Peter Corke

Robotics, Vision and Control

Fundamental Algorithms in MATLAB

Second, completely revised, extended and updated edition

With 492 Images

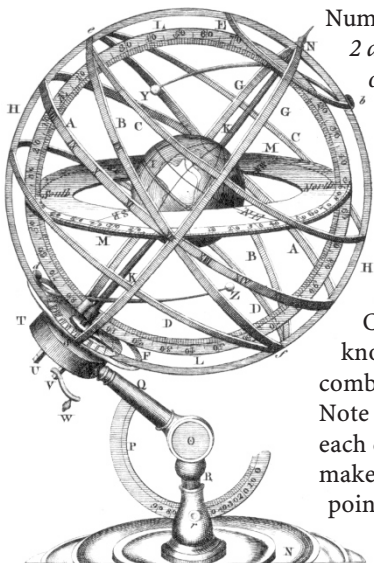
Additional material is provided at www.petercorke.com/RVC

Part I Foundations

Chapter 2 **Representing Position and Orientation**

Chapter 3 **Time and Motion**

2 Representing Position and Orientation



Numbers are an important part of mathematics. We use numbers for counting: *there are 2 apples*. We use *denominate numbers*, a number plus a unit, to specify distance: *the object is 2* m *away*. We also call this single number a *scalar*. We use a vector, a denominate number plus a direction, to specify a location: *the object is 2* m *due north*. We may also want to know the orientation of the object: *the object is 2* m *due north and facing west*. The combination of position and orientation we call *pose*.

A point in space is a familiar concept from mathematics and can be described by a coordinate vector, as shown in Fig. 2.1a. The vector represents the displacement of the point with respect to some reference coordinate frame – we call this a bound vector since it cannot be freely moved. A coordinate frame, or Cartesian coordinate system, is a set of orthogonal axes which intersect at a point known as the origin. A vector can be described in terms of its components, a linear combination of unit vectors which are parallel to the axes of the coordinate frame. Note that points and vectors are different types of mathematical objects even though each can be described by a tuple of numbers. We can add vectors but adding points makes no sense. The difference of two points is a vector, and we can add a vector to a point to obtain another point.

A point is an interesting mathematical abstraction, but a real object comprises infinitely many points. An object, unlike a point, also has an orientation. If we attach a coordinate frame to an object, as shown in Fig. 2.1b, we can describe every point within the object as a constant vector with respect to that frame.◂ Now we can describe the position and orientation – the pose – of that coordinate frame with respect to the reference coordinate frame. To distinguish the different frames we label them and in this case the object coordinate frame is labeled $\{B\}$ and its axes are labeled x_B and y_B, adopting the frame's label as their subscript.

We assume that the object is rigid, that is, the points do not move with respect to each other.

To completely describe the pose of a rigid object in a 3-dimensional world we need 6 not 3 dimensions: 3 to describe its position and 3 to describe its orientation. These dimensions behave quite differently. If we increase the value of one of the position dimensions the object will move continuously in a straight line, but if we increase the value of one of the orientation dimensions the object will rotate in some way and soon get back to its original orientation – this dimension is curved. We clearly need to treat the position and orientation dimensions quite differently.

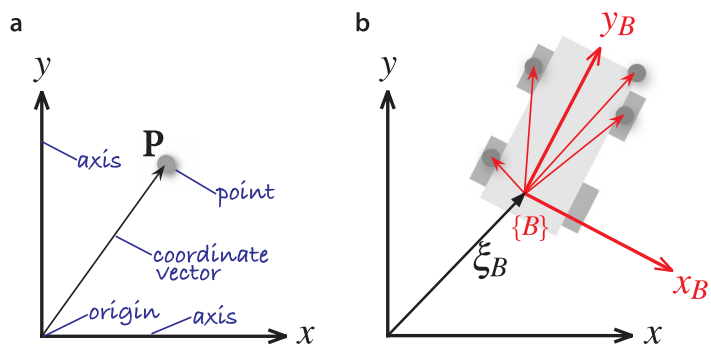


Fig. 2.1.
a The point **P** is described by a coordinate vector with respect to an absolute coordinate frame. **b** The points are described with respect to the object's coordinate frame $\{B\}$ which in turn is described by a relative pose ξ_B. Axes are denoted by thick lines with an open arrow, vectors by thin lines with a swept arrow head and a pose by a thick line with a solid head

The pose of the coordinate frame is denoted by the symbol ξ – pronounced ksi. Figure 2.2 shows two frames $\{A\}$ and $\{B\}$ and the relative pose ${}^{A}\xi_{B}$ which describes $\{B\}$ with respect to $\{A\}$. The leading superscript denotes the reference coordinate frame and the subscript denotes the frame being described. We could also think of ${}^{A}\xi_{B}$ as describing some motion – imagine picking up $\{A\}$ and applying a displacement and a rotation so that it is transformed to $\{B\}$. If the initial superscript is missing we assume that the change in pose is relative to the world coordinate frame which is generally denoted $\{O\}$.

The point **P** in Fig. 2.2 can be described with respect to *either* coordinate frame by the vectors ${}^{A}\boldsymbol{p}$ or ${}^{B}\boldsymbol{p}$ respectively. Formally they are related by

$$ {}^{A}\boldsymbol{p} = {}^{A}\xi_{B} \bullet {}^{B}\boldsymbol{p} \tag{2.1} $$

where the right-hand side expresses the motion from $\{A\}$ to $\{B\}$ and then to **P**. The operator $\bullet$ *transforms* the vector, resulting in a new vector that describes the same point but with respect to a different coordinate frame.

An important characteristic of relative poses is that they can be *composed* or *compounded*. Consider the case shown in Fig. 2.3. If one frame can be described in terms of another by a relative pose then they can be applied sequentially

$$ {}^{A}\xi_{C} = {}^{A}\xi_{B} \oplus {}^{B}\xi_{C} $$

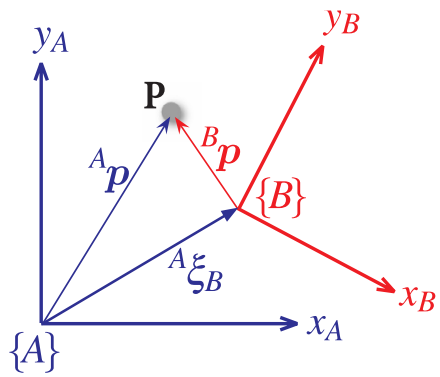


Fig. 2.2.
The point **P** can be described by coordinate vectors relative to either frame $\{A\}$ or $\{B\}$. The pose of $\{B\}$ relative to $\{A\}$ is ${}^{A}\xi_{B}$

In relative pose composition we can check that we have our reference frames correct by ensuring that the subscript and superscript on each side of the $\oplus$ operator are matched. We can then *cancel out* the intermediate subscripts and superscripts

$$ {}^{X}\xi_{Z} = {}^{X}\xi_{\cancel{Y}} \oplus {}^{\cancel{Y}}\xi_{Z} $$

leaving just the end most subscript and superscript which are shown highlighted.

Euclid of Alexandria (ca. 325 BCE–265 BCE) was a Greek mathematician, who was born and lived in Alexandria Egypt, and is considered the "father of geometry". His great work *Elements* comprising 13 books, captured and systematized much early knowledge about geometry and numbers. It deduces the properties of planar and solid geometric shapes from a set of 5 axioms and 5 postulates.

Elements is probably the most successful book in the history of mathematics. It describes plane geometry and is the basis for most people's first introduction to geometry and formal proof, and is the basis of what we now call Euclidean geometry. Euclidean distance is simply the distance between two points on a plane. Euclid also wrote *Optics* which describes geometric vision and perspective.

which says, in words, that the pose of {*C*} relative to {*A*} can be obtained by compounding the relative poses from {*A*} to {*B*} and {*B*} to {*C*}. We use the operator $\oplus$ to indicate *composition* of relative poses.

For this case the point **P** can be described by

$$ {}^{A}\boldsymbol{p} = \left({}^{A}\xi_{B} \oplus {}^{B}\xi_{C} \right) \bullet {}^{C}\boldsymbol{p} $$

Later in this chapter we will convert these abstract notions of ξ, $\bullet$ and $\oplus$ into standard mathematical objects and operators that we can implement in MATLAB®.

In the examples so far we have shown 2-dimensional coordinate frames. This is appropriate for a large class of robotics problems, particularly for mobile robots which operate in a planar world. For other problems we require 3-dimensional coordinate frames to describe objects in our 3-dimensional world such as the pose of a flying or underwater robot or the end of a tool carried by a robot arm.

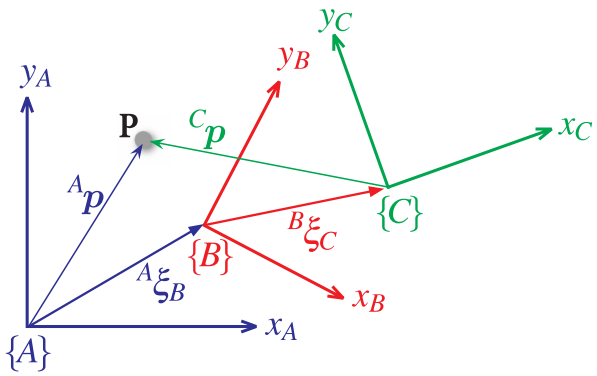
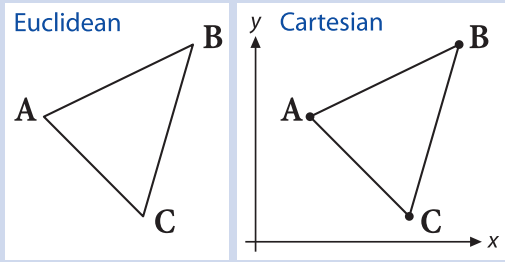


Fig. 2.3.
The point **P** can be described by coordinate vectors relative to either frame {*A*}, {*B*} or {*C*}. The frames are described by relative poses

Euclidean versus Cartesian geometry. Euclidean geometry is concerned with points and lines in the Euclidean plane (2D) or Euclidean space (3D). It is entirely based on a set of axioms and makes no use of arithmetic. Descartes added a coordinate system (2D or 3D) and was then able to describe points, lines and other curves in terms of algebraic equations. The study of such equations is called analytic geometry and is the basis of all modern geometry. The Cartesian plane (or space) is the Euclidean plane (or space) with all its axioms and postulates *plus* the extra facilities afforded by the added coordinate system. The term Euclidean geometry is often used to mean that Euclid's fifth postulate (parallel lines never intersect) holds, which is the case for a planar surface but not for a curved surface.



René Descartes (1596–1650) was a French philosopher, mathematician and part-time mercenary. He is famous for the philosophical statement "*Cogito, ergo sum*" or "*I am thinking, therefore I exist*" or "*I think, therefore I am*". He was a sickly child and developed a life-long habit of lying in bed and thinking until late morning. A possibly apocryphal story is that during one such morning he was watching a fly walk across the ceiling and realized that he could describe its position in terms of its distance from the two edges of the ceiling. This is the basis of the *Cartesian* coordinate system and modern (analytic) geometry, which he described in his 1637 book La Géométrie. For the first time mathematics and geometry were connected, and modern calculus was built on this foundation by Newton and Leibniz. In Sweden at the invitation of Queen Christina he was obliged to rise at 5 A.M., breaking his lifetime habit – he caught pneumonia and died. His remains were later moved to Paris, and are now lost apart from his skull which is in the Musée de l'Homme. After his death, the Roman Catholic Church placed his works on the Index of Prohibited Books.

Figure 2.4 shows a more complex 3-dimensional example in a graphical form where we have attached 3D coordinate frames to the various entities and indicated some relative poses. The fixed camera observes the object from its fixed viewpoint and estimates the object's pose ${}^{F}\xi_{B}$ relative to itself. The other camera is not fixed, it is attached to the robot at some constant relative pose and estimates the object's pose ${}^{C}\xi_{B}$ relative to itself.

An alternative representation of the spatial relationships is a directed graph (see Appendix I) which is shown in Fig. 2.5. Each node in the graph represents a pose and each edge represents a relative pose. An arrow from X to Y is denoted ${}^{X}\xi_{Y}$ and describes the pose of Y relative to X. Recalling that we can compose relative poses using the $\oplus$ operator we can write some spatial relationships

$$\xi_F \oplus {}^{F}\xi_{B} = \xi_R \oplus {}^{R}\xi_{C} \oplus {}^{C}\xi_{B}$$
$$\xi_F \oplus {}^{F}\xi_{R} = \xi_R$$

and each equation represents a loop in the graph with each side of the equation starting and ending at the same node. Each side of the first equation represents a path through the network from {0} to {B}, a sequence of edges (arrows) written in order.

It is quite possible that a pose graph can be inconsistent, that is, two paths through the graph give different results. In robotics these poses are only ever derived from noisy sensor data.

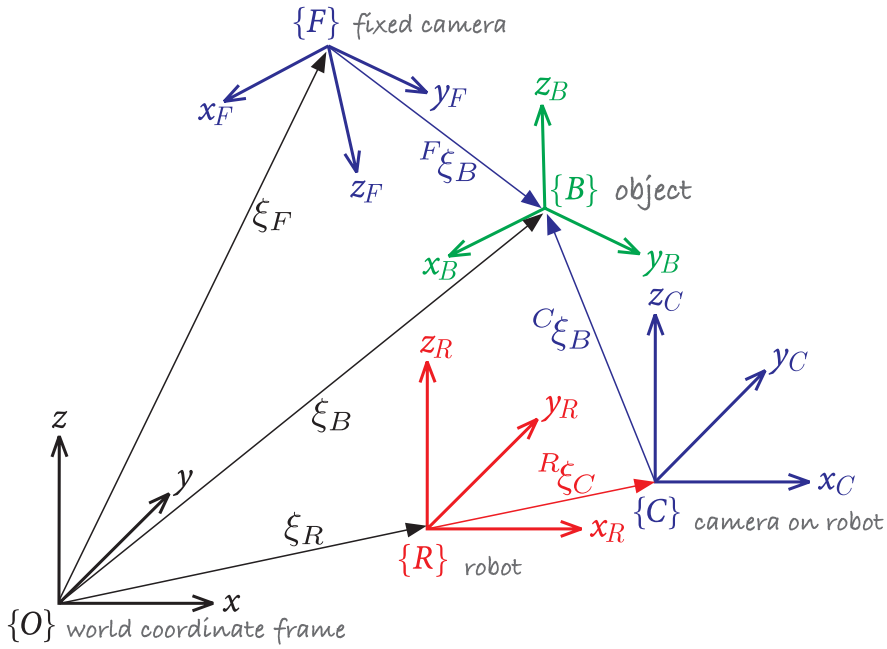


Fig. 2.4.
Multiple 3-dimensional coordinate frames and relative poses

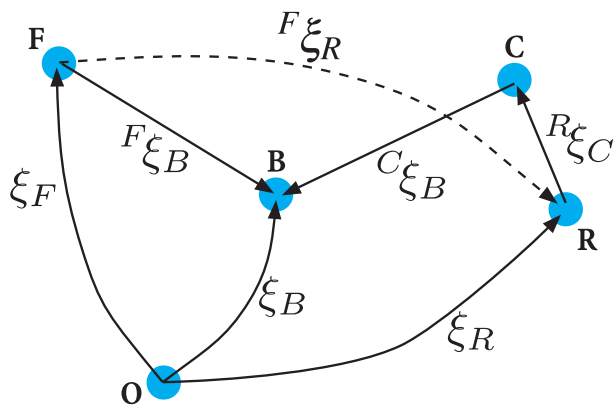


Fig. 2.5.
Spatial example of Fig. 2.4 expressed as a directed graph

In mathematical terms poses constitute a group – a set of objects that supports an associative binary operator (composition) whose result belongs to the group, an inverse operation and an identity element. In this case the group is the special Euclidean group in either 2 or 3 dimensions which are commonly referred to as **SE**(2) or **SE**(3) respectively.

There are just a few algebraic rules:

$$\xi \oplus 0 = \xi, \quad \xi \ominus 0 = \xi$$
$$\xi \ominus \xi = 0, \quad \ominus\xi \oplus \xi = 0$$

where 0 represents a zero relative pose. A pose has an inverse

$$\ominus {}^{X}\xi_{Y} = {}^{Y}\xi_{X}$$

which is represented graphically by an arrow from $\{Y\}$ to $\{X\}$. Relative poses can also be composed or compounded

$${}^{X}\xi_{Y} \oplus {}^{Y}\xi_{Z} = {}^{X}\xi_{Z}$$

It is important to note that the algebraic rules for poses are different to normal algebra and that composition is *not* commutative

$$\xi_1 \oplus \xi_2 \neq \xi_2 \oplus \xi_1$$

with the exception being the case where $\xi_1 \oplus \xi_2 = 0$. A relative pose can transform a point expressed as a vector relative to one frame to a vector relative to another

$${}^{X}\boldsymbol{p} = {}^{X}\xi_{Y} \bullet {}^{Y}\boldsymbol{p}$$

A very useful property of poses is the ability to perform algebra. The second loop equation says, in words, that the pose of the robot is the same as composing two relative poses: from the world frame to the fixed camera and from the fixed camera to the robot. We can subtract ξ_F from both sides of the equation by adding the inverse of ξ_F which we denote as $\ominus\xi_F$ and this gives

Order is important here, and we add $\ominus\xi_F$ to the left on each side of the equation.

$$\ominus\xi_F \oplus \xi_F \oplus {}^{F}\xi_{R} = \ominus\xi_F \oplus \xi_R$$
$${}^{F}\xi_{R} = \ominus\xi_F \oplus \xi_R$$

which is the pose of the robot relative to the fixed camera, shown as a dashed line in Fig. 2.5.

We can write these expressions quickly by inspection. To find the pose of node X with respect to node Y:

- find a path from Y to X and write down the relative poses on the edges in a left to right order;
- if you traverse the edge in the direction of its arrow precede it with the $\oplus$ operator, otherwise use $\ominus$.

So what is ξ? It can be any mathematical object that supports the algebra described above and is suited to the problem at hand. It will depend on whether we are considering a 2- or 3-dimensional problem. Some of the objects that we will discuss in the rest of this chapter will be familiar to us, for example vectors, but others will be more exotic mathematical objects such as homogeneous transformations, orthonormal rotation matrices, twists and quaternions. Fortunately all these mathematical objects are well suited to the mathematical programming environment of MATLAB.

To recap:

1. A point is described by a bound coordinate vector that represents its displacement from the origin of a reference coordinate system.
2. Points and vectors are different things even though they are each described by a tuple of numbers. We can add vectors but not points. The difference between two points is a vector.
3. A set of points that represent a rigid object can be described by a single coordinate frame, and its constituent points are described by constant vectors relative to that coordinate frame.
4. The position and orientation of an object's coordinate frame is referred to as its pose.
5. A relative pose describes the pose of one coordinate frame with respect to another and is denoted by an algebraic variable ξ.
6. A coordinate vector describing a point can be represented with respect to a different coordinate frame by applying the relative pose to the vector using the $\bullet$ operator.
7. We can perform algebraic manipulation of expressions written in terms of relative poses and the operators $\oplus$ and $\ominus$.

