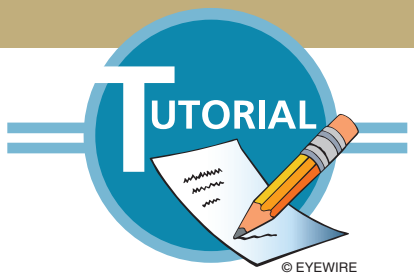

A Beginner's Guide to 6-D Vectors (Part 2)

From Equations to Software

BY ROY FEATHERSTONE

Spatial vectors are six-dimensional (6-D) vectors that describe the motions of rigid bodies and the forces acting upon them. In Part 1, we saw how spatial vectors can simplify the process of expressing and analyzing the dynamics of a simple rigid-body system. In this tutorial, we shall examine the application of spatial vectors to various problems in robot kinematics and dynamics. To demonstrate that spatial vectors are both a tool for analysis and a tool for computation, we shall consider both the mathematical solution of a problem and the computer code to calculate the answer.

To illustrate the power of spatial vectors, we shall consider the class of robots having branched connectivity. This class includes legged robots, humanoids and multifingered grippers, as well as traditional serial robot arms; however, it does not include robots with kinematic loops, such as parallel robots. To cope with this degree of generality, we shall take a model-based approach: the robot mechanism is described by means of a standard set of quantities stored in a model data structure, and the equations, algorithms, and computer code are designed to use those quantities in their calculations.

Following the same pattern as Part 1, this tutorial starts with a specific example and proceeds to analyze it in detail; the example in this instance being the computer code to implement a model-based inverse dynamics calculation using the recursive Newton–Euler algorithm. Subsequent sections then examine a variety of topics in kinematics and present the two main recursive algorithms for forward dynamics: the composite-rigid-body algorithm and the articulated-body algorithm.

It is assumed that the readers have already read Part 1 [6], or equivalent material, and therefore, they are familiar with the notation and basic concepts of spatial vector algebra.

Digital Object Identifier 10.1109/MRA.2010.939560

A Computational Example

Inverse dynamics is the problem of calculating the forces required to produce a given acceleration. It is a relatively easy problem, and therefore, a good place to start. A model-based inverse dynamics calculation can be expressed mathematically as

$$\boldsymbol{\tau} = \mathrm{ID}(\mathrm{model}, \boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}), \tag{1}$$

where $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$, $\ddot{\boldsymbol{q}}$, and $\boldsymbol{\tau}$ denote vectors of joint position, velocity, acceleration, and force variables, respectively, and *model* denotes a data structure containing a description of the robot. The objective is to calculate the numeric value of ID given the numeric values of its arguments.

Figure 1 shows the MATLAB source code for an implementation of (1) using the recursive Newton–Euler algorithm. This is a complete implementation: you could type it in right now (minus the line numbers) and get it to work, provided you also typed in the (very short) definitions of the functions `jcalc`, `crm`, and `crf`, which are discussed later in this tutorial. The code in Figure 1 can calculate the inverse dynamics of

1070-9932/10/$26.00©2010 IEEE

any robot mechanism in which the bodies are connected together in the manner of a topological tree, and each joint is either revolute, prismatic, or helical (a screw joint).

The code is clearly very short. This degree of brevity would not be possible using three-dimensional (3-D) vectors. Once the basics of spatial vectors are understood, code like this requires relatively little effort to write, test, and debug compared with the equivalent 3-D-vector code. Furthermore, code like this is relatively easy for others to read, understand, and adapt to other purposes.

```
 1  function  tau = ID( model, q, qd, qdd )
 2  for i = 1:model.N
 3    [ XJ, s{i} ] = jcalc( model.pitch(i), q(i) );
 4    vJ = s{i}*qd(i);
 5    Xup{i} = XJ * model.Xtree{i};
 6    pi = model.parent(i);
 7    if pi == 0
 8      v{i} = vJ;
 9      a{i} = Xup{i} * [0;0;0;0;0;9.81] + s{i}*qdd(i);
10    else
11      v{i} = Xup{i}*v{pi} + vJ;
12      a{i} = Xup{i}*a{pi} + s{i}*qdd(i) + crm(v{i})*vJ;
13    end
14    f{i} = model.I{i}*a{i} + crf(v{i})*model.I{i}*v{i};
15  end
16  for i = model.N:-1:1
17    tau(i,1) = s{i}' * f{i};
18    pi = model.parent(i);
19    if pi ~= 0
20      f{pi} = f{pi} + Xup{i}'*f{i};
21    end
22  end
```

Figure 1. *MATLAB code for inverse dynamics calculation.*

Model Data Structure

Before we study the code in detail, let us first examine the contents of the model data structure. This structure contains the following fields:

- `model.N`: an integer specifying the number of bodies in the mechanism
- `model.parent`: an array of integers, called the parent array, describing the connectivity of the mechanism
- `model.Xtree`: an array of Plücker coordinate transforms describing the relative locations of the joints within each body
- `model.pitch`: an array of floating point numbers describing the pitch (and therefore, the type) of each joint
- `model.I`: an array of spatial inertias giving the inertia of each body expressed in link coordinates.

This data is sufficient to describe a general kinematic tree in which the joints are revolute, prismatic, or helical. The term "kinematic tree" simply means a rigid-body system in which the connectivity is that of a topological tree. It is derived from the older term "kinematic chain." The small set of joint types is not quite as limiting as it appears, because many common joint types can be emulated by a chain of revolute and prismatic joints connected together by massless bodies. For example, a spherical joint can be emulated by a chain of three revolute joints with axes passing through the rotation center of the spherical joint. This works so long as the chain does not enter a kinematic singularity.

At this point, you might be wondering why helical joints have been included. The short answer is to demonstrate how easy it is, when using spatial vectors, to go beyond the basic repertoire of revolute and prismatic joints. A longer answer is that helical joints are more general than revolute or prismatic ones, so their inclusion represents a genuine increase in generality. Also, helical joints are an example of a joint type that requires a parameter (the pitch of the helix), so their inclusion provides an opportunity to include joint parameters in a robot model.

The next three subsections explain how the fields in the model data structure are used to model a robot mechanism, and then, we shall return to the code in Figure 1 and the algorithm it implements.

Connectivity

The connectivity of a robot mechanism can be represented by a connectivity graph, which is an undirected graph in which the nodes represent bodies and the arcs represent joints. A couple of examples are shown in Figure 2. If the robot is a kinematic tree, then its connectivity graph is a topological tree. To describe the connectivity of a kinematic tree, we first number the bodies and joints according to a standard scheme. For a robot having a fixed base, the numbering proceeds as follows:

1) The fixed base is assigned the number 0 and serves as the root node of the tree.

