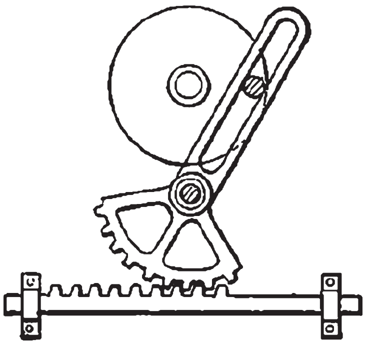
Chapter

7 Robot Arm Kinematics

Take to kinematics. It will repay you.
It is more fecund than geometry; it adds a fourth dimension to space.
Chebyshev to Sylvester 1873



Kinematics is the branch of mechanics that studies the motion of a body, or a system of bodies, without considering its mass or the forces acting on it.

From the Greek word for motion.

A robot arm, more formally a serial-link manipulator, comprises a chain of rigid links and joints. Each joint has one degree of freedom, either translational (a sliding or prismatic joint) or rotational (a revolute joint). Motion of the joint changes the relative pose of the links that it connects. One end of the chain, the base, is generally fixed and the other end is free to move in space and holds the tool or end-effector that does the useful work.

Figure 7.1 shows two modern arm-type robots that have six and seven joints respectively. Clearly the pose of the end-effector will be a complex function of the state of each joint and Sect. 7.1 describes how to compute the pose of the end-effector. Section 7.2 discusses the inverse problem, how to compute the position of each joint given the end-effector pose. Section 7.3 describes methods for generating smooth paths for the end-effector. The remainder of the chapter covers advanced topics and two complex applications: writing on a plane surface and a four-legged walking robot whose legs are simple robotic arms.

7.1 Forward Kinematics

Forward kinematics is the mapping from joint coordinates, or robot configuration, to end-effector pose. We start in Sect. 7.1.1 with conceptually simple robot arms that move in 2-dimensions in order to illustrate the principles, and in Sect. 7.1.2 extend this to more useful robot arms that move in 3-dimensions.

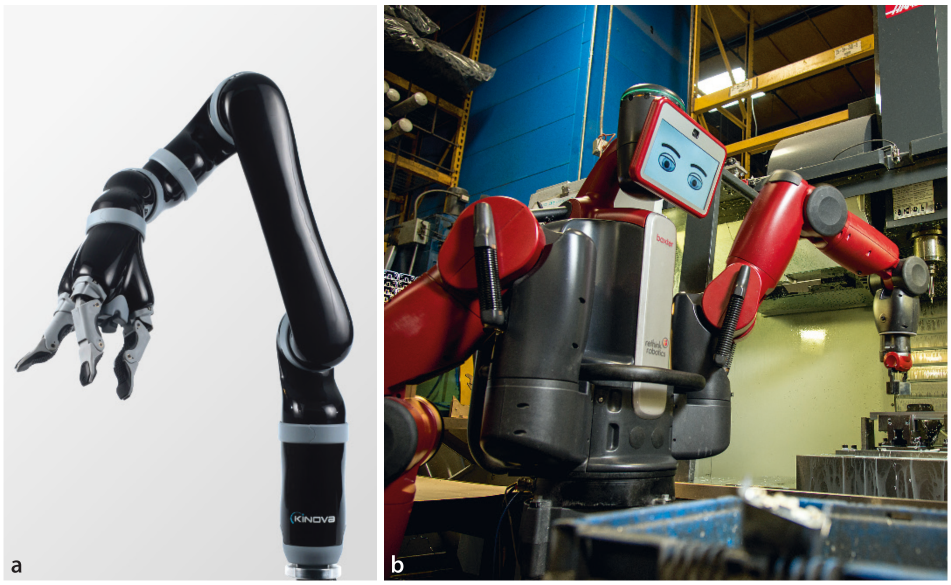


Fig. 7.1. **a** Mico 6-joint robot with 3-fingered hand (courtesy of Kinova Robotics). **b** Baxter 2-armed robotic coworker, each arm has 7 joints (courtesy of Rethink Robotics)

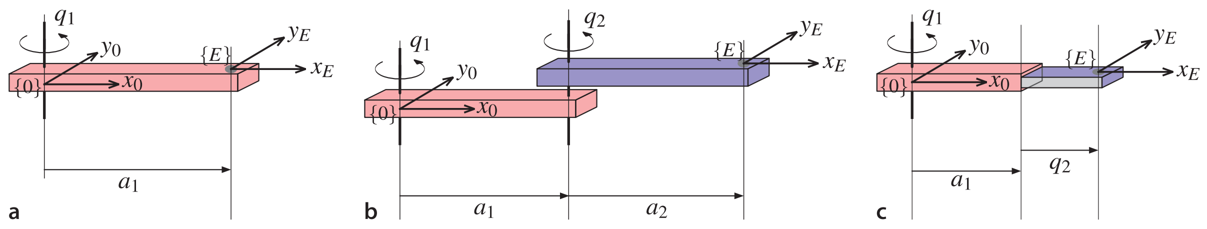


Fig. 7.2. Some simple planar robotic arms. **a** Planar arm with one rotational joint; **b** planar arm with two rotational joints; **c** planar arm with two joints: one rotational and one prismatic. The base {0} and end-effector {*E*} coordinate frames are shown. The joint variables, angle or prismatic extension, are generalized coordinates and denoted by q_j

We use the symbols $\mathcal{R}$, $\mathcal{T}_x$, $\mathcal{T}_y$ to denote relative poses in **SE**(2) that are respectively pure rotation and pure translation in the *x*- and *y*-directions.

7.1.1 2-Dimensional (Planar) Robotic Arms

Consider the simple robot arm shown in Fig. 7.2a which has a single rotational joint. We can describe the pose of its end-effector – frame {*E*} – by a sequence of relative poses: a rotation about the joint axis and then a translation by a_1 along the rotated *x*-axis

$$\xi_E(\boldsymbol{q}) = \mathcal{R}_z(q_1) \oplus \mathcal{T}_x(a_1)$$

The Toolbox allows us to express this, for the case $a_1 = 1$, by

```
>> import ETS2.*
>> a1 = 1;
>> E = Rz('q1') * Tx(a1)
```

which is a sequence of `ETS2` class objects. The argument to `Rz` is a string which indicates that its parameter is a joint variable whereas the argument to `Tx` is a constant numeric robot dimension.

The forward kinematics for a *particular* value of $q_1 = 30$ deg

```
>> E.fkine( 30, 'deg')
ans =
    0.8660   -0.5000     0.866
    0.5000    0.8660       0.5
         0         0         1
```

is an **SE**(2) homogeneous transformation matrix representing the pose of the end-effector – coordinate frame {*E*}.

An easy and intuitive way to understand how this simple robot behaves is interactively

```
>> E.teach
```

which generates a graphical representation of the robot arm as shown in Fig. 7.3. The rotational joint is indicated by a grey vertical cylinder and the link by a red horizontal pipe. You can adjust the joint angle q_1 using the slider and the arm pose and the displayed end-effector position and orientation will be updated. Clearly this is not a very useful robot arm since its end-effector can only reach points that lie on a circle.

Consider now a robot arm with two joints as shown in Fig. 7.2b. The pose of the end-effector is

$$\xi_E(\boldsymbol{q}) = \underbrace{\mathcal{R}_z(q_1)}_{\text{joint 1}} \oplus \underbrace{\mathcal{T}_x(a_1)}_{\text{link 1}} \oplus \underbrace{\mathcal{R}_z(q_2)}_{\text{joint 2}} \oplus \underbrace{\mathcal{T}_x(a_2)}_{\text{link 2}} \qquad (7.1)$$

We can represent this using the Toolbox as

```
>> a1 = 1; a2 = 1;
>> E = Rz('q1') * Tx(a1) * Rz('q2') * Tx(a2)
```

When computing the forward kinematics the joint angles are now specified by a vector

```
>> E.fkine( [30, 40], 'deg')
ans =
    0.3420   -0.9397     1.208
    0.9397    0.3420      1.44
         0         0         1
```

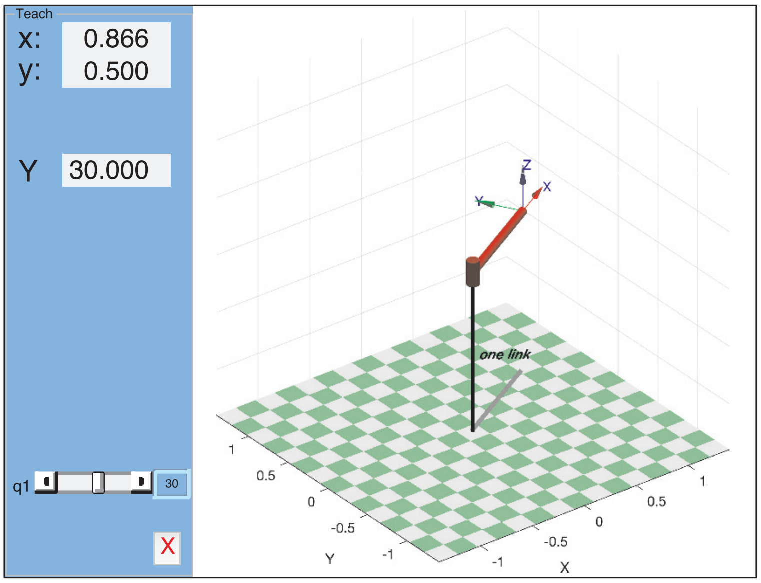


Fig. 7.3. Toolbox depiction of 1-joint planar robot using the `teach` method. The *blue panel* contains the joint angle slider and displays the position and orientation (yaw angle) of the end-effector (in degrees)

and the result is the end-effector pose when $q_1 = 30$ and $q_2 = 40$ deg. We could display the robot interactively as in the previous example, or noninteractively by

```
>> E.plot( [30, 40], 'deg')
```

The joint structure of a robot is often referred to by a shorthand comprising the letters R (for revolute) or P (for prismatic) to indicate the number and types of its joints. For this robot

```
>> E.structure
ans =
RR
```

indicates a revolute-revolute sequence of joints. The notation underneath the terms in Eq. 7.1 describes them in the context of a physical robot manipulator which comprises a series of joints and links.

You may have noticed a few characteristics of this simple planar robot arm. Firstly, most end-effector positions can be reached with two *different* joint angle vectors. Secondly, the robot can position the end-effector at any point within its reach but we cannot specify an arbitrary orientation. This robot has 2 degrees of freedom and its configuration space is $\mathcal{C} = \mathbb{S}^1 \times \mathbb{S}^1$. This is sufficient to achieve positions in the task space $\mathcal{T} \subset \mathbb{R}^2$ since $\dim \mathcal{C} = \dim \mathcal{T}$. However if our task space includes orientation $\mathcal{T} \subset \mathbf{SE}(2)$ then it is under-actuated since $\dim \mathcal{C} < \dim \mathcal{T}$ and the robot can access only a subset of the task space.

So far we have only considered revolute joints but we could use a prismatic joint instead as shown in Fig. 7.2c. The end-effector pose is

$$\xi_E(\boldsymbol{q}) = \underbrace{\mathcal{R}_z(q_1)}_{\text{joint 1}} \oplus \underbrace{\mathcal{T}_x(a_1)}_{\text{link 1}} \oplus \underbrace{\mathcal{T}_x(q_2)}_{\text{joint 2}}$$

Prismatic joints. Robot joints are commonly revolute (rotational) but can also be prismatic (linear, sliding, telescopic, etc.). The SCARA robot of Fig. III.1b has a prismatic third joint while the gantry robot of Fig. III.1c has three prismatic joints for motion in the *x*-, *y*- and *z*-directions.

The Stanford arm shown here has a prismatic third joint. It was developed at the Stanford AI Lab in 1972 by robotics pioneer Victor Scheinman who went on to design the PUMA robot arms. This type of arm supported a lot of important early research work in robotics and one can be seen in the Smithsonian Museum of American History, Washington DC. (Photo courtesy Oussama Khatib) and the Toolbox representation follows a familiar pattern

```
>> a1 = 1;
>> E = Rz('q1') * Tx(a1) * Tz('q2')
```

and the arm structure is now

```
>> E.structure
ans =
RP
```

which is commonly called a polar-coordinate robot arm.

We can easily add a third joint

$$\xi_e(\boldsymbol{q}) = \underbrace{\mathcal{R}_z(q_1)}_{\text{joint 1}} \oplus \underbrace{\mathcal{T}_x(a_1)}_{\text{link 1}} \oplus \underbrace{\mathcal{R}_z(q_2)}_{\text{joint 2}} \oplus \underbrace{\mathcal{T}_x(a_2)}_{\text{link 2}} \oplus \underbrace{\mathcal{R}_z(q_3)}_{\text{joint 3}} \oplus \underbrace{\mathcal{T}_x(a_3)}_{\text{link 3}}$$

and use the now familiar Toolbox functionality to represent and work with this arm. This robot has 3 degrees of freedom and is able to access all points in the task space $\mathcal{T} \subset \mathbf{SE}(2)$, that is, achieve any pose in the plane (limited by reach).

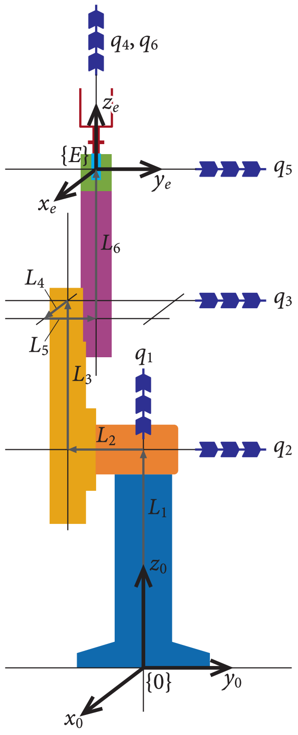


Fig. 7.4. Puma robot in the zero-joint-angle configuration showing dimensions and joint axes (indicated by *blue triple arrows*) (after Corke 2007)

7.1.2 3-Dimensional Robotic Arms

Truly useful robots have a task space $\mathcal{T} \subset \mathbf{SE}(3)$ enabling arbitrary position and orientation of the end-effector. This requires a robot with a configuration space $\dim \mathcal{C} \geq \dim \mathcal{T}$ which can be achieved by a robot with six or more joints. In this section we will use the Puma 560 as an exemplar of the class of all-revolute six-axis robot manipulators with $\mathcal{C} \subset (\mathbb{S}^1)^6$.

We can extend the technique from the previous section for a robot like the Puma 560 whose dimensions are shown in Fig. 7.4. Starting with the world frame {0} we move up, rotate about the waist axis (q_1), rotate about the shoulder axis (q_2), move to the left, move up and so on. As we go, we write down the transform expression

We use the symbols $\mathcal{R}_i$, $\mathcal{T}_i$, $i \in \{x, y, z\}$ to denote relative poses in **SE**(3) that are respectively pure rotation about, or pure translation along, the *i*-axis.

$$\xi_E = \mathcal{T}_z(L_1) \oplus \mathcal{R}_z(q_1) \oplus \mathcal{R}_y(q_2) \oplus \mathcal{T}_y(L_2) \oplus \mathcal{T}_z(L_3) \oplus \mathcal{R}_y(q_3)$$
$$\oplus \mathcal{T}_x(L_4) \oplus \mathcal{T}_y(L_5) \oplus \mathcal{T}_z(L_6) \oplus \underbrace{\mathcal{R}_z(q_4) \oplus \mathcal{R}_y(q_5) \oplus \mathcal{R}_z(q_6)}_{\text{wrist}}$$

The marked term represents the kinematics of the robot's wrist and should be familiar to us as a *ZYZ* Euler angle sequence from Sect. 2.2.1.2 – it provides an arbitrary orientation but is subject to a singularity when the middle angle $q_5 = 0$.