The remainder of this chapter discusses concrete representations of ξ for various common cases that we will encounter in robotics and computer vision. We start by considering the two-dimensional case which is comparatively straightforward and then extend those concepts to three dimensions. In each case we consider rotation first, and then add translation to create a description of pose.

2.1 Working in Two Dimensions (2D)

A 2-dimensional world, or plane, is familiar to us from high-school Euclidean geometry. We use a right-handed▸ Cartesian coordinate system or coordinate frame with orthogonal axes denoted x and y and typically drawn with the x-axis horizontal and the y-axis vertical. The point of intersection is called the origin. Unit-vectors parallel to the axes are denoted $\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{y}}$. A point is represented by its x- and y-coordinates (x, y) or as a bound vector

The relative orientation of the x- and y-axes obey the right-hand rule as shown on page 31.

$$\boldsymbol{p} = x\hat{\boldsymbol{x}} + y\hat{\boldsymbol{y}} \tag{2.2}$$

Figure 2.6 shows a red coordinate frame $\{B\}$ that we wish to describe with respect to the blue reference frame $\{A\}$. We can see clearly that the origin of $\{B\}$ has been displaced by the vector $\boldsymbol{t} = (x, y)$ and then rotated counter-clockwise by an angle θ.

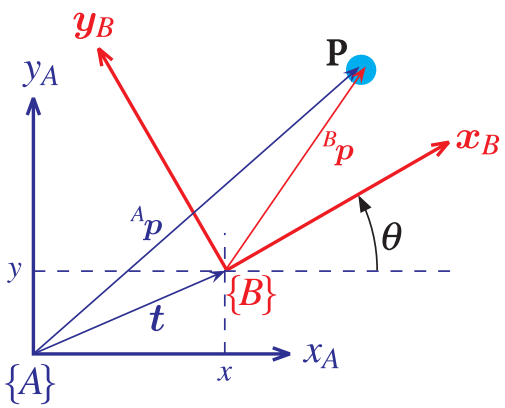


Fig. 2.6.
Two 2D coordinate frames $\{A\}$ and $\{B\}$ and a world point **P**. $\{B\}$ is rotated and translated with respect to $\{A\}$

A concrete representation of pose is therefore the 3-vector ${}^{A}\xi_{B} \sim (x, y, \theta)$, and we use the symbol $\sim$ to denote that the two representations are equivalent. Unfortunately this *representation* is not convenient for compounding since

$$(x_1, y_1, \theta_1) \oplus (x_2, y_2, \theta_2)$$

is a complex trigonometric function of both poses. Instead we will look for a different way to represent rotation and pose. We will consider the problem in two parts: rotation and then translation.

2.1.1 Orientation in 2-Dimensions

2.1.1.1 Orthonormal Rotation Matrix

Consider an arbitrary point **P** which we can express with respect to each of the coordinate frames shown in Fig. 2.6. We create a new frame $\{V\}$ whose axes are parallel to those of $\{A\}$ but whose origin is the same as $\{B\}$, see Fig. 2.7. According to Eq. 2.2 we can express the point **P** with respect to $\{V\}$ in terms of the unit-vectors that define the axes of the frame

$$\begin{aligned} {}^{V}p &= {}^{V}x\hat{\boldsymbol{x}}_V + {}^{V}y\hat{\boldsymbol{y}}_V \\ &= (\hat{\boldsymbol{x}}_V \quad \hat{\boldsymbol{y}}_V)\begin{pmatrix} {}^{V}x \\ {}^{V}y \end{pmatrix} \end{aligned} \tag{2.3}$$

which we have written as the product of a row and a column vector.

The coordinate frame $\{B\}$ is completely described by its two orthogonal axes which we represent by two unit vectors

$$\begin{aligned} \hat{\boldsymbol{x}}_B &= \cos\theta\hat{\boldsymbol{x}}_V + \sin\theta\hat{\boldsymbol{y}}_V \\ \hat{\boldsymbol{y}}_B &= -\sin\theta\hat{\boldsymbol{x}}_V + \cos\theta\hat{\boldsymbol{y}}_V \end{aligned}$$

which can be factorized into matrix form as

$$(\hat{\boldsymbol{x}}_B \quad \hat{\boldsymbol{y}}_B) = (\hat{\boldsymbol{x}}_V \quad \hat{\boldsymbol{y}}_V)\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \tag{2.4}$$

Using Eq. 2.2 we can represent the point **P** with respect to $\{B\}$ as

$$\begin{aligned} {}^{B}p &= {}^{B}x\hat{\boldsymbol{x}}_B + {}^{B}y\hat{\boldsymbol{y}}_B \\ &= (\hat{\boldsymbol{x}}_B \quad \hat{\boldsymbol{y}}_B)\begin{pmatrix} {}^{B}x \\ {}^{B}y \end{pmatrix} \end{aligned}$$

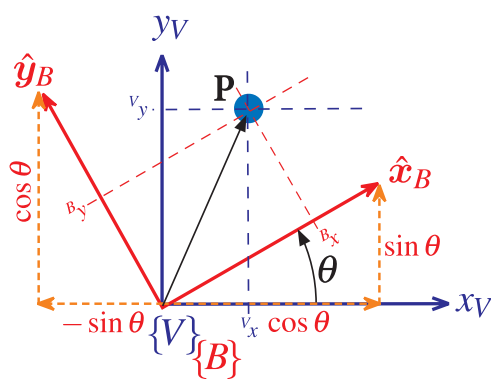


Fig. 2.7. Rotated coordinate frames in 2D. The point **P** can be considered with respect to the red or blue coordinate frame

and substituting Eq. 2.4 we write

$$ {}^{B}p = \begin{pmatrix} \hat{\boldsymbol{x}}_V & \hat{\boldsymbol{y}}_V \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} {}^{B}x \\ {}^{B}y \end{pmatrix} \tag{2.5} $$

Now by equating the coefficients of the right-hand sides of Eq. 2.3 and Eq. 2.5 we write

$$ \begin{pmatrix} {}^{V}x \\ {}^{V}y \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} {}^{B}x \\ {}^{B}y \end{pmatrix} $$

which describes how points are transformed from frame $\{B\}$ to frame $\{V\}$ when the frame is rotated. This type of matrix is known as a rotation matrix since it transforms a point from frame $\{V\}$ to $\{B\}$ and is denoted ${}^{V}\boldsymbol{R}_B$

$$ \begin{pmatrix} {}^{V}x \\ {}^{V}y \end{pmatrix} = {}^{V}\boldsymbol{R}_B \begin{pmatrix} {}^{B}x \\ {}^{B}y \end{pmatrix} \tag{2.6} $$

$$ {}^{X}\boldsymbol{R}_Y(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} $$

is a 2-dimensional rotation matrix with some special properties:

- it is *orthonormal* (also called *orthogonal*) since each of its columns is a unit vector and the columns are orthogonal.
- the columns are the unit vectors that define the axes of the rotated frame Y with respect to X and are by definition both unit-length and orthogonal.
- it belongs to the special orthogonal group of dimension 2 or $\boldsymbol{R} \in \mathbf{SO}(2) \subset \mathbb{R}^{2\times 2}$. This means that the product of any two matrices belongs to the group, as does its inverse.
- its *determinant* is $+1$, which means that the length of a vector is unchanged after transformation, that is, $\|{}^{Y}\boldsymbol{p}\| = \|{}^{X}\boldsymbol{p}\|, \forall\theta$.
- the inverse is the same as the transpose, that is, $\boldsymbol{R}^{-1} = \boldsymbol{R}^{T}$.

See Appendix B which provides a refresher on vectors, matrices and linear algebra.

We can rearrange Eq. 2.6 as

$$ \begin{pmatrix} {}^{B}x \\ {}^{B}y \end{pmatrix} = \left({}^{V}\boldsymbol{R}_B\right)^{-1} \begin{pmatrix} {}^{V}x \\ {}^{V}y \end{pmatrix} = \left({}^{V}\boldsymbol{R}_B\right)^{T} \begin{pmatrix} {}^{V}x \\ {}^{V}y \end{pmatrix} = {}^{B}\boldsymbol{R}_V \begin{pmatrix} {}^{V}x \\ {}^{V}y \end{pmatrix} $$

Note that inverting the matrix is the same as swapping the superscript and subscript, which leads to the identity $\boldsymbol{R}(-\theta) = \boldsymbol{R}(\theta)^{T}$.

It is interesting to observe that instead of representing an angle, which is a scalar, we have used a 2×2 matrix that comprises four elements, however these elements are not independent. Each column has a unit magnitude which provides two constraints. The columns are orthogonal which provides another constraint. Four elements and three constraints are effectively one independent value. The rotation matrix is an example of a nonminimum representation and the disadvantages such as the increased memory it requires are outweighed, as we shall see, by its advantages such as composability.

The Toolbox allows easy creation of these rotation matrices

```
>> R = rot2(0.2)
R =
    0.9801   -0.1987
    0.1987    0.9801
```

where the angle is specified in radians. We can observe some of the properties such as

```
>> det(R)
ans =
     1
```

and the product of two rotation matrices is also a rotation matrix

```
>> det(R*R)
ans =
     1
```

You will need to have the MATLAB Symbolic Math Toolbox™ installed.

The Toolbox also supports symbolic mathematics◀ for example

```
>> syms theta
>> R = rot2(theta)
R =
[ cos(theta), -sin(theta)]
[ sin(theta),  cos(theta)]
>> simplify(R*R)
ans =
[ cos(2*theta), -sin(2*theta)]
[ sin(2*theta),  cos(2*theta)]
>> simplify(det(R))
ans =
1
```

2.1.1.2 Matrix Exponential

Consider a pure rotation of 0.3 radians expressed as a rotation matrix

```
>> R = rot2(0.3)
ans =
    0.9553   -0.2955
    0.2955    0.9553
```

`logm` is different to the builtin function `log` which computes the logarithm of each element of the matrix. A logarithm can be computed using a power series, with a matrix rather than scalar argument. For a matrix the logarithm is not unique and `logm` computes the principal logarithm of the matrix.

We can compute the logarithm of this matrix using the MATLAB builtin function `logm`◀

```
>> S = logm(R)
S =
    0.0000   -0.3000
    0.3000    0.0000
```

and the result is a simple matrix with two elements having a magnitude of 0.3, which intriguingly is the original rotation angle. There is something deep and interesting going on here – we are on the fringes of Lie group theory which we will encounter throughout this chapter.

In 2 dimensions the skew-symmetric matrix is

$$[\omega]_\times = \begin{pmatrix} 0 & -\omega \\ \omega & 0 \end{pmatrix} \tag{2.7}$$

which has clear structure and only one unique element $\omega \in \mathbb{R}$. A simple example of Toolbox support for skew-symmetric matrices is

```
>> skew(2)
ans =
     0    -2
     2     0
```

and the inverse operation is performed using the Toolbox function `vex`

```
>> vex(ans)
ans =
     2
```

This matrix has a zero diagonal and is an example of a 2×2 skew-symmetric matrix. The matrix has only one unique element and we can unpack it using the Toolbox function `vex`

```
>> vex(S)
ans =
    0.3000
```

to recover the rotation angle.

The inverse of a logarithm is exponentiation and using the builtin MATLAB matrix exponential function `expm`

`expm` is different to the builtin function `exp` which computes the exponential of each element of the matrix. $\text{expm}(A) = I + A + A^2/2! + A^3/3! + \cdots$

```
>> expm(S)
ans =
    0.9553   -0.2955
    0.2955    0.9553
```

the result is, as expected, our original rotation matrix. In fact the command

```
>> R = rot2(0.3);
```

is equivalent to

```
>> R = expm(  skew(0.3)  );
```

Formally we can write

$$\boldsymbol{R} = \mathrm{e}^{[\theta]_\times} \in \mathbf{SO}(2)$$

where θ is the rotation angle, and the notation $[\cdot]_\times : \mathbb{R} \mapsto \mathbb{R}^{2\times 2}$ indicates a mapping from a scalar to a skew-symmetric matrix.

2.1.2 Pose in 2-Dimensions

2.1.2.1 Homogeneous Transformation Matrix

Now we need to account for the translation between the origins of the frames shown in Fig. 2.6. Since the axes $\{V\}$ and $\{A\}$ are parallel, as shown in Figs. 2.6 and 2.7, this is simply vectorial addition

$$\begin{pmatrix} {}^A x \\ {}^A y \end{pmatrix} = \begin{pmatrix} {}^V x \\ {}^V y \end{pmatrix} + \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.8}$$

$$= \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} {}^B x \\ {}^B y \end{pmatrix} + \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.9}$$

$$= \begin{pmatrix} \cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y \end{pmatrix} \begin{pmatrix} {}^B x \\ {}^B y \\ 1 \end{pmatrix} \tag{2.10}$$

or more compactly as

$$\begin{pmatrix} {}^A x \\ {}^A y \\ 1 \end{pmatrix} = \begin{pmatrix} {}^A\boldsymbol{R}_B & \boldsymbol{t} \\ \boldsymbol{0}_{1\times 2} & 1 \end{pmatrix} \begin{pmatrix} {}^B x \\ {}^B y \\ 1 \end{pmatrix} \tag{2.11}$$

where $\boldsymbol{t} = (x, y)$ is the translation of the frame and the orientation is ${}^A\boldsymbol{R}_B$. Note that ${}^A\boldsymbol{R}_B = {}^V\boldsymbol{R}_B$ since the axes of frames $\{A\}$ and $\{V\}$ are parallel. The coordinate vectors for point $\mathbf{P}$ are now expressed in homogeneous form and we write

A vector $\boldsymbol{p} = (x, y)$ is written in homogeneous form as $\tilde{\boldsymbol{p}} \in \mathbb{P}^2$, $\tilde{\boldsymbol{p}} = (x_1, x_2, x_3)$ where $x = x_1 / x_3$, $y = x_2 / x_3$ and $x_3 \neq 0$. The dimension has been increased by one and a point on a plane is now represented by a 3-vector. To convert a point to homogeneous form we typically append an element equal to one $\tilde{\boldsymbol{p}} = (x, y, 1)$. The tilde indicates the vector is homogeneous.

Homogeneous vectors have the important property that $\tilde{\boldsymbol{p}}$ is equivalent to $\lambda \tilde{\boldsymbol{p}}$ for all $\lambda \neq 0$ which we write as $\tilde{\boldsymbol{p}} \simeq \lambda \tilde{\boldsymbol{p}}$. That is $\tilde{\boldsymbol{p}}$ represents the same point in the plane irrespective of the overall scaling factor. Homogeneous representation is important for computer vision that we discuss in Part IV. Additional details are provided in Sect. C.2.

$$
\begin{aligned}
{}^A\tilde{\boldsymbol{p}} &= \begin{pmatrix} {}^A\boldsymbol{R}_B & \boldsymbol{t} \\ \boldsymbol{0}_{1\times 2} & 1 \end{pmatrix} {}^B\tilde{\boldsymbol{p}} \\
&= {}^A\boldsymbol{T}_B \, {}^B\tilde{\boldsymbol{p}}
\end{aligned}
$$

and ${}^A\boldsymbol{T}_B$ is referred to as a homogeneous transformation. The matrix has a very specific structure and belongs to the special Euclidean group of dimension 2 or $\boldsymbol{T} \in \mathbf{SE}(2) \subset \mathbb{R}^{3\times 3}$.

By comparison with Eq. 2.1 it is clear that ${}^A\boldsymbol{T}_B$ represents translation and orientation or relative pose. This is often referred to as a *rigid-body motion*.

$$
\boldsymbol{T} = \begin{pmatrix} \cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y \\ 0 & 0 & 1 \end{pmatrix}
$$

A concrete representation of relative pose ξ is $\xi \sim \boldsymbol{T} \in \mathbf{SE}(2)$ and $\boldsymbol{T}_1 \oplus \boldsymbol{T}_2 \mapsto \boldsymbol{T}_1 \boldsymbol{T}_2$ which is standard matrix multiplication

$$
\boldsymbol{T}_1 \boldsymbol{T}_2 = \begin{pmatrix} \boldsymbol{R}_1 & \boldsymbol{t}_1 \\ \boldsymbol{0}_{1\times 2} & 1 \end{pmatrix} \begin{pmatrix} \boldsymbol{R}_2 & \boldsymbol{t}_2 \\ \boldsymbol{0}_{1\times 2} & 1 \end{pmatrix} = \begin{pmatrix} \boldsymbol{R}_1 \boldsymbol{R}_2 & \boldsymbol{t}_1 + \boldsymbol{R}_1 \boldsymbol{t}_2 \\ \boldsymbol{0}_{1\times 2} & 1 \end{pmatrix}
$$

One of the algebraic rules from page 21 is $\xi \oplus 0 = \xi$. For matrices we know that $\boldsymbol{T}\boldsymbol{I} = \boldsymbol{T}$, where $\boldsymbol{I}$ is the identify matrix, so for pose $0 \mapsto \boldsymbol{I}$ the identity matrix. Another rule was that $\xi \ominus \xi = 0$. We know for matrices that $\boldsymbol{T}\boldsymbol{T}^{-1} = \boldsymbol{I}$ which implies that $\ominus \boldsymbol{T} \mapsto \boldsymbol{T}^{-1}$

$$
\boldsymbol{T}^{-1} = \begin{pmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}_{1\times 2} & 1 \end{pmatrix}^{-1} = \begin{pmatrix} \boldsymbol{R}^T & -\boldsymbol{R}^T \boldsymbol{t} \\ \boldsymbol{0}_{1\times 2} & 1 \end{pmatrix}
$$

For a point described by $\tilde{\boldsymbol{p}} \in \mathbb{P}^2$ then $\boldsymbol{T} \cdot \tilde{\boldsymbol{p}} \mapsto \boldsymbol{T}\tilde{\boldsymbol{p}}$ which is a standard matrix-vector product.

To make this more tangible we will show some numerical examples using MATLAB and the Toolbox. We create a homogeneous transformation which represents a translation of (1, 2) followed by a rotation of 30°

```
>> T1 = transl2(1, 2) * trot2(30, 'deg')
T1 =
    0.8660   -0.5000    1.0000
    0.5000    0.8660    2.0000
         0         0    1.0000
```

Many Toolbox functions have variants that return orthonormal rotation matrices or homogeneous transformations, for example, `rot2` and `trot2`.

