frame $\{B\}$ is *not* an inertial reference frame. In Newtonian mechanics, gravity is considered a real body force $m\boldsymbol{g}$– a free object will accelerate relative to the inertial frame.

Albert Einstein's equivalence principle is that "*we assume the complete physical equivalence of a gravitational field and a corresponding acceleration of the reference system*" – we are unable to distinguish between gravity and being on a rocket accelerating at 1 *g* far from the gravitational influence of any celestial object.

An everyday example of a noninertial reference frame is an accelerating car or airplane. Inside an accelerating vehicle we observe fictitious forces pushing objects around in a way that is not explained by Newton's law in an inertial reference frame. We also experience real forces acting on our body which, in this case, are provided by the seat and the restraint.

For a rotating reference frame things are more complex still. Imagine that you and a friend are standing on a large rotating turntable, and throwing a ball back and forth. You will observe that the ball follows a curved path in space. As a Newton-aware observer in this noninertial reference frame you would have to resort to invoking some magical force that explains why flying objects follow curved paths.

Of course if we look down onto the turntable from an inertial reference frame the ball is moving in a straight line.

If the reference frame $\{B\}$ is rotating with angular velocity $\boldsymbol{\omega}$ about its origin then Newton's second law Eq. 3.9 becomes

$$m\left({}^{B}\ddot{\boldsymbol{v}} + \underbrace{\boldsymbol{\omega}\times\left(\boldsymbol{\omega}\times{}^{B}\boldsymbol{p}\right)}_{\text{centripetal}} + \underbrace{2\boldsymbol{\omega}\times{}^{B}\boldsymbol{v}}_{\text{Coriolis}} + \underbrace{\frac{\mathrm{d}\boldsymbol{\omega}}{\mathrm{dt}}\times{}^{B}\boldsymbol{p}}_{\text{Euler}}\right) = {}^{0}\boldsymbol{f}$$

with three *new* acceleration terms. Centripetal acceleration always acts inward toward the origin. If the point is moving then Coriolis acceleration will be normal to its velocity. If rotational velocity is time varying then Euler acceleration will be normal to the position vector. Frequently the centripetal term is moved to the right-hand side in which case it becomes a fictitious outward centrifugal force. This complexity is symptomatic of being in a noninertial reference frame, and another definition of an inertial frame is one in which the "*physical laws hold good in their simplest form*".

Einstein, "*The foundation of the general theory of relativity*".

In robotics the term inertial frame and world coordinate frame tend to be used loosely and interchangeably to indicate a frame fixed to some point on the Earth. This is to distinguish it from the body-frame attached to the robot or vehicle. The surface of the Earth is an approximation of an inertial reference frame – the effect of the Earth's rotation is a finite acceleration less than 0.04 m s^{-2} due to centripetal acceleration. From the perspective of an Earth-bound observer a moving body will experience Coriolis acceleration. Both effects are small, dependent on latitude, and typically ignored.

Coriolis acceleration is significant for weather systems and meteorological prediction but below the sensitivity of low-cost sensors.

3.3 Creating Time-Varying Pose

In robotics we often need to generate a time-varying pose that moves smoothly in translation and rotation. A path is a spatial construct – a locus in space that leads from an initial pose to a final pose. A trajectory is a path with specified timing. For example there is a path from A to B, but there is a trajectory from A to B in 10 s or at 2 m s^{-1}.

An important characteristic of a trajectory is that it is *smooth* – position and orientation vary smoothly with time. We start by discussing how to generate smooth trajectories in one dimension. We then extend that to the multi-dimensional case and then to piecewise-linear trajectories that visit a number of intermediate points without stopping.

3.3.1 Smooth One-Dimensional Trajectories

We start our discussion with a scalar function of time. Important characteristics of this function are that its initial and final value are specified and that it is *smooth*. Smoothness in this context means that its first few temporal derivatives are continuous. Typically velocity and acceleration are required to be continuous and sometimes also the derivative of acceleration or jerk.

An obvious candidate for such a function is a polynomial function of time. Polynomials are simple to compute and can easily provide the required smoothness and boundary conditions. A quintic (fifth-order) polynomial is often used

$$s(t) = At^5 + Bt^4 + Ct^3 + Dt^2 + Et + F \tag{3.12}$$

where time $t \in [0, T]$. The first- and second-derivatives are also smooth polynomials

$$\dot{s}(t) = 5At^4 + 4Bt^3 + 3Ct^2 + 2D + E \tag{3.13}$$

$$\ddot{s}(t) = 20At^3 + 12Bt^2 + 6Ct + 2D \tag{3.14}$$

Time	s	$\dot{s}$	$\ddot{s}$
$t=0$	s_0	$\dot{s}_0$	$\ddot{s}_0$
$t=T$	s_T	$\dot{s}_T$	$\ddot{s}_T$

The trajectory has defined boundary conditions for position, velocity and acceleration and frequently the velocity and acceleration boundary conditions are all zero.

Writing Eq. 3.12 to Eq. 3.14 for the boundary conditions $t = 0$ and $t = T$ gives six equations which we can write in matrix form as

$$\begin{pmatrix} s_0 \\ s_T \\ \dot{s}_0 \\ \dot{s}_T \\ \ddot{s}_0 \\ \ddot{s}_T \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ T^5 & T^4 & T^3 & T^2 & T & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 5T^4 & 4T^3 & 3T^2 & 2T & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 20T^3 & 12T^2 & 6T & 2 & 0 & 0 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \\ D \\ E \\ F \end{pmatrix}$$

This is the reason for choice of quintic polynomial. It has six coefficients that enable it to meet the six boundary conditions on initial and final position, velocity and acceleration.

Since the matrix is square we can solve for the coefficient vector (A, B, C, D, E, F) using standard linear algebra methods such as the MATLAB \-operator. For a quintic polynomial acceleration will be a smooth cubic polynomial, and jerk will be a parabola.

