
```

```
% mArb.teach

%% Inverse Kinematics
% Example of inverse kinematics for the under actuated arm 'jarvis'

% A random pose
qn = [pi/4, 2 , pi/4];
T =  mArb.fkine(qn); % compute the transition matrix using the forward kin.
                     % the obtained cartesian point is surelly reachable.

% We can define a generic transformation matrix using
%  T = transl([5 5 0]);

% Inverse kinematic for an under actuated arm, using 'mask'
qi = mArb.ikine(T,'mask',[1 0 1 0 0 1]); % The pose to reach the
                                         % cartesian point
                                         % defined by T
Ti = mArb.fkine(qi);% Check the result,  Ti = T almost always since 'qn'
                    % is for sure reachable

% Plot of the 2 pose obtained
mArb_clone = SerialLink(L,'name','tony'); % clone of the arm
figure(3)
mArb_clone.plotopt={'workspace', [-20 20 -20 20 -10 20]};
mArb_clone.plot(qn) % plot the clone arm with config. 'qn'
figure(4)
mArb.plot(qi)% plot the original arm with config. 'qn'

%% Trjectory planning (joint space)
% we feed to the arm a trajectory to move from a configuration q1 to q2

q1 = [0, 8, 0]; % starting configuration
q2 = [pi/2, 3, 2*pi ]; % ending configuration
t = 0:0.05:2; % time instants

[q, qd, qdd] = jtraj( q1, q2, t); % trjectory: [pos., vel., acc.]

% obtain the cartesian position
T_jtraj = mArb.fkine(q); % An object with length(q) number of
                         % transformation matrices

% extract the translational part
p_E = transl(T_jtraj);

% trajectory animation
mArb.plotopt={'workspace', [-30 30 -30 30 -10 20]};
figure(5)
mArb.plot(q)

% plot of the joint trajectories
figure(6)
subplot(1,3,1)
```

```
plot(t,q(:,1),'LineWidth',3); hold on; plot(t,q(:,2),'LineWidth',3)
plot(t,q(:,3),'LineWidth',3); grid on;title('Joints Position')
legend('q_1','q_2','q_3');hold off
subplot(1,3,2)
plot(t,qd(:,1),'LineWidth',3); hold on; plot(t,qd(:,2),'LineWidth',3)
plot(t,qd(:,3),'LineWidth',3); grid on;title('Joints Velocities')
legend('qd_1','qd_2','qd_3');hold off
subplot(1,3,3)
plot(t,p_E(:,1),'LineWidth',3); hold on; plot(t,p_E(:,2),'LineWidth',3);
plot(t,p_E(:,3),'LineWidth',3);grid on;title('End-Eff. Pos')
legend('x_E','y_E','z_E');hold off

% plot in 2-D of the End-Effector position
figure(7)
plot(p_E(:,1),p_E(:,2),'LineWidth',3)
hold on
scatter(0,0,'filled','MarkerEdgeColor','r','SizeData',200,...
    'MarkerFaceColor','r') % plot the base of the arm
scatter(p_E(1,1),p_E(1,2),'MarkerEdgeColor','b','SizeData',100)
scatter(p_E(end,1),p_E(end,2),'MarkerEdgeColor','g','SizeData',100)
legend('E-E trajectory','Arm base','Start','Finish')
title('End-Effector Trajectory');grid on; axis equal;hold off

%% Trjectory planning (Cartesian space)

T_1 = T_jtraj(1);
T_2 = T_jtraj(end);

T_ctraj = ctraj(T_1, T_2, length(t)); % trajectory expressed through a
                                      % sequence  of transformation matrix

% extract the translational part
p_E_ctraj = transl(T_ctraj);

q_ctraj = mArb.ikine(T_ctraj,'mask',[1 0 1 0 0 1]); % joint state

figure(5)
mArb.plot(q_ctraj)

% plot of the joint trajectories
figure(8)
subplot(1,2,1)
plot(t,q_ctraj(:,1),'LineWidth',3); hold on;
plot(t,q_ctraj(:,2),'LineWidth',3)
plot(t,q_ctraj(:,3),'LineWidth',3); grid on;title('Joints Position')
legend('q_1','q_2','q_3');hold off
subplot(1,2,2)
plot(t,p_E_ctraj(:,1),'LineWidth',3); hold on;
plot(t,p_E_ctraj(:,2),'LineWidth',3);
plot(t,p_E_ctraj(:,3),'LineWidth',3);grid on;title('End-Eff. Pos')
legend('x_E','y_E','z_E');hold off

```

```
128 % plot in 2-D of the End-Effector position
129 figure(9)
130 plot(p_E_ctraj(:,1),p_E_ctraj(:,2),'LineWidth',3)
131 hold on
132 scatter(0,0,'filled','MarkerEdgeColor','r','SizeData',200,...
133     'MarkerFaceColor','r') % plot the base of the arm
134 scatter(p_E(1,1),p_E(1,2),'MarkerEdgeColor','b','SizeData',100)
135 scatter(p_E(end,1),p_E(end,2),'MarkerEdgeColor','g','SizeData',100)
136 legend('E-E trajectory','Arm base','Start','Finish')
137 title('End-Effector Trajectory');grid on; axis equal;hold off
138
139 %% Jacobian
140 % calculate the jacobian in the world frame given a certain pose "q"
141
142 J1 = mArb.jacob0(q1); J1% calculate for q1
143 J2 = mArb.jacob0(q2); J2% calculate for q2
144
145 %% Add gripper
146
147 mArb.tool = trotx(-pi/2)*transl([0,0,5])*trotz(pi/4);
148 figure
149 mArb.plot([0 5 0])
```