The function `transl2` creates a relative pose with a finite translation but zero rotation, while `trot2` creates a relative pose with a finite rotation but zero translation. We can plot this, relative to the world coordinate frame, by

```
>> plotvol([0 5 0 5]);
>> trplot2(T1, 'frame', '1', 'color', 'b')
```

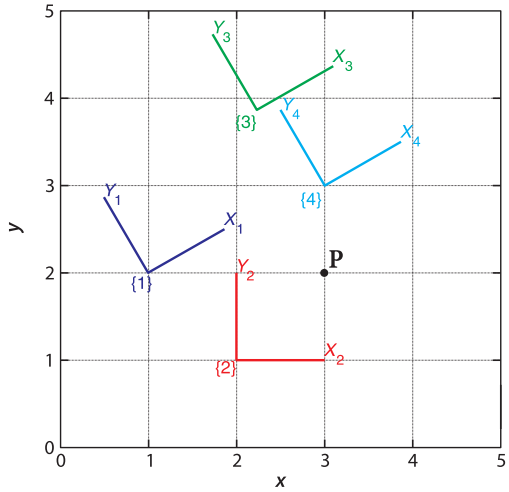


Fig. 2.8.
Coordinate frames drawn using the Toolbox function `trplot2`

The options specify that the label for the frame is {1} and it is colored blue and this is shown in Fig. 2.8. We create another relative pose which is a displacement of (2, 1) and zero rotation

```
>> T2 = transl2(2, 1)
T2 =
     1     0     2
     0     1     1
     0     0     1
```

which we plot in red

```
>> trplot2(T2, 'frame', '2', 'color', 'r');
```

Now we can compose the two relative poses

```
>> T3 = T1*T2
T3 =
    0.8660   -0.5000    2.2321
    0.5000    0.8660    3.8660
         0         0    1.0000
```

and plot it, in green, as

```
>> trplot2(T3, 'frame', '3', 'color', 'g');
```

We see that the displacement of (2, 1) has been applied with respect to frame {1}. It is important to note that our final displacement is not (3, 3) because the displacement is with respect to the rotated coordinate frame. The noncommutativity of composition is clearly demonstrated by

```
>> T4 = T2*T1;
>> trplot2(T4, 'frame', '4', 'color', 'c');
```

and we see that frame {4} is different to frame {3}.

Now we define a point (3, 2) relative to the world frame

```
>> P = [3 ; 2 ];
```

which is a column vector and add it to the plot

```
>> plot_point(P, 'label', 'P', 'solid', 'ko');
```

To determine the coordinate of the point with respect to {1} we use Eq. 2.1 and write down

$${}^0\boldsymbol{p} = {}^0\xi_1 \bullet {}^1\boldsymbol{p}$$

and then rearrange as

$$
\begin{aligned}
{}^{1}\boldsymbol{p} &= {}^{1}\xi_{0} \bullet {}^{0}\boldsymbol{p} \\
&= \left({}^{0}\xi_{1}\right)^{-1} \bullet {}^{0}\boldsymbol{p}
\end{aligned}
$$

Substituting numerical values

```
>> P1 = inv(T1) * [P; 1]
P1 =
    1.7321
   -1.0000
    1.0000
```

where we first converted the Euclidean point coordinates to *homogeneous form* by appending a one. The result is also in homogeneous form and has a negative y-coordinate in frame {1}. Using the Toolbox we could also have expressed this as

```
>> h2e( inv(T1) * e2h(P) )
ans =
    1.7321
   -1.0000
```

where the result is in Euclidean coordinates. The helper function `e2h` converts Euclidean coordinates to homogeneous and `h2e` performs the inverse conversion.

2.1.2.2 Centers of Rotation

We will explore the noncommutativity property in more depth and illustrate with the example of a pure rotation. First we create and plot a reference coordinate frame {0} and a target frame {X}

```
>> plotvol([-5 4 -1 5]);
>> T0 = eye(3,3);
>> trplot2(T0, 'frame', '0');
>> X = transl2(2, 3);
>> trplot2(X, 'frame', 'X');
```

and create a rotation of 2 radians (approximately 115°)

```
>> R = trot2(2);
```

and plot the effect of the two possible orders of composition

```
>> trplot2(R*X, 'framelabel', 'RX', 'color', 'r');
>> trplot2(X*R, 'framelabel', 'XR', 'color', 'r');
```

The results are shown as red coordinate frames in Fig. 2.9. We see that the frame {RX} has been rotated about the origin, while frame {XR} has been rotated about the origin of {X}.

What if we wished to rotate a coordinate frame about an arbitrary point? First of all we will establish a new point **C** and display it

```
>> C = [1 2]';
>> plot_point(C, 'label', ' C', 'solid', 'ko')
```

and then compute a transform to rotate about point **C**

```
>> RC = transl2(C) * R * transl2(-C)
RC =
   -0.4161   -0.9093    3.2347
    0.9093   -0.4161    1.9230
         0         0    1.0000
```

and applying this

```
>> trplot2(RC*X, 'framelabel', 'XC', 'color', 'r');
```

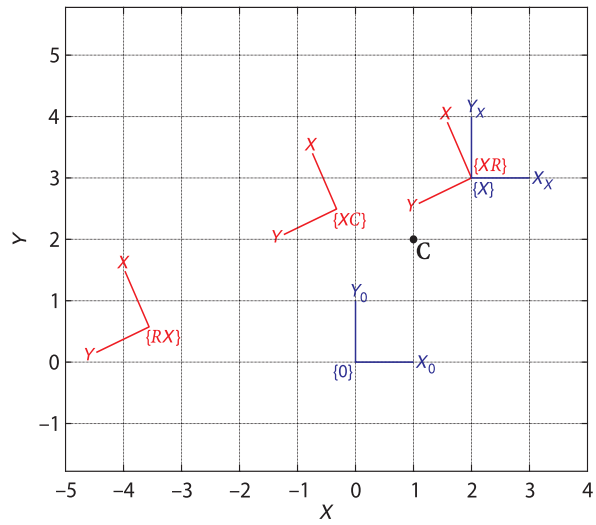


Fig. 2.9.
The frame $\{X\}$ is rotated by 2 radians about $\{0\}$ to give frame $\{RX\}$, about $\{X\}$ to give $\{XR\}$, and about point **C** to give frame $\{XC\}$

we see that the frame has indeed been rotated about point **C**. Creating the required transform was somewhat cumbersome and not immediately obvious. Reading from right to left▸ we first apply an origin shift, a translation from **C** to the origin of the reference frame, apply the rotation about that origin, and then apply the inverse origin shift, a translation from the reference frame origin back to **C**. A more descriptive way to achieve this is using twists.

`RC` left multiplies `X`, therefore the first transform applied to `X` is `transl(-C)`, then `R`, then `transl(C)`.

2.1.2.3 Twists in 2D

The corollary to what we showed in the last section is that, given any two frames we can find a rotational center that will *rotate* the first frame into the second. For the case of pure translational motion the rotational center will be at infinity. This is the key concept behind what is called a twist.

We can create a rotational twist about the point specified by the coordinate vector `C`

```
>> tw = Twist('R', C)
tw =
( 2 -1; 1 )
```

and the result is a `Twist` object that encodes a twist vector with two components: a 2-vector *moment* and a 1-vector *rotation*. The first argument `'R'` indicates a rotational twist is to be computed. This particular twist is a *unit twist* since the magnitude of the rotation, the last element of the twist, is equal to one.

To create an **SE**(2) transformation for a rotation about this unit twist by 2 radians we use the `T` method

```
>> tw.T(2)
ans =
   -0.4161   -0.9093    3.2347
    0.9093   -0.4161    1.9230
         0         0    1.0000
```

which is the same as that computed in the previous section, but more concisely specified in terms of the center of rotation. The center is also called the pole of the transformation and is encoded in the twist

```
>> tw.pole'
ans =
     1     2
```

For a unit-translational twist the rotation is zero and the moment is a unit vector.

If we wish to perform translational motion in the direction (1, 1) the relevant unit twist is

```
>> tw = Twist('T', [1 1])
tw =
( 0.70711 0.70711; 0 )
```

and for a displacement of $\sqrt{2}$ in the direction defined by this twist the SE(2) transformation is

```
>> tw.T(sqrt(2))
ans =
     1     0     1
     0     1     1
     0     0     1
```

which we see has a null rotation and a translation of 1 in the x- and y-directions.

For an arbitrary planar transform such as

```
>> T = transl2(2, 3) * trot2(0.5)
T =
    0.8776   -0.4794    2.0000
    0.4794    0.8776    3.0000
         0         0    1.0000
```

we can compute the twist vector

```
>> tw = Twist(T)
tw =
( 2.7082 2.4372; 0.5 )
```

and we note that the last element, the rotation, is not equal to one but is the required rotation angle of 0.5 radians. This is a nonunit twist. Therefore when we convert this to an SE(2) transform we don't need to provide a second argument since it is implicit in the twist

```
>>  tw.T
ans =
    0.8776   -0.4794    2.0000
    0.4794    0.8776    3.0000
         0         0    1.0000
```

and we have regenerated our original homogeneous transformation.

2.2 Working in Three Dimensions (3D)

The 3-dimensional case is an extension of the 2-dimensional case discussed in the previous section. We add an extra coordinate axis, typically denoted by z, that is orthogonal to both the x- and y-axes. The direction of the z-axis obeys the *right-hand rule* and forms a *right-handed coordinate frame*. Unit vectors parallel to the axes are denoted $\hat{\boldsymbol{x}}$, $\hat{\boldsymbol{y}}$ and $\hat{\boldsymbol{z}}$ such that

In all these identities, the symbols from left to right (across the equals sign) are a cyclic rotation of the sequence *xyz*.

$$\hat{\boldsymbol{z}} = \hat{\boldsymbol{x}} \times \hat{\boldsymbol{y}}, \quad \hat{\boldsymbol{x}} = \hat{\boldsymbol{y}} \times \hat{\boldsymbol{z}}; \quad \hat{\boldsymbol{y}} = \hat{\boldsymbol{z}} \times \hat{\boldsymbol{x}} \tag{2.12}$$

A point **P** is represented by its x-, y- and z-coordinates (x, y, z) or as a bound vector

$$\boldsymbol{p} = x\hat{\boldsymbol{x}} + y\hat{\boldsymbol{y}} + z\hat{\boldsymbol{z}}$$

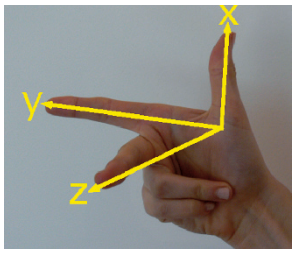


Figure 2.10 shows a red coordinate frame $\{B\}$ that we wish to describe with respect to the blue reference frame $\{A\}$. We can see clearly that the origin of $\{B\}$ has been

Right-hand rule. A right-handed coordinate frame is defined by the first three fingers of your right hand which indicate the relative directions of the x-, y- and z-axes respectively.

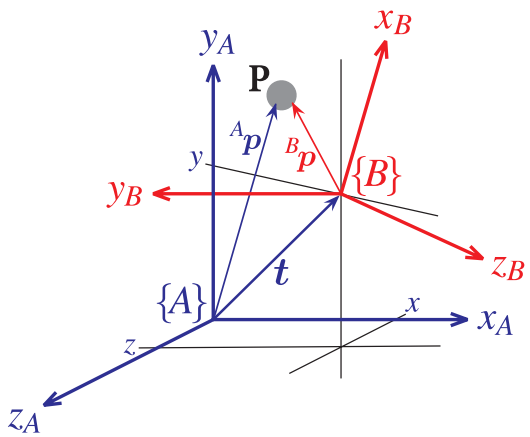


Fig. 2.10.
Two 3D coordinate frames $\{A\}$ and $\{B\}$. $\{B\}$ is rotated and translated with respect to $\{A\}$

displaced by the vector $\boldsymbol{t} = (x, y, z)$ and then rotated in some complex fashion. Just as for the 2-dimensional case the way we represent orientation is very important.

Our approach is to again consider an arbitrary point **P** with respect to each of the coordinate frames and to determine the relationship between ${}^A\boldsymbol{p}$ and ${}^B\boldsymbol{p}$. We will again consider the problem in two parts: rotation and then translation. Rotation is surprisingly complex for the 3-dimensional case and we devote all of the next section to it.

2.2.1 Orientation in 3-Dimensions

Any two independent orthonormal coordinate frames
can be related by a sequence of rotations (not more than ***three****)*
about coordinate axes, where no two successive rotations may be about the same axis.
Euler's rotation theorem (Kuipers 1999).

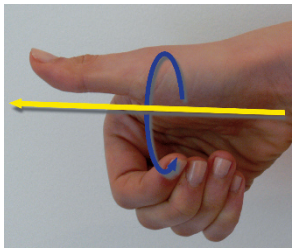
Figure 2.10 shows a pair of right-handed coordinate frames with very different orientations, and we would like some way to describe the orientation of one with respect to the other. We can imagine picking up frame $\{A\}$ in our hand and rotating it until it looked just like frame $\{B\}$. *Euler's rotation theorem* states that any rotation can be considered as a sequence of rotations about different coordinate axes.

We start by considering rotation about a single coordinate axis. Figure 2.11 shows a right-handed coordinate frame, and that same frame after it has been rotated by various angles about different coordinate axes.

The issue of rotation has some subtleties which are illustrated in Fig. 2.12. This shows a sequence of two rotations applied in different orders. We see that the final orientation depends on the order in which the rotations are applied. This is a deep and confounding characteristic of the 3-dimensional world which has intrigued mathematicians for a long time. There are implication for the pose algebra we have used in this chapter:

> In 3-dimensions rotation is not commutative – the order in which rotations are applied makes a difference to the result.

Mathematicians have developed many ways to represent rotation and we will discuss several of them in the remainder of this section: orthonormal rotation matrices, Euler and Cardan angles, rotation axis and angle, exponential coordinates, and unit quaternions. All can be represented as vectors or matrices, the natural datatypes of MATLAB or as a Toolbox defined class. The Toolbox provides many function to convert between these representations and these are shown in Tables 2.1 and 2.2 (pages 57, 58).



Rotation about a vector. Wrap your right hand around the vector with your thumb (your *x*-finger) in the direction of the arrow. The curl of your fingers indicates the direction of increasing angle.

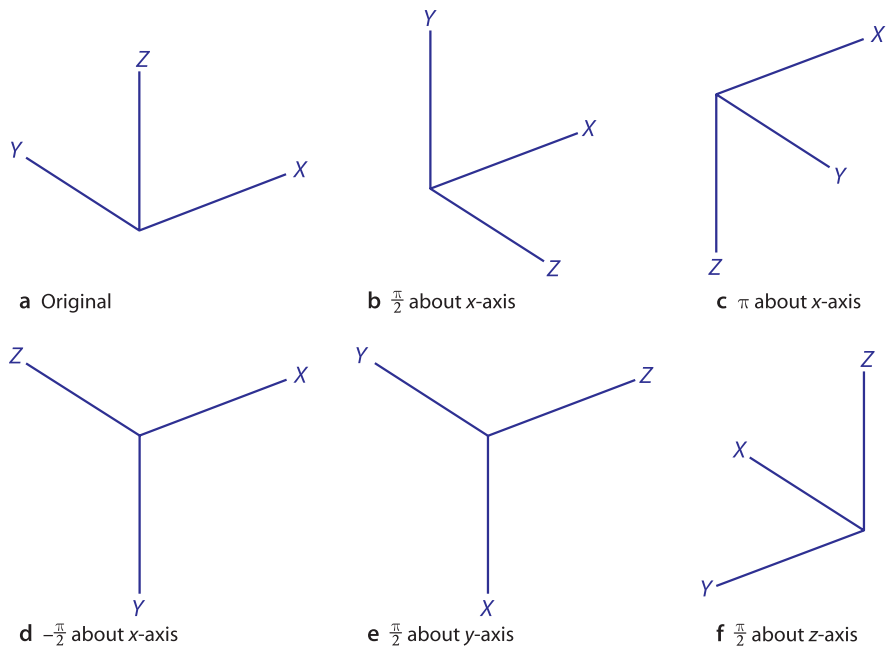


Fig. 2.11. Rotation of a 3D coordinate frame. **a** The original coordinate frame, **b–f** frame **a** after various rotations as indicated

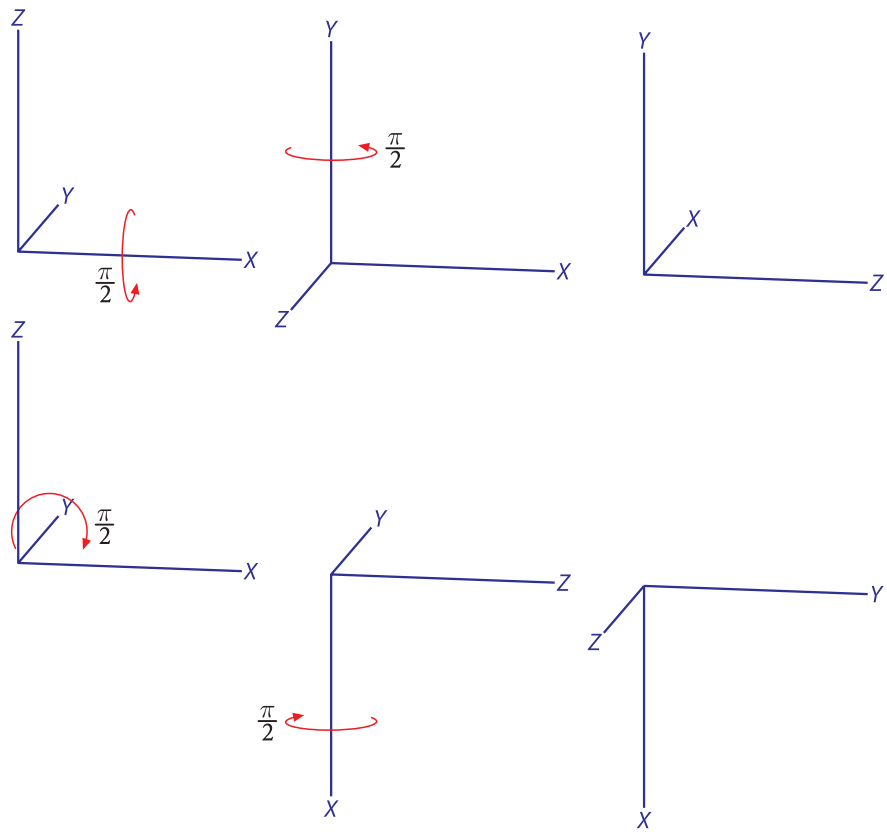


Fig. 2.12. Example showing the noncommutativity of rotation. In the top row the coordinate frame is rotated by $\frac{\pi}{2}$ about the *x*-axis and then $\frac{\pi}{2}$ about the *y*-axis. In the bottom row the order of rotations has been reversed. The results are clearly different

2.2.1.1 Orthonormal Rotation Matrix

Just as for the 2-dimensional case we can represent the orientation of a coordinate frame by its unit vectors expressed in terms of the reference coordinate frame. Each unit vector has three elements and they form the columns of a 3×3 *orthonormal matrix* ${}^{A}\boldsymbol{R}_{B}$

$$\begin{pmatrix} {}^{A}x \\ {}^{A}y \\ {}^{A}z \end{pmatrix} = {}^{A}\boldsymbol{R}_{B} \begin{pmatrix} {}^{B}x \\ {}^{B}y \\ {}^{B}z \end{pmatrix} \tag{2.13}$$

which transforms the description of a vector defined with respect to frame $\{B\}$ to a vector with respect to $\{A\}$.

A 3-dimensional rotation matrix ${}^{X}\boldsymbol{R}_{Y}$ has some special properties:

- it is *orthonormal* (also called *orthogonal*) since each of its columns is a unit vector and the columns are orthogonal.
- the columns are the unit vectors that define the axes of the rotated frame Y with respect to X and are by definition both unit-length and orthogonal.
- it belongs to the special orthogonal group of dimension 3 or $\boldsymbol{R} \in \mathbf{SO}(3) \subset \mathbb{R}^{3\times3}$. This means that the product of any two matrices within the group also belongs to the group, as does its inverse.
- its determinant is $+1$, which means that the length of a vector is unchanged after transformation, that is, $\|{}^{Y}\boldsymbol{p}\| = \|{}^{X}\boldsymbol{p}\|, \forall\theta$.
- the inverse is the same as the transpose, that is, $\boldsymbol{R}^{-1} = \boldsymbol{R}^{T}$.

