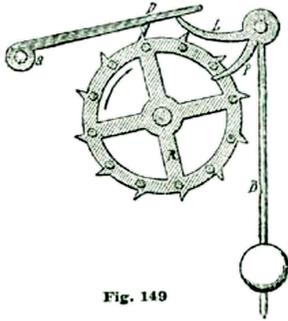# 3

Time and Motion

Fig. 149

In the previous chapter we learnt how to describe the pose of objects in 2- or 3-dimensional space. This chapter extends those concepts to objects whose pose is varying as a function of time.

For robots we wish to create a time varying pose that the robot can follow, for example the pose of a robot's end-effector should follow a path to the object that it is to grasp. Section 3.1 discusses how to generate a temporal sequence of poses, a trajectory, that smoothly changes from an initial pose to a final pose.

Section 3.2 discusses the concept of rate of change of pose, its temporal derivative, and how that relates to concepts from mechanics such as velocity and angular velocity. This allows us to solve the inverse problem – given measurements from velocity and angular velocity sensors how do we update the estimate of pose for a moving object. This is the principle underlying inertial navigation.

3.1 Trajectories

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 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 piece-wise-linear trajectories that visit a number of intermediate points without stopping.

3.1.1 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

$$S\langle t \rangle = At^5 + Bt^4 + Ct^3 + Dt^2 + Et + F \tag{3.1}$$

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

$$\dot{S}\langle t \rangle = 5At^4 + 4Bt^3 + 3Ct^2 + 2Dt + E \tag{3.2}$$

$$\ddot{S}\langle t \rangle = 20At^3 + 12Bt^2 + 6Ct + 2D \tag{3.3}$$

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

Writing Eq. 3.1 to Eq. 3.3 for the boundary conditions $t = 0$ and $t = T$ gives six equations which we can write in matrix form as

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

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

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.

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.
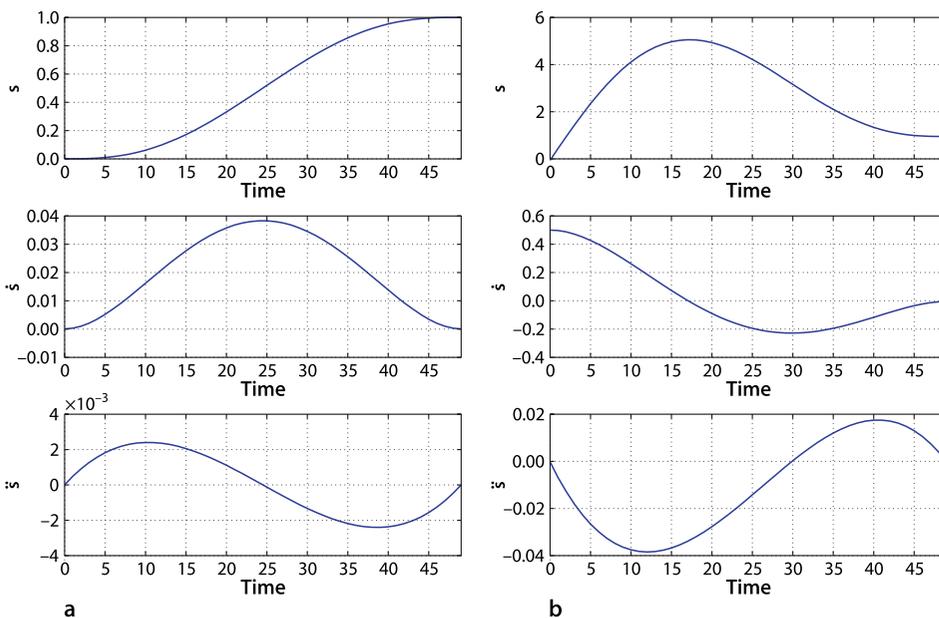
The Toolbox function `tpoly` generates a quintic polynomial trajectory as described by Eq. 3.1. For example

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

returns a 50 × 1 column vector with values varying smoothly from 0 to 1 in 50 time steps which we can plot

```
>> plot(s)
```

The corresponding velocity and acceleration can be returned via optional output arguments

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

as `sd` and `sdd` respectively. These are shown in Fig. 3.1a and we observe that the initial and final velocity and acceleration are all zero – the default value. The initial and final velocities can be set to non-zero values