We can represent this using the 3-dimensional version of the Toolbox class we used previously

```
>> import ETS3.*
>> L1 = 0; L2 = -0.2337; L3 = 0.4318; L4 = 0.0203; L5 = 0.0837; L6 = 0.4318;
>> E3 = Tz(L1) * Rz('q1') * Ry('q2') * Ty(L2) * Tz(L3) * Ry('q3')↵
    * Tx(L4) * Ty(L5) * Tz(L6) * Rz('q4') * Ry('q5')  * Rz('q6');
```

We can use the interactive teach facility or compute the forward kinematics

```
>> E3.fkine([0 0 0 0 0 0])
ans =
         1         0         0    0.0203
         0         1         0     -0.15
         0         0         1    0.8636
         0         0         0         1
```

While this notation is intuitive it does becomes cumbersome as the number of robot joints increases. A number of approaches have been developed to more concisely describe a serial-link robotic arm: Denavit-Hartenberg notation and product of exponentials.

Table 7.1. Denavit-Hartenberg parameters: their physical meaning, symbol and formal definition

Joint angle	θ_j	the angle between the x_{j-1} and x_j axes about the z_{j-1} axis	revolute joint variable
Link offset	d_j	the distance from the origin of frame $j-1$ to the x_j axis along the z_{j-1} axis	prismatic joint variable
Link length	a_j	the distance between the z_{j-1} and z_j axes along the x_j axis; for intersecting axes is parallel to $\hat{z}_{j-1} \times \hat{z}_j$	constant
Link twist	α_j	the angle from the z_{j-1} axis to the z_j axis about the x_j axis	constant
Joint type	σ_j	$\sigma = R$ for a revolute joint, $\sigma = P$ for a prismatic joint	constant

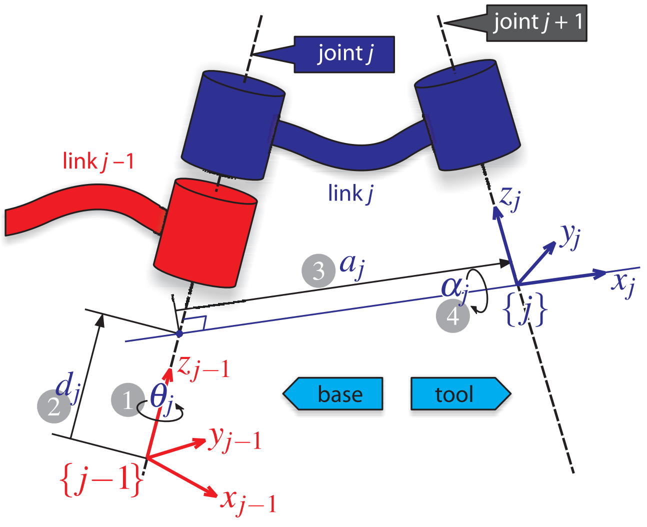


Fig. 7.5. Definition of standard Denavit and Hartenberg link parameters. The colors red and blue denote all things associated with links $j-1$ and j respectively. The numbers in circles represent the order in which the elementary transforms are applied. x_j is parallel to $z_{j-1} \times z_j$ and if those two axes are parallel then d_j can be arbitrarily chosen

7.1.2.1 Denavit-Hartenberg Parameters

One systematic way of describing the geometry of a serial chain of links and joints is Denavit-Hartenberg notation.

For a manipulator with N joints numbered from 1 to N, there are $N+1$ links, numbered from 0 to N. Joint j connects link $j-1$ to link j and moves them relative to each other. It follows that link ℓ connects joint ℓ to joint $\ell+1$. Link 0 is the base of the robot, typically fixed and link N, the last link of the robot, carries the end-effector or tool.

In Denavit-Hartenberg, notation a link defines the spatial relationship between two neighboring joint axes as shown in Fig. 7.5. A link is specified by four parameters. The relationship between two link coordinate frames would ordinarily entail six parameters, three each for translation and rotation. For Denavit-Hartenberg notation there are only four parameters but there are also two constraints: axis x_j intersects z_{j-1} and axis x_j is perpendicular to z_{j-1}. One consequence of these constraints is that sometimes the link coordinate frames are not actually located on the physical links of the robot. Another consequence is that the robot must be placed into a particular configuration – the zero-angle configuration – which is discussed further in Sect. 7.4.1. The Denavit-Hartenberg parameters are summarized in Table 7.1.

The coordinate frame $\{j\}$ is attached to the far (distal) end of link j. The z-axis of frame $\{j\}$ is aligned with the axis of joint $j+1$.

The transformation from link coordinate frame $\{j-1\}$ to frame $\{j\}$ is defined in terms of elementary rotations and translations as

$$ {}^{j-1}\xi_j\big(\theta_j, d_j, a_j, \alpha_j\big) = \mathcal{R}_z\big(\theta_j\big) \oplus \mathcal{T}_z\big(d_j\big) \oplus \mathcal{T}_x\big(a_j\big) \oplus \mathcal{R}_x\big(\alpha_j\big) \tag{7.2}$$

which can be expanded in homogeneous matrix form as

$$ {}^{j-1}\boldsymbol{A}_j = \begin{pmatrix} \cos\theta_j & -\sin\theta_j\cos\alpha_j & \sin\theta_j\sin\alpha_j & a_j\cos\theta_j \\ \sin\theta_j & \cos\theta_j\cos\alpha_j & -\cos\theta_j\sin\alpha_j & \alpha_j\sin\theta_j \\ 0 & \sin\alpha_j & \cos\alpha_j & d_j \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.3}$$

The parameters α_j and a_j are always constant. For a revolute joint, θ_j is the joint variable and d_j is constant, while for a prismatic joint, d_j is variable, θ_j is constant and $\alpha_j = 0$. In many of the formulations that follow, we use generalized joint coordinates q_j

$$ \text{if } \sigma_j = \begin{cases} R\colon & \theta_j \leftarrow q_j \\ P\colon & d_j \leftarrow q_j \end{cases} $$

For an N-axis robot, the generalized joint coordinates $\boldsymbol{q} \in \mathcal{C}$ where $\mathcal{C} \subset \mathbb{R}^N$ is called the joint space or configuration space.▸ For the common case of an all-revolute robot $\mathcal{C} \subset (\mathbb{S}^1)^N$ the joint coordinates are referred to as joint angles. The joint coordinates are also referred to as the *pose of the manipulator* which is different to the *pose of the end-effector* which is a Cartesian pose $\xi \in \mathbf{SE}(3)$. The term *configuration* is shorthand for *kinematic configuration* which will be discussed in Sect. 7.2.2.1.

This is the same concept as was introduced for mobile robots in Sect. 2.3.5.

Within the Toolbox a robot revolute joint and link can be created by

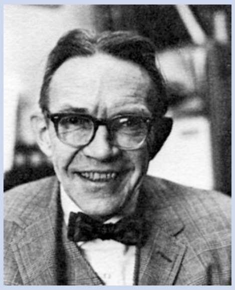
```
>> L = Revolute('a', 1)
L =
Revolute(std): theta=q, d=0, a=1, alpha=0, offset=0
```

which is a revolute-joint object of type `Revolute` which is a subclass of the generic `Link` object. The displayed value of the object shows the kinematic parameters (most of which have defaulted to zero), the joint type and that standard Denavit-Hartenberg convention is used (the tag `std`).▸

A slightly different notation, *modifed Denavit-Hartenberg* notation is discussed in Sect. 7.4.3.

Jacques Denavit and **Richard Hartenberg** introduced many of the key concepts of kinematics for serial-link manipulators in a 1955 paper (Denavit and Hartenberg 1955) and their later classic text *Kinematic Synthesis of Linkages* (Hartenberg and Denavit 1964).

Jacques Denavit (1930–2012) was born in Paris where he studied for his Bachelor degree before pursuing his masters and doctoral degrees in mechanical engineering at Northwestern University, Illinois. In 1958 he joined the Department of Mechanical Engineering and Astronautical Science at Northwestern where the collaboration with Hartenberg was formed. In addition to his interest in dynamics and kinematics Denavit was also interested in plasma physics and kinetics. After the publication of the book he moved to Lawrence Livermore National Lab, Livermore, California, where he undertook research on computer analysis of plasma physics problems. He retired in 1982.



Richard Hartenberg (1907–1997) was born in Chicago and studied for his degrees at the University of Wisconsin, Madison. He served in the merchant marine and studied aeronautics for two years at the University of Göttingen with space-flight pioneer Theodore von Kármán. He was Professor of mechanical engineering at Northwestern University where he taught for 56 years. His research in kinematics led to a revival of interest in this field in the 1960s, and his efforts helped put kinematics on a scientific basis for use in computer applications in the analysis and design of complex mechanisms. He also wrote extensively on the history of mechanical engineering.

A `Link` object has many parameters and methods which are described in the online documentation, but the most common ones are illustrated by the following examples. The link transform Eq. 7.3 for $q = 0.5$ rad is an `SE3` object

```
>> L.A(0.5)
ans =
    0.8776   -0.4794         0    0.8776
    0.4794    0.8776        -0    0.4794
         0         0         1         0
         0         0         0         1
```

representing the homogeneous transformation due to this robot link with the particular value of θ. Various link parameters can be read or altered, for example

```
>> L.type
ans =
     R
```

indicates that the link is revolute and

```
>> L.a
ans =
    1.0000
```

returns the kinematic parameter a. Finally a link can contain an offset

```
>> L.offset = 0.5;
>> L.A(0)
ans =
    0.8776   -0.4794         0    0.8776
    0.4794    0.8776        -0    0.4794
         0         0         1         0
         0         0         0         1
```

which is added to the joint variable before computing the link transform and will be discussed in more detail in Sect. 7.4.1.

The forward kinematics is a function of the joint coordinates and is simply the composition of the relative pose due to each link

$$ {}^0\xi_N = \mathcal{K}(\boldsymbol{q};\theta,\boldsymbol{d},\boldsymbol{a},\alpha,\sigma) = {}^0\xi_1 \oplus {}^1\xi_2 \cdots {}^{N-1}\xi_N \tag{7.4} $$

In this notation link 0 is the base of the robot and commonly for the first link $d_1 = 0$ but we could set $d_1 > 0$ to represent the height of the first joint above the world coordinate frame. The final link, link N, carries the tool – the parameters d_N, a_N and α_N provide a limited means to describe the tool-tip pose with respect to the $\{N\}$ frame. By convention the robot's tool points in the z-direction as shown in Fig. 2.16.

More generally we add two extra transforms to the chain◀

We have used W to denote the world frame in this case since 0 designates link 0, the base link.

$$ {}^W\xi_E = \underbrace{{}^W\xi_0}_{\xi_B} \oplus {}^0\xi_1 \oplus {}^1\xi_2 \cdots {}^{N-1}\xi_N \oplus \underbrace{{}^N\xi_E}_{\xi_T} $$

The base transform ξ_B puts the base of the robot arm at an arbitrary pose within the world coordinate frame. In a manufacturing system the base is usually fixed to the environment but it could be mounted on a mobile ground, aerial or underwater robot, a truck, or even a space shuttle.

The frame $\{N\}$ is often defined as the center of the spherical wrist mechanism, and the tool transform ξ_T describes the pose of the tool tip with respect to that. In practice ξ_T might consist of several components. Firstly, a transform to a tool-mounting flange on the physical end of the robot. Secondly, a transform from the flange to the end of the tool that is bolted to it, where the tool might be a gripper, screwdriver or welding torch.

In the Toolbox we connect `Link` class objects in series using the `SerialLink` class

```
>> robot = SerialLink( [ Revolute('a', 1) Revolute('a', 1) ],↵
 'name', 'my robot')
robot =
my robot:: 2 axis, RR, stdDH
+---+-----------+-----------+-----------+-----------+-----------+
| j |     theta |         d |         a |     alpha |    offset |
+---+-----------+-----------+-----------+-----------+-----------+
|  1|         q1|0          |1          |0          |0          |
|  2|         q2|0          |1          |0          |0          |
+---+-----------+-----------+-----------+-----------+-----------+
```

We have just recreated the 2-robot robot we looked at earlier, but now it is embedded in **SE**(3). The forward kinematics are

```
>> robot.fkine([30 40], 'deg')
ans =
    0.3420   -0.9397         0     1.208
    0.9397    0.3420         0      1.44
         0         0         1         0
         0         0         0         1
```

The Toolbox contains a large number of robot arm models defined in this way and they can be listed by

```
>> models
ABB, IRB140, 6DOF, standard_DH (mdl_irb140)
Aldebaran, NAO, humanoid, 4DOF, standard_DH (mdl_nao)
Baxter, Rethink Robotics, 7DOF, standard_DH (mdl_baxter)
 ...
```

where the name of the Toolbox script to load the model is given in parentheses at the end of each line, for example

```
>> mdl_irb140
```

The `models` function also supports searching by keywords and robot arm type. You can adjust the parameters of any model using the editing method, for example

```
>> robot.edit
```

Determining the Denavit-Hartenberg parameters for a particular robot is described in more detail in Sect. 7.4.2.

7.1.2.2 Product of Exponentials

In Chap. 2 we introduced twists. A twist is defined by a screw axis direction and pitch, and a point that the screw axis passes through. In matrix form the twist $S \in \mathbb{R}^6$

$$\boldsymbol{T}' = e^{[S]\theta}\boldsymbol{T}$$

rotates the coordinate frame described by the pose $\boldsymbol{T}$ about the screw axis by an angle θ.▸ This is exactly the case of the single-joint robot of Fig. 7.2a, where the screw axis *is* the joint axis and $\boldsymbol{T}$ is the pose of the end-effector when $q_1 = 0$. We can therefore write the forward kinematics as

For a prismatic twist, the motion is a displacement of θ along the screw axis. Here we are working in the plane so $\boldsymbol{T} \in \mathbf{SE}(2)$ and $S \in \mathbb{R}^3$.

$$\boldsymbol{T}_E = e^{[S_1]q_1}\boldsymbol{T}_E(0)$$

where $\boldsymbol{T}_E(0)$ is the end-effector pose in the zero-angle joint configuration: $q_1 = 0$.

For the 2-joint robot of Fig. 7.2b we would write

$$\boldsymbol{T}_E = e^{[S_1]q_1}\,\underbrace{e^{[S_2]q_2}\boldsymbol{T}_E(0)}$$

where S_1 and S_2 are the screws defined by the joint axes and $\boldsymbol{T}_E(0)$ is the end-effector pose in the zero-angle joint configuration: $q_1 = q_2 = 0$. The indicated term is similar to the single-joint robot above, and the first twist rotates that joint and link about S_1. In MATLAB we define the link lengths and compute $\boldsymbol{T}_E(0)$

```
>> a1 = 1; a2 = 1;
>> TE0 = SE2(a1+a2, 0, 0);
```

define the two twists, in **SE**(2), for this example

```
>> S1 = Twist( 'R', [0 0] );
>> S2 = Twist( 'R', [a1 0] );
```

and apply them to $\boldsymbol{T}_E(0)$

```
>> TE = S1.T(30, 'deg') * S2.T(40, 'deg') * TE0
TE =
    0.3420   -0.9397     1.208
    0.9397    0.3420      1.44
         0         0         1
```

For a general robot that moves in 3-dimensions we can write the forward kinematics in product of exponential (PoE) form as

$$\xi_E \sim {}^0\boldsymbol{T}_E = \mathrm{e}^{[S_1]q_1} \cdots \mathrm{e}^{[S_N]q_N}\, {}^0\boldsymbol{T}_E(0)$$

where ${}^0\boldsymbol{T}_E(0)$ is the end-effector pose when the joint coordinates are all zero and S_j is the twist for joint j expressed in the world frame. This can also be written as

The tool and base transform are effectively included in ${}^0\boldsymbol{T}_E(0)$, but an explicit base transform could be added if the screw axes are defined with respect to the robot's base rather than the world coordinate frame, or use the adjoint matrix to transform the screw axes from base to world coordinates.

$$\xi_E \sim {}^0\boldsymbol{T}_E = {}^0\boldsymbol{T}_E(0)\mathrm{e}^{\left[{}^ES_1\right]q_1} \cdots \mathrm{e}^{\left[{}^ES_N\right]q_N}$$

and ${}^ES_j$ is the twist for joint j expressed in the end-effector frame which is related to the twists above by ${}^ES_j = \mathrm{Ad}({}^E\xi_0)S_j$.