See Appendix B which provides a refresher on vectors, matrices and linear algebra.

The orthonormal rotation matrices for rotation of θ about the x-, y- and z-axes are

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}$$

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}$$

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The Toolbox provides functions to compute these elementary rotation matrices, for example $\boldsymbol{R}_x(\theta)$ is

```
>> R = rotx(pi/2)
R =
    1.0000         0         0
         0    0.0000   -1.0000
         0    1.0000    0.0000
```

and its effect on a reference coordinate frame is shown graphically in Fig. 2.11b. The functions `roty` and `rotz` compute $\boldsymbol{R}_y(\theta)$ and $\boldsymbol{R}_z(\theta)$ respectively.

If we consider that the rotation matrix represents a pose then the corresponding coordinate frame can be displayed graphically

```
>> trplot(R)
```

which is shown in Fig. 2.13a. We can visualize a rotation more powerfully using the Toolbox function `tranimate` which animates a rotation

```
>> tranimate(R)
```

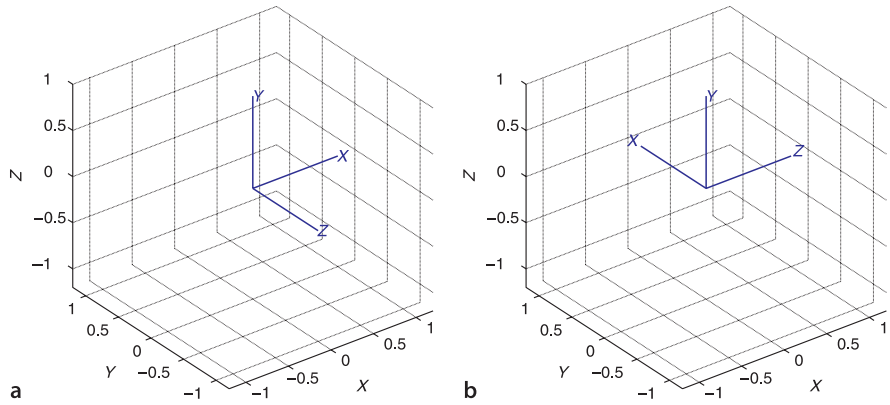
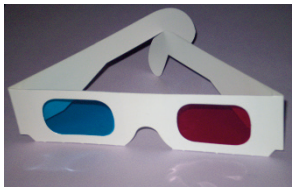


Fig. 2.13. Coordinate frames displayed using `trplot`. **a** Reference frame rotated by $\frac{\pi}{2}$ about the x-axis, **b** frame **a** rotated by $\frac{\pi}{2}$ about the y-axis



showing the world frame rotating into the specified coordinate frame. If you have a pair of anaglyph stereo glasses◀ you can see this in more realistic 3D by

```
>> tranimate(R, '3d')
```

To illustrate compounding of rotations we will rotate the frame of Fig. 2.13a again, this time around its y-axis

```
>> R = rotx(pi/2) * roty(pi/2)
R =
    0.0000         0    1.0000
    1.0000    0.0000   -0.0000
   -0.0000    1.0000    0.0000
>> trplot(R)
```

to give the frame shown in Fig. 2.13b. In this frame the x-axis now points in the direction of the world y-axis.

The noncommutativity of rotation can be shown by reversing the order of the rotations above

```
>> roty(pi/2)*rotx(pi/2)
ans =
    0.0000    1.0000    0.0000
         0    0.0000   -1.0000
   -1.0000    0.0000    0.0000
```

which has a very different value.

We recall that Euler's rotation theorem states that *any* rotation can be represented by *not more than three* rotations about coordinate axes. This means that in general an arbitrary rotation between frames can be decomposed into a sequence of three rotation angles and associated rotation axes – this is discussed in the next section.

The orthonormal rotation matrix has nine elements but they are not independent. The columns have unit magnitude which provides three constraints. The columns are orthogonal to each other which provides another three constraints.◀ Nine elements and six constraints is effectively three independent values.

If the column vectors are $\boldsymbol{c}_i, i \in 1 \cdots 3$ then $\boldsymbol{c}_1 \cdot \boldsymbol{c}_2 = \boldsymbol{c}_2 \cdot \boldsymbol{c}_3 = \boldsymbol{c}_3 \cdot \boldsymbol{c}_1 = 0$ and $\|\boldsymbol{c}_i\| = 1$.

Reading an orthonormal *rotation matrix*, the columns from left to right tell us the directions of the new frame's axes in terms of the current coordinate frame. For example if

```
R =
    1.0000         0         0
         0    0.0000   -1.0000
         0    1.0000    0.0000
```

the new frame has its x-axis in the old x-direction $(1, 0, 0)$, its y-axis in the old z-direction $(0, 0, 1)$, and the new z-axis in the old negative y-direction $(0, -1, 0)$. In this case the x-axis was unchanged since this is the axis around which the rotation occurred. The rows are the converse – the current frame axes in terms of the new frame axes.

2.2.1.2 Three-Angle Representations

Euler's rotation theorem requires successive rotation about three axes such that no two successive rotations are about the same axis. There are two classes of rotation sequence: Eulerian and Cardanian, named after Euler and Cardano respectively.

The Eulerian type involves repetition, but not successive, of rotations about one particular axis: XYX, XZX, YXY, YZY, ZXZ, or ZYZ. The Cardanian type is characterized by rotations about all three axes: XYZ, XZY, YZX, YXZ, ZXY, or ZYX.

> It is common practice to refer to all 3-angle representations as Euler angles but this is underspecified since there are twelve different types to choose from. The particular angle sequence is often a convention within a particular technological field.

The ZYZ sequence

$$\boldsymbol{R} = \boldsymbol{R}_z(\phi)\boldsymbol{R}_y(\theta)\boldsymbol{R}_z(\psi) \tag{2.14}$$

is commonly used in aeronautics and mechanical dynamics, and is used in the Toolbox. The Euler angles are the 3-vector $\boldsymbol{\Gamma} = (\phi, \theta, \psi)$.

For example, to compute the equivalent rotation matrix for $\boldsymbol{\Gamma} = (0.1, 0.2, 0.3)$ we write

```
>> R = rotz(0.1) * roty(0.2) * rotz(0.3);
```

or more conveniently

```
>> R = eul2r(0.1, 0.2, 0.3)
R =
    0.9021   -0.3836    0.1977
    0.3875    0.9216    0.0198
   -0.1898    0.0587    0.9801
```

The inverse problem is finding the Euler angles that correspond to a given rotation matrix

```
>> gamma = tr2eul(R)
gamma =
    0.1000    0.2000    0.3000
```

However if θ is negative

```
>> R = eul2r(0.1 , -0.2, 0.3)
R =
    0.9021   -0.3836   -0.1977
    0.3875    0.9216   -0.0198
    0.1898   -0.0587    0.9801
```

the inverse function

```
>> tr2eul(R)
ans =
   -3.0416    0.2000   -2.8416
```

returns a positive value for θ and quite different values for ϕ and ψ. However the corresponding rotation matrix

Leonhard Euler (1707–1783) was a Swiss mathematician and physicist who dominated eighteenth century mathematics. He was a student of Johann Bernoulli and applied new mathematical techniques such as calculus to many problems in mechanics and optics. He also developed the functional notation, $y = f(x)$, that we use today. In robotics we use his rotation theorem and his equations of motion in rotational dynamics.

He was prolific and his collected works fill 75 volumes. Almost half of this was produced during the last seventeen years of his life when he was completely blind.

```
>> eul2r(ans)
ans =
    0.9021   -0.3836   -0.1977
    0.3875    0.9216   -0.0198
    0.1898   -0.0587    0.9801
```

is the same – the two different sets of Euler angles correspond to the one rotation matrix. The mapping from a rotation matrix to Euler angles is not unique and the Toolbox *always* returns a positive angle for θ.

For the case where $\theta = 0$

```
>> R = eul2r(0.1, 0, 0.3)
R =
    0.9211   -0.3894         0
    0.3894    0.9211         0
         0         0    1.0000
```

the inverse function returns

```
>> tr2eul(R)
ans =
         0         0    0.4000
```

which is clearly quite different but the result is the same rotation matrix. The explanation is that if $\theta = 0$ then $\boldsymbol{R}_y = \boldsymbol{I}$ and Eq. 2.14 becomes

$$\boldsymbol{R} = \boldsymbol{R}_z(\phi)\boldsymbol{R}_z(\psi) = \boldsymbol{R}_z(\phi + \psi)$$

which is a function of the sum $\phi + \psi$. Therefore the inverse operation can do no more than determine this sum, and by convention we choose $\phi = 0$. The case $\theta = 0$ is a singularity and will be discussed in more detail in the next section.

Another widely used convention are the Cardan angles: roll, pitch and yaw. Confusingly there are two different versions in common use. Text books seem to define the roll-pitch-yaw sequence as ZYX or XYZ depending on whether they have a mobile robot or robot arm focus.◀ When describing the attitude of vehicles such as ships, aircraft and cars the convention is that the x-axis points in the forward direction and the z-axis points either up or down. It is intuitive to apply the rotations in the sequence: yaw (direction of travel), pitch (elevation of the front with respect to horizontal) and then finally roll (rotation about the forward axis of the vehicle). This leads to the ZYX angle sequence

Well known texts such as Siciliano et al. (2008), Spong et al. (2006) and Paul (1981) use the XYZ sequence. The Toolbox supports both formats by means of the `'xyz'` and `'zyx'` options. The ZYX order is default for Release 10, but for Release 9 the default was XYZ.

$$\boldsymbol{R} = \boldsymbol{R}_z\left(\theta_y\right)\boldsymbol{R}_y\left(\theta_p\right)\boldsymbol{R}_x\left(\theta_r\right) \tag{2.15}$$

Roll-pitch-yaw angles are also known as Tait-Bryan angles◀ or nautical angles, and for aeronautical applications they can be called bank, attitude and heading angles respectively.

Named after Peter Tait a Scottish physicist and quaternion supporter, and George Bryan an early Welsh aerodynamicist.

Gerolamo Cardano (1501–1576) was an Italian Renaissance mathematician, physician, astrologer, and gambler. He was born in Pavia, Italy, the illegitimate child of a mathematically gifted lawyer. He studied medicine at the University of Padua and later was the first to describe typhoid fever. He partly supported himself through gambling and his book about games of chance *Liber de ludo aleae* contains the first systematic treatment of probability as well as effective cheating methods. His family life was problematic: his eldest son was executed for poisoning his wife, and his daughter was a prostitute who died from syphilis (about which he wrote a treatise). He computed and published the horoscope of Jesus, was accused of heresy, and spent time in prison until he abjured and gave up his professorship.

He published the solutions to the cubic and quartic equations in his book *Ars magna* in 1545, and also invented the combination lock, the gimbal consisting of three concentric rings allowing a compass or gyroscope to rotate freely (see Fig. 2.15), and the Cardan shaft with universal joints – the drive shaft used in motor vehicles today.

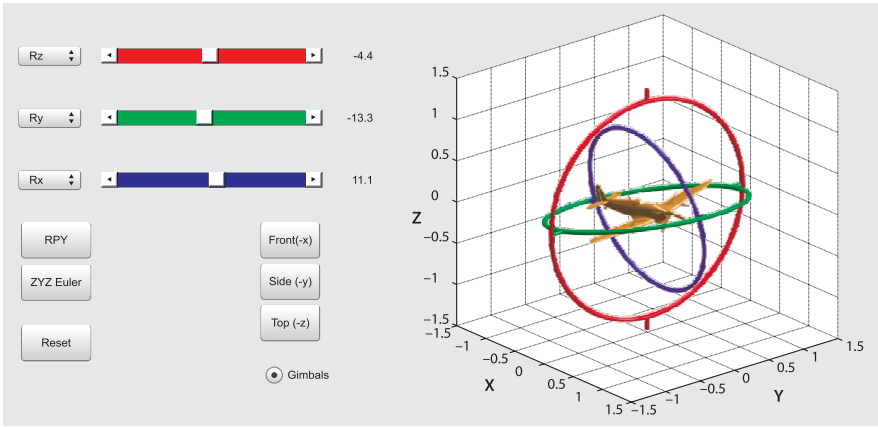


Fig. 2.14.
The Toolbox application `tripleangle` allows you to experiment with Euler angles and roll-pitch-yaw angles and see how the attitude of a body changes

When describing the attitude of a robot gripper, as shown in Fig. 2.16, the convention is that the z-axis points forward and the x-axis is either up or down. This leads to the XYZ angle sequence

$$\boldsymbol{R} = \boldsymbol{R}_x(\theta_y)\boldsymbol{R}_y(\theta_p)\boldsymbol{R}_z(\theta_r) \tag{2.16}$$

The Toolbox defaults to the ZYX sequence but can be overridden using the `'xyz'` option. For example

```
>> R = rpy2r(0.1, 0.2, 0.3)
R =
    0.9363   -0.2751    0.2184
    0.2896    0.9564   -0.0370
   -0.1987    0.0978    0.9752
```

and the inverse is

```
>> gamma = tr2rpy(R)
gamma =
    0.1000    0.2000    0.3000
```

The roll-pitch-yaw sequence allows all angles to have arbitrary sign and it has a singularity when $\theta_p = \pm\frac{\pi}{2}$ which is fortunately outside the range of feasible attitudes for most vehicles.

The Toolbox includes an interactive graphical tool

```
>> tripleangle
```

that allows you to experiment with Euler angles or roll-pitch-yaw angles and see their effect on the orientation of a body as shown in Fig. 2.14.

2.2.1.3 Singularities and Gimbal Lock

A fundamental problem with all the three-angle representations just described is singularity. This is also known as gimbal lock, a term made famous in the movie Apollo 13. This occurs when the rotational axis of the middle term in the sequence becomes parallel to the rotation axis of the first or third term.

A mechanical gyroscope used for spacecraft navigation is shown in Fig. 2.15. The innermost assembly is the *stable member* which has three orthogonal gyroscopes that hold it at a constant orientation with respect to the universe. It is mechanically connected to the spacecraft via a gimbal mechanism which allows the spacecraft to move around the stable platform without exerting any torque on it. The attitude of the spacecraft is determined directly by measuring the angles of the gimbal axes with respect to the stable platform – giving a direct indication of roll-pitch-yaw angles which in this design are a Cardanian YZX sequence.▶

"The LM Body coordinate system is right-handed, with the +X axis pointing up through the thrust axis, the +Y axis pointing right when facing forward which is along the +Z axis. The rotational transformation matrix is constructed by a 2-3-1 Euler sequence, that is: Pitch about Y, then Roll about Z and, finally, Yaw about X. Positive rotations are pitch up, roll right, yaw left." (Hoag 1963).

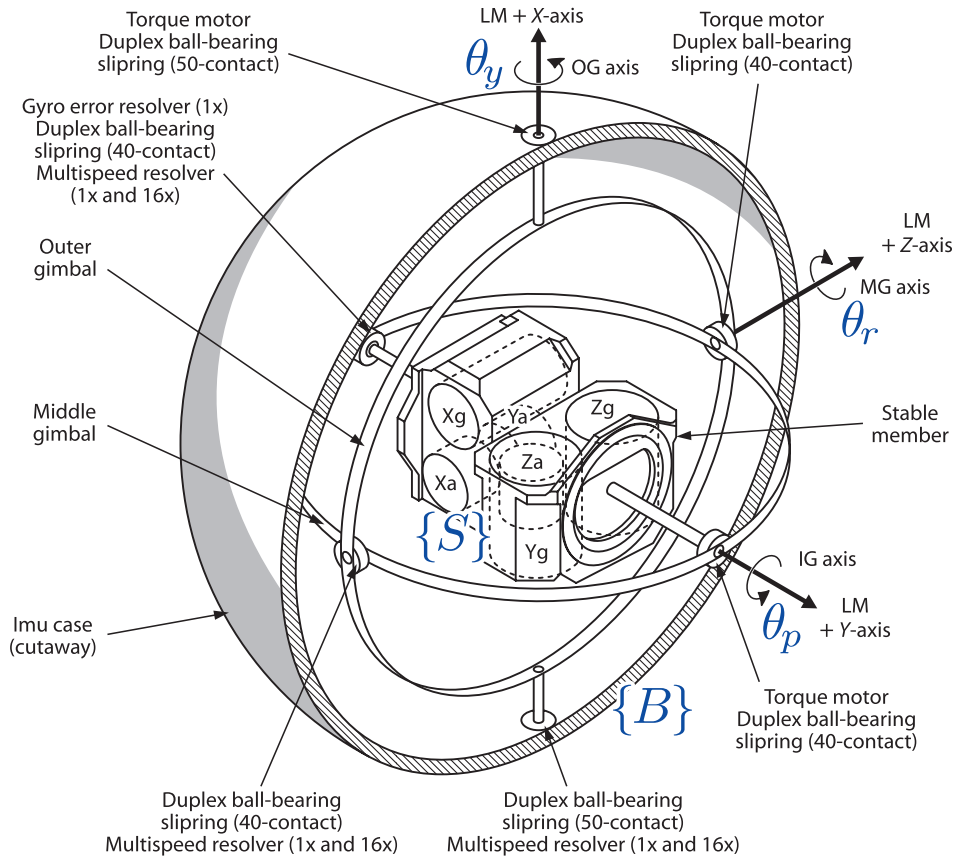


Fig. 2.15. Schematic of Apollo Lunar Module (LM) inertial measurement unit (IMU). The vehicle's coordinate system has the x-axis pointing up through the thrust axis, the z-axis forward, and the y-axis pointing right. Starting at the stable platform $\{S\}$ and working outwards toward the spacecraft's body frame $\{B\}$ the rotation angle sequence is YZX. The components labeled X_g, Y_g and Z_g are the x-, y- and z-axis gyroscopes and those labeled X_a, Y_a and Z_a are the x-, y- and z-axis accelerometers (redrawn after Apollo Operations Handbook, LMA790-3-LM)

Consider the situation when the rotation angle of the middle gimbal (rotation about the spacecraft's z-axis) is 90° – the axes of the inner and outer gimbals are aligned and they share the *same* rotation axis. Instead of the original three rotational axes, since two are parallel, there are now only two effective rotational axes – we say that one degree of freedom has been lost.◀

Operationally this was a significant limiting factor with this particular gyroscope (Hoag 1963) and could have been alleviated by adding a fourth gimbal, as was used on other spacecraft. It was omitted on the Lunar Module for reasons of weight and space.