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


Fig. 3.1.
Quintic polynomial trajectory. From top to bottom is position, velocity and acceleration versus time. **a** With zero-velocity boundary conditions, **b** initial velocity of 0.5 and a final velocity of 0
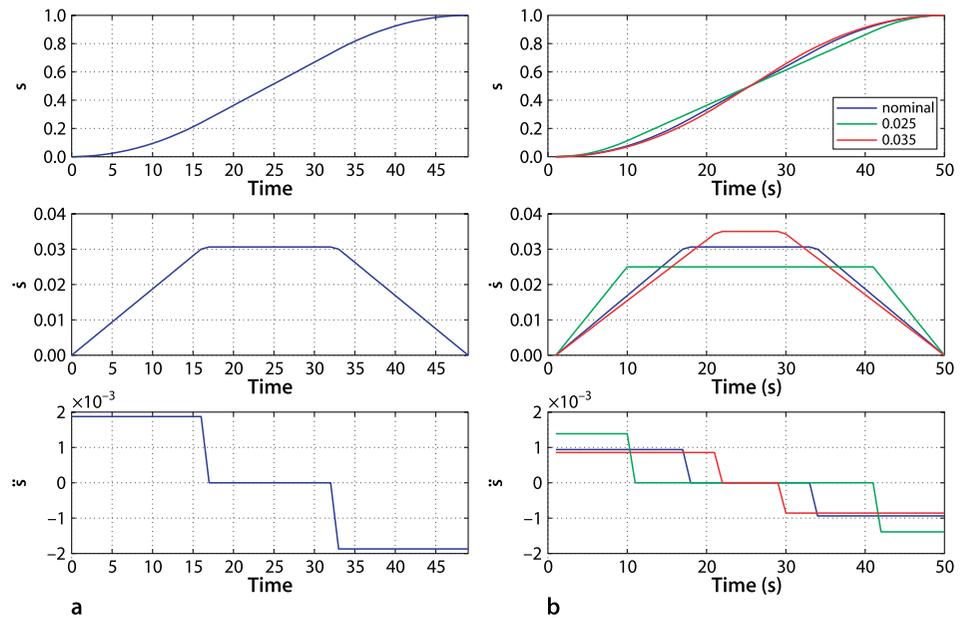
Fig. 3.2.
Linear segment with parabolic blend (LSPB) trajectory: **a** default velocity for linear segment; **b** specified linear segment velocity values

in this case, an initial velocity of 0.5 and a final velocity of 0. The results shown in Fig. 3.1b illustrate a problem with polynomials. The non-zero 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. 3.1a. The velocity peaks at $t = 25$ 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. 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

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

where the arguments have the same meaning as for `tpoly` and the trajectory is shown in Fig. 3.2a. 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 the velocity and acceleration

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

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

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 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. 3.2b. 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 is over-constrained, having five constraints (total time, initial and final position and velocity) but six degrees of freedom (blend time, three parabolic coefficients and two linear coefficients).

3.1.2   Multi-Dimensional Case

Most useful robots have more than one axis of motion or degree of freedom. We represent this in vector form as $x \in \mathbb{R}^M$ where M is the number of degrees of freedom. A wheeled mobile robot is characterised by its position (x, y) or pose (x, y, θ). The tool of an arm robot has position (x, y, z), orientation $(\theta_r, \theta_p, \theta_y)$ or pose $(x, y, z, \theta_r, \theta_p, \theta_y)$. We therefore require smooth multi-dimensional motion from an initial vector to a final vector.

It is quite straightforward to extend the smooth scalar trajectory to the vector case and in the Toolbox this is achieved using the function `mtraj`. For example to move from $(0, 2)$ to $(1, -1)$ in 50 steps

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

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

```
>> plot(x)
```

is shown in Fig. 3.3.

For a 3-dimensional problem we might consider converting a pose *T* to a 6-vector by

```
>> x = [transl(T); tr2rpy(T)']
```

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

3.1.3   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 applying a bead of sealant in a manufacturing application.
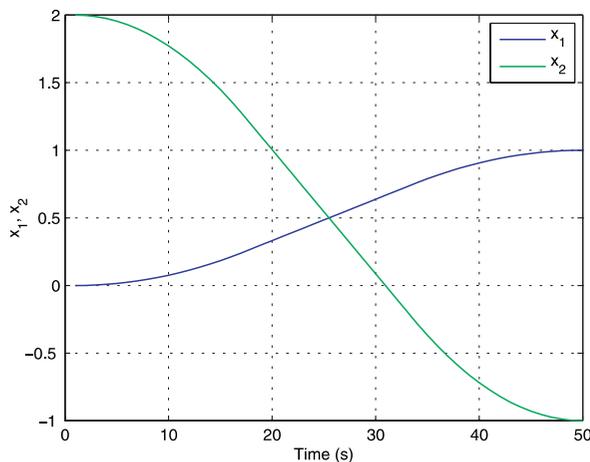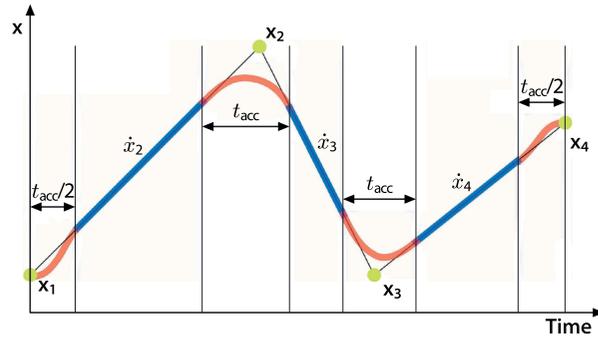