The Toolbox function `tpoly` generates a quintic polynomial trajectory as described by Eq. 3.12. For example

```
>> tpoly(0, 1, 50);
```

generates a polynomial trajectory and plots it, along with the corresponding velocity and acceleration, as shown in Fig. 3.2a. We can get these values into the workspace by providing output arguments

```
>> [s,sd,sdd] = tpoly(0, 1, 50);
```

where `s`, `sd` and `sdd` are respectively the trajectory, velocity and acceleration – each a 50×1 column vector. We observe that the initial and final velocity and acceleration

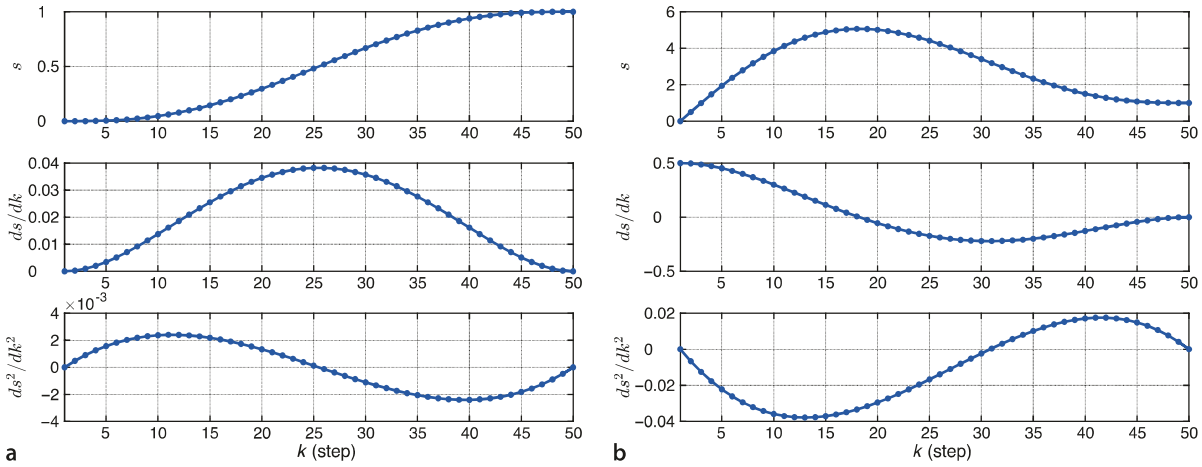

Fig. 3.2. Quintic polynomial trajectory. From top to bottom is position, velocity and acceleration versus time step. **a** With zero-velocity boundary conditions, **b** initial velocity of 0.5 and a final velocity of 0. Note that velocity and acceleration are in units of timestep not seconds

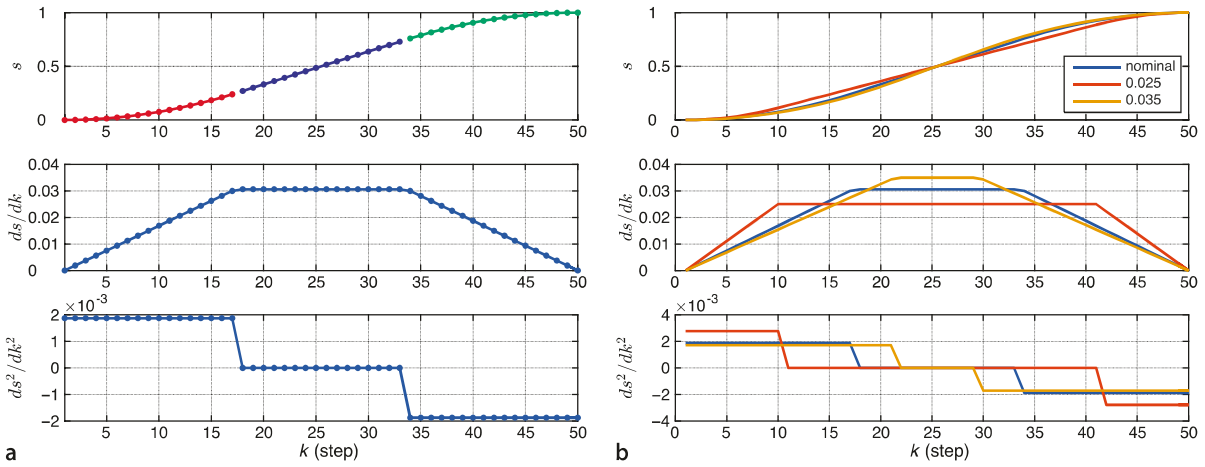

Fig. 3.3. Linear segment with parabolic blend (LSPB) trajectory: **a** default velocity for linear segment; **b** specified linear segment velocity values

are all zero – the default value. The initial and final velocities can be set to nonzero values

```
>> tpoly(0, 1, 50, 0.5, 0);
```

in this case, an initial velocity of 0.5 and a final velocity of 0. The results shown in Fig. 3.2b illustrate an important problem with polynomials. The nonzero initial velocity causes the polynomial to overshoot the terminal value – it peaks at 5 on a trajectory from 0 to 1.

Another problem with polynomials, a very practical one, can be seen in the middle graph of Fig. 3.2a. The velocity peaks at $k = 25$ which means that for most of the time the velocity is far less than the maximum. The mean velocity

```
>> mean(sd) / max(sd)
ans =
    0.5231
```

is only 52% of the peak so we are not using the motor as fully as we could. A real robot joint has a well defined maximum velocity and for minimum-time motion we want to be operating at that maximum for as much of the time as possible. We would like the velocity curve to be *flatter* on top.

A well known alternative is a hybrid trajectory which has a constant velocity segment with polynomial segments for acceleration and deceleration. Revisiting our first example the hybrid trajectory is

```
>> lspb(0, 1, 50);
```

where the arguments have the same meaning as for `tpoly` and the trajectory is shown in Fig. 3.3a. The trajectory comprises a linear segment (constant velocity) with parabolic blends, hence the name `lspb`. The term blend is commonly used to refer to a trajectory segment that smoothly joins linear segments. As with `tpoly` we can also return the trajectory and its velocity and acceleration

```
>> [s,sd,sdd] = lspb(0, 1, 50);
```

This type of trajectory is also referred to as trapezoidal due to the shape of the velocity curve versus time, and is commonly used in industrial motor drives.

The trapezoidal trajectory is smooth in velocity, but not in acceleration.

The function `lspb` has *chosen* the velocity of the linear segment to be

```
>> max(sd)
ans =
    0.0306
```

but this can be overridden by specifying it as a fourth input argument

```
>> s = lspb(0, 1, 50, 0.025);
>> s = lspb(0, 1, 50, 0.035);
```

The trajectories for these different cases are overlaid in Fig. 3.3b. We see that as the velocity of the linear segment increases its duration decreases and ultimately its duration would be zero. In fact the velocity cannot be chosen arbitrarily◀, too high or too low a value for the maximum velocity will result in an infeasible trajectory and the function returns an error.