A serial-link manipulator can be succinctly described by a table listing the 6 screw parameters for each joint as well as the zero-joint-coordinate end-effector pose.

7.1.2.3 6-Axis Industrial Robot

Truly useful robots have a task space $\mathcal{T} \subset \mathbf{SE}(3)$ enabling arbitrary position and attitude of the end-effector – the task space has six spatial degrees of freedom: three translational and three rotational. This requires a robot with a configuration space $\mathcal{C} \subset \mathbb{R}^6$ which can be achieved by a robot with six joints. In this section we will use the Puma 560 as an example of the class of all-revolute six-axis robot manipulators. We define an instance of a Puma 560 robot using the script

```
>> mdl_puma560
```

which creates a `SerialLink` object, `p560`, in the workspace. Displaying the variable shows the table of its Denavit-Hartenberg parameters

```
>> p560
Puma 560 [Unimation]:: 6 axis, RRRRRR, stdDH, slowRNE
 - viscous friction; params of 8/95;
+---+-----------+-----------+-----------+-----------+-----------+
| j |     theta |         d |         a |     alpha |    offset |
+---+-----------+-----------+-----------+-----------+-----------+
|  1|         q1|          0|          0|      1.571|          0|
|  2|         q2|          0|     0.4318|          0|          0|
|  3|         q3|       0.15|     0.0203|     -1.571|          0|
|  4|         q4|     0.4318|          0|      1.571|          0|
|  5|         q5|          0|          0|     -1.571|          0|
|  6|         q6|          0|          0|          0|          0|
+---+-----------+-----------+-----------+-----------+-----------+
```

The **Puma 560 robot** (Programmable Universal Manipulator for Assembly) released in 1978 was the first modern industrial robot and became enormously popular. It featured an anthropomorphic design, electric motors and a spherical wrist – the archetype of all that followed. It can be seen in the Smithsonian Museum of American History, Washington DC.

The Puma 560 catalyzed robotics research in the 1980s and it was a very common laboratory robot. Today it is obsolete and rare but in homage to its important role in robotics research we use it here. For our purposes the advantages of this robot are that it has been well studied and its parameters are very well known – it has been described as the "white rat" of robotics research.

Most modern 6-axis industrial robots are very similar in structure and can be accomodated simply by changing the Denavit-Hartenberg parameters. The Toolbox has kinematic models for a number of common industrial robots from manufacturers such as Rethink, Kinova, Motoman, Fanuc and ABB. (Puma photo courtesy Oussama Khatib)

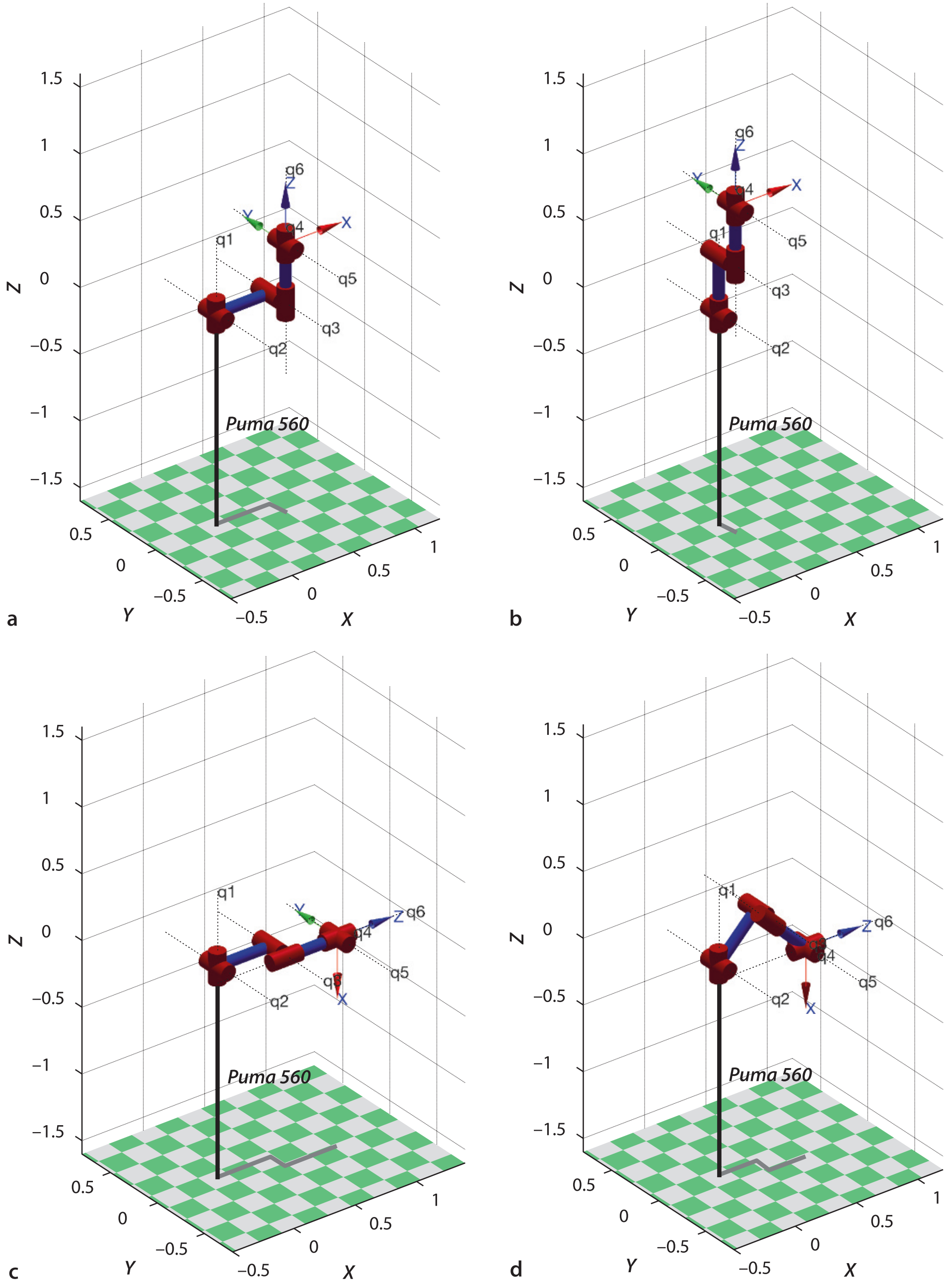


Fig. 7.6.
The Puma robot in 4 different poses. **a** *Zero angle*; **b** *ready* pose; **c** *stretch*; **d** *nominal*

Anthropomorphic means having human-like characteristics. The Puma 560 robot was designed to have approximately the dimensions and reach of a human worker. It also had a spherical joint at the wrist just as humans have.

Roboticists also tend to use anthropomorphic terms when describing robots. We use words like waist, shoulder, elbow and wrist when describing serial link manipulators. For the Puma these terms correspond respectively to joint 1, 2, 3 and 4–6.

Note that a_j and d_j are in SI units which means that the translational part of the forward kinematics will also have SI units.

The script `mdl_puma560` also creates a number of joint coordinate vectors in the workspace which represent the robot in some canonic configurations:

`qz`	$(0, 0, 0, 0, 0, 0)$	*zero angle*
`qr`	$(0, \frac{\pi}{2}, -\frac{\pi}{2}, 0, 0, 0)$	*ready*, the arm is straight and vertical
`qs`	$(0, 0, -\frac{\pi}{2}, 0, 0, 0)$	*stretch*, the arm is straight and horizontal
`qn`	$(0, \frac{\pi}{4}, -\pi, 0, \frac{\pi}{4}, 0)$	*nominal*, the arm is in a dextrous working pose◀

Well away from singularities, which will be discussed in Sect. 7.3.4.

and these are shown graphically in Fig. 7.6. These plots are generated using the `plot` method, for example

```
>> p560.plot(qz)
```

which shows a skeleton of the robot with pipes that connect the link coordinate frames as defined by the Denavit-Hartenberg parameters. The `plot` method has many options for showing the joint axes, wrist coordinate frame, shadows and so on. More realistic-looking plots such as shown in Fig. 7.7 can be created by the `plot3d` method for a limited set of Toolbox robot models.

Forward kinematics can be computed as before

```
>> TE = p560.fkine(qz)
TE =
    1.0000         0         0    0.4521
         0    1.0000         0   -0.1500
         0         0    1.0000    0.4318
         0         0         0    1.0000
```

where the joint coordinates are given as a row vector. This returns the homogeneous transformation corresponding to the end-effector pose. The origin of this frame, the link-6 coordinate frame {6}, is defined◀ as the point of intersection of the axes of the last 3 joints – physically this point is inside the robot's wrist mechanism. We can define a tool transform, from the $\boldsymbol{T}_6$ frame to the actual tool tip by

By the Denavit-Hartenberg parameters of the model in the `mdl_puma560` script.

```
>> p560.tool = SE3(0, 0, 0.2);
```

in this case a 200 mm extension in the $\boldsymbol{T}_6$ z-direction.◀ The pose of the tool tip, often referred to as the tool center point or TCP, is now

Alternatively we could change the kinematic parameter d_6. The tool transform approach is more general since the final link kinematic parameters only allow setting of d_6, a_6 and α_6 which provide z-axis translation, x-axis translation and x-axis rotation respectively.

```
>> p560.fkine(qz)
ans =
    1.0000         0         0    0.4521
         0    1.0000         0   -0.1500
         0         0    1.0000    0.6318
         0         0         0    1.0000
```

The kinematic definition we have used considers that the base of the robot is the intersection point of the waist and shoulder axes which is a point inside the structure of the robot. The Puma 560 robot includes a "30-inch" tall pedestal. We can shift the origin of the robot from the point inside the robot to the base of the pedestal using a base transform

```
>> p560.base = SE3(0, 0, 30*0.0254);
```

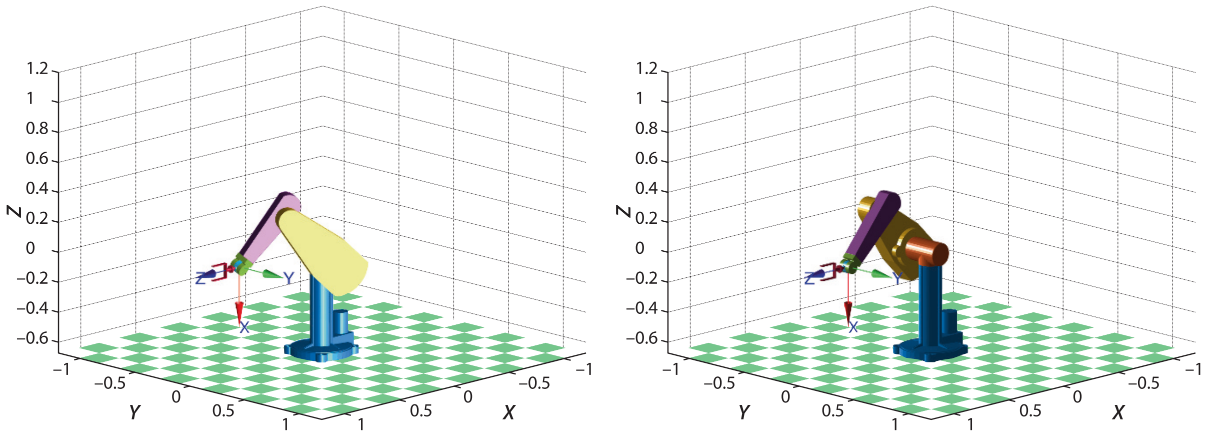


Fig. 7.7. These two different robot configurations result in the same end-effector pose. They are called the left- and right-handed configurations, respectively. These graphics, produced using the `plot3d` method, are available for a limited subset of robot models

where for consistency we have converted the pedestal height to SI units. Now, with both base and tool transform, the forward kinematics are

```
>> p560.fkine(qz)
ans =
    1.0000         0         0    0.4521
         0    1.0000         0   -0.1500
         0         0    1.0000    1.3938
         0         0         0    1.0000
```

and we can see that the *z*-coordinate of the tool is now greater than before.

We can also do more interesting things, for example

```
>> p560.base = SE3(0,0,3) * SE3.Rx(pi);
>> p560.fkine(qz)
ans =
    1.0000         0         0    0.4521
         0   -1.0000   -0.0000    0.1500
         0    0.0000   -1.0000    2.3682
         0         0         0    1.0000
```

which positions the robot's origin 3 m above the world origin with its coordinate frame rotated by 180° about the *x*-axis. This robot is now hanging from the ceiling!

The Toolbox supports joint angle time series, or trajectories, such as

```
>> q
q =
         0         0         0         0         0         0
         0    0.0365   -0.0365         0         0         0
         0    0.2273   -0.2273         0         0         0
         0    0.5779   -0.5779         0         0         0
         0    0.9929   -0.9929         0         0         0
         0    1.3435   -1.3435         0         0         0
         0    1.5343   -1.5343         0         0         0
         0    1.5708   -1.5708         0         0         0
```

where each row represents the joint coordinates at a different timestep and the columns represent the joints.▸ In this case the method `fkine`

Generated by the `jtraj` function, which is discussed in Sect. 7.3.1.

```
>> T = p560.fkine(q);
```

returns an array of SE3 objects

```
>> about T
T [SE3] : 1x8 (1.0 kB)
```

one per timestep. The homogeneous transform corresponding to the joint coordinates in the fourth row of `q` is

```
>> T(4)
ans =
    1.0000         0         0     0.382
         0        -1         0      0.15
         0         0   -1.0000     2.132
         0         0         0         1
```

Creating trajectories will be covered in Sect. 7.3.

7.2 Inverse Kinematics

We have shown how to determine the pose of the end-effector given the joint coordinates and optional tool and base transforms. A problem of real practical interest is the inverse problem: given the desired pose of the end-effector ξ_E what are the required joint coordinates? For example, if we know the Cartesian pose of an object, what joint coordinates does the robot need in order to reach it? This is the inverse kinematics problem which is written in functional form as

$$\boldsymbol{q} = \mathcal{K}^{-1}(\xi) \tag{7.5}$$

and in general this function is not unique, that is, several joint coordinate vectors $\boldsymbol{q}$ will result in the same end-effector pose.

Two approaches can be used to determine the inverse kinematics. Firstly, a closed-form or analytic solution can be determined using geometric or algebraic approaches. However this becomes increasingly challenging as the number or robot joints increases and for some serial-link manipulators no closed-form solution exists. Secondly, an iterative numerical solution can be used. In Sect. 7.2.1 we again use the simple 2-dimensional case to illustrate the principles and then in Sect. 7.2.2 extend these to robot arms that move in 3-dimensions.

7.2.1 2-Dimensional (Planar) Robotic Arms

We will illustrate inverse kinematics for the 2-joint robot of Fig. 7.2b in two ways: algebraic closed-form and numerical.