In mathematical, rather than mechanical, terms this problem can be seen using the definition of the Lunar module's coordinate system where the rotation of the spacecraft's body-fixed frame $\{B\}$ with respect to the stable platform frame $\{S\}$ is

$$ {}^{S}\boldsymbol{R}_B = \boldsymbol{R}_y\left(\theta_p\right)\boldsymbol{R}_z\left(\theta_r\right)\boldsymbol{R}_x\left(\theta_y\right) $$

For the case when $\theta_r = \frac{\pi}{2}$ we can apply the identity◀

$$ \boldsymbol{R}_y(\theta)\boldsymbol{R}_z\left(\tfrac{\pi}{2}\right) \equiv \boldsymbol{R}_z\left(\tfrac{\pi}{2}\right)\boldsymbol{R}_x(\theta) $$

leading to

$$ {}^{S}\boldsymbol{R}_B = \boldsymbol{R}_z\left(\tfrac{\pi}{2}\right)\boldsymbol{R}_x\left(\theta_p\right)\boldsymbol{R}_x\left(\theta_y\right) = \boldsymbol{R}_z\left(\tfrac{\pi}{2}\right)\boldsymbol{R}_x\left(\theta_p + \theta_y\right) $$

which is unable to represent any rotation about the y-axis. This is not a good thing because spacecraft rotation about the y-axis would rotate the stable element and thus ruin its precise alignment with the stars: hence the anxiety on Apollo 13.

The loss of a degree of freedom means that mathematically we cannot invert the transformation, we can only establish a linear relationship between two of the angles. In this case the best we can do is determine the sum of the pitch and yaw angles. We observed a similar phenomena with the Euler angle singularity earlier.

Rotations obey the cyclic rotation rules
$Rx(\frac{\pi}{2})\,Ry(\theta)\,Rx(\frac{\pi}{2})^T \equiv Rz(\theta)$
$Ry(\frac{\pi}{2})\,Rz(\theta)\,Ry(\frac{\pi}{2})^T \equiv Rx(\theta)$
$Rz(\frac{\pi}{2})\,Rx(\theta)\,Rz(\frac{\pi}{2})^T \equiv Ry(\theta)$
and anti-cyclic rotation rules
$Ry(\frac{\pi}{2})^T\,Rx(\theta)\,Ry(\frac{\pi}{2}) \equiv Rz(\theta)$
$Rz(\frac{\pi}{2})^T\,Ry(\theta)\,Rz(\frac{\pi}{2}) \equiv Rx(\theta)$.

Apollo 13 mission clock: 02 08 12 47

- **Flight:** "Go, Guidance."
- **Guido:** "He's getting close to gimbal lock there."
- **Flight:** "Roger. CapCom, recommend he bring up C3, C4, B3, B4, C1 and C2 thrusters, and advise he's getting close to gimbal lock."
- **CapCom:** "Roger."

Apollo 13, mission control communications loop (1970) (Lovell and Kluger 1994, p 131; NASA 1970).

All three-angle representations of attitude, whether Eulerian or Cardanian, suffer this problem of *gimbal lock* when two consecutive axes become aligned. For ZYZ-Euler angles this occurs when $\theta = k\pi$, $k \in \mathbb{Z}$ and for roll-pitch-yaw angles when pitch $\theta_p = \pm(2k+1)\frac{\pi}{2}$. The best that can be hoped for is that the singularity occurs for an attitude which does not occur during normal operation of the vehicle – it requires judicious choice of angle sequence and coordinate system.

Singularities are an unfortunate consequence of using a minimal representation. To eliminate this problem we need to adopt different representations of orientation. Many in the Apollo LM team would have preferred a four gimbal system and the clue to success, as we shall see shortly in Sect. 2.2.1.7, is to introduce a fourth parameter.

2.2.1.4 Two Vector Representation

For arm-type robots it is useful to consider a coordinate frame $\{E\}$ attached to the end-effector as shown in Fig. 2.16. By convention the axis of the tool is associated with the z-axis and is called the *approach vector* and denoted $\hat{\boldsymbol{a}} = (a_x, a_y, a_z)$. For some applications it is more convenient to specify the approach vector than to specify Euler or roll-pitch-yaw angles.

However specifying the direction of the z-axis is insufficient to describe the coordinate frame – we also need to specify the direction of the x- and y-axes. An orthogonal vector that provides orientation, perhaps between the two fingers of the robot's gripper is called the *orientation vector*, $\hat{\boldsymbol{o}} = (o_x, o_y, o_z)$. These two unit vectors are sufficient to completely define the rotation matrix

$$\boldsymbol{R} = \begin{pmatrix} n_x & o_x & a_x \\ n_y & o_y & a_y \\ n_z & o_z & a_z \end{pmatrix} \tag{2.17}$$

since the remaining column, the normal vector, can be computed using Eq. 2.12 as $\hat{\boldsymbol{n}} = \hat{\boldsymbol{o}} \times \hat{\boldsymbol{a}}$. Consider an example where the gripper's approach and orientation vectors are parallel to the world x- and y-directions respectively. Using the Toolbox this is implemented by

```
>> a = [1 0 0]';
>> o = [0 1 0]';
>> R = oa2r(o, a)
R =
     0     0     1
     0     1     0
    -1     0     0
```

Any two nonparallel vectors are sufficient to define a coordinate frame. Even if the two vectors $\hat{\boldsymbol{a}}$ and $\hat{\boldsymbol{o}}$ are not orthogonal they still define a plane and the computed $\hat{\boldsymbol{n}}$ is normal to that plane. In this case we need to compute a new value for $\hat{\boldsymbol{o}}' = \hat{\boldsymbol{a}} \times \hat{\boldsymbol{n}}$ which lies in the plane but is orthogonal to each of $\hat{\boldsymbol{a}}$ and $\hat{\boldsymbol{n}}$.

For a camera we might use the optical axis, by convention the z-axis, and the left side of the camera which is by convention the x-axis. For a mobile robot we might use

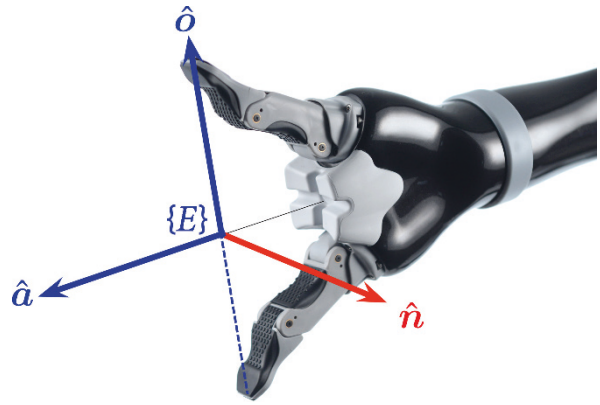


Fig. 2.16. Robot end-effector coordinate system defines the pose in terms of an *approach* vector $\hat{\boldsymbol{a}}$ and an *orientation* vector $\hat{\boldsymbol{o}}$, from which $\hat{\boldsymbol{n}}$ can be computed. $\hat{\boldsymbol{n}}$, $\hat{\boldsymbol{o}}$ and $\hat{\boldsymbol{a}}$ vectors correspond to the x-, y- and z-axes respectively of the end-effector coordinate frame. (courtesy of Kinova Robotics)

the gravitational acceleration vector (measured with accelerometers) which is by convention the z-axis and the heading direction (measured with an electronic compass) which is by convention the x-axis.

2.2.1.5 Rotation about an Arbitrary Vector

Two coordinate frames of arbitrary orientation are related by a *single* rotation about some axis in space. For the example rotation used earlier

```
>> R = rpy2r(0.1 , 0.2, 0.3);
```

we can determine such an angle and vector by

```
>> [theta, v] = tr2angvec(R)
th =
    0.3655
v =
    0.1886    0.5834    0.7900
```

This is not unique. A rotation of `-theta` about the vector `-v` results in the same orientation.

where `theta` is the angle of rotation and `v` is the vector◂ around which the rotation occurs.

This information is encoded in the eigenvalues and eigenvectors of $\boldsymbol{R}$. Using the built-in MATLAB function `eig`

```
>> [x,e] = eig(R)
x =
  -0.6944 + 0.0000i  -0.6944 + 0.0000i   0.1886 + 0.0000i
   0.0792 + 0.5688i   0.0792 - 0.5688i   0.5834 + 0.0000i
   0.1073 - 0.4200i   0.1073 + 0.4200i   0.7900 + 0.0000i
e =
   0.9339 + 0.3574i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.9339 - 0.3574i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i   1.0000 + 0.0000i
```

Both matrices are complex, but some elements are real (zero imaginary part).

the eigenvalues are returned on the diagonal of the matrix `e` and the corresponding eigenvectors are the corresponding columns of `x`.◂

From the definition of eigenvalues and eigenvectors we recall that

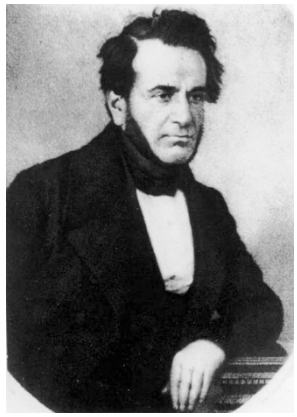
$$\boldsymbol{R}\boldsymbol{v} = \lambda \boldsymbol{v}$$

where $\boldsymbol{v}$ is the eigenvector corresponding to the eigenvalue λ. For the case $\lambda = 1$

$$\boldsymbol{R}\boldsymbol{v} = \boldsymbol{v}$$

which implies that the corresponding eigenvector $\boldsymbol{v}$ is *unchanged* by the rotation. There is only one such vector and that is the one *about which* the rotation occurs. In the example the third eigenvalue is equal to one, so the rotation axis is the third column of `x`.

Olinde Rodrigues (1795–1850) was a French banker and mathematician who wrote extensively on politics, social reform and banking. He received his doctorate in mathematics in 1816 from the University of Paris, for work on his first well known formula which is related to Legendre polynomials. His eponymous rotation formula was published in 1840 and is perhaps the first time the representation of a rotation as a scalar and a vector was articulated. His formula is sometimes, and inappropriately, referred to as the Euler-Rodrigues formula. He is buried in the Pere-Lachaise cemetery in Paris.



An orthonormal rotation matrix will always have one real eigenvalue at $\lambda = 1$ and in general a complex pair $\lambda = \cos\theta \pm i\sin\theta$ where θ is the rotation angle. The angle of rotation in this case is

```
>> theta = angle(e(1,1))
theta =
    0.3655
```

The inverse problem, converting from angle and vector to a rotation matrix, is achieved using Rodrigues' rotation formula

$$\boldsymbol{R} = \boldsymbol{I}_{3\times 3} + \sin\theta[\hat{\nu}]_\times + (1-\cos\theta)[\hat{\nu}]_\times^2 \tag{2.18}$$

where $[\hat{\nu}]_\times$ is a skew-symmetric matrix. We can use this formula to determine the rotation of $\frac{\pi}{2}$ about the x-axis

```
>> R = angvec2r(pi/2, [1 0 0])
R =
    1.0000         0         0
         0    0.0000   -1.0000
         0    1.0000    0.0000
```

It is interesting to note that this representation of an arbitrary rotation is parameterized by four numbers: three for the rotation axis, and one for the angle of rotation. This is far fewer than the nine numbers required by a rotation matrix. However the direction can be represented by a unit vector which has only two parameters and the angle can be encoded in the length to give a 3-parameter representation such as $\hat{\nu}\theta$, $\hat{\nu}\sin(\theta/2)$, $\hat{\nu}\tan(\theta)$ or the Rodrigues' vector $\hat{\nu}\tan(\theta/2)$. While these forms are minimal and efficient in terms of data storage they are analytically problematic and ill-defined when $\theta = 0$.

It can also be shown that the trace of a rotation matrix $tr(\boldsymbol{R}) = 1 + 2\cos\theta$ from which we can compute the magnitude of θ but not its sign.

Imagine a unit-sphere. All possible unit vectors from the center can be described by the latitude and longitude of the point at which they touch the surface of the sphere.

2.2.1.6 Matrix Exponentials

Consider an x-axis rotation expressed as a rotation matrix

```
>> R = rotx(0.3)
R =
    1.0000         0         0
         0    0.9553   -0.2955
         0    0.2955    0.9553
```

As we did for the 2-dimensional case we can compute the logarithm of this matrix using the MATLAB builtin function `logm`

```
>> S = logm(R)
S =
         0         0         0
         0    0.0000   -0.3000
         0    0.3000    0.0000
```

`logm` is different to the builtin function `log` which computes the logarithm of each element of the matrix. A logarithm can be computed using a power series, with a matrix rather than scalar argument. For a matrix the logarithm is not unique and `logm` computes the principal logarithm of the matrix.

and the result is a sparse matrix with two elements that have a magnitude of 0.3, which is the original rotation angle. This matrix has a zero diagonal and is another example of a skew-symmetric matrix, in this case 3×3.

Applying `vex` to the skew-symmetric matrix gives

```
>> vex(S)'
ans =
    0.3000         0         0
```

and we find the original rotation angle is in the first element, corresponding to the x-axis about which the rotation occurred. For the 3-dimensional case the Toolbox function `trlog` is equivalent◀

`trlog` uses a more efficient closed-form solution as well as being able to return the angle and axis information separately.

```
>> [th,w] = trlog(R)
th =
    0.3000
w =
    1.0000
         0
         0
```

The inverse of a logarithm is exponentiation and applying the builtin MATLAB matrix exponential function `expm`◀

`expm` is different to the builtin function `exp` which computes the exponential of each element of the matrix: $\text{expm}(A) = I + A + A^2/2! + A^3/3! + \cdots$

```
>> expm(S)
ans =
    1.0000         0         0
         0    0.9553   -0.2955
         0    0.2955    0.9553
```

we have regenerated our original rotation matrix. In fact the command

```
>> R = rotx(0.3);
```

is equivalent to

```
>> R = expm( skew([1 0 0]) * 0.3 );
```

where we have specified the rotation in terms of a rotation angle and a rotation axis (as a unit-vector). This generalizes to rotation about *any* axis and formally we can write

$$\boldsymbol{R} = \mathrm{e}^{[\hat{\omega}]_\times \theta} \in \mathbf{SO}(3)$$

where θ is the rotation angle, $\hat{\omega}$ is a unit-vector parallel to the rotation axis, and the notation $[\cdot]_\times : \mathbb{R}^3 \mapsto \mathbb{R}^{3\times 3}$ indicates a mapping from a vector to a skew-symmetric matrix. Since $[\boldsymbol{\omega}]_\times \theta = [\boldsymbol{\omega}\theta]_\times$ we can treat $\boldsymbol{\omega}\theta \in \mathbb{R}^3$ as a rotational parameter called exponential coordinates. For the 3-dimensional case, Rodrigues' rotation formula (Eq. 2.18) is a computationally efficient means of computing the matrix exponential for the special case where the argument is a skew-symmetric matrix, and this is used by the Toolbox function `trexp` which is equivalent to `expm`.

In 3-dimensions the skew-symmetric matrix has the form

$$[\boldsymbol{\omega}]_\times = \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix} \tag{2.19}$$

which has clear structure and only three unique elements $\boldsymbol{\omega} \in \mathbb{R}^3$. The matrix can be used to implement the vector cross product $\boldsymbol{v}_1 \times \boldsymbol{v}_2 = [\boldsymbol{v}_1]_\times \boldsymbol{v}_2$. A simple example of Toolbox support for skew-symmetric matrices is

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

and the inverse operation is performed using the Toolbox function `vex`

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

Both functions work for the 3D case, shown here, and the 2D case where the vector is a 1-vector.

2.2.1.7 Unit Quaternions

Quaternions came from Hamilton after his really good work had been done; and, though beautifully ingenious, have been an unmixed evil to those who have touched them in any way, including Clark Maxwell.
Lord Kelvin, 1892

Quaternions were discovered by Sir William Hamilton over 150 years ago and, while initially controversial, have great utility for robotics. The quaternion is an extension of the complex number – a hypercomplex number – and is written as a scalar plus a vector

$$\begin{aligned} \boldsymbol{q} &= s + \boldsymbol{v} \\ &= s + v_1 i + v_2 j + v_3 k \end{aligned} \tag{2.20}$$

where $s \in \mathbb{R}$, $\boldsymbol{v} \in \mathbb{R}^3$ and the orthogonal complex numbers i, j and k are defined such that

$$i^2 = j^2 = k^2 = ijk = -1 \tag{2.21}$$

and we denote a quaternion as

$$\boldsymbol{q} = s < v_1, v_2, v_3 >$$

In the Toolbox quaternions are implemented by the `Quaternion` class. Quaternions support addition and subtraction, performed element-wise, multiplication by a scalar and multiplication

$$\boldsymbol{q}_1 \circ \boldsymbol{q}_2 = s_1 s_2 - v_1 \cdot v_2 < s_1 v_2 + s_2 v_1 + v_1 \times v_2 >$$

which is known as the quaternion or Hamilton product.▸

If we write the quaternion as a 4-vector (s, v_1, v_2, v_2) then multiplication can be expressed as a matrix-vector product where

$$\mathring{\boldsymbol{q}} \circ \mathring{\boldsymbol{q}}' = \begin{pmatrix} s & -v_1 & -v_2 & -v_3 \\ v_1 & s & -v_3 & v_2 \\ v_2 & v_3 & s & -v_1 \\ v_3 & -v_2 & v_1 & s \end{pmatrix} \begin{pmatrix} s' \\ v_1' \\ v_2' \\ v_3' \end{pmatrix}$$

One early objection to quaternions was that multiplication was not commutative but as we have seen above this is exactly what we require for rotations. Despite the initial controversy quaternions are elegant, powerful and computationally straightforward and they are *widely* used for robotics, computer vision, computer graphics and aerospace navigation systems.

To represent rotations we use unit-quaternions denoted by $\mathring{\boldsymbol{q}}$. These are quaternions of unit magnitude; that is, those for which $\|\boldsymbol{q}\| = s^2 + v_1^2 + v_2^2 + v_3^2 = 1$. They can be considered as a rotation of θ about the unit vector $\hat{\boldsymbol{v}}$ which are related to the quaternion components by▸

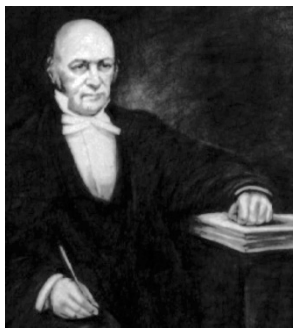
As for the angle-vector representation this is not unique. A rotation of θ about the vector $-v$ results in the same orientation. This is referred to as a double mapping or double cover.

Sir William Rowan Hamilton (1805–1865) was an Irish mathematician, physicist, and astronomer. He was a child prodigy with a gift for languages and by age thirteen knew classical and modern European languages as well as Persian, Arabic, Hindustani, Sanskrit, and Malay. Hamilton taught himself mathematics at age 17, and discovered an error in Laplace's Celestial Mechanics. He spent his life at Trinity College, Dublin, and was appointed Professor of Astronomy and Royal Astronomer of Ireland while still an undergraduate. In addition to quaternions he contributed to the development of optics, dynamics, and algebra. He also wrote poetry and corresponded with Wordsworth who advised him to devote his energy to mathematics.

According to legend the key quaternion equation, Eq. 2.21, occured to Hamilton in 1843 while walking along the Royal Canal in Dublin with his wife, and this is commemorated by a plaque on Broome bridge:

Here as he walked by on the 16th of October 1843 Sir William Rowan Hamilton in a flash of genius discovered the fundamental formula for quaternion multiplication $i^2 = j^2 = k^2 = i\,j\,k = -1$ & cut it on a stone of this bridge.