The system has one design degree of freedom. There are six degrees of freedom (blend time, three parabolic coefficients and two linear coefficients) and five constraints (total time, initial and final position and velocity).

3.3.2 Multi-Dimensional Trajectories

Most useful robots have more than one axis of motion and it is quite straightforward to extend the smooth scalar trajectory to the vector case. In terms of configuration space (Sect. 2.3.5), these axes of motion correspond to the dimensions of the robot's configuration space – to its degrees of freedom. We represent the robot's configuration as a vector $\boldsymbol{q} \in \mathbb{R}^N$ where N is the number of degrees of freedom. The configuration of a 3-joint robot would be its joint angles $\boldsymbol{q} = (q_1, q_2, q_3)$. The configuration vector of wheeled mobile robot might be its position $\boldsymbol{q} = (x, y)$ or its position and heading angle $\boldsymbol{q} = (x, y, \theta)$. For a 3-dimensional body that had an orientation in $\mathbf{SO}(3)$ we would use a configuration vector $\boldsymbol{q} = (\theta_r, \theta_p, \theta_y)$ or for a pose in $\mathbf{SE}(3)$ we would use $\boldsymbol{q} = (x, y, z, \theta_r, \theta_p, \theta_y)$◀. In all these cases we would require smooth multi-dimensional motion from an initial configuration vector to a final configuration vector.

Or an equivalent 3-angle representation.

In the Toolbox this is achieved using the function `mtraj` and to move from configuration $(0, 2)$ to $(1, -1)$ in 50 steps we write

```
>> q = mtraj(@lspb, [0 2], [1 -1], 50);
```

which results in a 50×2 matrix `q` with one row per time step and one column per axis. The first argument is a handle to a function that generates a *scalar* trajectory, `@lspb` as in this case or `@tpoly`. The trajectory for the `@lspb` case

```
>> plot(q)
```

is shown in Fig. 3.4.

If we wished to create a trajectory for 3-dimensional pose we might consider converting a pose $\boldsymbol{T}$ to a 6-vector by a command like

```
q = [T1.t' T1.torpy]
```

though as we shall see later interpolation of 3-angle representations has some limitations.

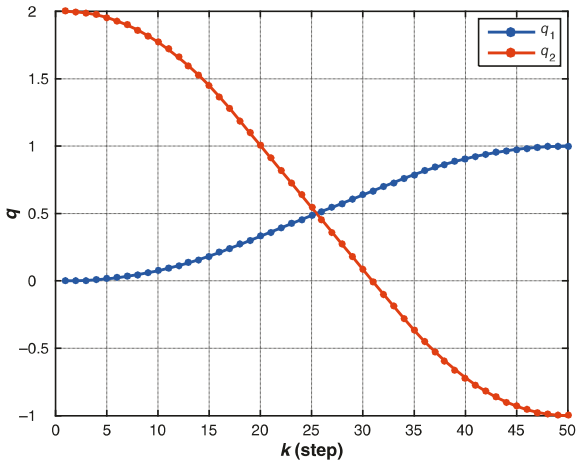

Fig. 3.4. Multi-dimensional motion. q_1 varies from $0 \rightarrow 1$ and q_2 varies from $2 \rightarrow -1$

3.3.3 Multi-Segment Trajectories

In robotics applications there is often a need to move smoothly along a path through one or more intermediate or *via* points without stopping. This might be to avoid obstacles in the workplace, or to perform a task that involves following a piecewise continuous trajectory such as welding a seam or applying a bead of sealant in a manufacturing application.

To formalize the problem consider that the trajectory is defined by M configurations $\boldsymbol{q}_k$, $k \in [1, M]$ and there are $M-1$ motion segments. As in the previous section $\boldsymbol{q}_k \in \mathbb{R}^N$ is a *vector* representation of configuration.

The robot starts from $\boldsymbol{q}_1$ at rest and finishes at $\boldsymbol{q}_M$ at rest, but moves through (or close to) the intermediate configurations without stopping. The problem is over constrained and in order to attain continuous velocity we surrender the ability to reach each intermediate configuration. This is easiest to understand for the 1-dimensional case shown in Fig. 3.5. The motion comprises linear motion segments with polynomial blends, like `lspb`, but here we choose quintic polynomials because they are able to match boundary conditions on position, velocity and acceleration at their start and end points.

The first segment of the trajectory accelerates from the initial configuration $\boldsymbol{q}_1$ and zero velocity, and joins the line heading toward the second configuration $\boldsymbol{q}_2$. The blend time is set to be a constant t_{acc} and $t_{\text{acc}}/2$ before reaching $\boldsymbol{q}_2$ the trajectory executes a polynomial blend, of duration t_{acc}, onto the line from $\boldsymbol{q}_2$ to $\boldsymbol{q}_3$, and the process repeats. The constant velocity $\dot{\boldsymbol{q}}_k$ can be specified for each segment. The average acceleration during the blend is

$$\ddot{\boldsymbol{q}} = \frac{\dot{\boldsymbol{q}}_{k+1} - \dot{\boldsymbol{q}}_k}{t_{\text{acc}}}$$

If the maximum acceleration capability of the axis is known then the minimum blend time can be computed.

The real limit of the axis will be its peak, rather than average, acceleration. The peak acceleration for the blend can be determined from Eq. 3.14 once the quintic coefficients are known.

On a particular motion segment each axis will have a different distance to travel and traveling at its maximum speed there will be a minimum time before it can reach its goal. The first step in planning a segment is to determine which axis will be the slowest to complete the segment, based on the distance that each axis needs to travel for the segment and its maximum achievable velocity. From this the duration of the segment can be computed and then the required velocity of each axis. This ensures that all axes reach the next target $\boldsymbol{q}_k$ at the *same time*.