Fig. 3.3.
Multi-dimensional motion. x_1 varies from $0 \rightarrow 1$ and x_2 varies from $2 \rightarrow -1$

Fig. 3.4.
Notation for multi-segment trajectory showing four points and three motion segments. Blue indicates constant velocity motion, red indicates regions of acceleration

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

The robot starts from x_1 at rest and finishes at x_N at rest, but moves through (or close to) the intermediate points without stopping. The problem is over constrained and in order to attain continuous velocity we surrender the ability to reach each intermediate point. This is easiest to understand for the one dimensional case shown in Fig. 3.4. 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 pose x_1 and zero velocity, and joins the line heading toward the second pose x_2. The blend time is set to be a constant t_{acc} and $t_{acc} / 2$ before reaching x_2 the trajectory executes a polynomial blend, of duration t_{acc}, onto the line from x_2 to x_3, and the process repeats. The constant velocity $\dot{x}_k$ can be specified for each segment. The average acceleration during the blend is

$$\ddot{x}_{av} = \frac{\dot{x}_{k+1} - \dot{x}_k}{t_{acc}}$$

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. 3.3 if the quintic coefficients are known.

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

For the multi-axis case it is likely that some axes will need to move further than others on a particular motion segment and this becomes complex if the joints have different velocity limits. The first step is to determine which axis will be the slowest to complete the motion 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 x_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 with four points can be generated by

```
>> via = [ 4,1; 4,4; 5,2; 2,5 ];
>> q = mstraj(via, [2,1], [], [4,1], 0.05, 0);
```

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 to a multiple of the time increment.

The first argument is the matrix of via points, one row per point. The remaining arguments are respectively: a vector of maximum speeds per axis, a vector of durations for each segment, ◄ the initial axis coordinates, the sample interval, and the acceleration time. ◄ The function `mstraj` returns a matrix with one row per time step and the columns correspond to the axes. If no output argument is provided `mstraj` will plot the trajectory as shown in Fig. 3.5a. The parameters in this example indicate that the first axis has a higher maximum speed than the second. However for the last segment both axes move at the same speed since the segment time is dominated by the slowest axis.
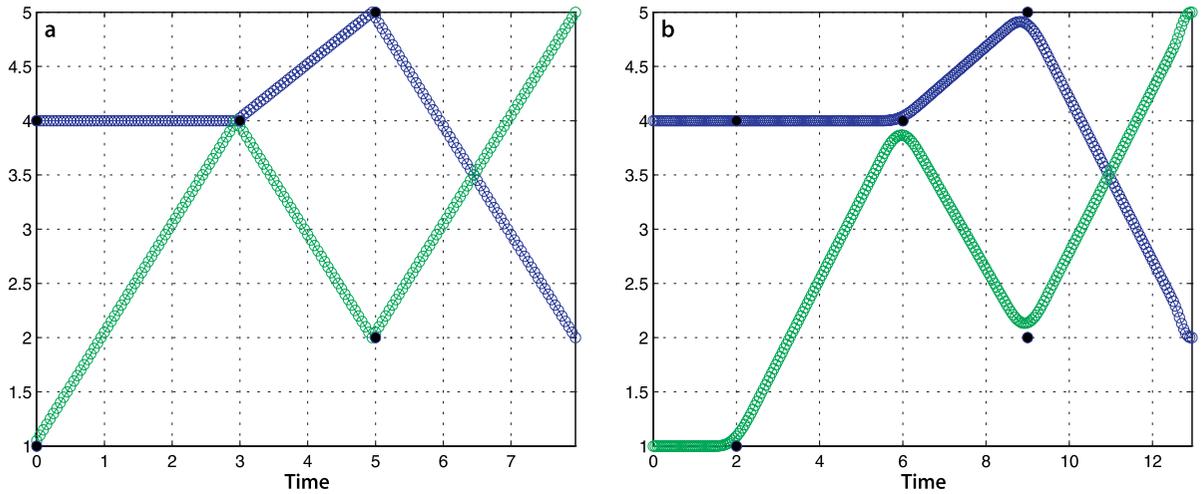
If we increase the acceleration time

```
>> q = mstraj(via, [2 1], [], [4 1], 0.05, 1);
```

the trajectory becomes more rounded, Fig. 3.5b, as the polynomial blending functions do their work, and the trajectory takes more time to complete. The function also accepts optional initial and final velocity arguments and t_{acc} can be a vector giving acceleration time 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.

3.1.4   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 ξ_0 to ξ_1. Using the notation from Chap. 2 we require some function $\xi(s) = \sigma(\xi_0, \xi_1, s)$ where $s \in [0, 1]$ which has the boundary conditions $\sigma(\xi_0, \xi_1, 0) = \xi_0$ and $\sigma(\xi_0, \xi_1, 1) = \xi_1$ and where $\sigma(\xi_0, \xi_1, s)$ varies *smoothly* for intermediate values of s. How we implement this depends very much on our concrete representation of ξ.