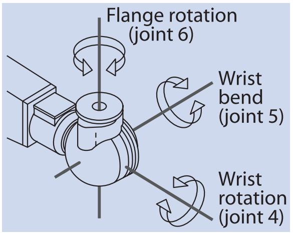
7.2.1.1 Closed-Form Solution

We start by computing the forward kinematics algebraically as a function of joint angles. We can do this easily, and in a familiar way

```
>> import ETS2.*
>> a1 = 1; a2 = 1;
>> E = Rz('q1') * Tx(a1) * Rz('q2') * Tx(a2)
```

but now using the MATLAB Symbolic Math Toolbox™ we define some real-valued symbolic variables to represent the joint angles

```
>> syms q1 q2 real
```



Spherical wrists are a key component of almost all modern arm-type robots. They have three axes of rotation that are orthogonal and intersect at a common point. This is a gimbal-like mechanism, and as discussed in Sect. 2.2.1.3 and will have a singularity.

The robot end-effector pose, position and an orientation, is defined at the center of the wrist. Since the wrist axes intersect at a common point they cause zero translation, therefore the position of the end-effector is a function only of the first three joints. This is a critical simplification that makes it possible to find closed-form inverse kinematic solutions for 6-axis industrial robots. An arbitrary end-effector orientation is achieved independently by means of the three wrist joints.

and then compute the forward kinematics

```
>> TE = E.fkine( [q1, q2] )
TE =
[ cos(q1 + q2), -sin(q1 + q2), cos(q1 + q2) + cos(q1)]
[ sin(q1 + q2),  cos(q1 + q2), sin(q1 + q2) + sin(q1)]
[            0,             0,                      1]
```

which is an algebraic representation of the robot's forward kinematics – the end-effector pose as a function of the joint variables.

We can define two more symbolic variables to represent the desired end-effector position (x, y)

```
>> syms x y real
```

and equate them with the results of the forward kinematics▸

With the MATLAB Symbolic Math Toolbox™ the == operator denotes equality, as opposed to = which denotes assignment.

```
>> e1 =  x == TE.t(1)
e1 =
x == cos(q1 + q2) + cos(q1)
>> e2 =  y == TE.t(2)
e2 =
y == sin(q1 + q2) + sin(q1)
```

which gives two scalar equations that we can solve simultaneously

```
>> [s1,s2] = solve( [e1 e2], [q1 q2] )
```

where the arguments are respectively the set of equations and the set of unknowns to solve for. The outputs are the solutions for q_1 and q_2 respectively. We observed in Sect. 7.1.1 that two different sets of joint angles give the same end-effector position, and this means that the inverse kinematics does not have a unique solution. Here MATLAB has returned

```
>> length(s2)
ans =
     2
```

indicating two solutions. One solution for q_2 is

```
>> s2(1)
ans =
-2*atan((-(x^2 + y^2)*(x^2 + y^2 - 4))^(1/2)/(x^2 + y^2))
```

and would be used in conjunction with the corresponding element of the solution vector for q_1 which is `s1(1)`.

As mentioned earlier the complexity of algebraic solution increases dramatically with the number of joints and more sophisticated symbolic solution approaches need to be used. The `SerialLink` class has a method `ikine_sym` that generates symbolic inverse kinematics solutions for a limited class of robot manipulators.

7.2.1.2 Numerical Solution

We can think of the inverse kinematics problem as one of adjusting the joint coordinates until the forward kinematics matches the desired pose. More formally this is an optimization problem – to minimize the error between the forward kinematic solution and the desired pose ξ^*

$$\boldsymbol{q}^* = \arg\min_{\boldsymbol{q}} \left\| \mathcal{K}(\boldsymbol{q}) \ominus \xi^* \right\|$$

For our simple 2-link example the error function comprises only the error in the end-effector position, not its orientation

$$E(q) = \left\| \left[\mathcal{K}(\boldsymbol{q})\right]_t - \left(x^* \ y^*\right)^T \right\|$$

We can solve this using the builtin MATLAB multi-variable minimization function fminsearch

```
>> pstar = [0.6; 0.7];
>> q = fminsearch( @(q) norm( E.fkine(q).t - pstar ), [0 0] )
q =
   -0.2295    2.1833
```

where the first argument is the error function, expressed here as a MATLAB anonymous function, that incorporates the desired end-effector position; and the second argument is the initial guess at the joint coordinates. The computed joint angles indeed give the desired end-effector position

```
>> E.fkine(q).print
t = (0.6, 0.7), theta = 111.9 deg
```

As already discussed there are two solutions for ***q*** but the solution that is found using this approach depends on the initial choice of ***q***.

7.2.2 3-Dimensional Robotic Arms

7.2.2.1 Closed-Form Solution

Closed-form solutions have been developed for most common types of 6-axis industrial robots and many are included in the Toolbox. A necessary condition for a closed-form solution of a 6-axis robot is a spherical wrist mechanism. We will illustrate closed-form inverse kinematics using the Denavit-Hartenberg model for the Puma robot

```
>> mdl_puma560
```

At the *nominal* joint coordinates shown in Fig. 7.6d

```
>> qn
qn =
         0    0.7854    3.1416         0    0.7854         0
```

the end-effector pose is

```
>> T = p560.fkine(qn)
T =
   -0.0000    0.0000    1.0000    0.5963
   -0.0000    1.0000   -0.0000   -0.1501
   -1.0000   -0.0000   -0.0000   -0.0144
         0         0         0    1.0000
```

Since the Puma 560 is a 6-axis robot arm with a spherical wrist we use the method ikine6s to compute the inverse kinematics using a closed-form solution.◂ The required joint coordinates to achieve the pose T are

The method ikine6s checks the Denavit-Hartenberg parameters to determine if the robot meets these criteria.

```
>> qi = p560.ikine6s(T)
qi =
    2.6486   -3.9270    0.0940    2.5326    0.9743    0.3734
```

Surprisingly, these are quite different to the joint coordinates we started with. However if we investigate a little further

```
>> p560.fkine(qi)
ans =
   -0.0000    0.0000    1.0000    0.5963
    0.0000    1.0000   -0.0000   -0.1500
   -1.0000    0.0000   -0.0000   -0.0144
         0         0         0    1.0000
```

we see that these two different sets of joint coordinates result in the *same* end-effector pose and these are shown in Fig. 7.7. The shoulder of the Puma robot is horizontally offset from the waist, so in one solution the arm is to the left of the waist, in the other

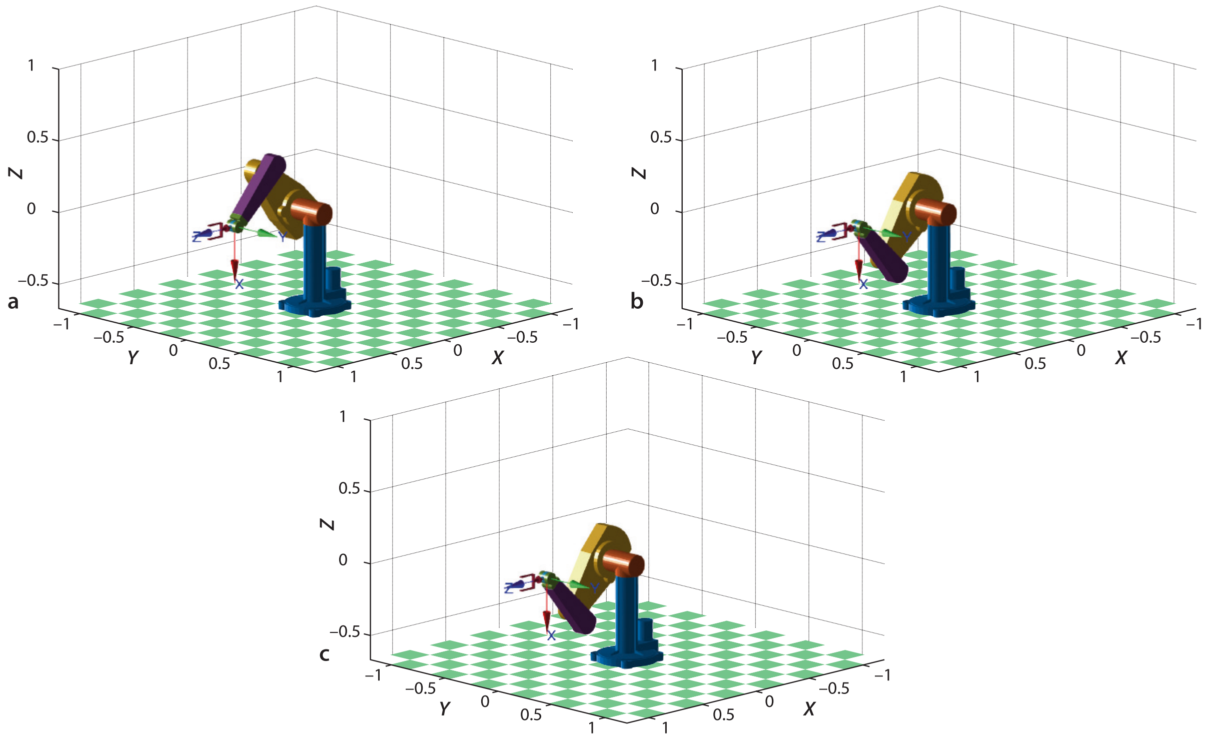


Fig. 7.8. Different configurations of the Puma 560 robot. **a** Right-up-noflip; **b** right-down-noflip; **c** right-down-flip

it is to the right. These are referred to as the left- and right-handed kinematic configurations respectively. In general there are eight sets of joint coordinates that give the same end-effector pose – as mentioned earlier the inverse solution is not unique.

We can *force* the right-handed solution

```
>> qi = p560.ikine6s(T, 'ru')
qi =
   -0.0000    0.7854    3.1416    0.0000    0.7854   -0.0000
```

which gives the original set of joint angles by specifying a *right handed* configuration with the elbow *up*.

In addition to the left- and right-handed solutions, there are solutions with the elbow either up or down, and with the wrist flipped or not flipped. For the Puma 560 robot the wrist joint, θ_4, has a large rotational range and can adopt one of two angles that differ by π radians.

More precisely the elbow is above or below the shoulder.

Some different various kinematic configurations are shown in Fig. 7.8. The kinematic configuration returned by `ikine6s` is controlled by one or more of the options:

left or right handed	`'l'`, `'r'`
elbow up or down	`'u'`, `'d'`
wrist flipped or not flipped	`'f'`, `'n'`

Due to mechanical limits on joint angles and possible collisions between links not all eight solutions are physically achievable. It is also possible that no solution can be achieved. For example

```
>> p560.ikine6s( SE3(3, 0, 0) )
Warning: point not reachable
ans =
   NaN   NaN   NaN   NaN   NaN   NaN
```

has failed because the arm is simply not long enough to reach this pose.

A pose may also be unachievable due to singularity where the alignment of axes reduces the effective degrees of freedom (the gimbal lock problem again). The Puma 560 has a wrist singularity when q_5 is equal to zero and the axes of joints 4 and 6 become aligned. In this case the best that `ikine6s` can do is to constrain $q_4 + q_6$ but their individual values are arbitrary. For example consider the configuration

```
>> q = [0 pi/4 pi 0.1 0 0.2];
```

for which $q_4 + q_6 = 0.3$. The inverse kinematic solution is

```
>> p560.ikine6s(p560.fkine(q), 'ru')
ans =
   -0.0000    0.7854    3.1416   -3.0409    0.0000   -2.9423
```

which has quite different values for q_4 and q_6 but their sum

```
>> q(4)+q(6)
ans =
    0.3000
```

remains the same.

7.2.2.2 Numerical Solution

For the case of robots which do not have six joints and a spherical wrist we need to use an iterative numerical solution. Continuing with the example of the previous section we use the method `ikine` to compute the general inverse kinematic solution

```
>> T = p560.fkine(qn)
ans =
   -0.0000    0.0000    1.0000    0.5963
   -0.0000    1.0000   -0.0000   -0.1501
   -1.0000   -0.0000   -0.0000   -0.0144
         0         0         0    1.0000
>> qi = p560.ikine(T)
qi =
    0.0000   -0.8335    0.0940   -0.0000   -0.8312    0.0000
```

which is different to the original value

```
>> qn
qn =
         0    0.7854    3.1416         0    0.7854         0
```

but does result in the correct tool pose

```
>> p560.fkine(qi)
ans =
   -0.0000    0.0000    1.0000    0.5963
   -0.0000    1.0000   -0.0000   -0.1501
   -1.0000   -0.0000   -0.0000   -0.0144
         0         0         0    1.0000
```

Plotting the pose

```
>> p560.plot(qi)
```

shows clearly that `ikine` has found the elbow-down configuration.

A limitation of this general numeric approach is that it does not provide explicit control over the arm's kinematic configuration as did the analytic approach – the only control is implicit via the initial value of joint coordinates (which defaults to zero). If we specify the initial joint coordinates

```
>> qi = p560.ikine(T, 'q0', [0 0 3 0 0 0])
qi =
    0.0000    0.7854    3.1416    0.0000    0.7854   -0.0000
```

we have forced the solution to converge on the elbow-up configuration.◂

When solving for a trajectory as on p. 204 the inverse kinematic solution for one point is used to initialize the solution for the next point on the path.

As would be expected the general numerical approach of `ikine` is considerably slower than the analytic approach of `ikine6s`. However it has the great advantage of being able to work with manipulators at singularities and manipulators with less than six or more than six joints. Details of the principle behind `ikine` is provided in Sect. 8.6.

7.2.2.3 Under-Actuated Manipulator

An under-actuated manipulator is one that has fewer than six joints, and is therefore limited in the end-effector poses that it can attain. SCARA robots such as shown on page 191 are a common example. They typically have an x-y-z-θ task space, $\mathcal{T} \subset \mathbb{R}^3 \times \mathbb{S}^1$ and a configuration space $\mathcal{C} \subset (\mathbb{S}^1)^3 \times \mathbb{R}$.

We will load a model of SCARA robot

```
>> mdl_cobra600
>> c600
c600 =
Cobra600 [Adept]:: 4 axis, RRPR, stdDH
+---+-----------+-----------+-----------+-----------+-----------+
| j |     theta |         d |         a |     alpha |    offset |
+---+-----------+-----------+-----------+-----------+-----------+
|  1|         q1|      0.387|      0.325|          0|          0|
|  2|         q2|          0|      0.275|      3.142|          0|
|  3|          0|         q3|          0|          0|          0|
|  4|         q4|          0|          0|          0|          0|
+---+-----------+-----------+-----------+-----------+-----------+
```

and then define a desired end-effector pose

```
>> T = SE3(0.4, -0.3, 0.2) * SE3.rpy(30, 40, 160, 'deg')
```

where the end-effector approach vector is pointing downward but is not vertically aligned. This *pose* is over-constrained for the 4-joint SCARA robot – the tool physically cannot meet the orientation requirement for an approach vector that is not vertically aligned. Therefore we require the `ikine` method to not consider rotation about the x- and y-axes when computing the end-effector pose error. We achieve this by specifying a mask vector as the fourth argument

```
>> q = c600.ikine(T, 'mask', [1 1 1 0 0 1])
q =
   -0.1110   -1.1760    0.1870   -0.8916
```

The elements of the mask vector correspond respectively to the three translations and three orientations: t_x, t_y, t_z, r_x, r_y, r_z in the end-effector coordinate frame. In this example we specified that rotation about the x- and y-axes are to be ignored (the zero elements). The resulting joint angles correspond to an achievable end-effector pose

```
>> Ta = c600.fkine(q);
>> Ta.print('xyz')
t = (0.4, -0.3, 0.2), RPY/xyz = (22.7, 0, 180) deg
```

which has the desired translation and yaw angle, but the roll and pitch angles are incorrect, as *we allowed* them to be. They are what the robot mechanism actually permits. We can also compare the desired and achievable poses graphically

```
>> trplot(T, 'color', 'b')
>> hold on
>> trplot(Ta, 'color', 'r')
```

7.2.2.4 Redundant Manipulator

A redundant manipulator is a robot with more than six joints. As mentioned previously, six joints is theoretically sufficient to achieve any desired pose in a Cartesian taskspace $\mathcal{T} \subset \mathbf{SE}(3)$. However practical issues such as joint limits and singularities mean that not all poses within the robot's reachable space can be achieved. Adding additional joints is one way to overcome this problem but results in an infinite number of joint-coordinate solutions. To find a single solution we need to introduce constraints – a common one is the minimum-norm constraint which returns a solution where the joint-coordinate vector elements have the smallest magnitude.