The Toolbox function `mstraj` generates a multi-segment multi-axis trajectory based on a matrix of via points. For example 2-axis motion via the corners of a rotated square can be generated by

```
>> via = SO2(30, 'deg') * [-1 1; 1 1; 1 -1; -1 -1]';
>> q0 = mstraj(via(:,[2 3 4 1])', [2,1], [], via(:,1)', 0.2, 0);
```

The first argument is the matrix of via points, each row is the coordinates of a point. The remaining arguments are respectively: a vector of maximum speeds per axis, a vector of

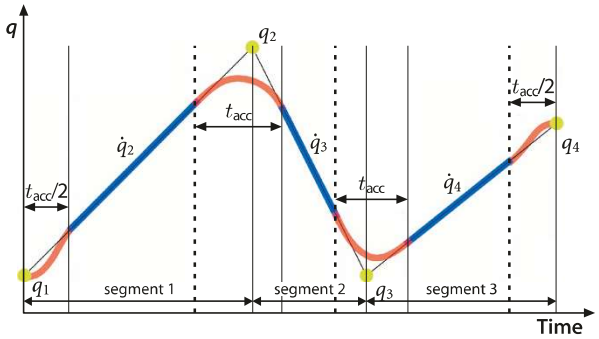

Fig. 3.5.
Notation for multi-segment trajectory showing four points and three motion segments. Blue indicates constant velocity motion, red indicates regions of acceleration

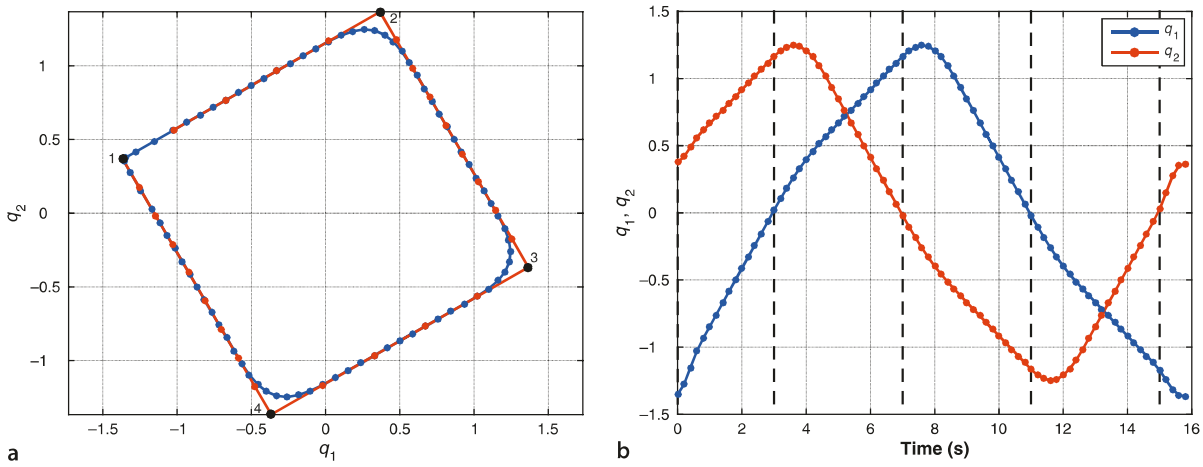

Fig. 3.6. Multi-segment multi-axis trajectories: **a** configuration of robot (tool position) for acceleration time of $t_{acc} = 0$ s (red) and $t_{acc} = 2$ s (blue), the via points are indicated by solid black markers; **b** configuration versus time with segment transitions ($t_{acc} = 2$ s) indicated by dashed black lines. The discrete-time points are indicated by dots

durations for each segment, the initial configuration, the sample time step, and the acceleration time. The function `mstraj` returns a matrix with one row per time step and the columns correspond to the axes. We can plot q_2 against q_1 to see the path of the robot

```
>> plot(q0(:,1), q0(:,2))
```

and is shown by the red path in Fig. 3.6a. If we increase the acceleration time

```
>> q2 = mstraj(via(:,[2 3 4 1])', [2,1], [], via(:,1)', 0.2, 2);
```

the trajectory becomes more rounded (blue path) as the polynomial blending functions do their work. The smoother trajectory also takes more time to complete.

```
>> [numrows(q0) numrows(q2)]
ans =
   28   80
```

Only one of the maximum axis speed or time per segment can be specified, the other is set to MATLAB's empty matrix `[]`.

Acceleration time if given is rounded up internally to a multiple of the time step.

The configuration variables as a function of time are shown in Fig. 3.6b. This function also accepts optional initial and final velocity arguments and t_{acc} can be a vector specifying different acceleration times for each of the N blends.

Keep in mind that this function simply interpolates pose represented as a vector. In this example the vector was assumed to be Cartesian coordinates, but this function could also be applied to Euler or roll-pitch-yaw angles but this is not an ideal way to interpolate rotation. This leads us nicely to the next section where we discuss interpolation of orientation.

3.3.4 Interpolation of Orientation in 3D

In robotics we often need to interpolate orientation, for example, we require the end-effector of a robot to smoothly change from orientation $\boldsymbol{\xi}_0$ to $\boldsymbol{\xi}_1$ in $\mathbf{SO}(3)$. We require some function $\boldsymbol{\xi}(s) = \sigma(\boldsymbol{\xi}_0, \boldsymbol{\xi}_1, s)$ where $s \in [0, 1]$ which has the boundary conditions $\sigma(\boldsymbol{\xi}_0, \boldsymbol{\xi}_1, 0) = \boldsymbol{\xi}_0$ and $\sigma(\boldsymbol{\xi}_0, \boldsymbol{\xi}_1, 1) = \boldsymbol{\xi}_1$ and where $\sigma(\boldsymbol{\xi}_0, \boldsymbol{\xi}_1, s)$ varies *smoothly* for intermediate values of s. How we implement this depends very much on our concrete representation of $\boldsymbol{\xi}$.

If pose is represented by an orthonormal rotation matrix, $\boldsymbol{\xi} \sim \boldsymbol{R} \in \mathbf{SO}(3)$, we might consider a simple linear interpolation $\sigma(\boldsymbol{R}_0, \boldsymbol{R}_1, s) = (1 - s)\boldsymbol{R}_0 + s\boldsymbol{R}_1$ but this would not, in general, be a valid orthonormal matrix which has strict column norm and inter-column orthogonality constraints.