If pose is represented by an orthonormal rotation matrix, $\xi \sim R \in SO(3)$, we might consider a simple linear interpolation $\sigma(R_0, R_1, s) = (1 - s)R_0 + sR_1$ 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 alternative is to consider a 3-angle representation such as Euler or roll-pitch-yaw angles, $\xi \sim \Gamma \in \mathbb{S}^3$ and use linear interpolation

$$\sigma(\Gamma_0, \Gamma_1, s) = (1 - s)\Gamma_0 + s\Gamma_1$$

For example we define two orientations

```
>> R0 = rotz(-1) * roty(-1);
>> R1 = rotz(1) * roty(1);
```

and find the equivalent roll-pitch-yaw angles

```
>> rpy0 = tr2rpy(R0);   rpy1 = tr2rpy(R1);
```

and create a trajectory between them over 50 time steps

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

which is mostly easily visualized as an animation◄

```
>> tranimate( rpy2tr(rpy) );
```

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.

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 = Quaternion(R0);
>> q1 = Quaternion(R1);
```

and then interpolate them

```
>> q = interp(q0, q1, [0:49]'/49);
>> about(q)
q [Quaternion] : 1x50 (1656 bytes)◄
```

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

```
>> tranimate(q)
```

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.

3.1.5   Cartesian Motion

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 = transl(0.4, 0.2, 0) * trotx(pi);
>> T1 = transl(-0.4, -0.2, 0.3) * troty(pi/2)*trotz(-pi/2);
```

The Toolbox function trinterp provides interpolation for normalized distance along the path $s \in [0, 1]$, for example the mid pose is

```
>> trinterp(T0, T1, 0.5)
ans =
   -0.0000    1.0000         0         0
         0   -0.0000   -1.0000         0
   -1.0000         0   -0.0000    0.1500
         0         0         0    1.0000
```
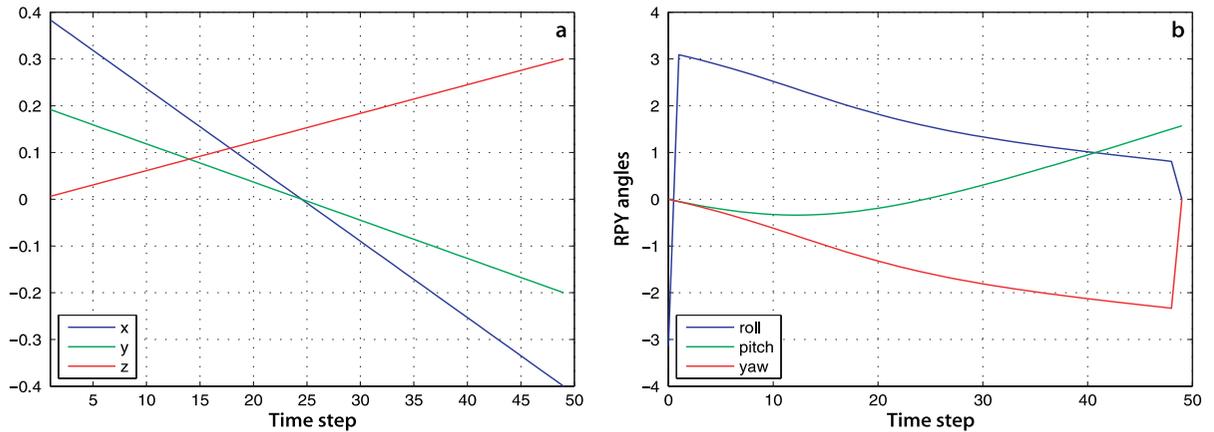
where the translational component is linearly interpolated and the rotation is is spherically interpolated using the quaternion interpolation method interp.

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

```
>> Ts = trinterp(T0, T1, [0:49]/49);
```

where the arguments are the initial and final pose and a path length varying linearly from zero to one. The resulting trajectory Ts is a matrix with three dimensions

```
>> about(Ts)
Ts [double] : 4x4x50 (6400 bytes)
```

Fig. 3.6. Cartesian motion. **a** Cartesian position versus time, **b** roll-pitch-yaw angles versus time

representing the homogeneous transformation (first 2 indices) for each time step (third index). The homogeneous transformation for the first point on the path is

```
>> Ts(:,:,1)
ans =
    1.0000         0         0    0.4000
         0   -1.0000         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

```
>> tranimate(Ts)
```

which shows the coordinate frame moving and rotating from pose `T1` to pose `T2`.

The translational part of this trajectory is obtained by

```
>> P = transl(Ts);
```

which returns the Cartesian position for the trajectory in matrix form

```
>> about(P)
P [double] : 50x3 (1200 bytes)
```

which has one row per time step, and each row is the corresponding position vector. This is plotted

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

in Fig. 3.6 along with the orientation in roll-pitch-yaw format

```
>> rpy = tr2rpy(Ts);
>> plot(rpy);
```

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

However the translational motion has a velocity and acceleration *discontinuity* at the first and last points. The problem is that while the trajectory is smooth in space the distance *s* along the trajectory 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 third argument to `trinterp`

```
>> Ts = trinterp(T0, 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 roll-pitch-yaw angles do not vary linearly with time because they represent a non-linear transformation of the linearly varying quaternion.