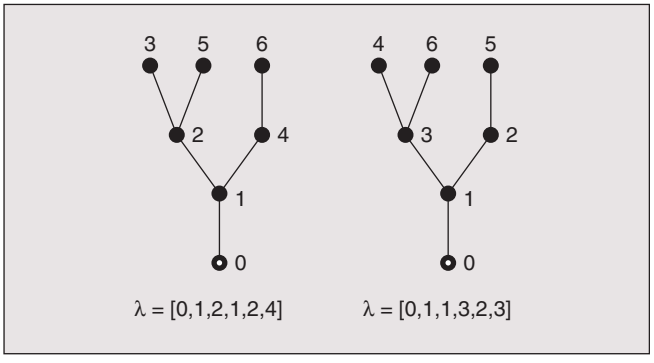

Figure 2. *Two numberings of a simple tree and their corresponding parent arrays.*

2) The remaining bodies are numbered consecutively from 1 to N in any order such that each body has a higher number than its parent.
3) The joints are then numbered from 1 to N such that joint i is the joint that connects body i to its parent.

Having numbered the bodies and joints, the connectivity can be described using a parent array, called λ, which is defined such that $\lambda(i)$ is the body number of the parent of body i. This array has the following special property, which is a consequence of the numbering scheme:

$$0 \leq \lambda(i) < i \qquad \text{for all } 1 \leq i \leq N. \tag{2}$$

Many algorithms rely on this property. Note that neither the numbering nor the parent array is unique. To illustrate this, Figure 2 shows two possible numberings of the same graph and their corresponding parent arrays. For the tree on the left, we have $\lambda(1) = 0$, $\lambda(2) = 1$, $\lambda(3) = 2$, and so on, indicating that body 0 is the parent of body 1, body 1 is the parent of body 2, and so on. Joint numbers have not been shown, because they can be deduced from the body numbers: joint i connects between bodies i and $\lambda(i)$.

To model a mobile robot, we first place a Cartesian coordinate frame at any convenient fixed location in space. This frame serves as a virtual fixed base. We then introduce a six-degree-of-freedom (six-DoF) joint between the fixed base and any one body of the mobile robot. The chosen body is then called the floating base. Now, a six-DoF joint does not introduce any kinematic constraints, so it does not restrict the mobility of the mobile robot. Instead, its purpose is to supply the extra variables needed to identify the position and orientation of the robot relative to its virtual fixed base. Having made these two modifications, the mobile robot can be treated as a fixed-base robot and numbered as described above. (The floating base will therefore be body number 1.)

Although λ alone already provides a complete description of the connectivity, it is often helpful to supplement λ with the following sets:

- $\mu(i)$: the set of children of body i,
- $\kappa(i)$: the set of joints on the path between body i and the root, and
- $\nu(i)$: the set of bodies in the subtree starting at body i.

For the left-hand tree in Figure 2, we have $\mu(2) = \{3, 5\}$, $\kappa(3) = \{1, 2, 3\}$, $\nu(4) = \{4, 6\}$, $\mu(5) = \emptyset$ (the empty set), and so on.

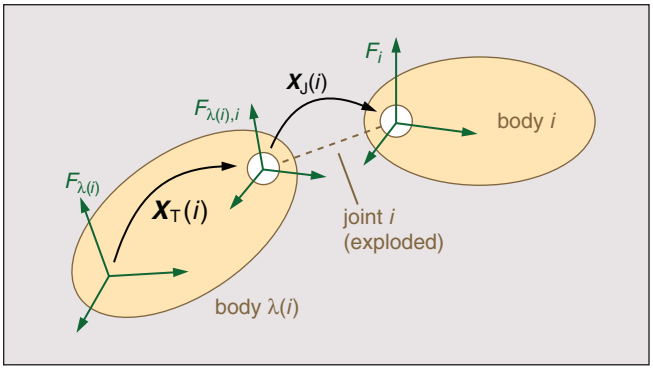

Figure 3. *Coordinate frames and transforms associated with joint i.*

Geometry

The geometrical part of a robot model is the part that specifies the relative locations of the joints in each body. It is also the part that defines a link coordinate system for each body so that quantities like `model.I{i}` can be expressed and stored in link-i coordinates. (Link is the technical term for a body in a mechanical linkage, so link and body can be used interchangeably here.)

To describe the geometry, the first step is to introduce a pair of coordinate frames for each joint: one fixed in each of the two bodies connected by the joint. For joint i, which connects between bodies i and $\lambda(i)$, we introduce a frame F_i that is fixed in body i and a frame $F_{\lambda(i),i}$ that is fixed in body $\lambda(i)$ (see Figure 3). We also introduce a special frame, F_0, which is fixed in body 0 and which serves as an absolute, world, or reference frame (take your pick) for the whole robot. For a mobile robot, F_0 is the frame that was introduced earlier to serve as a virtual fixed base.

The frames can be located anywhere in their respective bodies, provided they satisfy the following rules:

1) Frames F_i and $F_{\lambda(i),i}$ must coincide when the joint variable of joint i is zero.
2) Frames F_i and $F_{\lambda(i),i}$ must comply with the joint-specific alignment requirements of joint i.

As an example of rule 2, if joint i is revolute or helical, then the z-axes of F_i and $F_{\lambda(i),i}$ must lie on the joint's rotation or screw axis. If, instead, joint i is prismatic, then the two z-axes must be parallel to the joint's direction of translation. The purpose of this rule is to ensure that the joint coordinate transform (labeled $\boldsymbol{X}_J(i)$ in Figure 3) takes a canonical form for each joint type. For a revolute joint, $\boldsymbol{X}_J(i)$ is a pure rotation about the z-axis. (More information on this topic is provided in the next subsection.)

The above rules stipulate only the minimum necessary constraints on the placement of coordinate frames and do not constrain them completely. It is therefore possible to introduce additional rules for the purpose of further constraining their locations. The most well-known example is the scheme of Denavit and Hartenberg, which has the special property that the location of F_i relative to $F_{\lambda(i)}$ is a function of only four parameters, one of which serves as the joint variable [1], [3], [9].

At the end of this process, there are $2N + 1$ frames in total, of which $N + 1$ have names of the form F_i, and N have names of the form $F_{i,j}$, where $j \in \mu(i)$ (which is the same condition as $i = \lambda(j)$). Every body in the system, including the fixed base, now contains exactly one frame F_i plus a variable number of frames $F_{i,j}$, one for each $j \in \mu(i)$. At this point, we select F_i to define the link coordinate system for body i, i.e., link-i coordinates. Thus, the spatial inertia stored in `model.I{i}` is expressed in the (Plücker) coordinate system defined by frame F_i.