A workable and commonly used approach is to consider a 3-angle representation such as Euler or roll-pitch-yaw angles, $\boldsymbol{\xi} \sim \boldsymbol{\Gamma} \in \mathbb{S}^1 \times \mathbb{S}^1 \times \mathbb{S}^1$ and use linear interpolation

$$\sigma(\boldsymbol{\Gamma}_0, \boldsymbol{\Gamma}_1, s) = (1 - s)\boldsymbol{\Gamma}_0 + s\boldsymbol{\Gamma}_1$$

and converting the interpolated angles back to a rotation matrix always results in a valid form. For example we define two orientations

```
>> R0 = SO3.Rz(-1) * SO3.Ry(-1);
>> R1 = SO3.Rz(1) * SO3.Ry(1);
```

and find the equivalent roll-pitch-yaw angles

```
>> rpy0 = R0.torpy();  rpy1 = R1.torpy();
```

and create a trajectory between them over 50 time steps

```
>> rpy = mtraj(@tpoly, rpy0, rpy1, 50);
```

which is most easily visualized as an animation▸

```
>> SO3.rpy( rpy ).animate;
```

`rpy` is a 50×3 matrix and the result of `SO3.rpy` is a 1×50 vector of `SO3` objects, and their `animate` method is then called.

For large orientation changes we see that the axis around which the coordinate frame rotates changes along the trajectory. The motion, while smooth, sometimes looks uncoordinated. There will also be problems if either ξ_0 or ξ_1 is close to a singularity in the particular 3-angle system being used. This particular trajectory passes very close to the singularity, at around steps 24 and 25, and a symptom of this is the very rapid rate of change of roll-pitch-yaw angles at this point. The frame is not rotating faster at this point – you can verify that in the animation – the rotational parameters are changing very quickly and this is consequence of the particular representation.

Interpolation of unit-quaternions is only a little more complex than for 3-angle vectors and produces a change in orientation that is a rotation around a *fixed* axis in space. Using the Toolbox we first find the two equivalent quaternions

```
>> q0 = R0.UnitQuaternion;  q1 = R1.UnitQuaternion;
```

and then interpolate them

```
>> q = interp(q0, q1, 50);
>> about(q)
q [UnitQuaternion] : 1x50 (1.7 kB)
```

which results in a vector of 50 `UnitQuaternion` objects which we can animate by

```
>> q.animate
```

Quaternion interpolation is achieved using spherical linear interpolation (*slerp*) in which the unit quaternions follow a great circle path on a 4-dimensional hypersphere. The result in 3-dimensions is rotation about a fixed axis in space.

3.3.4.1 Direction of Rotation

When traveling on a circle we can move clockwise or counter-clockwise to reach the goal – the result is the same but the distance traveled may be different. On a sphere or hypersphere the principle is the same but now we are traveling on a great circle▸. In this example we animate a rotation about the z-axis, from an angle of -2 radians to $+2$ radians

```
>> q0 = UnitQuaternion.Rz(-2);  q1 = UnitQuaternion.Rz(2);
>> q = interp(q0, q1, 50);
>> q.animate()
```

A great circle on a sphere is the intersection of the sphere and a plane that passes through the center. On Earth the equator and all lines of longitude are great circles. Ships and aircraft prefer to follow great circles because they represent the shortest path between two points on the surface of a sphere.

but this is taking the long way around the circle, moving 4 radians when we could travel $2\pi - 4 \approx 2.28$ radians in the opposite direction. The `'shortest'` option requests the rotational interpolation to select the shortest path

```
>> q = interp(q0, q1, 50, 'shortest');
>> q.animate()
```

and the animation clearly shows the difference.

3.3.5 Cartesian Motion in 3D

Another common requirement is a smooth path between two poses in SE(3) which involves change in position as well as in orientation. In robotics this is often referred to as Cartesian motion.

We represent the initial and final poses as homogeneous transformations

```
>> T0 = SE3([0.4, 0.2, 0]) * SE3.rpy(0, 0, 3);
>> T1 = SE3([-0.4, -0.2, 0.3]) * SE3.rpy(-pi/4, pi/4, -pi/2);
```

The `SE3` object has a method `interp` that interpolates between two poses for normalized distance $s \in [0, 1]$ along the path, for example the midway pose between `T0` and `T1` is

```
>> interp(T0, T1, 0.5)
ans =
    0.0975   -0.7020    0.7055         0
    0.7020    0.5510    0.4512         0
   -0.7055    0.4512    0.5465    0.1500
         0         0         0         0
```

where the translational component is linearly interpolated and the rotation is spherically interpolated using the unit-quaternion interpolation method `interp`.

A trajectory between the two poses in 50 steps is created by

```
>> Ts = interp(T0, T1, 50);
```

This could also be written as `T0.interp(T1, 50)`.

where the arguments are the initial and final pose and the trajectory length.◀ The resulting trajectory `Ts` is a vector of `SE3` objects

```
>> about(Ts)
Ts [SE3] : 1x50 (6.5 kB)
```

representing the pose at each time step. The homogeneous transformation for the first point on the path is

```
>> Ts(1)
ans =
   -0.9900   -0.1411         0    0.4000
    0.1411   -0.9900         0    0.2000
         0         0    1.0000         0
         0         0         0    1.0000
```

and once again the easiest way to visualize this is by animation

```
>> Ts.animate
```

which shows the coordinate frame moving and rotating from pose `T0` to pose `T1`.

The translational part of this trajectory is obtained by◀

```
>> P = Ts.transl;
```

The `.t` property applied to a vector of `SE3` objects returns a MATLAB comma-separated list of translation vectors. The `.transl` method returns the translations in a more useful matrix form.

which returns the Cartesian position for the trajectory in matrix form

```
>> about(P)
P [double] : 50x3 (1.2 kB)
```

which has one row per time step that is the corresponding position vector. This is plotted

```
>> plot(P);
```

in Fig. 3.7 along with the orientation in roll-pitch-yaw format

```
>> rpy = Ts.torpy;
>> plot(rpy);
```