The discontinuity in roll angle after the first point is due to angle wrapping around the circle, moving from $-\pi$ to $+\pi$.

The discontinuity between the last two points is because the final orientation is a singularity for roll-pitch-yaw angles.
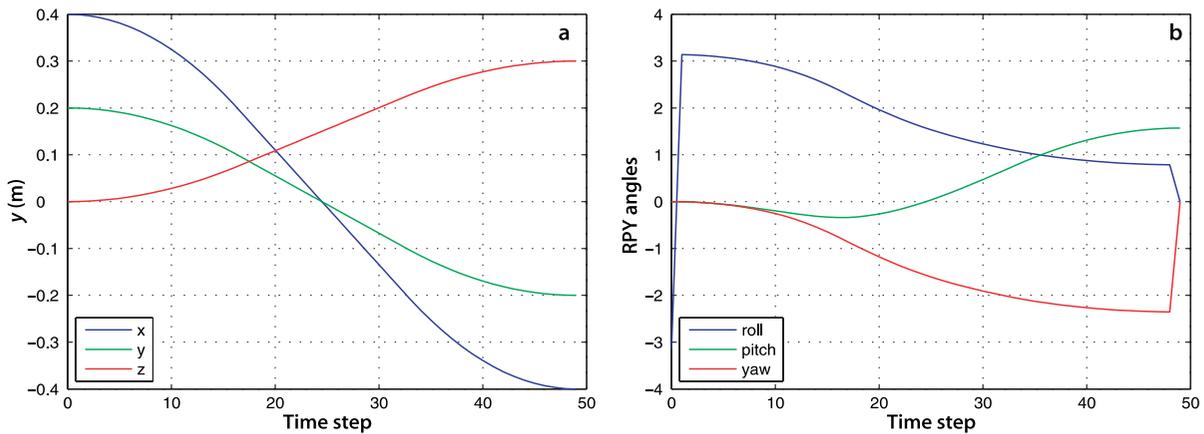
Fig. 3.7. Cartesian motion with LSPB path distance profile. **a** Cartesian position versus time, **b** roll-pitch-yaw angles versus time

the translational components of the trajectory shown in Fig. 3.7b. 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.

3.2   Time Varying Coordinate Frames

The previous section discussed the generation of coordinate frame motion which has a translational and rotational velocity component. The translational velocity is the rate of change of the position of the origin of the coordinate frame. Rotational velocity is a little more complex.

3.2.1   Rotating Coordinate Frame

A body rotating in 3-dimensional space has an angular velocity which is a vector quantity $\omega = (\omega_x, \omega_y, \omega_z)$. The direction of this vector defines the instantaneous axis of rotation, that is, the axis about which the coordinate frame is rotating at a particular instant of time. In general this axis changes with time. The magnitude of the vector is the rate of rotation about the axis – in this respect it is similar to the angle-axis representation for rotation introduced in Sect. 2.2.1.5. From mechanics there is a well known expression for the derivative of a time-varying rotation matrix

$$\dot{R}\langle t \rangle = S(\omega)R\langle t \rangle \tag{3.4}$$

Known as the angular velocity matrix.

where $R\langle t \rangle \in SO(2)$ or $SO(3)$ and $S(\cdot)$ is a skew-symmetric matrix◄ that, for the 3-dimensional case, has the form

$$S(\omega) = \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix} \tag{3.5}$$

and its properties are described in Appendix D. Using the Toolbox we can write

```
>> S = skew([1 2 3])
S =
        0       -3        2
        3        0       -1
       -2        1        0
```

The Toolbox function `vex` performs the inverse function▶ of converting a skew-symmetric matrix to a vector

```
>> vex(S)'
ans =
     1     2     3
```

We might ask what does $\dot{R}$ mean? Consider the approximation to the derivative

$$\dot{R} \approx \frac{R\langle t+\delta_t\rangle - R\langle \delta t\rangle}{\delta_t} \tag{3.6}$$

which we rearrange as

$$R\langle t+\delta_t\rangle \approx \delta_t \dot{R} + R\langle t\rangle$$

and substituting Eq. 3.4 we obtain

$$R\langle t+\delta_t\rangle \approx \delta_t S(\omega)R\langle t\rangle + R\langle t\rangle \approx \left(\delta_t S(\omega) + I_{3\times3}\right)R\langle t\rangle \tag{3.7}$$

which describes how the orthnormal rotation matrix changes as a function of angular velocity.