A complete description of the robot's geometry can now be obtained as follows. Let $\boldsymbol{X}_T(i)$ be the Plücker coordinate transform from link-$\lambda(i)$ coordinates to the coordinate system defined

by frame $F_{\lambda(i),i}$, as shown in Figure 3. Now, a Plücker transform implicitly describes the relative locations of two coordinate frames; so $\boldsymbol{X}_{\mathrm{T}}(i)$ serves to locate $F_{\lambda(i),i}$ relative to F_i, and a complete set of transforms, $\boldsymbol{X}_{\mathrm{T}}(1), \ldots, \boldsymbol{X}_{\mathrm{T}}(N)$, serves to locate every $F_{i,j}$ relative to its corresponding F_i. These transforms are stored in the array `model.Xtree`, so that `model.Xtree{i}` $= \boldsymbol{X}_{\mathrm{T}}(i)$.

Let ${}^{i}\boldsymbol{X}_{\lambda(i)}$ denote the Plücker coordinate transform from link-$\lambda(i)$ to link-i coordinates for motion vectors (the corresponding transform for force vectors is ${}^{i}\boldsymbol{X}^{*}_{\lambda(i)}$). This transform locates F_i relative to $F_{\lambda(i)}$ and therefore locates body i relative to body $\lambda(i)$. From Figure 3, we can see that

$$ {}^{i}\boldsymbol{X}_{\lambda(i)} = \boldsymbol{X}_{\mathrm{J}}(i)\,\boldsymbol{X}_{\mathrm{T}}(i). \tag{3} $$

Equation (3) brings together the connectivity data, $\lambda(i)$, the geometry data, $\boldsymbol{X}_{\mathrm{T}}(i)$, and the joint data, via $\boldsymbol{X}_{\mathrm{J}}(i)$, to express the relative locations of adjacent bodies as a function of the joint position variables. This calculation appears on line 5 of Figure 1, where ${}^{i}\boldsymbol{X}_{\lambda(i)}$ is calculated and stored in the variable `Xup{i}`. Quantities like ${}^{i}\boldsymbol{X}_{\lambda(i)}$ are called link-to-link coordinate transforms.

Joint Models

A joint is a kinematic constraint between two bodies. To identify them individually, we call one body the predecessor and the other the successor. In a kinematic tree, the predecessor of joint i is body $\lambda(i)$, and its successor is body i. By convention, we define the velocity across a joint to be the velocity of the successor relative to the predecessor, and the force across a joint to be a force transmitted from the predecessor to the successor. Thus, if $\boldsymbol{v}_{\mathrm{J}i}$ and $\boldsymbol{f}_{\mathrm{J}i}$ are the spatial velocity and force across joint i, then

$$ \boldsymbol{v}_{\mathrm{J}i} = \boldsymbol{v}_i - \boldsymbol{v}_{\lambda(i)}, \tag{4} $$

where $\boldsymbol{v}_i$ is the velocity of body i, and $\boldsymbol{f}_{\mathrm{J}i}$ is the force transmitted from body $\lambda(i)$ to body i through the joint.

A mathematical model of a joint consists of two quantities: a coordinate transform, $\boldsymbol{X}_{\mathrm{J}}$, and a motion subspace matrix, $\boldsymbol{S}$ (also known as a free-modes matrix). For joint i, $\boldsymbol{X}_{\mathrm{J}}(i)$ is the coordinate transform from $F_{\lambda(i),i}$ to F_i, as shown in Figure 3, and $\boldsymbol{S}_i$ defines the following relationships between the joint variables and spatial vectors:

$$ \boldsymbol{v}_{\mathrm{J}i} = \boldsymbol{S}_i\,\dot{\boldsymbol{q}}_{\mathrm{i}} \tag{5} $$

and

$$ \boldsymbol{\tau}_i = \boldsymbol{S}_i^{\mathrm{T}}\boldsymbol{f}_{\mathrm{J}i}, \tag{6} $$

where $\dot{\boldsymbol{q}}_i$ and $\boldsymbol{\tau}_i$ are the subvectors of $\dot{\boldsymbol{q}}$ and $\boldsymbol{\tau}$ that contain the velocity and force variables, respectively, for joint i. We can see an instance of (5) and (6) on lines 4 and 17, respectively, in Figure 1. Incidentally, $\dot{\boldsymbol{q}}_i$ and $\boldsymbol{\tau}_i$ also satisfy

$$ \boldsymbol{\tau}_i \cdot \dot{\boldsymbol{q}}_i = \boldsymbol{f}_{\mathrm{J}i} \cdot \boldsymbol{v}_{\mathrm{J}i}, \tag{7} $$

which is known as the power-balance equation. The scalar on the left is the mechanical power delivered to the robot mechanism at joint i, expressed in terms of joint variables, and the scalar on the right is the same physical quantity expressed in terms of spatial vectors. The two are necessarily equal.

```
function  [XJ,s] = jcalc( pitch, q )
if pitch == 0           % revolute joint
  XJ = rotz(q);
  s = [0;0;1;0;0;0];
elseif pitch == inf     % prismatic joint
  XJ = xlt([0 0 q]);
  s = [0;0;0;0;0;1];
else                    % helical joint
  XJ = rotz(q)*xlt([0 0 q*pitch]);
  s = [0;0;1;0;0;pitch];
end
```

Figure 4. *MATLAB code for function* `jcalc`.

A computational model of a joint consists of a piece of code (such as `jcalc` in Figure 4) that computes the numeric values of $\boldsymbol{X}_{\mathrm{J}}$ and $\boldsymbol{S}$ as a function of the numeric values of the joint variables and parameters (if any). If $\boldsymbol{S}$ varies as a function of the joint's position variables, then it is also necessary to compute the numeric value of a term that depends on $\partial \boldsymbol{S}/\partial \boldsymbol{q}$ (see $\boldsymbol{c}_{\mathrm{J}}$ on p. 80 of [3]).

If joint i permits n_i degrees of motion freedom, then $\boldsymbol{S}_i$ is a $6 \times n_i$ matrix and $\dot{\boldsymbol{q}}_i$ is an $n_i \times 1$ vector. The total number of joint variables is then

$$ n = \sum_{i=1}^{N} n_i\,. \tag{8} $$

This is the dimension of the vectors in (1). However, in this tutorial, we have chosen to limit the repertoire of joint types to revolute, prismatic, and helical. These are all single-DoF joints, so we have $n_i = 1$ for every joint in the mechanism and therefore also $n = N$. Two more simplifications are:

1) the motion subspace matrix simplifies to a joint axis vector $\boldsymbol{s}_i$, and
2) the variables for joint i are the ith elements of their corresponding joint-space vectors.