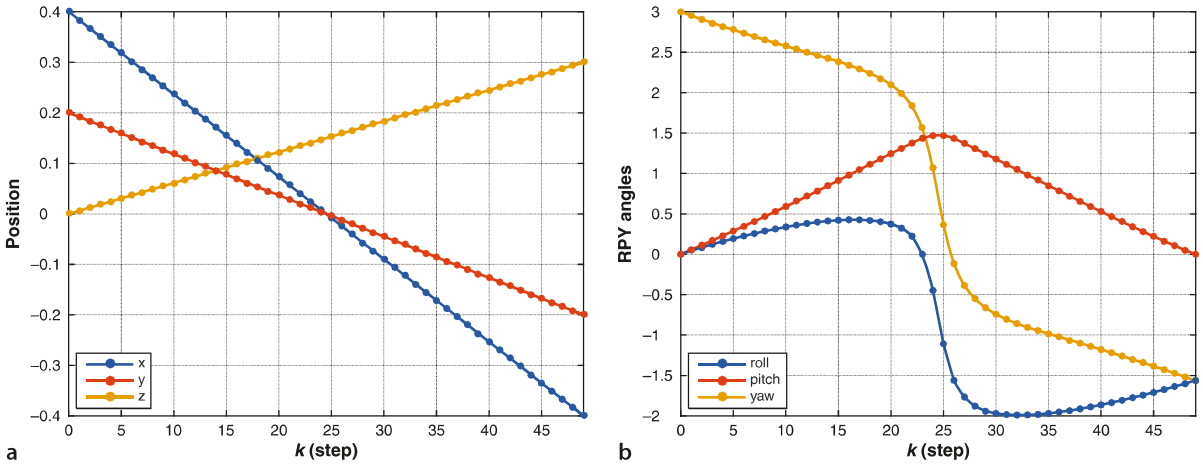

Fig. 3.7. Cartesian motion. **a** Cartesian position versus time, **b** roll-pitch-yaw angles versus time

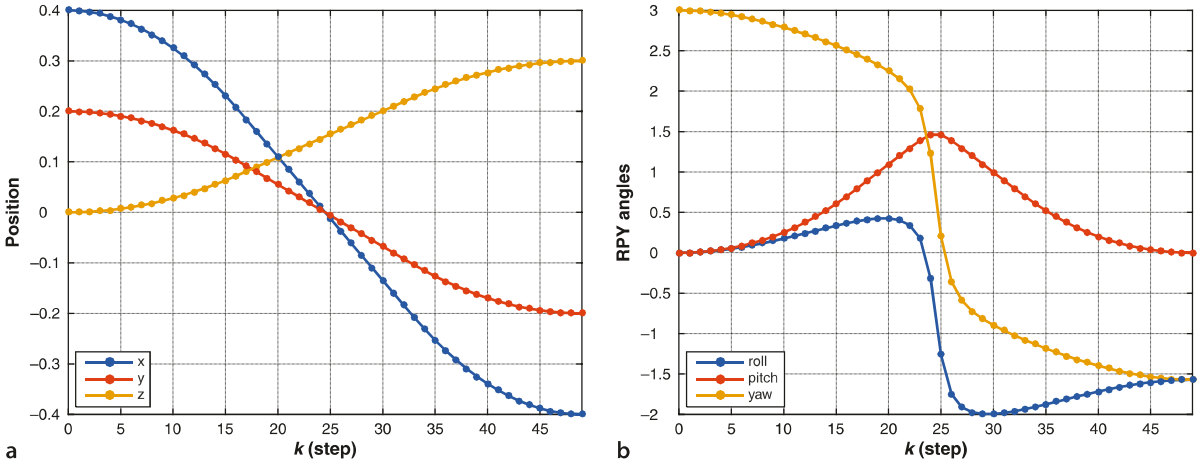

Fig. 3.8. Cartesian motion with LSPB path distance profile. **a** Cartesian position versus time, **b** roll-pitch-yaw angles versus time

We see that the position coordinates vary smoothly and linearly with time and that orientation varies smoothly with time.

However the motion has a velocity and acceleration *discontinuity* at the first and last points. While the path is smooth in space the distance s along the path is not smooth in time. Speed along the path jumps from zero to some finite value and then drops to zero at the end – there is no initial acceleration or final deceleration. The scalar functions `tpoly` and `lspb` discussed earlier can be used to generate s so that motion *along* the path is smooth. We can pass a vector of normalized distances along the path as the second argument to `interp`

```
>> Ts = T0.interp(T1, lspb(0, 1, 50) );
```

The trajectory is unchanged but the coordinate frame now accelerates to a constant speed along the path and then decelerates and this is reflected in smoother curves for the trajectory shown in Fig. 3.8. The Toolbox provides a convenient shorthand `ctraj` for the above

```
>> Ts = ctraj(T0, T1, 50);
```

where the arguments are the initial and final pose and the number of time steps.

The roll-pitch-yaw angles do not vary linearly with time because they represent a nonlinear transformation of the linearly varying quaternion.

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

Consider the end-effector moving between two Cartesian poses◂

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.

```
>> 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)
```