3.2.2 Incremental Motion

Consider a coordinate frame that undergoes a small rotation from R_0 to R_1. We can write Eq. 3.7 as

$$R_1 = \left(\delta_t S(\omega) + I_{3\times3}\right)R_0$$

and rearrange it as

$$\delta_t S(\omega) = R_1 R_0^T - I_{3\times3}$$

and then apply the vex operator, the inverse of $S(\cdot)$, to both sides

$$\delta_\Theta = \text{vex}(R_1 R_0^T - I_{3\times3}) \tag{3.8}$$

where $\delta_\Theta = \delta_t\omega$ is a 3-vector with units of angle that represents an infinitesimal rotation about the world x-, y- and z-axes.▶

We have strongly, and properly, cautioned about the non-commutivity of rotations but for infinitesimal angular changes multiplication *is commutative*. We can demonstrate this numerically by

```
>> Rdelta = rotx(0.001)*roty(0.002)*rotz(0.003)
ans =
    1.0000   -0.0030    0.0020
    0.0030    1.0000   -0.0010
   -0.0020    0.0010    1.0000
```

which is, to four significant figures, the same as

```
>> roty(0.002) * rotx(0.001)*rotz(0.003)
ans =
    1.0000   -0.0030    0.0020
    0.0030    1.0000   -0.0010
   -0.0020    0.0010    1.0000
```

Using Eq. 3.8 we can recover the infinitesimal rotation angle δ_Θ

```
>> vex( Rdelta - eye(3,3) )'
ans =
    0.0010    0.0020    0.0030
```

Given two poses ξ_0 and ξ_1 that differ infinitesimally we can represent the difference between them as a 6-vector

$$\boldsymbol{\delta} = \Delta(\xi_0, \xi_1) = (\boldsymbol{\delta}_d, \boldsymbol{\delta}_\Theta) \tag{3.9}$$

comprising the incremental displacement and the incremental rotation. The quantity $\boldsymbol{\delta} \in \mathbb{R}^6$ is effectively the spatial velocity which we discuss further in Chap. 8 multiplied by δ_t. If the poses are represented in homogeneous transformation form then the difference is

$$\boldsymbol{\delta} = \Delta(\boldsymbol{T}_0, \boldsymbol{T}_1) = \begin{pmatrix} \boldsymbol{t}_1 - \boldsymbol{t}_0 \\ \mathrm{vex}(\boldsymbol{R}_1 \boldsymbol{R}_0^T - \boldsymbol{I}_{3\times3}) \end{pmatrix} \tag{3.10}$$

where $\boldsymbol{T}_0 = (\boldsymbol{R}_0, \boldsymbol{t}_0)$ and $\boldsymbol{T}_1 = (\boldsymbol{R}_1, \boldsymbol{t}_1)$. In the Toolbox this is the function `tr2delta`.

The inverse operation is

$$\xi = \Delta^{-1}(\boldsymbol{\delta}) \tag{3.11}$$

and for homogenous transformation representation is

$$\boldsymbol{T} = \begin{pmatrix} \boldsymbol{S}(\boldsymbol{\delta}_\Theta) & \boldsymbol{\delta}_d \\ \boldsymbol{0}_{3\times1} & 0 \end{pmatrix} + \boldsymbol{I}_{3\times3} \tag{3.12}$$

and in the Toolbox this is the function `delta2tr`.

For example

```
>> T0 = transl(1,2,3)*trotx(1)*troty(1)*trotz(1);
>> T1 = T0*transl(0.01,0.02,0.03)*trotx(0.001)*troty(0.002)*trotz(0.003)
T1 =
    0.2889   -0.4547    0.8425    1.0191
    0.8372   -0.3069   -0.4527    1.9887
    0.4644    0.8361    0.2920    3.0301
         0         0         0    1.0000
```

the function $\Delta(\cdot)$ is computed by the Toolbox function `tr2delta`

```
>> d = tr2delta(T0, T1);
>> d'
ans =
    0.0191   -0.0113    0.0301    0.0019   -0.0011    0.0030
```

which comprises the incremental translation and rotation expressed in the world coordinate frame. Given this displacement and the initial pose, the final pose is

```
>> delta2tr(d) * T0
    0.2889   -0.4547    0.8425    1.0096
    0.8372   -0.3069   -0.4527    1.9859
    0.4644    0.8361    0.2920    3.0351
         0         0         0    1.0000
```

which is quite close to the true value given above and the error is due to the fact that the displacement is not infinitesimal. The displacement 6-vector is used in the next section and several times in Chap. 8.

3.2.3   Inertial Navigation Systems

An inertial navigation system is a "black box" that estimates its velocity, orientation and position with respect to the inertial reference frame (the universe). Importantly it has no external inputs such as radio signals from satellites and this makes it well suited to applications such as submarine, spacecraft and missile guidance. An inertial navigation system works by measuring accelerations and angular velocities and integrating them over time.