Item 2 refers to expressions like `q(i)` on line 3 of Figure 1 and `qd(i)` on line 4. These expressions simply extract the ith element of `q`, `qd`, etc. In the general case, the variables for joint i would be n_i-dimensional subvectors of `q`, `qd`, etc., and expressions such as `q(i)` and `qd(i)` would have to be replaced with something a little more complicated.

Another simplification is that revolute and prismatic joints can be regarded as helical joints having zero pitch and infinite pitch, respectively; so it is possible to use the array `model.pitch` both to define the type of each joint and to supply the pitch parameter for each helical joint. This tactic can be seen in the source code of `jcalc`, which is shown in Figure 4. As you can see from this code, a revolute joint implements a pure rotation about the (local) z-axis, a prismatic joint implements a pure translation in the z-direction, and a helical joint implements a screwing motion about the z-axis, in which the pitch

Table 1. Instant spatial vector arithmetic (based on Table A.2 of [3]).

3-D vector formulae $(c = \cos(\theta),\ s = \sin(\theta))$

$$\mathrm{rx}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & s \\ 0 & -s & c \end{bmatrix} \qquad \mathrm{rz}(\theta) = \begin{bmatrix} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathrm{ry}(\theta) = \begin{bmatrix} c & 0 & -s \\ 0 & 1 & 0 \\ s & 0 & c \end{bmatrix} \qquad \boldsymbol{r}\times = \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}$$

function $\boldsymbol{X} = \mathrm{rotx}(\theta)$
$\boldsymbol{E} = \mathrm{rx}(\theta)$
$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{E} & \boldsymbol{0}_{3\times3} \\ \boldsymbol{0}_{3\times3} & \boldsymbol{E} \end{bmatrix}$
end

function $\boldsymbol{X} = \mathrm{roty}(\theta)$
$\boldsymbol{E} = \mathrm{ry}(\theta)$
$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{E} & \boldsymbol{0}_{3\times3} \\ \boldsymbol{0}_{3\times3} & \boldsymbol{E} \end{bmatrix}$
end

function $\boldsymbol{X} = \mathrm{rotz}(\theta)$
$\boldsymbol{E} = \mathrm{rz}(\theta)$
$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{E} & \boldsymbol{0}_{3\times3} \\ \boldsymbol{0}_{3\times3} & \boldsymbol{E} \end{bmatrix}$
end

function $\boldsymbol{X} = \mathrm{xlt}(\boldsymbol{r})$
$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{1}_{3\times3} & \boldsymbol{0}_{3\times3} \\ -\boldsymbol{r}\times & \boldsymbol{1}_{3\times3} \end{bmatrix}$
end

function $\boldsymbol{v}\times = \mathrm{crm}(\boldsymbol{v})$
$\boldsymbol{v}\times = \begin{bmatrix} \boldsymbol{v}_{1:3}\times & \boldsymbol{0}_{3\times3} \\ \boldsymbol{v}_{4:6}\times & \boldsymbol{v}_{1:3}\times \end{bmatrix}$
end

function $\boldsymbol{v}\times^* = \mathrm{crf}(\boldsymbol{v})$
$\boldsymbol{v}\times^* = -\mathrm{crm}(\boldsymbol{v})^{\mathrm{T}}$
end

function $\boldsymbol{I} = \mathrm{mcI}(m, \boldsymbol{c}, \boldsymbol{I}_C)$
$\boldsymbol{I} = \begin{bmatrix} \boldsymbol{I}_C - m\,\boldsymbol{c}\times\boldsymbol{c}\times & m\,\boldsymbol{c}\times \\ -m\,\boldsymbol{c}\times & m\boldsymbol{1}_{3\times3} \end{bmatrix}$
end

function $\boldsymbol{v} = \mathrm{XtoV}(\boldsymbol{X})$
$\boldsymbol{v} = \frac{1}{2}\begin{bmatrix} X_{23} - X_{32} \\ X_{31} - X_{13} \\ X_{12} - X_{21} \\ X_{53} - X_{62} \\ X_{61} - X_{43} \\ X_{42} - X_{51} \end{bmatrix}$
end

parameter determines the lead (per radian) of the screw. The functions `rotz` and `xlt` are defined in Table 1.

To represent a broader range of joint types, we could replace `model.pitch` with an array of joint-descriptor data structures: each descriptor contains a joint-type code and zero or more parameter values, the number of parameters being determined by the type code. More information on joint models can be found in [3], [4], and [8].

The code in Figure 1 shows a clear separation between the code that implements the dynamics algorithm and the code that handles joint-dependent calculations, the latter being hived off into the function `jcalc`. This organizational feature is standard practice with 6-D vectors, but it is harder to achieve using 3-D vectors. As a result, algorithms that are expressed using 3-D vectors tend to be restricted to revolute and prismatic joints, and their descriptions typically contain many statements of the form

if joint type is revolute **then**
  variable = one expression
else
  variable = another expression.

(For example, see the original description of the recursive Newton–Euler algorithm in [7].) Clearly, it is a nontrivial exercise to extend such an algorithm to accommodate a third joint type. This intertwining of algorithms with joint-specific details is an impediment to the development of clean, extensible, general-purpose code, and it is yet another reason to prefer 6-D vectors over 3-D vectors.

Algorithm

We now return to the code in Figure 1. As mentioned earlier, this code implements the recursive Newton–Euler algorithm for calculating inverse dynamics. The equations for this algorithm, expressed using spatial vectors, are as follows:

$$\boldsymbol{v}_i = \boldsymbol{v}_{\lambda(i)} + \boldsymbol{s}_i\dot{\boldsymbol{q}}_i \qquad (\boldsymbol{v}_0 = \boldsymbol{0}) \tag{9}$$

$$\boldsymbol{a}_i = \boldsymbol{a}_{\lambda(i)} + \boldsymbol{s}_i\ddot{\boldsymbol{q}}_i + \boldsymbol{v}_i \times \boldsymbol{s}_i\dot{\boldsymbol{q}}_i \qquad (\boldsymbol{a}_0 = -\boldsymbol{a}_g) \tag{10}$$

$$\boldsymbol{f}_{\mathrm{B}i} = \boldsymbol{I}_i\boldsymbol{a}_i + \boldsymbol{v}_i \times^* \boldsymbol{I}_i\boldsymbol{v}_i \tag{11}$$

$$\boldsymbol{f}_{\mathrm{J}i} = \boldsymbol{f}_{\mathrm{B}i} + \sum_{j\in\mu(i)} \boldsymbol{f}_{\mathrm{J}j} \tag{12}$$

$$\tau_i = \boldsymbol{s}_i^{\mathrm{T}}\boldsymbol{f}_{\mathrm{J}i} \tag{13}$$

Equation (9) states that the velocity of body i is the sum of the velocity of its parent and the velocity across joint i [cf. (4) and (5)]. Equation (10) says the same for accelerations and is simply the derivative of (9). Observe that $\dot{\boldsymbol{s}}_i = \boldsymbol{v}_i \times \boldsymbol{s}_i$, because $\boldsymbol{s}_i$ is fixed in body i. The starting condition for (10) is $\boldsymbol{a}_0 = -\boldsymbol{a}_g$, where $\boldsymbol{a}_g$ is the acceleration caused by gravity. This is a trick that exploits the fact that a uniform gravitational field is indistinguishable from a constant linear acceleration. Therefore, instead of calculating the gravitational force acting on each body and incorporating those forces into (11), we can simply offset every body's spatial acceleration by giving the fixed base a fictitious acceleration of $-\boldsymbol{a}_g$ (see line 9 in Figure 1).