His original carving is no longer visible, but the bridge is a pilgrimage site for mathematicians and physicists.



For the case of unit quaternions our generalized pose is a rotation $\xi \sim \mathring{q} \in \mathbb{S}^3$ and

$$\mathring{q}_1 \oplus \mathring{q}_2 \mapsto \mathring{q}_1 \circ \mathring{q}_2$$

and

$$\ominus \mathring{q} \mapsto \mathring{q}^{-1} = s \ <-v>$$

which is the quaternion conjugate. The zero rotation $0 \mapsto 1 <0, 0, 0>$ which is the identity quaternion. A vector $v \in \mathbb{R}^3$ is rotated by

$$\mathring{q} \bullet v \mapsto \mathring{q} \circ \mathring{v} \circ \mathring{q}^{-1}$$

where $\mathring{v} = 0 <v>$ is known as a pure quaternion.

$$\mathring{q} = \cos\tfrac{\theta}{2} < \hat{v} \sin\tfrac{\theta}{2} > \tag{2.22}$$

and has similarities to the angle-axis representation of Sect. 2.2.1.5.

In the Toolbox these are implemented by the `UnitQuaternion` class and the constructor converts a passed argument such as a rotation matrix to a unit quaternion, for example

```
>> q = UnitQuaternion( rpy2tr(0.1, 0.2, 0.3)  )
q =
0.98335 < 0.034271, 0.10602, 0.14357 >
```

Compounding two orthonormal rotation matrices requires 27 multiplications and 18 additions. The quaternion form requires 16 multiplications and 12 additions. This saving can be particularly important for embedded systems.

This class overloads a number of standard methods and functions. Quaternion multiplication◀ is invoked through the overloaded multiplication operator

```
>> q = q * q;
```

and inversion, the conjugate of a unit quaternion, is

```
>> inv(q)
ans =
0.93394 < -0.0674, -0.20851, -0.28236 >
```

Multiplying a quaternion by its inverse yields the identity quaternion

```
>> q*inv(q)
ans =
1 < 0, 0, 0 >
```

which represents a null rotation, or more succinctly

```
>> q/q
ans =
1 < 0, 0, 0 >
```

The quaternion can be converted to an orthonormal rotation matrix by

```
>> q.R
ans =
    0.7536   -0.4993    0.4275
    0.5555    0.8315   -0.0081
   -0.3514    0.2436    0.9040
```

and we can also plot the orientation represented by a quaternion

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

which produces a result similar in style to that shown in Fig. 2.13. A vector is rotated by a quaternion using the overloaded multiplication operator

```
>> q*[1 0 0]'
ans =
    0.7536
    0.5555
   -0.3514
```

The Toolbox implementation is quite complete and the `UnitQuaternion` class has many methods and properties which are described fully in the online documentation.

2.2.2 Pose in 3-Dimensions

We return now to representing relative pose in three dimensions – the position and orientation change between the two coordinate frames as shown in Fig. 2.10. This is often referred to as a rigid-body displacement or rigid-body motion.

We have discussed several different representations of orientation, and we need to combine one of these with translation, to create a tangible representation of relative pose.

2.2.2.1 Homogeneous Transformation Matrix

The derivation for the homogeneous transformation matrix is similar to the 2D case of Eq. 2.11 but extended to account for the z-dimension. $\boldsymbol{t} \in \mathbb{R}^3$ is a vector defining the origin of frame $\{B\}$ with respect to frame $\{A\}$, and $\boldsymbol{R}$ is the 3×3 orthonormal matrix which describes the orientation of the axes of frame $\{B\}$ with respect to frame $\{A\}$.

$$\begin{pmatrix} {}^A x \\ {}^A y \\ {}^A z \\ 1 \end{pmatrix} = \begin{pmatrix} {}^A\boldsymbol{R}_B & \boldsymbol{t} \\ \boldsymbol{0}_{1\times 3} & 1 \end{pmatrix} \begin{pmatrix} {}^B x \\ {}^B y \\ {}^B z \\ 1 \end{pmatrix}$$

If points are represented by homogeneous coordinate vectors then

$$\begin{aligned} {}^A\tilde{\boldsymbol{p}} &= \begin{pmatrix} {}^A\boldsymbol{R}_B & \boldsymbol{t} \\ \boldsymbol{0}_{1\times 3} & 1 \end{pmatrix} {}^B\tilde{\boldsymbol{p}} \\ &= {}^A\boldsymbol{T}_B \, {}^B\tilde{\boldsymbol{p}} \end{aligned} \tag{2.23}$$

and ${}^A\boldsymbol{T}_B$ is a 4×4 homogeneous transformation matrix. This matrix has a very specific structure and belongs to the special Euclidean group of dimension 3 or $\boldsymbol{T} \in \mathbf{SE}(3) \subset \mathbb{R}^{4\times 4}$.

A concrete representation of relative pose is $\xi \sim \boldsymbol{T} \in \mathbf{SE}(3)$ and $\boldsymbol{T}_1 \oplus \boldsymbol{T}_2 \mapsto \boldsymbol{T}_1\boldsymbol{T}_2$ which is standard matrix multiplication.

$$\boldsymbol{T}_1\boldsymbol{T}_2 = \begin{pmatrix} \boldsymbol{R}_1 & \boldsymbol{t}_1 \\ \boldsymbol{0}_{1\times 3} & 1 \end{pmatrix} \begin{pmatrix} \boldsymbol{R}_2 & \boldsymbol{t}_2 \\ \boldsymbol{0}_{1\times 3} & 1 \end{pmatrix} = \begin{pmatrix} \boldsymbol{R}_1\boldsymbol{R}_2 & \boldsymbol{t}_1 + \boldsymbol{R}_1\boldsymbol{t}_2 \\ \boldsymbol{0}_{1\times 3} & 1 \end{pmatrix} \tag{2.24}$$

One of the rules of pose algebra from page 21 is $\xi \oplus 0 = \xi$. For matrices we know that $\boldsymbol{T}\boldsymbol{I} = \boldsymbol{T}$, where $\boldsymbol{I}$ is the identify matrix, so for pose $0 \mapsto \boldsymbol{I}$ the identity matrix. Another rule of pose algebra was that $\xi \ominus \xi = 0$. We know for matrices that $\boldsymbol{T}\boldsymbol{T}^{-1} = \boldsymbol{I}$ which implies that $\ominus \boldsymbol{T} \mapsto \boldsymbol{T}^{-1}$

$$\boldsymbol{T}^{-1} = \begin{pmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}_{1\times 3} & 1 \end{pmatrix}^{-1} = \begin{pmatrix} \boldsymbol{R}^T & -\boldsymbol{R}^T\boldsymbol{t} \\ \boldsymbol{0}_{1\times 3} & 1 \end{pmatrix} \tag{2.25}$$

The 4×4 homogeneous transformation is very commonly used in robotics, computer graphics and computer vision. It is supported by the Toolbox and will be used throughout this book as a concrete representation of 3-dimensional pose.

The Toolbox has many functions to create homogeneous transformations. For example we can demonstrate composition of transforms by

```
>> T = transl(1, 0, 0) * trotx(pi/2) * transl(0, 1, 0)
T =
    1.0000         0         0    1.0000
         0    0.0000   -1.0000    0.0000
         0    1.0000    0.0000    1.0000
         0         0         0    1.0000
```

The function `transl` creates a relative pose with a finite translation but no rotation, while `trotx` creates a relative pose corresponding to a rotation of $\frac{\pi}{2}$ about the x-axis with zero translation.◂ We can think of this expression as representing a walk along the x-axis for 1 unit, then a rotation by 90° about the x-axis and then a walk of 1 unit along the new y-axis which was the previous z-axis. The result, as shown in the last column of the resulting matrix is a translation of 1 unit along the original x-axis and 1 unit along the original z-axis. The orientation of the final pose shows the effect of the rotation about the x-axis. We can plot the corresponding coordinate frame by

Many Toolbox functions have variants that return orthonormal rotation matrices or homogeneous transformations, for example, `rotx` and `trotx`, `rpy2r` and `rpy2tr` etc. Some Toolbox functions accept an orthonormal rotation matrix or a homogeneous transformation and ignore the translational component, for example, `tr2rpy`.

```
>> trplot(T)
```

The rotation matrix component of `T` is

```
>> t2r(T)
ans =
    1.0000         0         0
         0    0.0000   -1.0000
         0    1.0000    0.0000
```

and the translation component is a column vector

```
>> transl(T)'
ans =
    1.0000    0.0000    1.0000
```

2.2.2.2 Vector-Quaternion Pair

This representation is not implemented in the Toolbox.

A compact and practical representation is the vector and unit quaternion pair. It represents pose using just 7 numbers, is easy to compound, and singularity free.◂

For the vector-quaternion case $\xi \sim (\boldsymbol{t}, \mathring{q})$ where $\boldsymbol{t} \in \mathbb{R}^3$ is a vector defining the frame's origin with respect to the reference coordinate frame, and $\mathring{q} \in \mathbb{S}^3$ is the frame's orientation with respect to the reference frame.

Composition is defined by

$$\xi_1 \oplus \xi_2 = \left(\boldsymbol{t}_1 + \mathring{q}_1 \bullet \boldsymbol{t}_2, \mathring{q}_1 \circ \mathring{q}_2\right)$$

and negation is

$$\ominus \xi = \left(-\mathring{q}^{-1} \bullet \boldsymbol{t}, \mathring{q}^{-1}\right)$$

and a point coordinate vector is transformed to a coordinate frame by

$${}^{X}\boldsymbol{p} = {}^{X}\xi_{Y} \bullet {}^{Y}\boldsymbol{p} = \mathring{q} \bullet {}^{Y}\boldsymbol{p} + \boldsymbol{t}$$

2.2.2.3 Twists

Pure translation can be considered as rotation about a point at infinity.

In Sect. 2.1.2.3 we introduced twists for the 2D case. Any rigid-body motion in 3D space is equivalent to a screw motion – motion about and along some line in space.◂ We represent a screw as a pair of 3-vectors $\boldsymbol{s} = (\boldsymbol{v}, \boldsymbol{\omega}) \in \mathbb{R}^6$.

The $\boldsymbol{\omega}$ component of the twist vector is the direction of the screw axis. The $\boldsymbol{v}$ component is called the moment and encodes the position of the line of the twist axis in space and also the pitch of the screw. The pitch is the ratio of the distance along the screw axis to the rotation about the screw axis.

Consider the example of a rotation of 0.3 radians about the x-axis. We first specify a unit twist▸ with an axis that is parallel to the x-axis and passes through the origin

A rotational unit twist has $\|\boldsymbol{\omega}\| = 1$.

```
>> tw = Twist('R', [1 0 0], [0 0 0])
tw =
( -0 -0 -0; 1 0 0 )
```

which we convert, for the required rotation angle, to an SE(3)-homogeneous transformation

```
>>  tw.T(0.3)
ans =
    1.0000         0         0         0
         0    0.9553   -0.2955         0
         0    0.2955    0.9553         0
         0         0         0    1.0000
```

and has the same value we would obtain using `trotx(0.3)`.

For pure translation in the y-direction the unit twist▸ would be

A translational unit twist has $\|\boldsymbol{v}\| = 1$ and $\boldsymbol{\omega} = 0$.

```
>> tw = Twist('T', [0 1 0])
tw =
( 0 1 0; 0 0 0 )
```

which we convert, for the required translation distance, to an SE(3)-homogeneous transformation.

```
>> tw.T(2)
ans =
     1     0     0     0
     0     1     0     2
     0     0     1     0
     0     0     0     1
```

which is, as expected, an identity matrix rotational component (no rotation) and a translational component of 2 in the y-direction.

To illustrate the underlying screw model we define a coordinate frame $\{X\}$

```
>> X = transl(3, 4, -4);
```

which we will rotate by a range of angles

```
>> angles = [0:0.3:15];
```

around a screw axis parallel to the z-axis, direction (0, 0, 1), through the point (2, 3, 2) and with a pitch of 0.5

```
>> tw = Twist('R', [0 0 1], [2 3 2], 0.5);
```

The next line packs a lot of functionality. For values of θ drawn successively from the vector `angles` we use an anonymous function to evaluate the twist for each value of θ and apply it to the frame $\{X\}$. This sequence is animated and each frame in the sequence is retained

```
>> tranimate( @(theta) tw.T(theta) * X, angles, ...
       'length', 0.5, 'retain', 'rgb', 'notext');
```

and the result is shown in Fig. 2.17. We can clearly see the screw motion in the successive poses of the displaced reference frame as it is rotated about the screw axis.

The screw axis is the line

```
>> L = tw.line
L =
{ 3  -2  0; 0  0  1 }
```

which is described in terms of its Plücker coordinates which we can plot

```
>> L.plot('k:', 'LineWidth', 2)
```

Finally we can convert an arbitrary homogeneous transformation to a nonunit twist

```
>> T = transl(1, 2, 3) * eul2tr(0.3, 0.4, 0.5);
>> tw = Twist(T)
tw =
( 1.1204 1.6446 3.1778; 0.041006 0.4087 0.78907 )
```

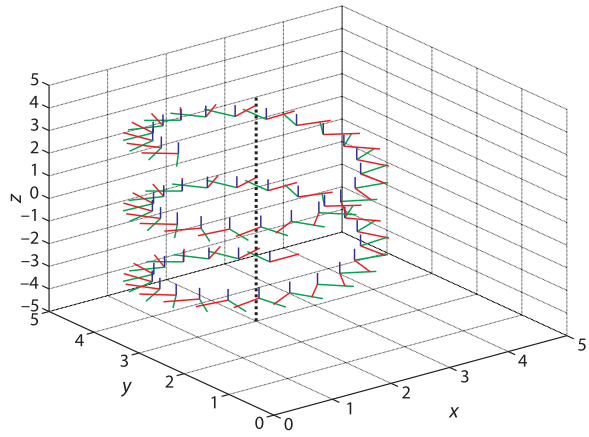


Fig. 2.17. A coordinate frame {*X*} displayed for different values of θ about a screw parallel to the *z*-axis and passing through the point (2, 3, 2). The *x*-, *y*- and *z*-axes are indicated by red, green and blue lines respectively

which has a pitch of

```
>> tw.pitch
ans =
    3.2256
```

and the rotation about the axis is

```
>> tw.theta
ans =
    0.8896
```

and a point lying on the twist axis is

```
>> tw.pole'
ans =
    0.0011    0.8473   -0.4389
```

2.3 Advanced Topics

2.3.1 Normalization

The IEEE standard for double precision floating point, the standard MATLAB numeric format, has around 16 decimal digits of precision.

Floating-point arithmetic has finite precision◂ and consecutive operations will accumulate error. A rotation matrix has by definition, a determinant of one

```
>> R = eye(3,3);
>> det(R) - 1
ans =
     0
```

but if we repeatedly multiply by a valid rotation matrix the result

```
>> for i=1:100
     R = R * rpy2r(0.2, 0.3, 0.4);
   end
>> det(R) - 1
ans =
  4.4409e-15
```

indicates a small error – the determinant is no longer equal to one and the matrix is no longer a proper orthonormal rotation matrix. To fix this we need to normalize the matrix, a process which enforces the constraints on the columns $\boldsymbol{c}_i$ of an orthonormal matrix $\boldsymbol{R} = [\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3]$. We need to assume that one column has the correct direction

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

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 all normalized to unit magnitude

$$\boldsymbol{c}_i'' = \frac{\boldsymbol{c}_1'}{\|\boldsymbol{c}_1'\|}, \quad i = 1 \cdots 3$$

In the Toolbox normalization is implemented by

```
>> R = trnorm(R);
```

and the determinant is now much closer to one▸

This error is now at the limit of double precision arithmetic which is 2.2204×10^{-16} and given by the MATLAB function `eps`.

```
>> det(R) - 1
ans =
  -2.2204e-16
```

A similar issue arises for unit quaternions when the norm, or magnitude, of the unit quaternion is no longer equal to one. However this is much easier to fix since normalizing the quaternion simply involves dividing all elements by the norm

$$\mathring{q}' = \frac{\mathring{q}}{\|\mathring{q}\|}$$

which is implemented by the `unit` method

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

The `UnitQuaternion` class also supports a variant of multiplication

```
>> q = q .* q2;
```

which performs an explicit normalization after the multiplication.

Normalization does not need to be done after every multiplication since it is an expensive operation. However for situations like the example above where one transform is being repeatedly updated it is advisable.

2.3.2 Understanding the Exponential Mapping

In this chapter we have glimpsed some connection between rotation matrices, skew-symmetric matrices and matrix exponentiation. The basis for this lies in the mathematics of Lie groups which are covered in text books on algebraic geometry and algebraic topology. These require substantial knowledge of advanced mathematics and many people starting out in robotics will find their content quite inaccessible. An introduction to the essentials of this topic is given in Appendix D. In this section we will use an intuitive approach, based on undergraduate engineering mathematics, to shed some light on these relationships.

Consider a point **P**, defined by a coordinate vector $\boldsymbol{p}$, being rotated with an angular velocity $\boldsymbol{\omega}$ which is a vector whose direction defines the axis of rotation and whose magnitude $\|\boldsymbol{\omega}\|$ specifies the rate of rotation about the axis which we assume passes through the origin.▸ We wish to rotate the point by an angle θ about this axis and the velocity of the point is known from mechanics to be

Angular velocity will be properly introduced in the next chapter.

$$\dot{\boldsymbol{p}} = \boldsymbol{\omega} \times \boldsymbol{p}$$

and we replace the cross product with a skew-symmetric matrix giving a matrix-vector product

$$\dot{\boldsymbol{p}} = [\boldsymbol{\omega}]_\times \boldsymbol{p} \tag{2.26}$$

We can find the solution to this first-order differential equation by analogy to the simple scalar case

$$\dot{x} = ax$$

whose solution is

$$x(t) = \mathrm{e}^{at} x(0)$$

This implies that the solution to Eq. 2.26 is

$$\boldsymbol{p}(t) = \mathrm{e}^{[\boldsymbol{\omega}]_\times t} \boldsymbol{p}(0)$$

If $\|\boldsymbol{\omega}\| = 1$ then after t seconds the vector will have rotated by t radians. We require a rotation by θ so we can set $t = \theta$ to give

$$\boldsymbol{p}(\theta) = \mathrm{e}^{[\hat{\boldsymbol{\omega}}]_\times \theta} \boldsymbol{p}(0)$$

which describes the vector $\boldsymbol{p}(0)$ being rotated to $\boldsymbol{p}(\theta)$. A matrix that rotates a vector is a rotation matrix, and this implies that our matrix exponential is a rotation matrix

$$\boldsymbol{R}(\theta, \hat{\boldsymbol{\omega}}) = \mathrm{e}^{[\hat{\boldsymbol{\omega}}]_\times \theta} \in \mathbf{SO}(3)$$

Now consider the more general case of rotational and translational motion. We can write

$$\dot{\boldsymbol{p}} = [\boldsymbol{\omega}]_\times \boldsymbol{p} + \boldsymbol{v}$$

and rearranging into matrix form

$$\begin{pmatrix} \dot{\boldsymbol{p}} \\ 0 \end{pmatrix} = \begin{pmatrix} [\boldsymbol{\omega}]_\times & \boldsymbol{v} \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{p} \\ 1 \end{pmatrix}$$

and introducing homogeneous coordinates this becomes

$$\begin{aligned} \dot{\tilde{p}} &= \begin{pmatrix} [\boldsymbol{\omega}]_\times & \boldsymbol{v} \\ 0 & 0 \end{pmatrix} \tilde{\boldsymbol{p}} \\ &= \boldsymbol{\Sigma} \tilde{p} \end{aligned}$$

where $\boldsymbol{\Sigma}$ is a 4×4 augmented skew-symmetric matrix. Again, by analogy with the scalar case we can write the solution as

$$\tilde{\boldsymbol{p}}(\theta) = \mathrm{e}^{\Sigma\theta} \tilde{\boldsymbol{p}}(0)$$

A matrix that rotates and translates a point in homogeneous coordinates is a homogeneous transformation matrix, and this implies that our matrix exponential is a homogeneous transformation matrix

$$\boldsymbol{T}(\theta, \hat{\boldsymbol{\omega}}, \boldsymbol{v}) = \mathrm{e}^{\begin{pmatrix} [\hat{\boldsymbol{\omega}}]_\times & \boldsymbol{v} \\ 0 & 0 \end{pmatrix} \theta} \in \mathbf{SE}(3)$$

where $[\hat{\boldsymbol{\omega}}]_\times \theta$ defines the magnitude and axis of rotation and $\boldsymbol{v}\theta$ is the translation.