Early inertial navigation systems, such as shown in Fig. 2.13, used mechanical gimbals to keep the accelerometers at a constant attitude with respect to the stars using a gyro-stabilized platform. The acceleration measured on this platform was integrated to obtain the velocity of the platform, and integrated again to obtain its position. In order to achieve accurate position estimates over periods of hours or days the gimbals and gyroscopes had to be of extremely high quality so that the stable platform did not drift, and the acceleration sensors needed to be extremely accurate.

In a modern strapdown inertial measurement system the acceleration and angular velocity sensors are rigidly attached to the vehicle. The three orthogonally mounted gyroscopes measure the components of the angular velocity ω and use Eq. 3.7 to continuously update the estimated orientation ${}^{0}R_B$ of the vehicle's body-fixed frame {B} with respect to the stars {0}.

A discrete-time version of Eq. 3.7 such as

$$R\langle k+1\rangle = \delta_t S(\omega)R\langle k\rangle + R\langle k\rangle \tag{3.13}$$

is used to numerically integrate changes in pose in order to estimate the orientation of the vehicle. The measured acceleration ${}^{B}a$ of the vehicle's body frame is rotated into the inertial frame

$$ {}^{0}a = {}^{0}R_B \, {}^{B}a $$

and can then be integrated twice to update the estimate of the vehicle's position in the inertial frame. Practical systems work at high sample rate, typically hundreds of Hertz, and would employ higher-order numerical integration techniques rather than the simple rectangular integration of Eq. 3.13.

In Eq. 3.13 we added the matrix $\delta_t S(\omega)R(t)$ to an orthonormal rotation matrix and this is not quite proper – the result *will not* be an orthonormal matrix. However if the added term is small▶ the result will be close to orthonormal and we can *straighten it up*. This process is called normalization and enforces the constraints on the elements of an orthonormal matrix. It involves the following steps where c_i is the ith column of R. We first assume that column 3 is correct

Which is why inertial navigation systems operate at a high sample rate and δ_t is small.

An **Inertial Navigation System** (INS) comprises a computer and motion sensors and continuously calculates the position, orientation, and velocity of a moving object via dead reckoning. It has no need for external references which is important for vehicles, such as submarines or spacecraft, that are unable to communicate with radio navigation aids or which must be immune to radio jamming.

A gyroscope or gyro is a sensor that measures angular velocity about a particular axis. Early rotational sensors actually employed a spinning disk, like the childhood toy, but today the term gyroscope is generically applied to all angular velocity sensors. To measure the angular velocity a triaxial assembly of gyroscopes is used – three gyroscopes with their measurement axes in orthogonal directions. Modern gyroscopes are based on a variety of physical principles such as tiny vibrating beams or relativistic effects in ring-laser and fibre-optic gyros. An accelerometer is a sensor that measures acceleration along a particular axis. Accelerometers work by measuring the forces on a small proof mass within the sensor. The vehicle's acceleration vector is measured using a triaxial assembly of accelerometers. The sensors are collectively referred to as an inertial measurement unit (IMU).

Much important development was undertaken by the MIT Instrumentation Laboratory under the leadership of Charles Stark Draper. In 1953 the Space Inertial Reference Equipment (SPIRE) system, 1 200 kg of equipment, guided a B-29 bomber on a 12 hour trip from Massachusetts to Los Angeles without the aid of a pilot and with Draper aboard. In 1954 the first self-contained submarine navigation system (SINS) was introduced to service. The Instrumentation Lab also developed the Apollo Guidance Computer, a one-cubic-foot computer that guided the Apollo Lunar Module to the surface of the Moon in 1969.

Today high-performance inertial navigation systems based on fibre-optic gyroscopes are widely available and weigh around one 1 kg and low-cost systems based on MEMS technology can weigh less than 100 g and cost tens of dollars.

$$\boldsymbol{c}_3' = \boldsymbol{c}_3$$

then the first column is made orthogonal to the last two

$$\boldsymbol{c}_1' = \boldsymbol{c}_2 \times \boldsymbol{c}_3'$$

However the last two columns may not have been orthogonal so

$$\boldsymbol{c}_2' = \boldsymbol{c}_1' \times \boldsymbol{c}_3'$$

Finally the columns are normalized to unit magnitude

$$\boldsymbol{c}_i'' = \frac{\boldsymbol{c}_i'}{|\boldsymbol{c}_i'|}, \ \ i = 1 \cdots 3$$

In the Toolbox normalization is implemented by

```
>> T = trnorm(T);
```

and is quite similar to the problem of representing pose using two unit vectors discussed in Sect. 2.2.1.4. In an orientation estimation system using Eq. 3.13 the attitude **R** should be normalized after each integration step.◄

Rotation matrices also become denormalized from repeated compounding due to the accumulation of errors from finite precision arithmetic.