In (11)–(13), $\boldsymbol{f}_{\mathrm{B}i}$ is the net force acting on body i, and $\boldsymbol{f}_{\mathrm{J}i}$ is the force transmitted across joint i. As mentioned earlier, the spatial force across a joint is defined to be a force transmitted from its predecessor body to its successor; so $\boldsymbol{f}_{\mathrm{J}i}$ is a force transmitted from body $\lambda(i)$ to body i. In other words, joint i is causing a force of $+\boldsymbol{f}_{\mathrm{J}i}$ to act on body i and a force of $-\boldsymbol{f}_{\mathrm{J}i}$ to act on body $\lambda(i)$.

The equation of motion for body i is given by (11). As the accelerations are already known, the purpose of this equation is to work out the force required to produce the given acceleration. Equation (12) then calculates the (spatial) joint forces from the body forces. It works as follows: $\boldsymbol{f}_{\mathrm{B}i}$ is the net force acting on body i, so it must be the sum of all the individual forces acting on body i. Having accounted for gravity by means of a fictitious acceleration, instead of a gravitational force field, the only force acting on body i is those transmitted to it via the joints. So, $\boldsymbol{f}_{\mathrm{B}i} = \boldsymbol{f}_{\mathrm{J}i} + \sum_{j\in\mu(i)} (-\boldsymbol{f}_{\mathrm{J}j})$. A small rearrangement of this equation yields (12). Finally, (13) calculates the joint force variable for joint i from the spatial force transmitted across the joint, per (6).

Equations (9)–(13) provide a mathematical description of the recursive Newton–Euler algorithm; they describe the algorithm in principle but omit some calculation details. Translating these equations into a more explicit description of

the algorithm produces the pseudocode shown in Figure 5. On comparing the pseudocode with the equations, we can see that the following things have changed.

1) The vectors in (9)–(13) are tacitly assumed to be expressed in a single common coordinate system; however, their counterparts in Figure 5 are expressed in link coordinates, and the necessary link-to-link coordinate transforms (${}^{i}\boldsymbol{X}_{\lambda(i)}$ and ${}^{\lambda(i)}\boldsymbol{X}_{i}^{*}$) have been incorporated into lines 6, 7, and 13.
2) The method of calculating the link-to-link transforms has been made explicit in lines 4 and 5.
3) The sum over $\mu(i)$ in (12) has been replaced with code that performs the same summation but using $\lambda(i)$ instead.

Item 3 refers to lines 8 and 12–14. Line 8 initializes every variable $\boldsymbol{f}_i$ to have the value of $\boldsymbol{f}_{Bi}$ as given by (11). However, by the time $\boldsymbol{f}_i$ is used on line 11 to calculate τ_i, its value is equal to $\boldsymbol{f}_{Ji}$ as given in (12). The change is effected by lines 12–14, which add each vector $\boldsymbol{f}_i$ ($=\boldsymbol{f}_{Ji}$) back to its parent. By the time $\boldsymbol{f}_i$ is used on line 11, the contributions from all of its children have already been added in.

On comparing the pseudocode in Figure 5 with the MATLAB source code in Figure 1, it can be seen that the translation from pseudocode to source code is entirely straightforward. Again, look how short everything is: five equations have given rise to 15 lines of pseudocode and 22 lines of source code. This degree of brevity would not be possible using 3-D vectors. Of course, if you want source code in a language such as C or C++, then the code will be somewhat longer; however, the translation from pseudocode to source code will still be straightforward, provided you have access to a suitable library of spatial arithmetic functions.

Arithmetic

To perform arithmetic with spatial vectors, you need a spatial arithmetic library. Most arithmetic operations on spatial vectors are just standard matrix arithmetic. Therefore, if you are using a programming language that already has matrix arithmetic built in (such as MATLAB or Octave), then only a small number of additional functions are needed. Table 1 presents a small but sufficient set.

The functions `rotx`, `roty`, `rotz`, and `xlt` construct Plücker coordinate transforms (for motion vectors) from a current coordinate system to one that has been rotated or translated, as appropriate, relative to the current one. Examples of their use can be found in Figure 4. Note that the formulae listed for `rx`, `ry`, and `rz` are coordinate rotation matrices; they rotate the coordinate system in which the vector is represented. In many robotics textbooks (e.g., [1], p. 372), you will find formulae for rotation matrices that rotate the vector itself. These two types of matrix are inverses of each other.

The functions `crm` and `crf` implement the two spatial cross-product operators. Examples of their use can be found in Figure 1. The symbols $\boldsymbol{v}\times$ and $\boldsymbol{v}\times^{*}$ appearing in these two functions are the names of the return values. The expressions $\boldsymbol{v}_{1:3}$ and $\boldsymbol{v}_{4:6}$ are 3-D vectors formed from the first and last three Plücker coordinates of $\boldsymbol{v}$.

```
1  v0 = 0
2  a0 = -ag
3  for i = 1 to N do
4     [XJ, Si] = jcalc(hi,qi)
5     iXλ(i) = XJXT(i)
6     vi = iXλ(i)vλ(i) + sidq̇i
7     ai = iXλ(i)aλ(i) + sidq̈i + vi ×sidq̇i
8     fi = Iiai + vi ×*Iivi
9  end
10 for i = N to 1 do
11    τi  = Si^T fi
12    If λ(i) ≠ 0 then
13       fλ(i) = fλ(i) + λ(i)Xi* fi
14    end
15 end
```

Figure 5. *The recursive Newton–Euler algorithm.*

The function `mcI` constructs a spatial rigid-body inertia from arguments giving the body's mass (m), the position of its center of mass ($\boldsymbol{c}$), and its rotational inertia about its center of mass ($\boldsymbol{I}_C$). You would use this function to initialize the inertia matrices in a robot model data structure.