The exponential of a scalar can be computed using a power series, and the matrix case is analogous and relatively straightforward to compute. The MATLAB function `expm` uses a polynomial approximation for the general matrix case. If $\boldsymbol{A}$ is skew-symmetric or augmented-skew-symmetric then an efficient closed-form solution for a rotation matrix – the Rodrigues' rotation formula (Eq. 2.18) – can be used and this is implemented by the Toolbox function `trexp`.

2.3.3 More About Twists

In this chapter we introduced and applied twists and here we will more formally define them. We also highlight the very close relationship between twists and homogeneous transformation matrices via the exponential mapping.

The key concept comes from Chasle's theorem: "*any displacement of a body in space can be accomplished by means of a rotation of the body about a unique line in space accompanied by a translation of the body parallel to that line*". Such a line is called a screw axis and is illustrated in Fig. 2.18. The mathematics of screw theory was developed by Sir Robert Ball in the late 19th century for the analysis of mechanisms. At the core of screw theory are pairs of vectors: angular and linear velocity; forces and moments; and Plücker coordinates (see Sect. C.1.2.2).

The general displacement of a rigid body in 3D can be represented by a twist vector

$$\boldsymbol{S} = (\boldsymbol{v}, \omega) \in \mathbb{R}^6$$

where $\boldsymbol{v} \in \mathbb{R}^3$ is referred to as the moment and encodes the position of the action line in space and the pitch of the screw and $\boldsymbol{\omega} \in \mathbb{R}^3$ is the direction of the screw axis.

For rotational motion where the screw axis is parallel to the vector $\hat{\boldsymbol{a}}$, passes through a point $\mathbf{Q}$ defined by its coordinate vector $\boldsymbol{q}$, and the screw pitch p is the ratio of the distance along the screw axis to the rotation about the axis, the twist elements are

$$\boldsymbol{S} = (\boldsymbol{q} \times \hat{\boldsymbol{a}} + p\hat{\boldsymbol{a}}, \hat{\boldsymbol{a}})$$

and the pitch can be recovered by

$$p = \hat{\boldsymbol{w}}^T \boldsymbol{v}$$

For the case of pure rotation the pitch of the screw is zero and the unit twist is

$$\boldsymbol{S} = (\boldsymbol{q} \times \hat{\boldsymbol{a}}, \hat{\boldsymbol{a}})$$

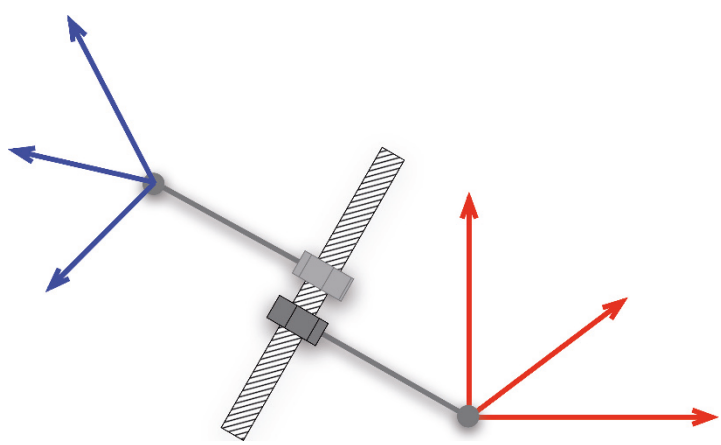
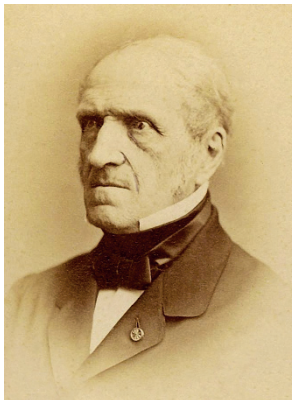


Fig. 2.18.
Conceptual depiction of a screw. A coordinate frame is attached to a nut by a rigid rod and rotated around the screw thread. The pose changes from the red frame to the blue frame. The corollary is that given any two frames we can determine a screw axis to rotate one into the other



Michel Chasles (1793–1880) was a French mathematician born at Épernon. He studied at the École Polytechnique in Paris under Poisson and in 1814 was drafted to defend Paris in the War of the Sixth Coalition. In 1837 he published a work on the origin and development of methods in geometry, which gained him considerable fame and he was appointed as professor at the École Polytechnique in 1841, and at the Sorbonne in 1846.

He was an avid collector and purchased over 27 000 forged letters purporting to be from Newton, Pascal and other historical figures – all written in French! One from Pascal claimed he had discovered the laws of gravity before Newton, and in 1867 Chasles took this to the French Academy of Science but scholars recognized the fraud. Eventually Chasles admitted he had been deceived and revealed he had spent nearly 150 000 francs on the letters. He is buried in Cimetière du Père Lachaise in Paris.

For purely translational motion in the direction parallel to the vector $\boldsymbol{a}$, the pitch is infinite which leads to a zero rotational component and the unit twist is

$$\boldsymbol{S} = (\hat{\boldsymbol{a}}, 0)$$

A twist is related to the rigid-body displacement in $\mathbf{SE}(3)$ by the exponential mapping already discussed.

$$\boldsymbol{T}(\theta, \boldsymbol{S}) = \mathrm{e}^{[\boldsymbol{S}]\theta} \in \mathbf{SE}(3)$$

where the augmented skew-symmetric matrix

$$[\boldsymbol{S}] = \left(\begin{array}{ccc|c} 0 & -\omega_3 & \omega_2 & v_1 \\ \omega_3 & 0 & -\omega_1 & v_2 \\ -\omega_2 & \omega_1 & 0 & v_3 \\ \hline 0 & 0 & 0 & 0 \end{array}\right) \in \mathbf{se}(3)$$

belongs to the Lie algebra $\mathbf{se}(3)$ and is the *generator* of the rigid-body displacement. The matrix exponential has an efficient closed-form

$$\boldsymbol{T}(\theta, \boldsymbol{S}) = \begin{pmatrix} \boldsymbol{R}(\theta, \hat{\boldsymbol{\omega}}) & \left(\boldsymbol{I}_{3\times 3}\theta + (1-\cos\theta)[\hat{\boldsymbol{\omega}}]_\times + (\theta - \sin\theta)[\hat{\boldsymbol{\omega}}]_\times^2\right)\boldsymbol{v} \\ 0 & 1 \end{pmatrix}$$

where $\boldsymbol{R}(\theta, \hat{\omega})$ is computed using Rodrigues' rotation formula (Eq. 2.18). For a nonunit rotational twist, that is $\|\boldsymbol{\omega}\| \neq 1$, then $\theta = \|\boldsymbol{\omega}\|$.

For real numbers, if $x = \log X$ and $y = \log Y$ then

$$Z = XY = \mathrm{e}^x \mathrm{e}^y = \mathrm{e}^{x+y}$$

but for the matrix case this is *only true* if the matrices commute, and rotation matrices do not, therefore

$$Z = XY = \mathrm{e}^x \mathrm{e}^y \neq \mathrm{e}^{x+y} \quad \text{if } x, y \in \mathbf{so}(n) \text{ or } \mathbf{se}(n)$$

The bottom line is that there is no shortcut to compounding rotations, we must compute $z = \log(\mathrm{e}^x \mathrm{e}^y)$ not $z = x + y$.

The Toolbox provides many ways to create twists and to convert them to rigid-body displacements expressed as homogeneous transformations. Now that we understand more about the exponential mapping we will revisit the example from page 48

```
>> tw = Twist('R', [1 0 0], [0 0 0])
tw =
( -0 -0 -0; 1 0 0 )
```

A unit twist describes a *family* of motions that have a single parameter, either a rotation and translation about and along some screw axis, or a pure translation in some direction. We can visualize it as a mechanical screw in space, or represent it as a 6-vector $\boldsymbol{S} = (\boldsymbol{v}, \boldsymbol{\omega})$ where $\|\boldsymbol{\omega}\| = 1$ for a rotational twist and $\|\boldsymbol{v}\| = 1, \boldsymbol{\omega} = 0$ for a translational twist.

A *particular* rigid-body motion is described by a unit-twist $\boldsymbol{s}$ and a motion parameter θ which is a scalar specifying the amount of rotation or translation. The motion is described by the twist $\boldsymbol{S}\theta$ which is in general not a unit-twist. The exponential of this in 4×4 matrix format is the 4×4 homogeneous transformation matrix describing that particular rigid-body motion in $\mathbf{SE}(3)$.

which is a unit twist that describes rotation about the x-axis in $\mathbf{SE}(3)$. The `Twist` has a number of properties

```
>> tw.S'
ans =
     0     0     0     1     0     0
>> tw.v'
ans =
     0     0     0
>> tw.w'
ans =
     1     0     0
```

as well as various methods. We can create the $\mathbf{se}(3)$ Lie algebra using the `se` method of this class

```
>> tw.se
ans =
     0     0     0     0
     0     0    -1     0
     0     1     0     0
     0     0     0     0
```

which is the augmented skew-symmetric version of $\boldsymbol{S}$. The method `T` performs the exponentiation▸ of this to create an $\mathbf{SE}(3)$ homogeneous transformation for the specified rotation about the unit twist

The `expm` method is synonomous and both invoke the Toolbox function `trexp`.

```
>>  tw.T(0.3)
ans =
    1.0000         0         0         0
         0    0.9553   -0.2955         0
         0    0.2955    0.9553         0
         0         0         0    1.0000
```

The Toolbox functions `trexp` and `trlog` are respectively closed-form alternatives to `expm` and `logm` when the arguments are in $\mathbf{so}(3)/\mathbf{se}(3)$ or $\mathbf{SO}(3)/\mathbf{SE}(3)$.

The `line` method returns a `Plucker` object that represents the line of the screw in Plücker coordinates

```
>> tw.line
ans =
{ 0  0  0; 1  0  0 }
```

Finally, the overloaded multiplication operator for the `Twist` class will compound two twists.

```
>> t2 = tw * tw
t2 =
( -0  -0  -0; 2  0  0 )
>> tr2angvec(t2.T)
Rotation: 2.000000 rad x [1.000000 0.000000 0.000000]
```

and the result in this case is a nonunit twist of two units, or 2 rad, about the x-axis.

2.3.4 Dual Quaternions

1845–1879, an English mathematician and geometer.

Quaternions were developed by William Hamilton in 1843 and we have already seen their utility for representing orientation, but using them to represent pose proved more difficult. One early approach was Hamilton's bi-quaternion where the quaternion coefficients were complex numbers. Somewhat later William Clifford◂ developed the dual number, defined as an ordered pair $d = (x, y)$ which can be written as $d = x + y\varepsilon$ where $\varepsilon^2 = 0$ and for which specific addition and multiplication rules exist. Clifford created a quaternion dual number with $x, y \in \mathbb{H}$ which he also called a bi-quaternion but is today called a dual quaternion

$$\mathring{q}_\xi = r + \tfrac{1}{2}\varepsilon \mathring{t} \circ \mathring{r}$$

where $\mathring{r} \in \mathbb{H}$ is a unit quaternion representing the rotational part of the pose and $\mathring{t} \in \mathbb{H}$ is a pure quaternion representing translation. This type of mathematical object has been largely eclipsed by modern matrix and vector approaches, but there seems to be a recent resurgence of interest in alternative approaches. The dual quaternion is quite compact, requiring just 8 numbers; it is easy to compound using a special multiplication table; and it is easy to renormalize to eliminate the effect of imprecise arithmetic. However it has no real useful computational advantage over matrix methods.

2.3.5 Configuration Space

We have so far considered the pose of objects in terms of the position and orientation of a coordinate frame affixed to them. For an arm-type robot we might affix a coordinate frame to its end-effector, while for a mobile robot we might affix a frame to its body – its body-fixed frame. This is sufficient to describe the state of the robot in the familiar 2D or 3D Euclidean space which is referred to as the task space or operational space since it is where the robot performs tasks or operates.

An alternative way of thinking about this comes from classical mechanics and is referred to as the *configuration* of a system. The configuration is the smallest set of parameters, called generalized coordinates, that are required to fully describe the position of *every* particle in the system. This is not as daunting as it may appear since in general a robot comprises one or more rigid elements, and in each of these the particles maintain a constant relative offset to each other.

If the *system* is a train moving along a track then all the particles comprising the train move together and we need only a single generalized coordinate q, the distance along the track from some datum, to describe their location. A robot arm with a fixed base and two rigid links, connected by two rotational joints has a configuration that is completely described by two generalized coordinates – the two joint angles (q_1, q_2). The generalized coordinates can, as their name implies, represent displacements or rotations.

Sir Robert Ball (1840–1913) was an Irish astronomer born in Dublin. He became Professor of Applied Mathematics at the Royal College of Science in Dublin in 1867, and in 1874 became Royal Astronomer of Ireland and Andrews Professor of Astronomy at the University of Dublin. In 1892 he was appointed Lowndean Professor of Astronomy and Geometry at Cambridge University and became director of the Cambridge Observatory. He was a Fellow of the Royal Society and in 1900 became the first president of the Quaternion Society.

He is best known for his contributions to the science of kinematics described in his treatise "The Theory of Screws" (1876), but he also published "A Treatise on Spherical Astronomy" (1908) and a number of popular articles on astronomy. He is buried at the Parish of the Ascension Burial Ground in Cambridge.

The number of independent▸ generalized coordinates N is known as the number of degrees of freedom of the system. Any configuration of the system is represented by a point in its N-dimensional configuration space, or C-space, denoted by $\mathcal{C}$ and $\boldsymbol{q} \in \mathcal{C}$. We can also say that $\dim \mathcal{C} = N$. For the train example $\mathcal{C} \subset \mathbb{R}$ which says that the displacement is a bounded real number. For the 2-joint robot the generalized coordinates are both angles so $\mathcal{C} \subset \mathbb{S}^1 \times \mathbb{S}^1$.

That is, there are no holonomic constraints on the system.

Any point in the configuration space can be mapped to a point in the task space $q \in \mathcal{C} \mapsto \tau \in \mathcal{T}$ but the inverse is not necessarily true. This mapping depends on the task space that we choose and this, as its name suggests, is task specific.

Consider again the train moving along its rail. We might be interested to describe the train in terms of its position on a plane in which case the task space would be $\mathcal{T} \subset \mathbb{R}^2$, or in terms of its latitude and longitude, in which case the task space would be $\mathcal{T} \subset \mathbb{S}^1 \times \mathbb{S}^1$. We might choose a 3-dimensional task space $\mathcal{T} \subset \mathbf{SE}(3)$ to account for height changes as the train moves up and down hills and its orientation changes as it moves around curves. However in all these case the dimension of the task space exceeds the dimension of the configuration space $\dim \mathcal{T} > \dim \mathcal{C}$ and this means that the train cannot *access* all points in the task space. While every point along the rail line can be mapped to the task space, most points in the task space will not map to a point on the rail line. The train is constrained by its fixed rails to move in a subset of the task space.

The simple 2-joint robot arm can access a subset of points in a plane so a useful task space might be $\mathcal{T} \subset \mathbb{R}^2$. The dimension of the task space equals the dimension of the configuration space $\dim \mathcal{T} = \dim \mathcal{C}$ and this means that the mapping between task and configuration spaces is bi-directional but it is not necessarily unique – for this type of robot, in general, two different configurations map to a single point in task space. Points in the task space beyond the physical reach of the robot are not mapped to the configuration space. If we chose a task space with more dimensions such as $\mathbf{SE}(2)$ or $\mathbf{SE}(3)$ then $\dim \mathcal{T} > \dim \mathcal{C}$ and the robot would only be able to access points within a subset of that space.

Now consider a snake-robot arm, such as shown in Fig. 8.9, with 20 joints and $\mathcal{C} \subset \mathbb{S}^1 \times \cdots \times \mathbb{S}^1$ and $\dim \mathcal{T} < \dim \mathcal{C}$. In this case an infinite number of configurations in a $20 - 6 = 14$-dimensional subspace of the 20-dimensional configuration space will map to the same point in task space. This means that in addition to the task of positioning the robot's end-effector we can *simultaneously* perform motion in the configuration subspace to control the shape of the arm to avoid obstacles in the environment. Such a robot is referred to as over-actuated or redundant and this topic is covered in Sect. 8.4.2.

The body of a quadrotor, such as shown in Fig. 4.19d, is a single rigid-body whose configuration is completely described by six generalized coordinates, its position and orientation in 3D space $\mathcal{C} \subset \mathbb{R}^3 \times \mathbb{S}^1 \times \mathbb{S}^1 \times \mathbb{S}^1$ where the orientation is expressed in some three-angle representation. For such a robot the most logical task space would be $\mathbf{SE}(3)$ which is equivalent to the configuration space and $\dim \mathcal{T} = \dim \mathcal{C}$. However the quadrotor has only four actuators which means it cannot *directly* access all the points in its configuration space and hence its task space. Such a robot is referred to as under-actuated and we will revisit this in Sect. 4.2.

2.4 Using the Toolbox

The Toolbox supports all the different representations discussed in this chapter as well as conversions between many of them. The representations and possible conversions are shown in tabular form in Tables 2.1 and 2.2 for the 2D and 3D cases respectively.