We will illustrate this with the Baxter robot shown in Fig. 7.1b. This is a two armed robot, and each arm has 7 joints. We load the Toolbox model

```
>> mdl_baxter
```

which defines two SerialLink objects in the workspace, one for each arm. We will work with the left arm

```
>> left
left =
Baxter LEFT [Rethink Robotics]:: 7 axis, RRRRRRR, stdDH
+---+-----------+-----------+-----------+-----------+-----------+
| j |     theta |         d |         a |     alpha |    offset |
+---+-----------+-----------+-----------+-----------+-----------+
|  1|         q1|       0.27|      0.069|     -1.571|          0|
|  2|         q2|          0|          0|      1.571|      1.571|
|  3|         q3|      0.364|      0.069|     -1.571|          0|
|  4|         q4|          0|          0|      1.571|          0|
|  5|         q5|      0.374|       0.01|     -1.571|          0|
|  6|         q6|          0|          0|      1.571|          0|
|  7|         q7|       0.28|          0|          0|          0|
+---+-----------+-----------+-----------+-----------+-----------+
base:    t = (0.064614,0.25858,0.119), RPY/xyz = (0, 0, 45) deg
```

which we can see has a base offset that reflects where the arm is attached to Baxter's torso. We want the robot to move to this pose

```
>> TE = SE3(0.8, 0.2, -0.2) * SE3.Ry(pi);
```

which has its approach vector downward. The required joint angles are obtained using the numerical inverse kinematic solution and

```
>> q = left.ikine(TE)
q =
    0.0895   -0.0464   -0.4259    0.6980   -0.4248    1.0179    0.2998
```

is the joint-angle vector with the smallest norm that results in the desired end-effector pose. We can verify this by computing the forward kinematics or plotting

```
>> left.fkine(q).print('xyz')
t = (0.8, 0.2, -0.2), RPY/xyz = (180, 180, 180) deg
>> left.plot(q)
```

7.3 Trajectories

One of the most common requirements in robotics is to move the end-effector smoothly from pose A to pose B. Building on what we learned in Sect. 3.3 we will discuss two approaches to generating such trajectories: straight lines in configuration space and straight lines in task space. These are known respectively as joint-space and Cartesian motion.

7.3.1 Joint-Space Motion

In this robot configuration, similar to Fig. 7.6d, we specify the pose to include a rotation so that the end-effector *z*-axis is not pointing straight up in the world *z*-direction. For the Puma 560 robot this would be physically impossible to achieve in the elbow-up configuration.

Consider the end-effector moving between two Cartesian poses◂

```
>> T1 = SE3(0.4,  0.2, 0) * SE3.Rx(pi);
>> T2 = SE3(0.4, -0.2, 0) * SE3.Rx(pi/2);
```

which describe points in the *xy*-plane with different end-effector orientations. The joint coordinate vectors associated with these poses are

```
>> q1 = p560.ikine6s(T1);
>> q2 = p560.ikine6s(T2);
```

and we require the motion to occur over a time period of 2 seconds in 50 ms time steps

```
>> t = [0:0.05:2]';
```

A joint-space trajectory is formed by smoothly interpolating between the joint configurations `q1` and `q2`. The scalar interpolation functions `tpoly` or `lspb` from Sect. 3.3.1 can be used in conjunction with the multi-axis *driver* function `mtraj`

```
>> q = mtraj(@tpoly, q1, q2, t);
```

or

```
>> q = mtraj(@lspb, q1, q2, t);
```

which each result in a 50 × 6 matrix `q` with one row per time step and one column per joint. From here on we will use the equivalent `jtraj` convenience function

```
>> q = jtraj(q1, q2, t);
```

This is equivalent to `mtraj` with `tpoly` interpolation but optimized for the multi-axis case and also allowing initial and final velocity to be set using additional arguments.

For `mtraj` and `jtraj` the final argument can be a time vector, as here, or an integer specifying the number of time steps.

We can obtain the joint velocity and acceleration vectors, as a function of time, through optional output arguments

```
>> [q,qd,qdd] = jtraj(q1, q2, t);
```

An even more concise way to achieve the above steps is provided by the `jtraj` method of the `SerialLink` class

```
>> q = p560.jtraj(T1, T2, t)
```

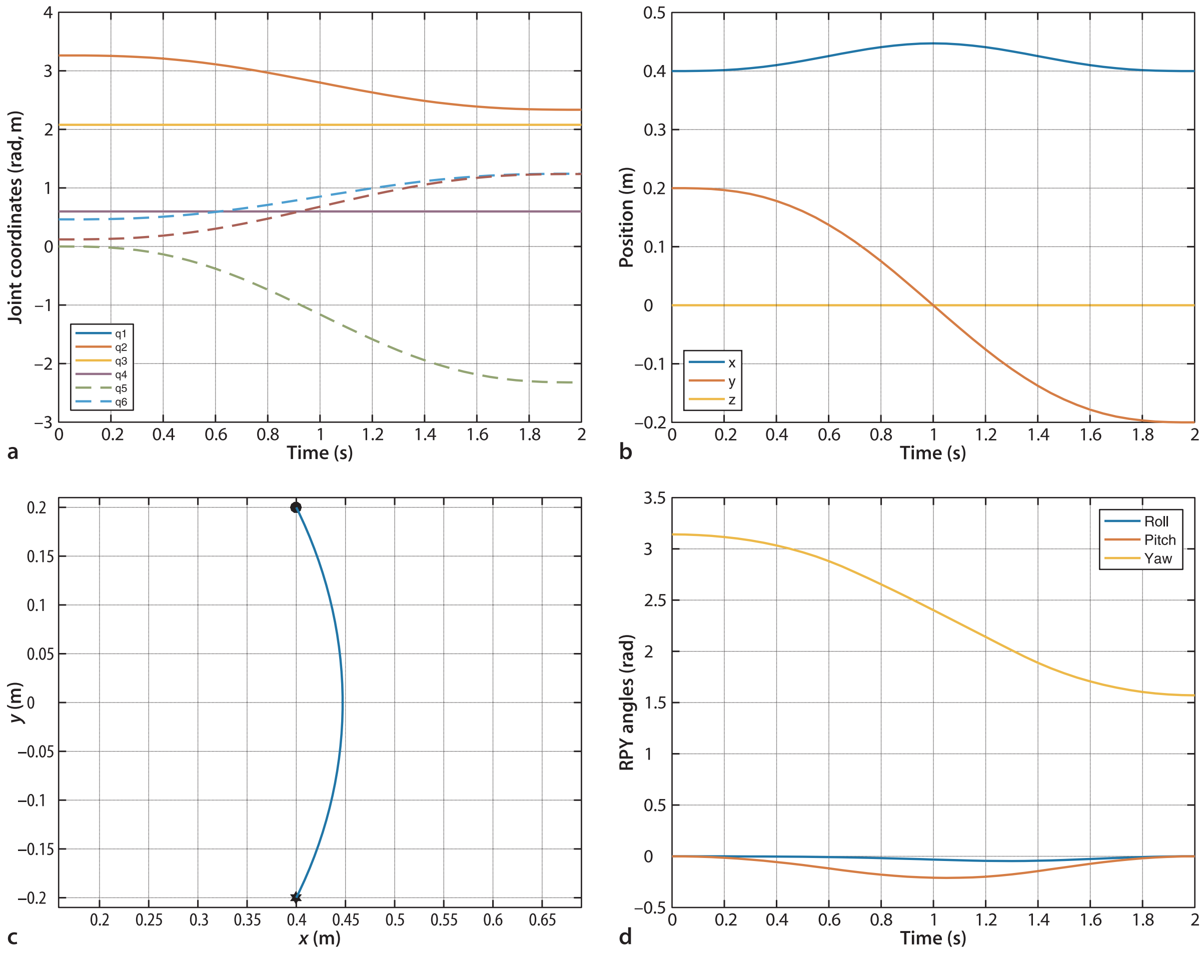


Fig. 7.9. Joint-space motion. **a** Joint coordinates versus time; **b** Cartesian position versus time; **c** Cartesian position locus in the *xy*-plane **d** roll-pitch-yaw angles versus time

The trajectory is best viewed as an animation

```
>> p560.plot(q)
```

but we can also plot the joint angle, for instance q_2, versus time

```
>> plot(t, q(:,2))
```

or all the angles versus time

```
>> qplot(t, q);
```

as shown in Fig. 7.9a. The joint coordinate trajectory is smooth but we do not know how the robot's end-effector will move in Cartesian space. However we can easily determine this by applying forward kinematics to the joint coordinate trajectory

```
>> T = p560.fkine(q);
```

which results in an array of SE3 objects. The translational part of this trajectory is

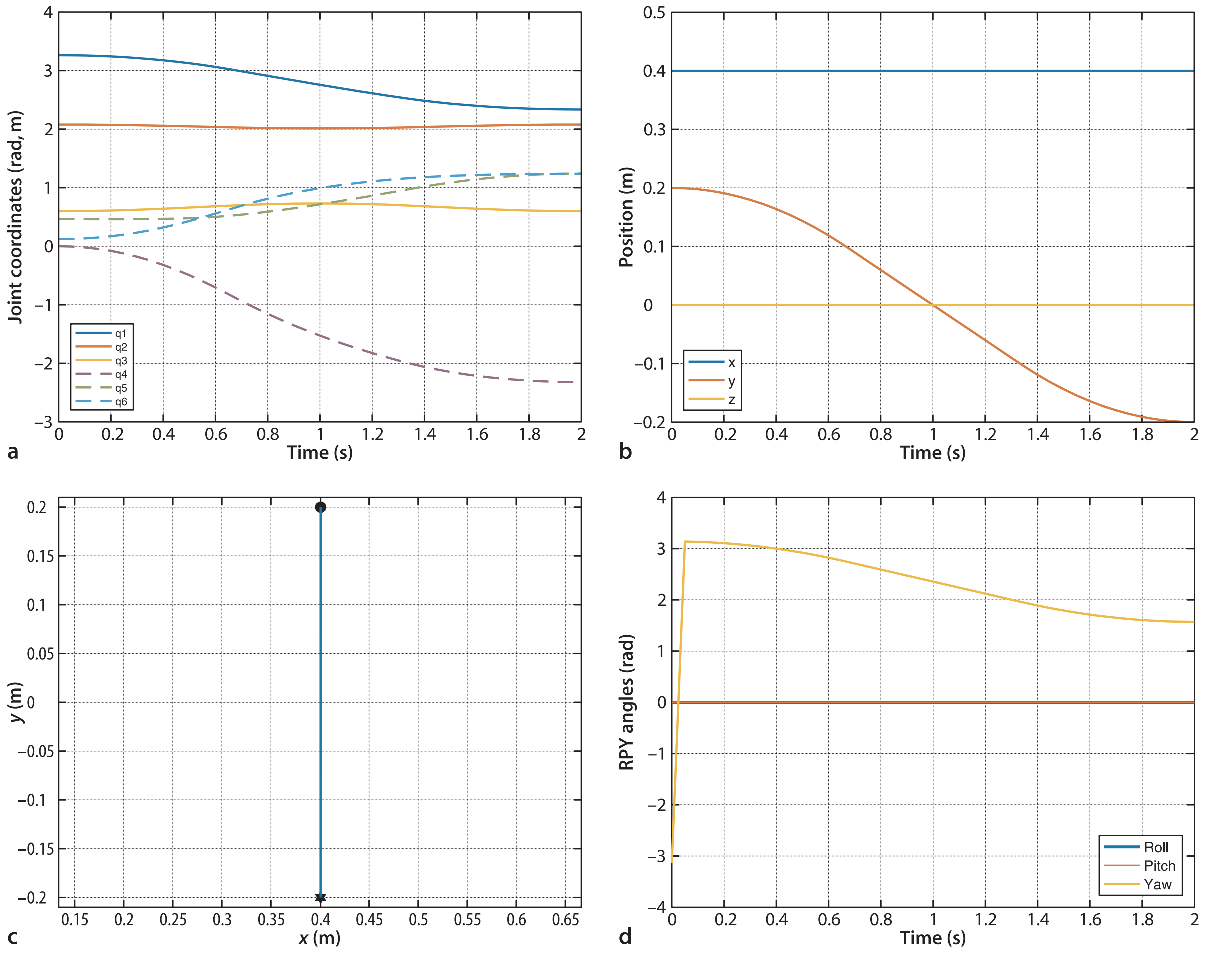
```
>> p = T.transl;
```

which is in matrix form

```
>> about(p)
p [double] : 41x3 (984 bytes)
```

and has one column per time step, and each column is the end-effector position vector. This is plotted against time in Fig. 7.9b. The path of the end-effector in the *xy*-plane

Fig. 7.10. Cartesian motion. **a** Joint coordinates versus time; **b** Cartesian position versus time; **c** Cartesian position locus in the *xy*-plane; **d** roll-pitch-yaw angles versus time



```
>> plot(p(1,:), p(2,:))
```

is shown in Fig. 7.9c and it is clear that the path is not a straight line. This is to be expected since we only specified the Cartesian coordinates of the end-points. As the robot rotates about its waist joint during the motion the end-effector will naturally follow a circular arc. In practice this could lead to collisions between the robot and nearby objects even if they do not lie on the path between poses A and B. The orientation of the end-effector, in XYZ roll-pitch-yaw angle form, can also be plotted against time

```
>> plot(t, T.torpy('xyz'))
```

as shown in Fig. 7.9d. Note that the yaw angle▸ varies from 0 to $\frac{\pi}{2}$ radians as we specified. However while the roll and pitch angles have met their boundary conditions they have varied along the path.

Rotation about *x*-axis for a robot end-effector from Sect. 2.2.1.2.

7.3.2 Cartesian Motion

For many applications we require straight-line motion in Cartesian space which is known as Cartesian motion. This is implemented using the Toolbox function `ctraj` which was introduced in Sect. 3.3.5. Its usage is very similar to `jtraj`

```
>> Ts = ctraj(T1, T2, length(t));
```

where the arguments are the initial and final pose and the *number of* time steps and it returns the trajectory as an array of `SE3` objects.

As for the previous joint-space example we will extract and plot the translation

```
>> plot(t, Ts.transl);
```

and orientation components

```
>> plot(t, Ts.torpy('xyz'));
```

of this motion which is shown in Fig. 7.10 along with the path of the end-effector in the *xy*-plane. Compared to Fig. 7.9 we note some important differences. Firstly the end-effector follows a straight line in the *xy*-plane as shown in Fig. 7.10c. Secondly the roll and pitch angles shown in Fig. 7.10d are constant at zero along the path.

The corresponding joint-space trajectory is obtained by applying the inverse kinematics

```
>> qc = p560.ikine6s(Ts);
```

and is shown in Fig. 7.10a. While broadly similar to Fig. 7.9a the minor differences are what result in the straight line Cartesian motion.