Finally, `XtoV` calculates a small-magnitude motion vector from the Plücker transform for a small change of coordinates. If A and B denote two Cartesian frames, and also the Plücker coordinate systems defined by those frames, then we can define `XtoV` as follows: if A and B are close together, and $\boldsymbol{X}$ is the Plücker coordinate transform from A to B, then `XtoV` ($\boldsymbol{X}$) approximates to the velocity vector that would move frame A to coincide with B after one time unit. The returned value also happens to be an invariant of $\boldsymbol{X}$ (i.e., $\boldsymbol{v}=\boldsymbol{X}\boldsymbol{v}$) so it has the same value in both A and B coordinates. An example of this function's use appears in the next section.

If you want to perform spatial arithmetic in a programming language such as C or C++, then you will need a more extensive library. Some guidelines on how to build such a library, and a collection of formulae for implementing highly efficient spatial arithmetic, can be found in [3], and a less comprehensive version appears in [3]. Implementations of the functions in Table 1 can be found in [5].

Kinematics

Spatial vectors can be used both for positional kinematics and for instantaneous kinematics. We have already encountered the latter in (9) and (10), which present recursive formulae for calculating body velocities and accelerations from joint velocity and acceleration variables. Body positions can be calculated recursively using the formula

$$ {}^{i}\boldsymbol{X}_{0} = {}^{i}\boldsymbol{X}_{\lambda(i)}\,{}^{\lambda(i)}\boldsymbol{X}_{0}, \qquad (\lambda(i)\neq 0) \tag{14} $$

which calculates the coordinate transform from reference coordinates (frame F_0) to the body coordinate frame (F_i) of each body in the mechanism. As mentioned earlier, a coordinate transform implicitly defines the relative locations of two

coordinate frames; so ${}^{i}\boldsymbol{X}_0$ effectively locates F_i, and therefore also body i, relative to F_0.

Suppose we want to move a particular body, say body b, so that its body coordinate frame, F_b, coincides with a given target frame, F_{d}. We can express the location of F_{d} by means of the coordinate transform ${}^{\mathrm{d}}\boldsymbol{X}_0$, which is assumed to be given. If F_b is already close to F_{d}, then we can use the function XtoV in Table 1 to calculate a small displacement, $\Delta\boldsymbol{p}$, as follows:

$$\Delta\boldsymbol{p} = \mathrm{XtoV}({}^{\mathrm{d}}\boldsymbol{X}_b) = \mathrm{XtoV}({}^{\mathrm{d}}\boldsymbol{X}_0\ {}^{0}\boldsymbol{X}_b). \tag{15}$$

This vector approximates to the small screw displacement that would bring F_b into coincidence with F_{d}, the error in the approximation diminishing quadratically with the angular magnitude of $\Delta\boldsymbol{p}$. Alternatively, $\Delta\boldsymbol{p}$ can be regarded as approximating the velocity that would bring F_b into coincidence with F_{d} in one time unit. As $\Delta\boldsymbol{p}$ is an invariant of ${}^{\mathrm{d}}\boldsymbol{X}_b$, it has the same value in both F_b and F_{d} coordinates.

```
 1  function  q = IKpos( model, body, Xd, q0 )
 2  q = q0;
 3  dpos = ones(6,1);
 4  while norm(dpos) > 1e-10
 5    X = bodypos( model, body, q );
 6    J = bodyJac( model, body, q );
 7    J = X * J;
 8    dpos = XtoV( Xd / X );
 9    dq = J' * ((J*J') \ dpos);
10    q = q + dq;
11  end
```

```
 1  function  X = bodypos( model, b, q )
 2  X = eye(6);
 3  while b > 0
 4    XJ = jcalc( model.pitch(b), q(b) );
 5    X = X * XJ * model.Xtree{b};
 6    b = model.parent(b);
 7  end
```

```
 1  function  J = bodyJac( model, body, q )
 2  e = zeros(1,model.N);
 3  while body ~= 0
 4    e(body) = 1;
 5    body = model.parent(body);
 6  end
 7  J = zeros(6,model.N);
 8  for i = 1:model.N
 9    if e(i)
10      [XJ,S] = jcalc( model.pitch(i), q(i) );
11      Xa{i} = XJ * model.Xtree{i};
12      if model.parent(i) ~= 0
13        Xa{i} = Xa{i} * Xa{model.parent(i)};
14      end
15      J(:,i) = Xa{i} \ S;
16    end
17  end
```

Figure 6. *MATLAB code for iterative inverse kinematics.*

If F_{d} is a reachable position for body b, then there will be a joint position change, $\Delta\boldsymbol{q}$, that causes a displacement of $\Delta\boldsymbol{p}$ in body b. In general, $\Delta\boldsymbol{q}$ will not be unique. The exact relationship between $\Delta\boldsymbol{p}$ and $\Delta\boldsymbol{q}$ may be difficult to obtain, but a first-order approximation is given by

$$\Delta\boldsymbol{p} = {}^{b}\boldsymbol{J}_b\,\Delta\boldsymbol{q}, \tag{16}$$

where ${}^{b}\boldsymbol{J}_b$ is the Jacobian for body b expressed in b coordinates (Jacobians are discussed in the next section). Equations (15) and (16) form the basis for an iterative inverse-kinematics algorithm as follows:

while not close enough **do**
  calculate ${}^{\mathrm{d}}\boldsymbol{X}_b$ and ${}^{b}\boldsymbol{J}_b$
  $\Delta\boldsymbol{p} = \mathrm{XtoV}({}^{\mathrm{d}}\boldsymbol{X}_b)$
  $\Delta\boldsymbol{q} = {}^{b}\boldsymbol{J}_b^{+}\Delta\boldsymbol{p}$
  $\boldsymbol{q} = \boldsymbol{q} + \Delta\boldsymbol{q}$
end,

where ${}^{b}\boldsymbol{J}_b^{+}$ is the pseudoinverse of ${}^{b}\boldsymbol{J}_b$.

Some MATLAB source code to implement this calculation is shown in Figure 6. The variables `q0` and `q` are the initial guess and computed final value of $\boldsymbol{q}$, and the variables `body` and `Xd` contain b and ${}^{\mathrm{d}}\boldsymbol{X}_0$. Line 3 sets `dpos` (= $\Delta\boldsymbol{p}$) to a dummy value that will pass the test on line 4, and the functions `bodypos` and `bodyJac` calculate ${}^{b}\boldsymbol{X}_0$ and ${}^{0}\boldsymbol{J}_b$, respectively. Line 7 calculates ${}^{b}\boldsymbol{J}_b$ from ${}^{0}\boldsymbol{J}_b$, line 8 calculates $\Delta\boldsymbol{p}$, and line 9 calculates $\Delta\boldsymbol{q}$ using the pseudoinverse of ${}^{b}\boldsymbol{J}_b$. Bear in mind that this is not a serious inverse-kinematics function, as it fails to check for a variety of things that can go wrong, such as singularities and unreachable positions.