Alternatively we could use unit-quaternions and things, as is generally the case, are a little simpler. The derivative of a quaternion, the quaternion equivalent of Eq. 3.4 is defined as

$$\dot{\mathring{q}} = \frac{1}{2}\mathring{q}(\omega)\mathring{q} \tag{3.14}$$

which is implemented by the dot method

```
>> qd = q.dot(omega);
```

Integration of quaternion rates is achieved by

$$\mathring{q}\langle k+1 \rangle = \delta_t \dot{\mathring{q}} + \mathring{q}\langle k \rangle$$

As with Eq. 3.7 the addition is not quite proper and the result will no longer be a unit quaternion. Normalization is achieved by ensuring that the quaternion norm is unity, a straightforward division of all elements by the quaternion norm

$$\mathring{q}\langle k+1 \rangle' = \frac{\mathring{q}\langle k+1 \rangle}{|\mathring{q}\langle k+1 \rangle|}$$

or in the Toolbox

```
>> q = q.unit();
```

Quaternions are more commonly used in the rotation update equations for strapdown inertial navigation systems than orthonormal rotation matrices. The results will of course be the same but the computational cost for the quaternion version is significantly less.

3.3   Wrapping Up

In this chapter we have considered pose that varies as a function of time from two perspectives. The first perspective was to create a sequence of poses, a trajectory, that a robot can follow. An important characteristic of a trajectory is that it is *smooth* – the position and orientation varies smoothly with time. We start by discussing how to generate smooth trajectories in one dimension and then extended that to the multi-dimensional case and then to piecewise-linear trajectories that visit a number of intermediate points.

The second perspective was to examine the temporal derivative of an orthonormal rotation matrix and how that relates to concepts from mechanics such as velocity and angular velocity. This allows us to solve the inverse problem, given measurements from sensors we are able to update the estimate of pose for a moving object – the principle underlying inertial navigation. We introduced the infinitesimal motion δ which is related to spatial velocity and which we will encounter again in Chap. 8.

Interpolation of rotation and integration of angular velocity was treated using both orthonormal rotation matrices and quaternions. The results are equivalent but the quaternion formulation is more elegant and computationally more efficient.

Further Reading

The earliest work on manipulator Cartesian trajectory generation was by Paul (1972, 1979) and Taylor (1979). The muti-segment trajectory is discussed by Paul (1979, 1981) and the concept of segment transitions or blends is discussed by Lloyd and Hayward (1991). These early papers, and others, are included in the compilation on Robot Motion (Brady et al. 1982). Polynomial and LSPB trajectories are described in detail by Spong et al. (2006) and multi-segment trajectories are covered at length in Siciliano et al. (2008) and Craig (2004).

The relationship between orthonormal rotation matrices, the skew-symmetric matrix and angular velocity is well described in Spong et al. (2006).

The principles of inertial navigation are covered in the book by Groves (2008) which also covers GPS and other radio-based localization systems which are the subject of Part II. The book *Digital Apollo* (Mindell 2008) is a very readable story of the development of the inertial navigation system for the Apollo Moon landings. The paper by Corke et al. (2007) describes the principles of inertial sensors and the functionally equivalent sensors located in the inner ear of mammals that play a key role in maintaining balance.

Exercises

1. For a `tpoly` trajectory from 0 to 1 in 50 steps explore the effects of different initial and final velocities, both positive and negative. Under what circumstances does the quintic polynomial overshoot and why?
2. For a `lspb` trajectory from 0 to 1 in 50 steps explore the effects of specifying the velocity for the constant velocity segment. What are the minimum and maximum bounds possible?
3. For a trajectory from 0 to 1 and given a maximum possible velocity of 0.025 compare how many time steps are required for each of the `tpoly` and `lspb` trajectories?

4. Use `tranimate` to compare rotational interpolation using quaternions, Euler angles and roll-pitch-yaw angles. Hint: use the quaternion `interp` method, and `mtraj` with `tr2eul` and `eul2tr`
   a) Repeat for the case choose where the final orientation is at a singularity. What happens?
5. Repeat for the case where the interpolation passes through a singularity. What happens?
6. Develop a method to quantitatively compare the performance of the different orientation interpolation methods. Hint: plot the locus followed by $\hat{z}$ on a unit sphere.
7. For the `mstraj` example (page 47)
   a) Repeat with different initial and final velocity.
   b) Investigate the effect of increasing the acceleration time. Plot total time as a function of acceleration time.
8. Modify `mstraj` so that acceleration limits are taken into account when determining the segment time.
9. Implement an inertial measurement system. First create an angular velocity signal as a function of time, for example

   ```
   >> t = [0:0.01:10]';
   >> w = [0.1*sin(t) 0.2*sin(0.6*t) 0.3*sin(0.4*t)];
   ```

   a) Estimate the rotation matrix at each time step, then animate it.
   b) Repeat using quaternions.
   c) Add a small amount of Gaussian noise $w = w + \text{randn}(\text{size}(w)) * 0.001$ and repeat. What changes?
   d) Investigate performance for increasing amounts of noise.