In this chapter we have mostly used native MATLAB matrices to represent rotations and homogeneous transformations▸ and historically this has been what the Toolbox supported – the Toolbox *classic* functions. From Toolbox release 10 there are classes that

Quaternions and twists are implemented as classes not native types, but in very old versions of the Toolbox quaternions were 1×4 vectors.

represent rotations and homogeneous transformations, named respectively SO2 and SE2 for 2 dimensions and SO3 and SE3 for 3 dimensions. These provide real advantages in terms of code readability and type safety and can be used in an almost identical fashion to the native matrix types. They are also polymorphic meaning they support many of the same operations which makes it very easy to switch between using say rotation matrices and quaternions or lifting a solution from 2- to 3-dimensions. A quick illustration of the new functionality is the example from page 27 which becomes

```
>> T1 = SE2(1, 2, 30, 'deg');
>> about T1
T1 [SE2] : 1x1 (176 bytes)
```

The size of the object in bytes, shown in parentheses, will vary between MATLAB. versions and computer types.

which results in an SE2 class object not a 3×3 matrix. If we display it however it does look like a 3×3 matrix

If you have the cprintf package from MATLAB File Exchange installed then the rotation submatrix will be colored red.

```
>> T1
 T1 =
    0.8660   -0.5000         1
    0.5000    0.8660         2
         0         0         1
```

The matrix is encapsulated within the object and we can extract it readily if required

```
>> T1.T
ans =
    0.8660   -0.5000    1.0000
    0.5000    0.8660    2.0000
         0         0    1.0000
>> about ans
ans [double] : 3x3 (72 bytes)
```

Returning to that earlier example we can quite simply transform the vector

```
>> inv(T1) * P
ans =
    1.7321
   -1.0000
```

and the class handles the details of converting the vector between Euclidean and homogeneous forms.

This new functionality is also covered in Tables 2.1 and 2.2, and Table 2.3 is a map between the classic and new functionality to assist you in using the Toolbox. From here on the book will use a mixture of classic functions and the newer classes.

Table 2.1. Toolbox supported data types for representing 2D pose: constructors and conversions

	Output type							
Input type	*t*	θ	*R*	*T*	Twist vector	Twist	SO2	SE2
t (2-vector)				transl2		**Twist('T')**		**SE2()**
θ (scalar)			rot2	trot2		**Twist('R')**	**SO2()**	**SE2()**
R (2×2 matrix)				r2t			**SO2()**	**SE2()**
T (3×3 matrix)	transl2		t2r			**Twist()**		**SE2()**
Twist vector (1- or 3-vector)			trexp2	trexp2		**Twist()**	SO2.exp()	**SE3**.exp()
Twist				.T	.S			.SE
SO2		.theta	.R	.T	.log			.SE2
SE2	.t	.theta	.R	.T	.log	.Twist	.SO2	

Dark grey boxes are not possible conversions. Light grey boxes are possible conversions but the Toolbox has no direct conversion, you need to convert via an intermediate type. Red text indicates classical Robotics Toolbox functions that work with native MATLAB® vectors and matrices. **Bold text** indicates a Toolbox class. Class.type() indicates a static factory method that constructs a Class object from input of that type. Functions shown starting with a dot are a method on the class corresponding to that row.

Input type	Output type: t	Euler	RPY	θ, v	R	T	Twist vector	Twist	Unit-Quaternion	SO3	SE3
t (3-vector)						`transl`		`Twist('T')`			`SE3()`
Euler (3-vector)					`eul2r`	`eul2tr`			`UnitQuaternion.eul()`	`SO3.eul()`	`SE3.eul()`
RPY (3-vector)					`rpy2r`	`rpy2tr`			`UnitQuaternion.rpy()`	`SO3.rpy()`	`SE3.rpy()`
θ, v (scalar + 3-vector)					`angvec2r`	`angvec2tr`			`UnitQuaternion.angvec()`	`SO3.angvec()`	`SE3.angvec()`
R (3×3 matrix)		`tr2eul`	`tr2rpy`	`tr2angvec`		`r2t`	`trlog`		`UnitQuaternion()`	`SO3()`	`SE3()`
T (4×4 matrix)	`transl`	`tr2eul`	`tr2rpy`	`tr2angvec`	`t2r`		`trlog`	`Twist()`	`UnitQuaternion()`	`SO3()`	`SE3()`
Twist vector (3- or 6-vector)					`trexp`	`trexp`		`Twist()`		`SO3.exp()`	`SE3.exp()`
Twist						`.T`	`.S`				`.SE`
Unit-Quaternion		`.toeul`	`.torpy`	`.toangvec`	`.R`	`.T`				`.SO3`	`.SE3`
SO3		`.toeul`	`.torpy`	`.toangvec`	`.R`	`.T`	`.log`		`.UnitQuaternion`		`.SE3`
SE3	`.t`	`.toeul`	`.torpy`	`.toangvec`	`.R`	`.T`	`.log`	`.Twist`	`.UnitQuaternion`	`.SO3`	

Dark grey boxes are not possible conversions. Light grey boxes are possible conversions but the Toolbox has no direct conversion, you need to convert via an intermediate type. Red text indicates classical Robotics Toolbox functions that work with native MATLAB® vectors and matrices. Class.type() indicates a static factory method that constructs a Class object from input of that type. Functions shown starting with a dot are a method on the class corresponding to that row.

Table 2.2. Toolbox supported data types for representing 3D pose: constructors and conversions

2.5 Wrapping Up

In this chapter we learned how to represent points and poses in 2- and 3-dimensional worlds. Points are represented by coordinate vectors relative to a coordinate frame. A set of points that belong to a rigid object can be described by a coordinate frame, and its constituent points are described by constant vectors in the object's coordinate frame. The position and orientation of any coordinate frame can be described relative to another coordinate frame by its relative pose ξ. We can think of a relative pose as a motion – a rigid-body motion – and these motions can be applied sequentially (composed or compounded). It is important to remember that composition is noncommutative – the order in which relative poses are applied is important.

We have shown how relative poses can be expressed as a pose graph or manipulated algebraically. We can also use a relative pose to transform a vector from one coordinate frame to another. A simple graphical summary of key concepts is given in Fig. 2.19.

We have discussed a variety of mathematical objects to tangibly represent pose. We have used orthonormal rotation matrices for the 2- and 3-dimensional case to represent orientation and shown how it can rotate a points' coordinate vector from one coordinate frame to another. Its extension, the homogeneous transformation matrix, can be used to represent both orientation and translation and we have shown how it can rotate and translate a point expressed in homogeneous coordinates from one frame

Orientation		Pose	
Classic	New	Classic	New
`rot2`	`SO2`	`trot2`	`SE2`
		`transl2`	`SE2`
`trplot2`	`.plot`	`trplot2`	`.plot`
`rotx, roty, rotz`	`SO3.Rx, SO3.Ry, SO3.Rz`	`trotx, troty, trotz`	`SE3.Rx, SE3.Ry, SE3.Rz`
		`T = transl(v)`	`SE3(v)`
`eul2r, rpy2r`	`SO3.eul, SO3.rpy`	`eul2tr, rpy2tr`	`SE3.eul, SE3.rpy`
`angvec2r`	`SO3.angvec`	`angvec2tr`	`SE3.angvec`
`oa2r`	`SO3.oa`	`oa2tr`	`SE3.oa`
		`v = transl(T)`	`.t, .transl`
`tr2eul, tr2rpy`	`.toeul, .torpy`	`tr2eul, tr2rpy`	`.toeul, .torpy`
`tr2angvec`	`.toangvec`	`tr2angvec`	`.toangvec`
`trexp`	`SO3.exp`	`trexp`	`SE3.exp`
`trlog`	`.log`	`trlog`	`.log`
`trplot`	`.plot`	`trplot`	`.plot`

Functions starting with dot are methods on the new objects. You can use them in functional form `toeul(R)` or in dot form `R.toeul()` or `R.toeul`. It's a personal preference. The trailing parentheses are not required if no arguments are passed, but it is a useful convention and reminder that you that you are invoking a method not reading a property. The old function `transl` appears twice since it maps a vector to a matrix as well as the inverse.

Table 2.3. Table of subsitutions from classic Toolbox functions that operate on and return a matrix, to the corresponding new classes and methods

to another. Rotation in 3-dimensions has subtlety and complexity and we have looked at various parameterizations such as Euler angles, roll-pitch-yaw angles and unit quaternions. Using Lie group theory we showed that rotation matrices, from the group $\mathbf{SO}(2)$ or $\mathbf{SO}(3)$, are the result of exponentiating skew-symmetric generator matrices. Similarly, homogeneous transformation matrices, from the group $\mathbf{SE}(2)$ or $\mathbf{SE}(3)$, are the result of exponentiating augmented skew-symmetric generator matrices. We have also introduced twists as a concise way of describing relative pose in terms of rotation around a screw axis, a notion that comes to us from screw theory and these twists are the unique elements of the generator matrices.

There are two important lessons from this chapter. The first is that there are *many* mathematical objects that can be used to represent pose and these are summarized in Table 2.4. There is no right or wrong – each has strengths and weaknesses and we typically choose the representation to suit the problem at hand. Sometimes we wish for a vectorial representation, perhaps for interpolation, in which case (x, y, θ) or $(x, y, z, \boldsymbol{\Gamma})$ might be appropriate, but this representation cannot be easily compounded. Sometime we may only need to describe 3D rotation in which case $\boldsymbol{\Gamma}$ or $\mathring{q}$ is appropriate. Converting between representations is easy as shown in Tables 2.1 and 2.2.

The second lesson is that coordinate frames are your friend. The essential first step in many vision and robotics problems is to assign coordinate frames to all objects of interest, indicate the relative poses as a directed graph, and write down equations for the loops. Figure 2.20 shows you how to build a coordinate frame out of paper that you can pick up and rotate – making these ideas more tangible. Don't be shy, embrace the coordinate frame.

We now have solid foundations for moving forward. The notation has been defined and illustrated, and we have started our hands-on work with MATLAB. The next chapter discusses motion and coordinate frames that change with time, and after that we are ready to move on and discuss robots.

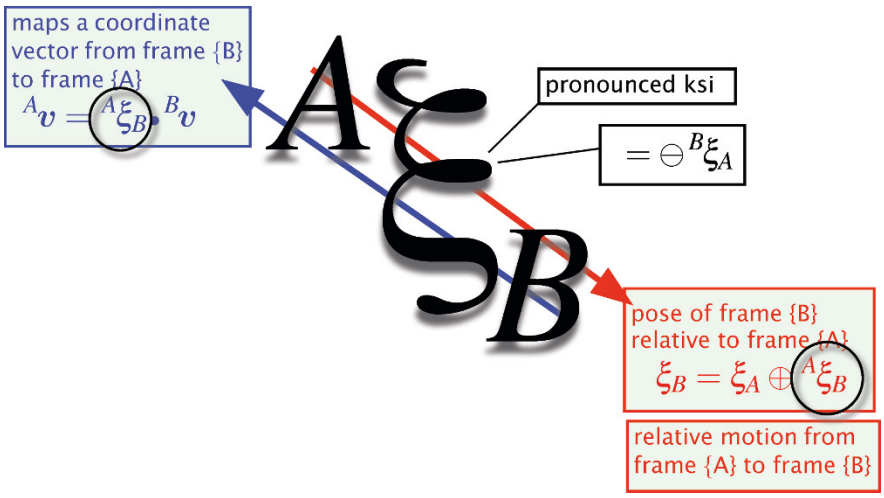


Fig. 2.19. Everything you need to know about pose

	2D	Composition	3D	Composition
Position	2-vector	+	3-vector	+
Orientation	Angle 3 × 3 rotation matrix	+ *	3 angles $\boldsymbol{\Gamma}$: Euler, RPY, etc. 2 vectors: OA (angle, vector) UnitQuaternion $\mathring{q}$ 3 × 3 rotation matrix	☹ ☹ ☹ * *
Pose	(angle, 2-vector) 4 × 4 transformation matrix	☹ *	(3 angles, 3-vector) (3-vector, UnitQuaternion) 4 × 4 transformation matrix	☹ 😐 *

Toolbox composition operators are shown in blue. Composition operators shown in red are ☹ difficult to implement, 😐 less difficult to implement.

Table 2.4. Summary of the various concrete representations of pose ξ introduced in this chapter

Further Reading

The treatment in this chapter is a hybrid mathematical and graphical approach that covers the 2D and 3D cases by means of abstract representations and operators which are later made tangible. The standard robotics textbooks such as Kelly (2013), Siciliano et al. (2009), Spong et al. (2006), Craig (2005), and Paul (1981) all introduce homogeneous transformation matrices for the 3-dimensional case but differ in their approach. These books also provide good discussion of the other representations such as angle-vector and 3-angle representations. Spong et al. (2006, sect. 2.5.1) have a good discussion of singularities. The book Lynch and Park (2017) covers the standard matrix approaches but also introduces twists and screws. Siegwart et al. (2011) explicitly cover the 2D case in the context of mobile robotics.

Quaternions are discussed in Kelly (2013) and briefly in Siciliano et al. (2009). The book by Kuipers (1999) is a very readable and comprehensive introduction to quaternions. Quaternion interpolation is widely used in computer graphics and animation and the classic paper by Shoemake (1985) is very readable introduction to this topic. The first publication about quaternions for robotics is probably Taylor (1979), and followed up in subsequent work by Funda (1990).

You will encounter a wide variety of different notation for rotations and transformations in textbooks and research articles. This book uses ${}^{A}\boldsymbol{T}_{B}$ to denote a transform giving the pose of frame $\{B\}$ with respect to frame $\{A\}$. A common alternative notation is $\boldsymbol{T}_{B}^{A}$ or even ${}_{B}^{A}\boldsymbol{T}$. To denote points this book uses ${}^{A}\boldsymbol{p}_{B}$ to denote a vector from the origin of frame $\{A\}$ to the point **B** whereas others use $\boldsymbol{p}_{B}^{A}$, or even ${}^{C}\boldsymbol{p}_{B}^{A}$ to denote a vector

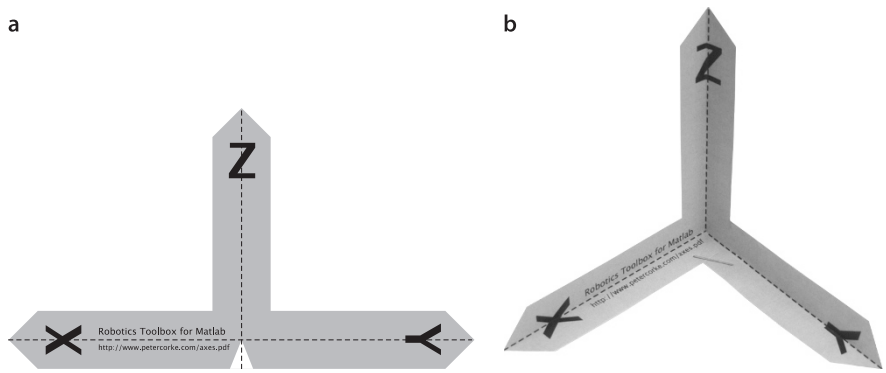


Fig. 2.20. Build your own coordinate frame. **a** Get the PDF file from http://www.petercorke.com/axes.pdf; **b** cut it out, fold along the dotted lines and add a staple. Voila!

from the origin of frame $\{A\}$ to the point **B** but with respect to coordinate frame $\{C\}$. Twists can be written as either $(\boldsymbol{v}, \boldsymbol{\omega})$ as in this book, or as $(\boldsymbol{\omega}, \boldsymbol{v})$.

Historical and general. Hamilton and his supporters, including Peter Tait, were vigorous in defending Hamilton's precedence in inventing quaternions, and for opposing the concept of vectors which were then beginning to be understood and used. Rodrigues developed his eponymous formula in 1840 although Gauss discovered it in 1819 but, as usual, did not publish it. It was published in 1900. Quaternions had a tempestuous beginning. The paper by Altmann (1989) is an interesting description on this tussle of ideas, and quaternions have even been woven into fiction (Pynchon 2006).

Exercises

1. Explore the many options associated with `trplot`.
2. Animate a rotating cube
   a) Write a function to plot the edges of a cube centered at the origin.
   b) Modify the function to accept an argument which is a homogeneous transformation which is applied to the cube vertices before plotting.
   c) Animate rotation about the x-axis.
   d) Animate rotation about all axes.
3. Create a vector-quaternion class to describe pose and which supports composition, inverse and point transformation.
4. Create a 2D rotation matrix. Visualize the rotation using `trplot2`. Use it to transform a vector. Invert it and multiply it by the original matrix; what is the result? Reverse the order of multiplication; what is the result? What is the determinant of the matrix and its inverse?
5. Create a 3D rotation matrix. Visualize the rotation using `trplot` or `tranimate`. Use it to transform a vector. Invert it and multiply it by the original matrix; what is the result? Reverse the order of multiplication; what is the result? What is the determinant of the matrix and its inverse?
6. Compute the matrix exponential using the power series. How many terms are required to match the result shown to standard MATLAB precision?
7. Generate the sequence of plots shown in Fig. 2.12.
8. For the 3-dimensional rotation about the vector $[2, 3, 4]$ by 0.5 rad compute an $\mathbf{SO}(3)$ rotation matrix using: the matrix exponential functions `expm` and `trexp`, Rodrigues' rotation formula (code this yourself), and the Toolbox function `angvec2tr`. Compute the equivalent unit quaternion.
9. Create two different rotation matrices, in 2D or 3D, representing frames $\{A\}$ and $\{B\}$. Determine the rotation matrix ${}^{A}\boldsymbol{R}_{B}$ and ${}^{B}\boldsymbol{R}_{A}$. Express these as a rotation axis and angle, and compare the results. Express these as a twist.

10. Create a 2D or 3D homogeneous transformation matrix. Visualize the rigid-body displacement using `tranimate`. Use it to transform a vector. Invert it and multiply it by the original matrix, what is the result? Reverse the order of multiplication; what happens?
11. Create two different rotation matrices, in 2D or 3D, representing frames $\{A\}$ and $\{B\}$. Determine the rotation matrix ${}^{A}\boldsymbol{R}_{B}$ and ${}^{B}\boldsymbol{R}_{A}$. Express these as a rotation axis and angle and compare the results. Express these as a twist.
12. Create three symbolic variables to represent roll, pitch and yaw angles, then use these to compute a rotation matrix using `rpy2r`. You may want to use the `simplify` function on the result. Use this to transform a unit vector in the z-direction. Looking at the elements of the rotation matrix devise an algorithm to determine the roll, pitch and yaw angles. Hint – find the pitch angle first.
13. Experiment with the `tripleangle` application in the Toolbox. Explore roll, pitch and yaw motions about the nominal attitude and at singularities.
14. If you have an iPhone or iPad download from the App Store the free "Euler Angles" app by École de Technologie Supérieure and experiment with it.
15. Using Eq. 2.24 show that $\boldsymbol{T}\boldsymbol{T}^{-1} = \boldsymbol{I}$.
16. Is the inverse of a homogeneous transformation matrix equal to its transpose?
17. In Sect. 2.1.2.2 we rotated a frame about an arbitrary point. Derive the expression for computing `RC` that was given.
18. Explore the effect of negative roll, pitch or yaw angles. Does transforming from RPY angles to a rotation matrix then back to RPY angles give a different result to the starting value as it does for Euler angles?
19. From page 53 show that $e^{x} e^{y} \neq e^{x+y}$ for the case of matrices. Hint – expand the first few terms of the exponential series.
20. A camera has its z-axis parallel to the vector $[0, 1, 0]$ in the world frame, and its y-axis parallel to the vector $[0, 0, -1]$. What is the attitude of the camera with respect to the world frame expressed as a rotation matrix and as a unit quaternion?