Jacobians

In common robotics usage, a Jacobian is a matrix that maps the joint-space velocity vector, $\dot{\boldsymbol{q}}$, to some other kind of velocity. We have used the term body Jacobian, and the symbol $\boldsymbol{J}_b$, to refer to the matrix that maps $\dot{\boldsymbol{q}}$ to the spatial velocity of body b, as in

$$\boldsymbol{v}_b = \boldsymbol{J}_b\dot{\boldsymbol{q}}. \tag{17}$$

To obtain a formula for $\boldsymbol{J}_b$, we first express $\boldsymbol{v}_b$ in nonrecursive form:

$$\boldsymbol{v}_b = \sum_{i\in\kappa(b)} \boldsymbol{s}_i\dot{\boldsymbol{q}}_i. \tag{18}$$

This equation simply states that $\boldsymbol{v}_b$ is the sum of the joint velocities of all the joints on the path between body b and the fixed base. Rewriting this equation as

$$\boldsymbol{v}_b = \sum_{i=1}^{N} e_{bi}\boldsymbol{s}_i\dot{\boldsymbol{q}}_i, \tag{19}$$

where

$$e_{bi} = \begin{cases} 1 & \text{if } i \in \kappa(b) \\ 0 & \text{otherwise} \end{cases} \tag{20}$$

yields the following expression for $\boldsymbol{J}_b$:

$$\boldsymbol{J}_b = \begin{bmatrix} e_{b1}\boldsymbol{s}_1 & e_{b2}\boldsymbol{s}_2 & \cdots & e_{bN}\boldsymbol{s}_N \end{bmatrix}. \tag{21}$$

Thus, the body Jacobian for body b is the $6 \times N$ matrix whose ith column is either $\boldsymbol{s}_i$ or zero, depending on whether joint i is or is not on the path between the fixed base and body b. More generally, a body Jacobian is a $1 \times N$ block matrix in which the ith block is the $6 \times n_i$ matrix ${}_{ei}\boldsymbol{S}_i$, and the overall dimension of the Jacobian is $6 \times n$ [cf. (8)]. The code for `bodyJac` in Figure 6 should now make sense: lines 2–6 calculate e_{bi} and lines 8–17 calculate the nonzero columns. For the special case where b is the end effector of a serial robot, we have $e_{bi} = 1$ for all i, which implies the following simplified formula for the end-effector Jacobian:

$$\boldsymbol{J}_{\text{ee}} = \begin{bmatrix} \boldsymbol{s}_1 & \boldsymbol{s}_2 & \dots & \boldsymbol{s}_N \end{bmatrix}. \tag{22}$$

If we need to be explicit about the coordinate system, then (17) and (21) can be written as

$${}^{A}\boldsymbol{v}_b = {}^{A}\boldsymbol{J}_b\,\dot{\boldsymbol{q}} \tag{23}$$

and

$${}^{A}\boldsymbol{J}_b = \begin{bmatrix} e_{b1}{}^{A}\boldsymbol{s}_1 & e_{b2}{}^{A}\boldsymbol{s}_2 & \dots & e_{bN}{}^{A}\boldsymbol{s}_N \end{bmatrix}, \tag{24}$$

where A is the name of a coordinate system. These equations show that every column of a body Jacobian must be expressed in the same coordinate system as the velocity vector it maps to. The coordinate transformation rule for a body Jacobian is therefore

$${}^{B}\boldsymbol{J}_b = {}^{B}\boldsymbol{X}_A\,{}^{A}\boldsymbol{J}_b. \tag{25}$$

Still on the subject of coordinate systems, here is a popular trap for the unwary. You will occasionally encounter an equation of the form

$$\begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v} \end{bmatrix} = \boldsymbol{J}\dot{\boldsymbol{q}}, \tag{26}$$

where $\boldsymbol{\omega}$ and $\boldsymbol{v}$ are described as the angular and linear velocity of the end effector (or some other body) expressed in absolute, reference, or base coordinates (i.e., the Cartesian coordinate system defined by frame F_0). In translating this equation from 3-D to spatial vectors, it is tempting to regard the left-hand side as being the Plücker coordinates of a spatial velocity expressed in frame F_0. However, this is nearly always incorrect, because the 3-D vector $\boldsymbol{v}$ nearly always refers to some particular point in the end effector, such as the tool center point, which does not coincide with the origin of F_0. The correct translation is this: the left-hand side of (26) contains the Plücker coordinates of the spatial velocity of the end effector expressed in a coordinate system that is parallel to absolute coordinates but has its origin at the particular point in the end effector to which $\boldsymbol{v}$ refers.

Jacobians can also map forces. If a robot makes contact with its environment through body b, and the environment responds by exerting a force of $\boldsymbol{f}_{\text{e}}$ on body b, then the effect of that force on the robot is equivalent to a joint-space force of $\boldsymbol{\tau}_{\text{e}}$ given by

$$\boldsymbol{\tau}_{\text{e}} = \boldsymbol{J}_b^{\text{T}}\boldsymbol{f}_{\text{e}}. \tag{27}$$

The robot's control system can resist this force by adding $-\boldsymbol{\tau}_{\text{e}}$ to its joint-force command. This works because $\boldsymbol{f}_{\text{e}}$ acting on body b has the same effect on the robot as $\boldsymbol{\tau}_{\text{e}}$ acting at the joints, and the two forces $+\boldsymbol{\tau}_{\text{e}}$ and $-\boldsymbol{\tau}_{\text{e}}$ cancel. In applications like this, it is important to be clear and unambiguous about whether a force is being exerted by the environment on the robot or the other way around. In this example, the environment exerts a force of $\boldsymbol{f}_{\text{e}}$ on the robot, and the robot exerts a force of $-\boldsymbol{f}_{\text{e}}$ on the environment.