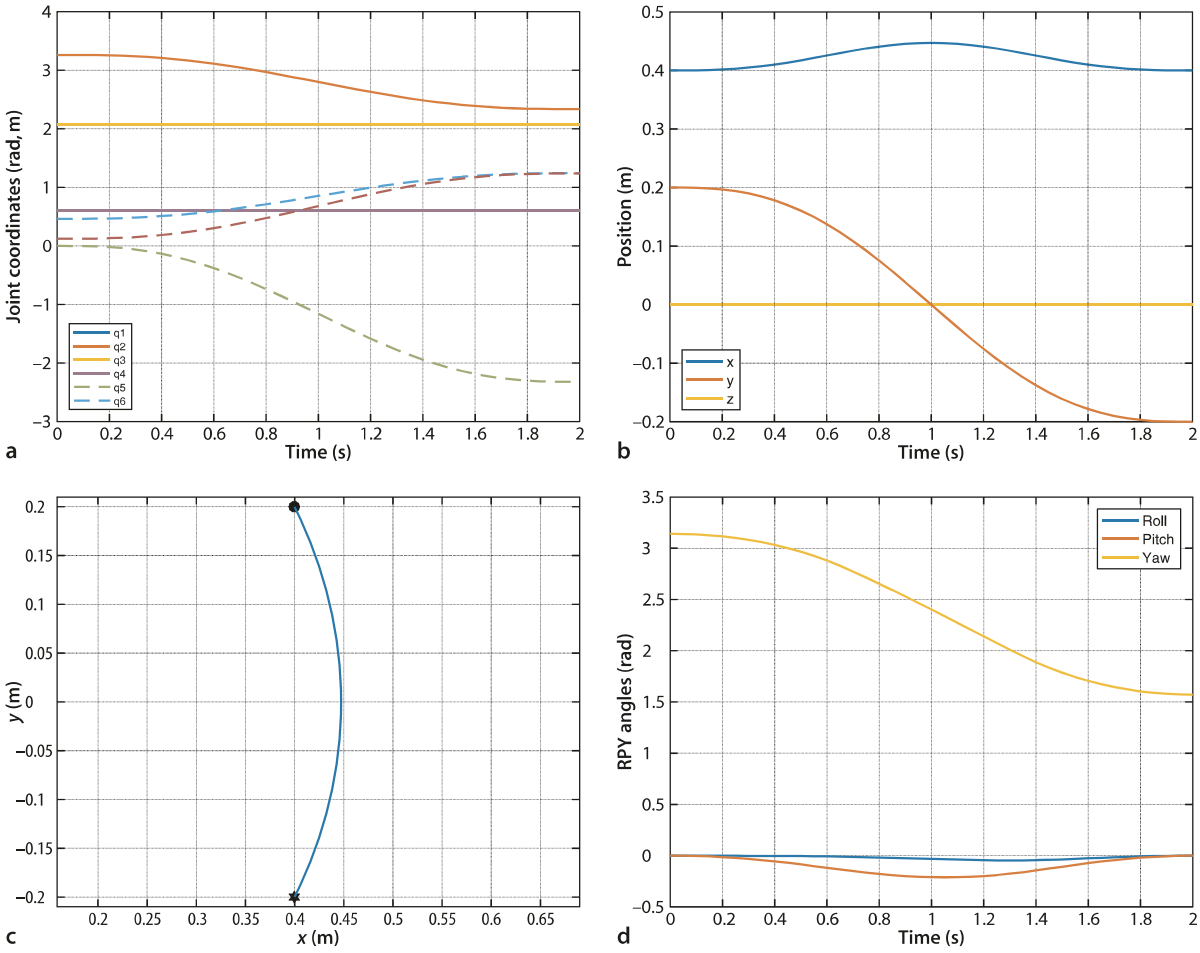

Fig. 7.9. Joint-space motion. **a** Joint coordinates versus time; **b** Cartesian position versus time; **c** Cartesian position locus in the *xy*-plane **d** roll-pitch-yaw angles versus time