7.3.3 Kinematics in Simulink

We can also implement this example in Simulink®

```
>> sl_jspace
```

and the block diagram model is shown in Fig. 7.11. The parameters of the `jtraj` block are the initial and final values for the joint coordinates and the time duration of the motion segment. The smoothly varying joint angles are wired to a `plot` block which will animate a robot in a separate window, and to an `fkine` block to compute the forward kinematics. Both the `plot` and `fkine` blocks have a parameter which is a `SerialLink` object, in this case `p560`. The Cartesian position of the end-effector pose is extracted using the `T2xyz` block which is analogous to the Toolbox function `transl`. The `XY Graph` block plots `y` against `x`.

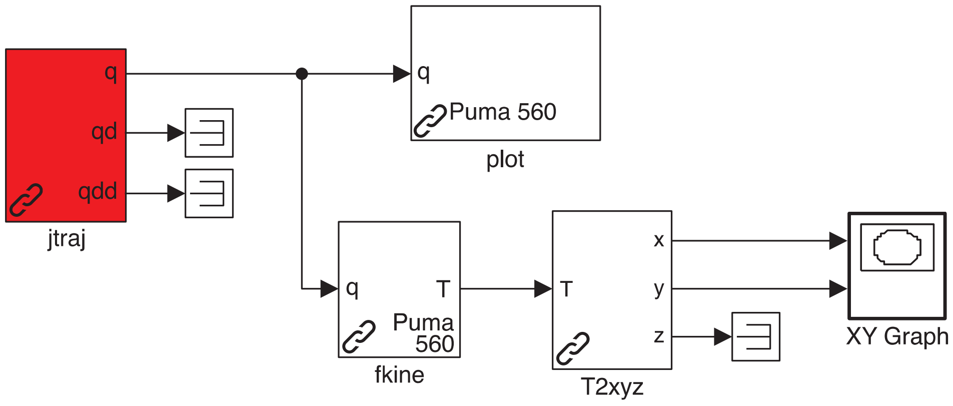


Fig. 7.11. Simulink model `sl_jspace` for joint-space motion

7.3.4 Motion through a Singularity

We have already briefly touched on the topic of singularities (page 209) and we will revisit it again in the next chapter. In the next example we deliberately choose a trajectory that moves through a robot wrist singularity. We change the Cartesian end-points of the previous example to

```
>> T1 = SE3(0.5,  0.3, 0.44) * SE3.Ry(pi/2);
>> T2 = SE3(0.5, -0.3, 0.44) * SE3.Ry(pi/2);
```

which results in motion in the y-direction with the end-effector z-axis pointing in the world x-direction. The Cartesian path is

```
>> Ts = ctraj(T1, T2, length(t));
```

which we convert to joint coordinates

```
>> qc = p560.ikine6s(Ts)
```

and is shown in Fig. 7.12a. At time $t \approx 0.7$ s we observe that the rate of change of the wrist joint angles q_4 and q_6 has become very high.◂ The cause is that q_5 has become almost zero which means that the q_4 and q_6 rotational axes are almost aligned – another gimbal lock situation or singularity.

q_6 has increased rapidly, while q_4 has decreased rapidly and wrapped around from $-\pi$ to π. This counter-rotational motion of the two joints means that the gripper does not rotate but the two motors are working hard.

The joint axis alignment means that the robot has lost one degree of freedom and is now effectively a 5-axis robot. Kinematically we can only solve for the sum $q_4 + q_6$ and there are an infinite number of solutions for q_4 and q_6 that would have the same sum. From Fig. 7.12b we observe that the generalized inverse kinematics method `ikine` handles the singularity with far less unnecessary joint motion. This is a consequence of the minimum-norm solution which has returned the smallest magnitude q_4 and q_6 which have the correct sum. The joint-space motion between the two poses, Fig. 7.12c, is immune to this problem since it is does not involve inverse kinematics. However it will not maintain the orientation of the tool in the x-direction for the whole path – only at the two end points.

The dexterity of a manipulator, its ability to move easily in any direction, is referred to as its manipulability. It is a scalar measure, high is good, and can be computed for each point along the trajectory

```
>> m = p560.maniplty(qc);
```

and is plotted in Fig. 7.12d. This shows that manipulability was almost zero around the time of the rapid wrist joint motion. Manipulability and the generalized inverse kinematics function are based on the manipulator's Jacobian matrix which is the topic of the next chapter.

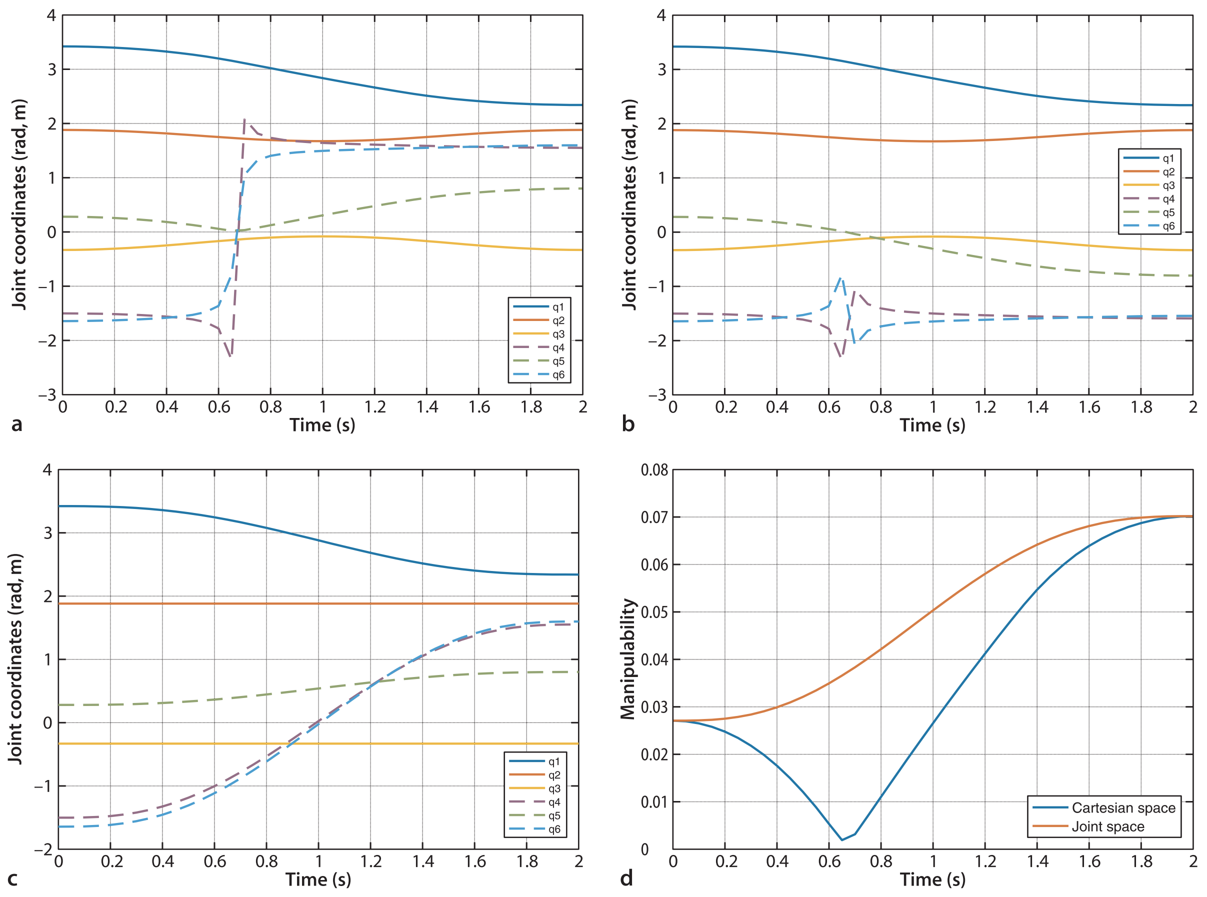


Fig. 7.12. Cartesian motion through a wrist singularity. **a** Joint coordinates computed by inverse kinematics (`ikine6s`); **b** joint coordinates computed by numerical inverse kinematics (`ikine`); **c** joint coordinates for joint-space motion; **d** manipulability

7.3.5 Configuration Change

Earlier (page 208) we discussed the kinematic configuration of the manipulator arm and how it can work in a left- or right-handed manner and with the elbow up or down. Consider the problem of a robot that is working for a while left-handed at one work station, then working right-handed at another. Movement from one configuration to another ultimately results in no change in the end-effector pose since both configuration have the same forward kinematic solution – therefore we *cannot* create a trajectory in Cartesian space. Instead we must use joint-space motion.

For example to move the robot arm from the right- to left-handed configuration we first define some end-effector pose

```
>> T = SE3(0.4, 0.2, 0) * SE3.Rx(pi);
```

and then determine the joint coordinates for the right- and left-handed elbow-up configurations

```
>> qr = p560.ikine6s(T, 'ru');
>> ql = p560.ikine6s(T, 'lu');
```

and then create a joint-space trajectory between these two joint coordinate vectors

```
>> q = jtraj(qr, ql, t);
```

Although the initial and final end-effector pose is the same, the robot makes some quite significant joint space motion as shown in Fig. 7.13 – in the real world you need to be careful the robot doesn't hit something. Once again, the best way to visualize this is in animation

```
>> p560.plot(q)
```

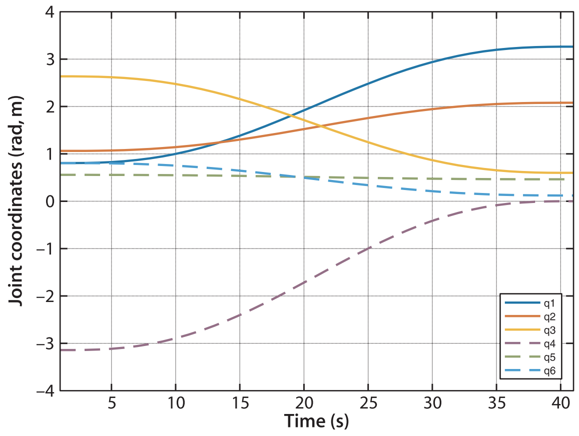


Fig. 7.13. Joint space motions for configuration change from right-handed to left-handed

7.4 Advanced Topics

7.4.1 Joint Angle Offsets

The pose of the robot with zero joint angles is an arbitrary decision of the robot designer and might even be a mechanically unachievable pose. For the Puma robot the zero-angle pose is a nonobvious *L-shape* with the upper arm horizontal and the lower arm vertically upward as shown in Fig. 7.6a. This is a consequence of constraints imposed by the Denavit-Hartenberg formalism.

The joint coordinate offset provides a mechanism to set an arbitrary configuration for the zero joint coordinate case. The offset vector, $\boldsymbol{q}_0$, is added to the user specified joint angles before any kinematic or dynamic function is invoked,◂ for example

It is actually implemented within the `Link` object.

$$\xi_E = \mathcal{K}(\boldsymbol{q} + \boldsymbol{q}_0) \tag{7.6}$$

Similarly it is subtracted after an operation such as inverse kinematics

$$\boldsymbol{q} = \mathcal{K}^{-1}(\xi_E) - \boldsymbol{q}_0 \tag{7.7}$$

The offset is set by assigning the `offset` property of the `Link` object, or giving the `'offset'` option to the `SerialLink` constructor.

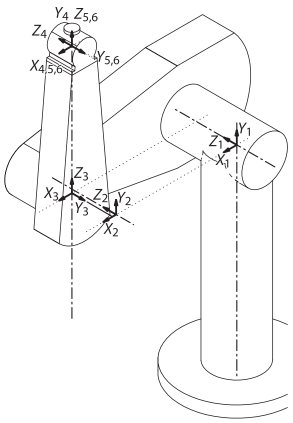


Fig. 7.14. Puma 560 robot coordinate frames. Standard Denavit-Hartenberg link coordinate frames for Puma in the zeroangle pose (Corke 1996b)

7.4.2 Determining Denavit-Hartenberg Parameters

The classical method of determining Denavit-Hartenberg parameters is to systematically assign a coordinate frame to each link. The link frames for the Puma robot using the standard Denavit-Hartenberg formalism are shown in Fig. 7.14. However there are strong constraints on placing each frame since joint rotation must be about the *z*-axis and the link displacement must be in the *x*-direction. This in turn imposes constraints on the placement of the coordinate frames for the base and the end-effector, and ultimately dictates the zero-angle pose just discussed. Determining the Denavit-Hartenberg parameters and link coordinate frames for a completely new mechanism is therefore more difficult than it should be – even for an experienced roboticist.

An alternative approach, supported by the Toolbox, is to simply describe the manipulator as a series of elementary translations and rotations from the base to the tip of the end-effector as we discussed in Sect. 7.1.2. Some of the elementary operations are constants such as translations that represent link lengths or offsets, and some are functions of the generalized joint coordinates q_j. Unlike the conventional approach we impose no constraints on the axes about which these rotations or translations can occur.

For the Puma robot shown in Fig. 7.4 we first define a convenient coordinate frame at the base and then write down the sequence of translations and rotations, from "toe to tip", in a string▸

All lengths must start with *L* and negative signs cannot be used in the string, but the values themselves can be negative. You can generate this string from an ETS3 sequence (page 196) using its `string` method.

```
>> s = 'Tz(L1) Rz(q1) Ry(q2) Ty(L2) Tz(L3) Ry(q3) Tx(L4) Ty(L5)
              Tz(L6) Rz(q4) Ry(q5) Rz(q6)'
```

Note that we have described the second joint as `Ry(q2)`, a rotation about the *y*-axis, which is not permissible using the Denavit-Hartenberg formalism.

This string is input to a symbolic algebra function▸

Written in Java, the MATLAB® Symbolic Math Toolbox™ is not required.

```
>> dh = DHFactor(s);
```

which returns a `DHFactor` object▸ that holds the kinematic structure of the robot that has been factorized into Denavit-Hartenberg parameters. We can display this in a human-readable form

Actually a Java object.

```
>> dh
dh =
DH(q1, L1, 0, -90).DH(q2+90, 0, -L3, 0).DH(q3-90, L2+L5, L4, 90).
DH(q4, L6, 0, -90).DH(q5, 0, 0, 90).DH(q6, 0, 0, 0)
```

which shows the Denavit-Hartenberg parameters for each joint in the order θ, d, a and α. Joint angle offsets (the constants added to or subtracted from joint angle variables such as `q2` and `q3`) are generated automatically, as are base and tool transformations. The object can generate a string that is a complete Toolbox command to create the robot named "puma"

```
>> cmd = dh.command('puma')
cmd =
SerialLink([0, L1, 0, -pi/2, 0; 0, 0, -L3, 0, 0; 0, L2+L5, L4,↵
pi/2, 0; 0, L6, 0, -pi/2, 0; 0, 0, 0, pi/2, 0; 0, 0, 0, 0, 0; ], ...
 'name', 'puma', ...
 'base', eye(4,4), 'tool', eye(4,4), ...
 'offset', [0 pi/2 -pi/2 0 0 0 ])
```

which can be executed

```
>> robot = eval(cmd)
```

to create a workspace variable called `robot` that is a `SerialLink` object.▸

The length parameters `L1` to `L6` must be defined in the workspace first.

7.4.3 Modified Denavit-Hartenberg Parameters

The Denavit-Hartenberg notation introduced in this chapter is commonly used and described in many robotics textbooks. Craig (1986) first introduced the modified Denavit-Hartenberg parameters where the link coordinate frames shown in Fig. 7.15 are attached to the near (proximal), rather than the far (distal) end of each link. This modified notation is in some ways clearer and tidier and is also now commonly used. However its introduction has increased the scope for confusion, particularly for those who are new to robot kinematics. The root of the problem is that the algorithms for kinematics, Jacobians and dynamics depend on the kinematic conventions used. According to Craig's convention the link transform matrix is

$$^{j-1}\xi_j\left(\alpha_{j-1}, a_{j-1}, d_j, \theta_j\right) = \mathcal{R}_x\left(\alpha_{j-1}\right) \oplus \mathcal{T}_x\left(a_{j-1}\right) \oplus \mathcal{T}_z\left(d_j\right) \oplus \mathcal{R}_z\left(\theta_j\right) \tag{7.8}$$

denoted in that book as $^{j-1}_{\ \ j}\boldsymbol{A}$. This has the same terms as Eq. 7.2 but in a different order – remember rotations are not commutative – and this is the nub of the problem. a_j is

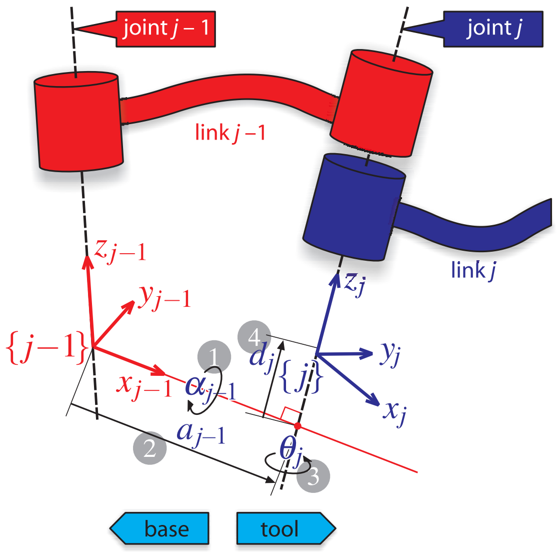


Fig. 7.15. Definition of modified Denavit and Hartenberg link parameters. The colors *red* and *blue* denote all things associated with links $j-1$ and j respectively. The *numbers in circles* represent the order in which the elementary transforms are applied

always the length of link j, but it is the displacement between the origins of frame $\{j\}$ and frame $\{j+1\}$ in one convention, and frame $\{j-1\}$ and frame $\{j\}$ in the other.

If you intend to build a Toolbox robot model from a table of kinematic parameters provided in a research paper it is really important to know which convention the author of the table used. Too often this important fact is not mentioned. An important clue lies in the column headings. If they all have the same subscript, i.e. θ_j, d_j, a_j and α_j then this is standard Denavit-Hartenberg notation. If half the subscripts are different, i.e. θ_j, d_j, a_{j-1} and α_{j-1} then you are dealing with modified Denavit-Hartenberg notation. In short, you must know which kinematic convention your Denavit-Hartenberg parameters conform to.

You can also help the field when publishing by stating clearly which kinematic convention is used for your parameters.

The Toolbox can handle either form, it only needs to be specified, and this is achieved using variant classes when creating a link object

```
>> L1 = RevoluteMDH('d', 1)
L1 =
Revolute(mod): theta=q, d=1, a=0, alpha=0, offset=0
```

Everything else from here on, creating the robot object, kinematic and dynamic functions works as previously described.

The two forms can be interchanged by considering the link transform as a string of elementary rotations and translations as in Eq. 7.2 or Eq. 7.8. Consider the transformation chain for standard Denavit-Hartenberg notation

$$\underbrace{\mathcal{R}_z(\theta_1) \oplus \mathcal{T}_z(d_1) \oplus \mathcal{T}_x(a_1) \oplus \mathcal{R}_x(\alpha_1)}_{\mathrm{DH}_1} \oplus \underbrace{\mathcal{R}_z(\theta_2) \oplus \mathcal{T}_z(d_2) \oplus \mathcal{T}_x(a_2) \oplus \mathcal{R}_x(\alpha_2)}_{\mathrm{DH2}} \cdots$$

which we can regroup as

$$\underbrace{\mathcal{R}_z(\theta_1) \oplus \mathcal{T}_z(d_1)}_{\mathrm{base}} \oplus \underbrace{\mathcal{T}_x(a_1)\mathcal{R}_x(\alpha_1) \oplus \mathcal{R}_z(\theta_2) \oplus \mathcal{T}_z(d_2)}_{\mathrm{MDH}_1} \oplus \underbrace{\mathcal{T}_x(a_2) \oplus \mathcal{R}_x(\alpha_2) \cdots}_{\mathrm{MDH}_2}$$

where the terms marked as MDH_j have the form of Eq. 7.8 taking into account that translation along, and rotation about the same axis *is* commutative, that is, $\mathcal{R}_i(\theta) \oplus \mathcal{T}_i(d) = \mathcal{T}_i(d) \oplus \mathcal{R}_i(\theta)$ for $i \in \{x, y, z\}$.

7.5 Applications

7.5.1 Writing on a Surface [examples/drawing.m]

Our goal is to create a trajectory that will allow a robot to draw a letter. The Toolbox comes with a preprocessed version of the Hershey font▸

Developed by Dr. Allen V. Hershey at the Naval Weapons Laboratory in 1967, data from http://paulbourke.net/dataformats/hershey.

```
>> load hershey
```

as a cell array of character descriptors. For an upper-case 'B'

```
>> B = hershey{'B'}
B = 
    stroke: [2x23 double]
     width: 0.8400
       top: 0.8400
    bottom: 0
```

the structure describes the dimensions of the character, vertically from 0 to 0.84 and horizontally from 0 to 0.84▸. The path to be drawn is

This is a variable-width font, and all characters fit within a unit-height rectangle.

```
>> B.stroke
ans =
  Columns 1 through 11
    0.1600    0.1600       NaN    0.1600    0.5200    0.6400    ...
    0.8400         0       NaN    0.8400    0.8400    0.8000    ...
```

where the rows are the x- and y-coordinates respectively, and a column of NaNs indicates the end of a segment – the pen is lifted and placed down again at the beginning of the next segment. We perform some processing

```
>> path = [ 0.25*B.stroke; zeros(1,numcols(B.stroke))];
>> k = find(isnan(path(1,:)));
>> path(:,k) = path(:,k-1); path(3,k) = 0.2;
```

to scale the path by 0.25 so that the character is around 20 cm tall, append a row of zeros (add z-coordinates to this 2-dimensional path), find the columns that contain NaNs and replace them with the preceding column but with the z-coordinate set to 0.2 in order to lift the pen off the surface.

Next we convert this to a continuous trajectory

```
>> traj = mstraj(path(:,2:end)', [0.5 0.5 0.5], [], path(:,1)',↵
  0.02, 0.2);
```

where the second argument is the maximum speed in the x-, y- and z-directions, the fourth argument is the initial coordinate followed by the sample interval and the acceleration time. The number of steps in the interpolated path is

```
>> about(traj)
traj [double] : 487x3 (11.7 kB)
```

and will take

```
>> numrows(traj) * 0.02
ans =
   9.7400
```

seconds to execute at the 20 ms sample interval. The trajectory can be plotted

```
>> plot3(traj(:,1), traj(:,2), traj(:,3))
```

as shown in Fig. 7.16.

We now have a sequence of 3-dimensional points but the robot end-effector has a pose, not just a position, so we need to attach a coordinate frame to every point. We assume that the robot is writing on a horizontal surface so these frames must have their approach vector pointing downward, that is, $\boldsymbol{a} = [0, 0, -1]$, with the gripper ar-

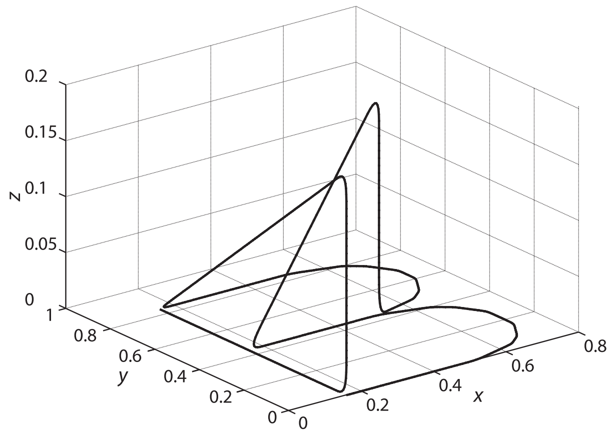


Fig. 7.16. The end-effector path drawing the letter 'B'

bitrarily oriented in the y-direction with $\boldsymbol{o} = [0, 1, 0]$. The character is also placed at (0.6, 0, 0) in the workspace, and all this is achieved by

```
>> Tp = SE3(0.6, 0, 0) * SE3(traj) * SE3.oa( [0 1 0], [0 0 -1]);
```

Now we can apply inverse kinematics

```
>> q = p560.ikine6s(Tp);
```

to determine the joint coordinates and then animate it

```
>> p560.plot(q)
```

We have not considered the force that the robot-held pen exerts on the paper, we cover force control in Chap. 9. In a real implementation of this example it would be prudent to use a spring to push the pen against the paper with sufficient force to allow it to write.

The Puma is drawing the letter 'B', and lifting its pen in between strokes! The approach is quite general and we could easily change the size of the letter, write whole words and sentences, write on an arbitrary plane or use a robot with quite different kinematics.◀

7.5.2 A Simple Walking Robot [examples/walking.m]

Four legs good, two legs bad!
Snowball the pig, Animal Farm by George Orwell

Our goal is to create a four-legged walking robot. We start by creating a 3-axis robot *arm* that we use as a leg, plan a trajectory for the leg that is suitable for walking, and then instantiate four instances of the leg to create the walking robot.

Kinematics

Kinematically a robot leg is much like a robot arm. For this application a three joint serial-link manipulator is sufficient since the foot has point contact with the ground and orientation is not important. Determining the Denavit-Hartenberg parameters, even for a simple robot like this, is an involved procedure and the zero-angle offsets need to be determined in a separate step. Therefore we will use the procedure introduced in Sect. 7.4.2.

As always we start by defining our coordinate frame. This is shown in Fig. 7.17 along with the robot leg in its zero-angle pose. We have chosen the aerospace coordinate convention which has the x-axis forward and the z-axis downward, constraining the y-axis to point to the right-hand side. The first joint will be hip motion, forward and backward, which is rotation about the z-axis or $\boldsymbol{R}_z(q1)$. The second joint is hip motion up and down, which is rotation about the x-axis, $\boldsymbol{R}_x(q_2)$. These form a

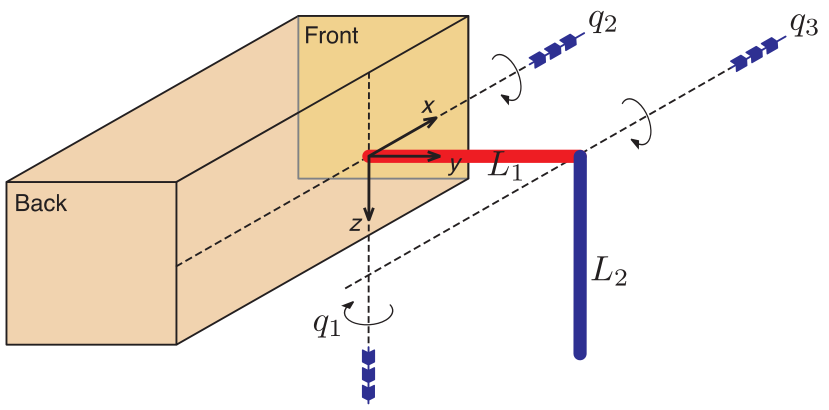


Fig. 7.17. The coordinate frame and axis rotations for the simple leg. The leg is shown in its zero angle pose

spherical hip joint since the axes of rotation intersect. The knee is translated by L_1 in the y-direction or $\boldsymbol{T}_y(L_1)$. The third joint is knee motion, toward and away from the body, which is $\boldsymbol{R}_x(q_3)$. The foot is translated by L_2 in the z-direction or $\boldsymbol{T}_z(L_2)$. The transform sequence of this robot, from hip to toe, is therefore $\boldsymbol{R}_z(q1)\boldsymbol{R}_x(q_2)\boldsymbol{T}_y(L_1)\boldsymbol{R}_x(q_3)\boldsymbol{T}_z(L_2)$.

Using the technique of Sect. 7.4.2 we write this sequence as the string

```
>> s = 'Rz(q1) Rx(q2) Ty(L1) Rx(q3) Tz(L2)';
```

The string can be automatically manipulated into Denavit-Hartenberg factors

```
>> dh = DHFactor(s)
DH(q1+90, 0, 0, 90).DH(q2, 0, L1, 0).DH(q3-90, 0, -L2, 0)↵
.Rz(+90).Rx(-90).Rz(-90)
```

The last three terms in this factorized sequence is a tool transform

```
 >> dh.tool
ans =
trotz(pi/2)*trotx(-pi/2)*trotz(-pi/2)
```

that changes the orientation of the frame at the foot. However for this problem the foot is simply a point that contacts the ground so we are not concerned about its orientation. The method `dh.command` generates a string that is the Toolbox command to create a `SerialLink` object

```
>> dh.command('leg')
ans =
SerialLink([0, 0, 0, pi/2, 0; 0, 0, L1, 0, 0; 0, 0, -L2, 0, 0; ],↵
 'name', 'leg', 'base', eye(4,4),↵
 'tool', trotz(pi/2)*trotx(-pi/2)*trotz(-pi/2),↵
 'offset', [pi/2 0 -pi/2 ])
```

which is input to the MATLAB `eval` command

```
>> L1 = 0.1; L2 = 0.1;
>> leg = eval( dh.command('leg') )
>> leg
leg =
leg:: 3 axis, RRR, stdDH, slowRNE
+---+-----------+-----------+-----------+-----------+-----------+
| j |     theta |         d |         a |     alpha |    offset |
+---+-----------+-----------+-----------+-----------+-----------+
|  1|         q1|          0|          0|     1.5708|     1.5708|
|  2|         q2|          0|        0.1|          0|          0|
|  3|         q3|          0|       -0.1|          0|    -1.5708|
+---+-----------+-----------+-----------+-----------+-----------+
tool:    t = (0, 0, 0), RPY/zyx = (0, -90, 0) deg
```

after first setting the length of each leg segment to 100 mm in the MATLAB workspace.

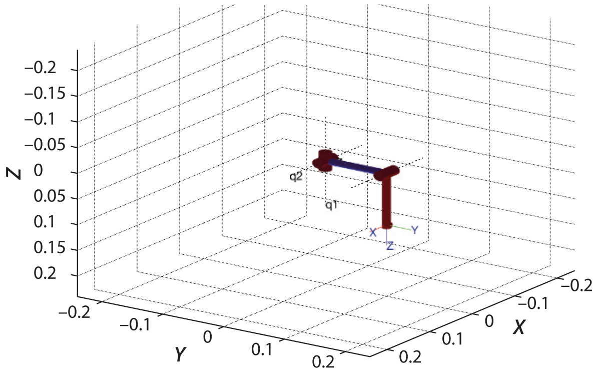


Fig. 7.18. Robot leg in its zero angle pose. Note that the *z*-axis points downward

We perform a quick sanity check of our robot. For zero joint angles the foot is at

```
>> transl( leg.fkine([0,0,0]) )
ans =
         0    0.1000    0.1000
```

as we designed it. We can visualize the zero-angle pose

```
>> leg.plot([0,0,0], 'nobase', 'noshadow', 'notiles')
>> set(gca, 'Zdir', 'reverse'); view(137,48);
```

which is shown in Fig. 7.18. Now we should test that the other joints result in the expected motion. Increasing q_1

```
>> transl( leg.fkine([0.2,0,0]) )
ans =
   -0.0199    0.0980    0.1000
```

results in motion in the *xy*-plane, and increasing q_2

```
>> transl( leg.fkine([0,0.2,0]) )
ans =
   -0.0000    0.0781    0.1179
```

results in motion in the *yz*-plane, as does increasing q_3

```
>> transl( leg.fkine([0,0,0.2]) )
ans =
   -0.0000    0.0801    0.0980
```

We have now created and verified a simple robot leg.