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

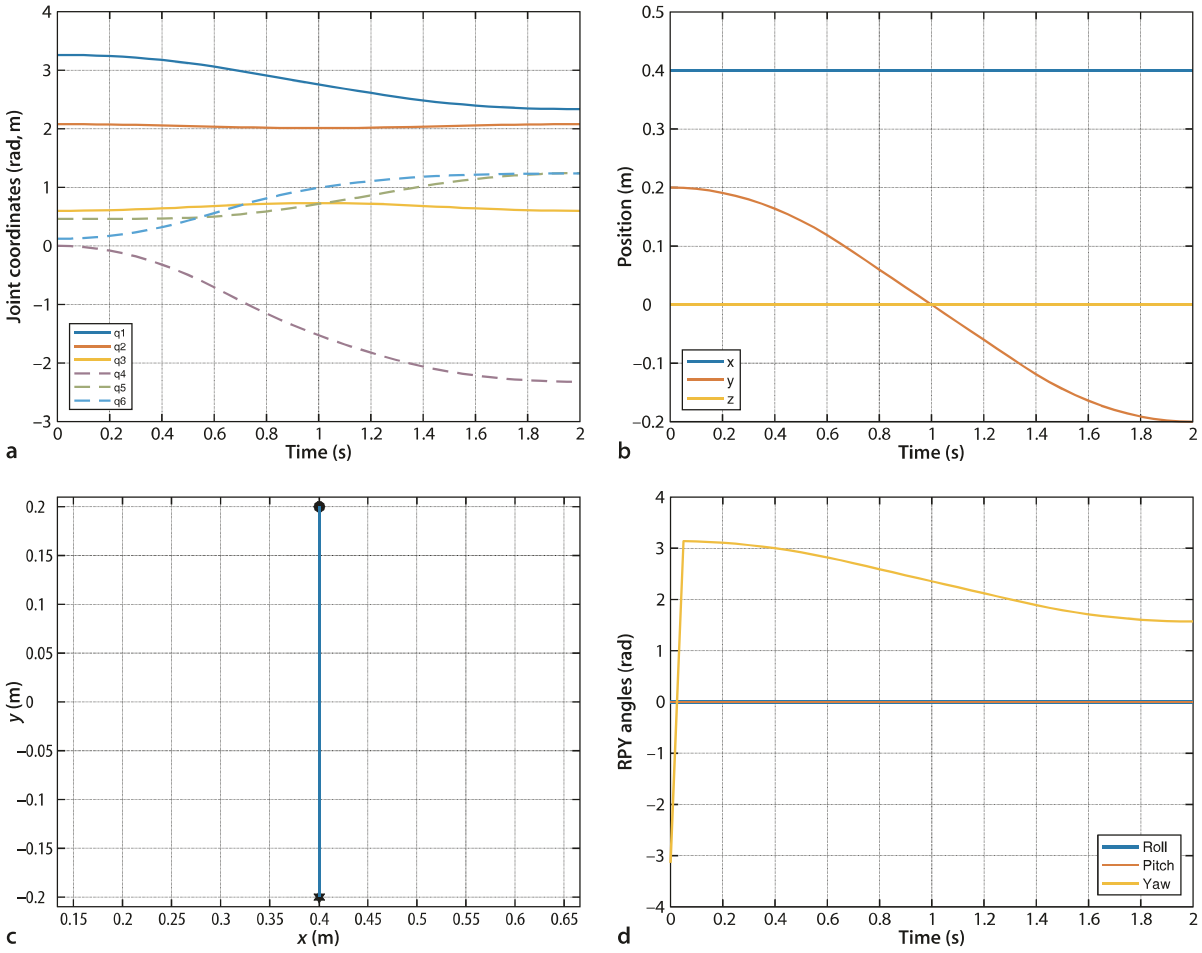

Fig. 7.10. Cartesian motion. **a** Joint coordinates versus time; **b** Cartesian position versus time; **c** Cartesian position locus in the *xy*-plane; **d** roll-pitch-yaw angles versus time

```
>> 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`.

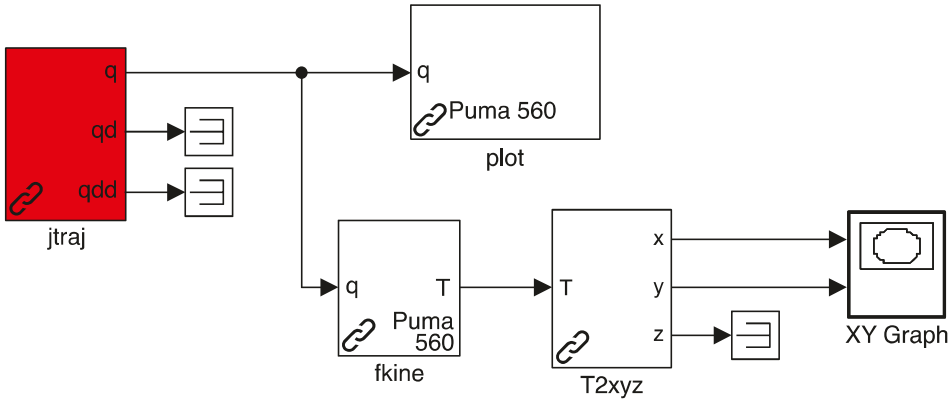

Fig. 7.11. Simulink model `sl_jspace` for joint-space motion

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 endpoints 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)
```

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.

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.

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.

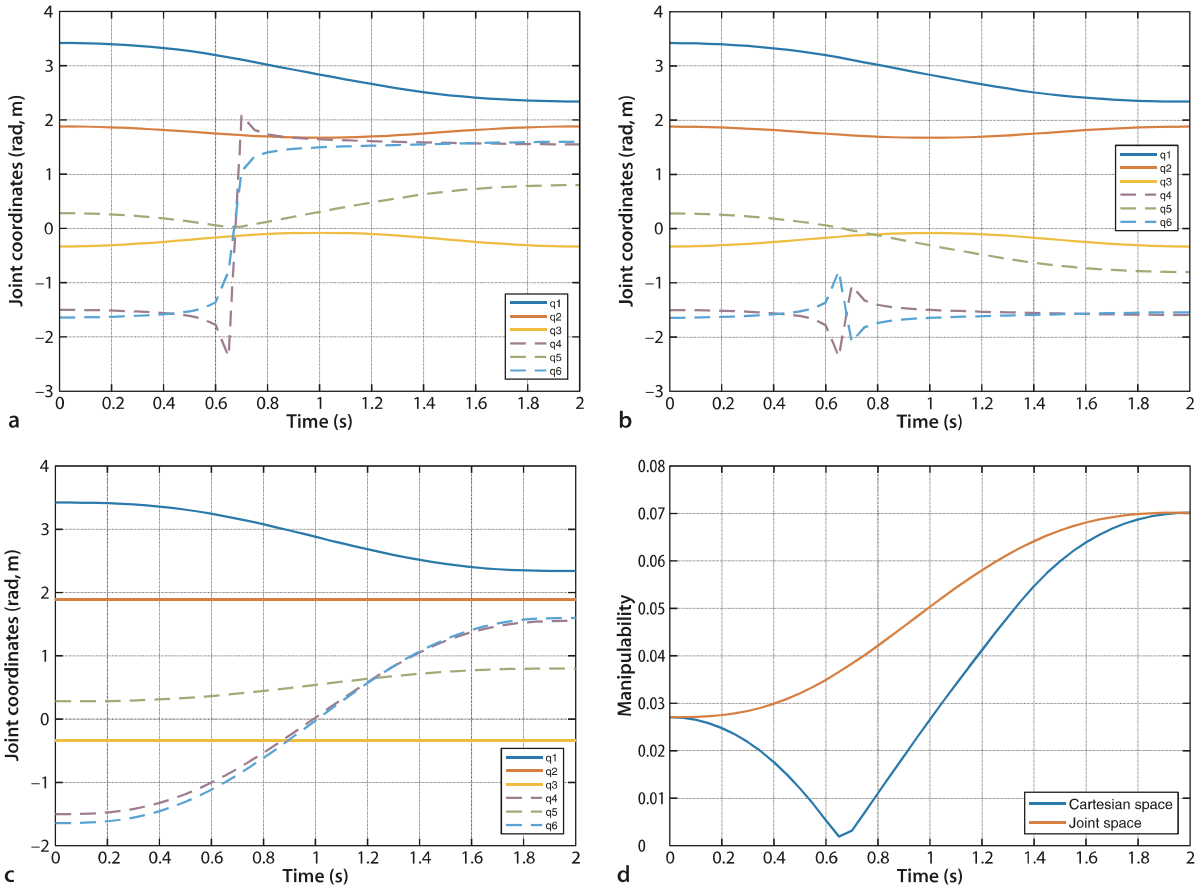

Fig. 7.12. Cartesian motion through a wrist singularity. **a** Joint coordinates computed by inverse kinematics (`ikine6s`); **b** joint coordinates computed by numerical inverse kinematics (`ikine`); **c** joint coordinates for joint-space motion; **d** manipulability

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)
```

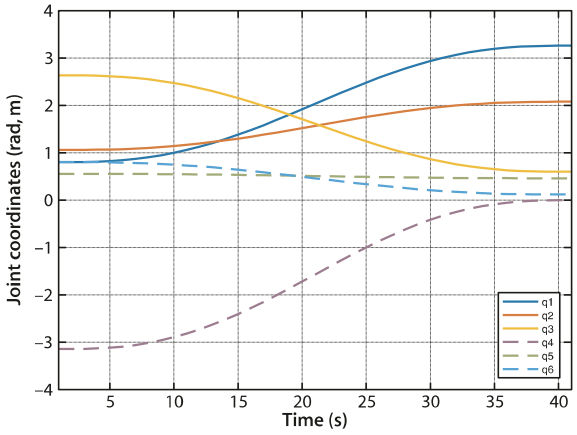

Fig. 7.13. Joint space motions for configuration change from right-handed to left-handed

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.

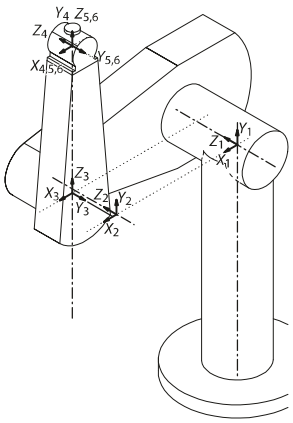

Fig. 7.14. Puma 560 robot coordinate frames. Standard Denavit-Hartenberg link coordinate frames for Puma in the zeroangle pose (Corke 1996b)

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