Motion of One Leg

The next step is to define the path that the end-effector of the leg, its foot, will follow. The first consideration is that the end-effector of all feet move backwards at the same speed in the ground plane – propelling the robot's body forward without its feet slipping. Each leg has a limited range of movement so it cannot move backward for very long. At some point we must reset the leg – lift the foot, move it forward and place it on the ground again. The second consideration comes from static stability – the robot must have at least three feet on the ground at all times so each leg must take its turn to reset. This requires that any leg is in contact with the ground for ¾ of the cycle and is resetting for ¼ of the cycle. A consequence of this is that the leg has to move much faster during reset since it has a longer path and less time to do it in.

The required trajectory is defined by the via points

```
>> xf = 50; xb = -xf;  y = 50; zu = 20; zd = 50;
>> path = [xf y zd; xb y zd; xb y zu; xf y zu; xf y zd] * 1e-3;
```

where `xf` and `xb` are the forward and backward limits of leg motion in the *x*-direction (in units of mm), `y` is the distance of the foot from the body in the *y*-direction, and `zu` and `zd`

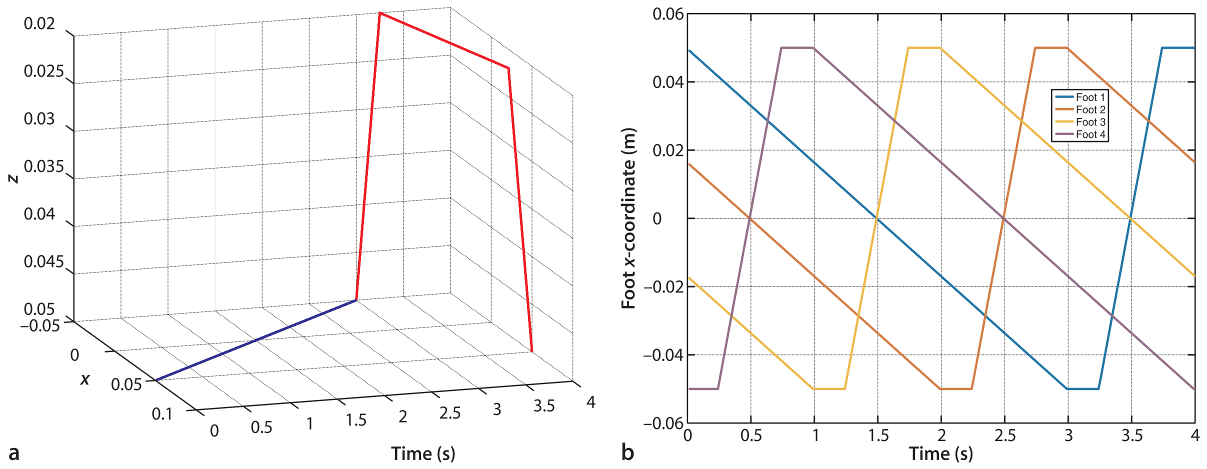


Fig. 7.19. a Trajectory taken by a single foot. Recall from Fig. 7.17 that the *z*-axis is downward. The red segments are the leg reset. **b** The *x*-direction motion of each leg (offset vertically) to show the gait. The leg reset is the period of high *x*-direction velocity

are respectively the height of the foot motion in the *z*-direction for foot up and foot down. In this case the foot moves from 50 mm forward of the hip to 50 mm behind. When the foot is down, it is 50 mm below the hip and it is raised to 20 mm below the hip during reset. The points in `path` define a complete cycle: the start of the stance phase, the end of stance, top of the leg lift, top of the leg return and the start of stance. This is shown in Fig. 7.19a.

Next we sample the multi-segment path at 100 Hz

```
>> p = mstraj(path, [], [0, 3, 0.25, 0.5, 0.25]', path(1,:), 0.01, 0);
```

In this case we have specified a vector of desired segment times rather than maximum joint velocities.▸ The final three arguments are the initial leg configuration, the sample interval and the acceleration time. This trajectory has a total time of 4 s and therefore comprises 400 points.

This way we can ensure that the reset takes exactly one quarter of the cycle.

We apply inverse kinematics to determine the joint angle trajectories required for the foot to follow the computed Cartesian trajectory. This robot is under-actuated so we use the generalized inverse kinematics `ikine` and set the mask so as to solve only for end-effector translation

```
>> qcycle = leg.ikine( SE3(p), 'mask', [1 1 1 0 0 0] );
```

We can view the motion of the leg in animation

```
>> leg.plot(qcycle, 'loop')
```

to verify that it does what we expect: slow motion along the ground, then a rapid lift, forward motion and foot placement. The `'loop'` option displays the trajectory in an endless loop and you need to type control-C to stop it.

Motion of Four Legs

Our robot has width and length

```
>> W = 0.1; L = 0.2;
```

We create multiple instances of the leg by cloning the `leg` object we created earlier, and providing different base transforms so as to attach the legs to different points on the body

```
>> legs(1) = SerialLink(leg, 'name', 'leg1');
>> legs(2) = SerialLink(leg, 'name', 'leg2', 'base', SE3(-L, 0, 0));
>> legs(3) = SerialLink(leg, 'name', 'leg3', 'base', SE3(-L, -W, 0) ↵
 *SE3.Rz(pi));
>> legs(4) = SerialLink(leg, 'name', 'leg4', 'base', SE3(0, -W, 0) ↵
 *SE3.Rz(pi));
```

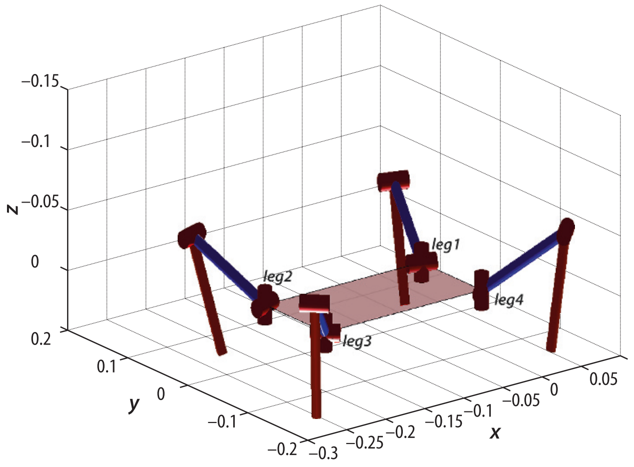


Fig. 7.20. The walking robot

The result is a vector of `SerialLink` objects. Note that legs 3 and 4, on the left-hand side of the body have been rotated about the *z*-axis so that they point away from the body.

As mentioned earlier each leg must take its turn to reset. Since the trajectory is a cycle, we achieve this by having each leg run the trajectory with a phase shift equal to one quarter of the total cycle time. Since the total cycle has 400 points, each leg's trajectory is offset by 100, and we use modulo arithmetic to index into the cyclic gait for each leg. The result is the gait pattern shown in Fig. 7.19b.

The core of the walking program is

```
clf; k = 1;
while 1
    legs(1).plot( gait(qcycle, k, 0,   false) );
    if k == 1, hold on; end
    legs(2).plot( gait(qcycle, k, 100, false) );
    legs(3).plot( gait(qcycle, k, 200, true) );
    legs(4).plot( gait(qcycle, k, 300, true) );
    drawnow
    k = k+1;
end
```

where the function

```
gait(q, k, ph, flip)
```

returns the `k+ph`th element of `q` with modulo arithmetic that considers `q` as a cycle. The argument `flip` reverses the sign of the joint 1 motion for the legs on the left-hand side of the robot. A snapshot from the simulation is shown in Fig. 7.20. The entire implementation, with some additional refinement, is in the file `examples/walking.m` and detailed explanation is provided by the comments.

7.6 Wrapping Up

In this chapter we have learned how to determine the forward and inverse kinematics of a serial-link manipulator arm. Forward kinematics involves compounding the relative poses due to each joint and link, giving the pose of the robot's end-effector relative to its base. Commonly the joint and link structure is expressed in terms of Denavit-Hartenberg parameters for each link. Inverse kinematics is the problem of determining the joint coordinates given the end-effector pose. For simple robots, or those with six joints and a spherical wrist we can compute the inverse kinematics using an analytic solution. This inverse is not unique and the robot may have several joint configurations that result in the same end-effector pose.

For robots which do not have six joints and a spherical wrist we can use an iterative numerical approach to solving the inverse kinematics. We showed how this could be applied to an under-actuated 4-joint SCARA robot and a redundant 7-link robot. We also touched briefly on the topic of singularities which are due to the alignment of joint axes.

We also learned about creating paths to move the end-effector smoothly between poses. Joint-space paths are simple to compute but in general do not result in straight line paths in Cartesian space which may be problematic for some applications. Straight line paths in Cartesian space can be generated but singularities in the workspace may lead to very high joint rates.

Further Reading

Serial-link manipulator kinematics are covered in all the standard robotics textbooks such as the Robotics Handbook (Siciliano and Khatib 2016), Siciliano et al. (2009), Spong et al. (2006) and Paul (1981). Craig's text (2005) is also an excellent introduction to robot kinematics and uses the modified Denavit-Hartenberg notation, and the examples in the third edition are based on an older version of the Robotics Toolbox. Lynch and Park (2017) and Murray et al. (1994) cover the product of exponential approach. An emerging alternative to Denavit-Hartenberg notation is URDF (unified robot description format) which is described at http://wiki.ros.org/urdf.

Siciliano et al. (2009) provide a very clear description of the process of assigning Denavit-Hartenberg parameters to an arbitrary robot. The alternative approach described here based on symbolic factorization was described in detail by Corke (2007). The definitive values for the parameters of the Puma 560 robot are described in the paper by Corke and Armstrong-Hélouvry (1995).

Robotic walking is a huge field in its own right and the example given here is very simplistic. Machines have been demonstrated with complex gaits such as running and galloping that rely on dynamic rather than static balance. A good introduction to legged robots is given in the Robotics Handbook (Siciliano and Khatib 2016, § 17). Robotic hands, grasping and manipulation is another large topic which we have not covered – there is a good introduction in the Robotics Handbook (Siciliano and Khatib 2016, §37, 38).

Parallel-link manipulators were mentioned only briefly on page 192 and have advantages such as increased actuation force and stiffness (since the actuators form a truss-like structure). For this class of mechanism the inverse kinematics is usually closed-form and it is the forward kinematics that requires numerical solution. Useful starting points for this class of robots are the handbook (Siciliano and Khatib 2016, §18), a brief section in Siciliano et al. (2009) and Merlet (2006).

Closed-form inverse kinematic solutions can be derived symbolically by writing down a number of kinematic relationships and solving for the joint angles, as described in Paul (1981). Software packages to automatically generate the forward and inverse kinematics for a given robot have been developed and these include Robotica (Nethery and Spong 1994) which is now obsolete, and SYMORO (Khalil and Creusot 1997) which is now available as open-source.

Historical. The original work by Denavit and Hartenberg was their 1955 paper (Denavit and Hartenberg 1955) and their textbook (Hartenberg and Denavit 1964). The book has an introduction to the field of kinematics and its history but is currently out of print, although a version can be found online. The first full description of the kinematics of a six-link arm with a spherical wrist was by Paul and Zhang (1986).

MATLAB and Toolbox Notes

The workhorse of the Toolbox is the `SerialLink` class which has considerable functionality and very many methods – we will use it extensively for the remainder of Part III. The classes `ETS2` and `ETS3` used in the early parts of this chapter were designed to illustrate principles as concisely as possible and have limited functionality, but the names of their methods are the same as their equivalents in the `SerialLink` class.

The `plot` method draws a *stick figure* robot and needs only Denavit-Hartenberg parameters. However the joints depicted are associated with the link frames and don't necessarily correspond to physical joints on the robot, but that is a limitation of the Denavit-Hartenberg parameters. A small number of robots have more realistic 3-dimensional models defined by STL files and these can be displayed using the `plot3d`. The models shipped with the Toolbox were created by Arturo Gil and are also shipped with his ARTE Toolbox.

The numerical inverse kinematics method `ikine` can handle over- and under-actuated robot arms, but does not handle joint coordinate limits which can be set in the `SerialLink` object. The alternative inverse kinematic method `ikcon` respects joint limits but requires the MATLAB Optimization Toolbox™.

The MATLAB Robotics System Toolbox™ provides a `RigidBodyTree` class to represent a serial-link manipulator, and this also supports branched mechanisms such as a humanoid robot. It also provides a general `InverseKinematics` class to solve inverse kinematic problems and can handle joint limits.

Exercises

1. Forward kinematics for planar robot from Sect. 7.1.1.
   a) For the 2-joint robot use the `teach` method to determine the two sets of joint angles that will position the end-effector at (0.5, 0.5).
   b) Experiment with the three different models in Fig. 7.2 using the `fkine` and `teach` methods.
   c) Vary the models: adjust the link lengths, create links with a translation in the *y*-direction, or create links with a translation in the *x*- and *y*-direction.
2. Experiment with the `teach` method for the Puma 560 robot.
3. Inverse kinematics for the 2-link robot on page 206.
   a) Compute forward and inverse kinematics with a_1 and a_2 as symbolic rather than numeric values.
   b) What happens to the solution when a point is out of reach?
   c) Most end-effector positions can be reached by two different sets of joint angles. What points can be reached by only one set?
4. Compare the solutions generated by `ikine6s` and `ikine` for the Puma 560 robot at different poses. Is there any difference in accuracy? How much slower is `ikine`?
5. For the Puma 560 at configuration `qn` demonstrate a configuration change from elbow up to elbow down.
6. For a Puma 560 robot investigate the errors in end-effector pose due to manufacturing errors.
   a) Make link 2 longer by 0.5 mm. For 100 random joint configurations what is the mean and maximum error in the components of end-effector pose?
   b) Introduce an error of 0.1 degrees in the joint 2 angle and repeat the analysis above.
7. Investigate the redundant robot models `mdl_hyper2d` and `mdl_hyper3d`. Manually control them using the `teach` method, compute forward kinematics and numerical inverse kinematics.

8. If you have the MATLAB Optimization Toolbox™ experiment with the `ikcon` method which solves inverse kinematics for the case where the joint coordinates have limits (modeling mechanical end stops). Joint limits are set with the `qlim` property of the `Link` class.
9. Drawing a 'B' (page 220)
   a) Change the size of the letter.
   b) Write a word or sentence.
   c) Write on a vertical plane.
   d) Write on an inclined plane.
   e) Change the robot from a Puma 560 to the Fanuc 10L.
   f) Write on a sphere. Hint: Write on a tangent plane, then project points onto the sphere's surface.
   g) This writing task does not require 6DOF since the rotation of the pen about its axis is not important. Remove the final link from the Puma 560 robot model and repeat the exercise.
10. Walking robot (page 221)
    a) Shorten the reset trajectory by reducing the leg lift during reset.
    b) Increase the stride of the legs.
    c) Figure out how to steer the robot by changing the stride length on one side of the body.
    d) Change the gait so the robot moves sideways like a crab.
    e) Add another pair of legs. Change the gait to reset two legs or three legs at a time.
    f) Currently in the simulation the legs move but the body does not move forward. Modify the simulation so the body moves.
    g) A robot hand comprises a number of fingers, each of which is a small serial-link manipulator. Create a model of a hand with 2, 3 or 5 fingers and animate the finger motion.
11. Create a simulation with two robot arms next to each other, whose end-effectors are holding a basketball at diametrically opposite points in the horizontal plane. Write code to move the robots so as to rotate the ball about the vertical axis.
12. Create STL files to represent your own robot and integrate them into the Toolbox. Check out the code in `SerialLink.plot3d`.