Acceleration

Equation (10) provides us with a recursive formula for calculating body accelerations. A nonrecursive formula can be obtained by differentiating (18):

$$\boldsymbol{a}_b = \sum_{i \in \kappa(b)} (\boldsymbol{s}_i\ddot{\boldsymbol{q}}_i + \dot{\boldsymbol{s}}_i\dot{\boldsymbol{q}}_i). \tag{28}$$

If we assume that $\dot{\boldsymbol{s}}_i = \boldsymbol{v}_i \times \boldsymbol{s}_i$, then this equation can be further expanded to

$$\begin{aligned} \boldsymbol{a}_b &= \sum_{i \in \kappa(b)} \boldsymbol{s}_i\ddot{\boldsymbol{q}}_i + \sum_{i \in \kappa(b)} \Big(\sum_{j \in \kappa(i)} \boldsymbol{s}_j\dot{\boldsymbol{q}}_j \Big) \times \boldsymbol{s}_i\dot{\boldsymbol{q}}_i \\ &= \sum_{i \in \kappa(b)} \boldsymbol{s}_i\ddot{\boldsymbol{q}}_i + \sum_{i \in \kappa(b)} \sum_{j \in \kappa(i)} \boldsymbol{s}_j\dot{\boldsymbol{q}}_j \times \boldsymbol{s}_i\dot{\boldsymbol{q}}_i. \end{aligned} \tag{29}$$

It is sometimes useful to define a velocity-product acceleration, $\boldsymbol{a}_b^{\text{vp}}$, equal to the velocity terms on the right-hand side:

$$\boldsymbol{a}_b^{\text{vp}} = \sum_{i \in \kappa(b)} \sum_{j \in \kappa(i)} \boldsymbol{s}_j\dot{\boldsymbol{q}}_j \times \boldsymbol{s}_i\dot{\boldsymbol{q}}_i. \tag{30}$$

In dynamics applications, this quantity might also include the fictitious acceleration that simulates gravity. Velocity-product accelerations can be calculated efficiently by the recursive formula

$$\boldsymbol{a}_i^{\text{vp}} = \boldsymbol{a}_{\lambda(i)}^{\text{vp}} + \boldsymbol{v}_i \times \boldsymbol{s}_i\dot{\boldsymbol{q}}_i, \qquad (\boldsymbol{a}_0^{\text{vp}} = \boldsymbol{0} \text{ or } -\boldsymbol{a}_g) \tag{31}$$

which is obtained from (10) by setting $\ddot{\boldsymbol{q}}_i = 0$.

Another equation for the acceleration of body b can be obtained by differentiating (17):

$$\boldsymbol{a}_b = \boldsymbol{J}_b\,\ddot{\boldsymbol{q}} + \dot{\boldsymbol{J}}_b\,\dot{\boldsymbol{q}}. \tag{32}$$

At first sight, the term $\dot{\boldsymbol{J}}_b\,\dot{\boldsymbol{q}}$ looks like it might be difficult to calculate. However, a moments thought reveals that $\dot{\boldsymbol{J}}_b\,\dot{\boldsymbol{q}} = \boldsymbol{a}_b^{\text{vp}}$; so this equation can be written as

$$\boldsymbol{a}_b = \boldsymbol{J}_b\,\ddot{\boldsymbol{q}} + \boldsymbol{a}_b^{\text{vp}}. \tag{33}$$

If a two-handed robot has rigidly grasped a single object with both hands, then the kinematic acceleration constraint on that robot is $\boldsymbol{a}_l = \boldsymbol{a}_r$, where l and r are the body numbers of the left and right hands, respectively. Using (33), we can express this as a constraint on the joint accelerations as follows:

$$(\boldsymbol{J}_l - \boldsymbol{J}_r)\ddot{\boldsymbol{q}} = \boldsymbol{a}_r^{\mathrm{vp}} - \boldsymbol{a}_l^{\mathrm{vp}}. \tag{34}$$

If a robot mechanism contains kinematic loops, then the loop-closure constraints can be formulated in a similar manner to (34). However, if one wishes to simulate such a mechanism, then the acceleration constraints must be stabilized to prevent accumulation of position and velocity errors [3], [8].

Dynamics

We have already examined inverse dynamics in some detail, so let us now look at forward dynamics, which is the problem of calculating a robot's acceleration response to applied forces. In analogy with (1), we can express the forward-dynamics problem mathematically as

$$\ddot{\boldsymbol{q}} = \mathrm{FD}(\mathrm{model}, \boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{\tau}), \tag{35}$$

where the objective is to calculate the numeric value of the function FD from the numeric values of its arguments. There are many ways to do this; however, we shall consider only the two most efficient ways.

Composite Rigid-Body Algorithm

The joint-space equation of motion for a kinematic tree can be expressed in the following canonical form:

$$\boldsymbol{\tau} = \boldsymbol{H}\ddot{\boldsymbol{q}} + \boldsymbol{C}, \tag{36}$$

where $\boldsymbol{H}$ is the joint-space inertia matrix, and $\boldsymbol{C}$ is a vector containing the Coriolis, centrifugal, and gravitational terms. If we can calculate $\boldsymbol{H}$ and $\boldsymbol{C}$, then we can solve the forward-dynamics problem simply by solving (36) for $\ddot{\boldsymbol{q}}$. We already know how to calculate $\boldsymbol{C}$, because

$$\boldsymbol{C} = \mathrm{ID}(\mathrm{model}, \boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{0}) \tag{37}$$

[cf. (1)], so the only remaining problem is how to calculate $\boldsymbol{H}$. The best algorithm for this job is called the composite-rigid-body algorithm, which we shall now derive.

One of the defining properties of the joint-space inertia matrix is that the kinetic energy of a robot mechanism is given by

$$T = \frac{1}{2}\dot{\boldsymbol{q}}^{\mathrm{T}}\boldsymbol{H}\dot{\boldsymbol{q}} = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} H_{ij}\dot{\boldsymbol{q}}_i\dot{\boldsymbol{q}}_j. \tag{38}$$

However, the kinetic energy is also the sum of the kinetic energies of the individual bodies, which can be written in spatial-vector notation as

$$T = \sum_{k=1}^{N}\frac{1}{2}\boldsymbol{v}_k^{\mathrm{T}}\boldsymbol{I}_k\boldsymbol{v}_k. \tag{39}$$

Substituting for $\boldsymbol{v}_k$ using (18) gives

$$\begin{aligned} T &= \frac{1}{2}\sum_{k=1}^{N}\Big(\sum_{i\in\kappa(k)}\boldsymbol{s}_i\dot{\boldsymbol{q}}_i\Big)^{\mathrm{T}}\boldsymbol{I}_k\Big(\sum_{j\in\kappa(k)}\boldsymbol{s}_j\dot{\boldsymbol{q}}_j\Big) \\ &= \frac{1}{2}\sum_{k=1}^{N}\sum_{i\in\kappa(k)}\sum_{j\in\kappa(k)}\boldsymbol{s}_i^{\mathrm{T}}\boldsymbol{I}_k\boldsymbol{s}_j\dot{\boldsymbol{q}}_i\dot{\boldsymbol{q}}_j\,. \end{aligned} \tag{40}$$

Now, the expression on the right-hand side is a sum over all i, j, k triples in which both i and j are elements of $\kappa(k)$. This same set of triples can also be described as the set of all i, j, k triples in which $k \in \nu(i)$ and $k \in \nu(j)$. So we can rewrite (40) as follows:

$$T = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k\in\nu(i)\cap\nu(j)}\boldsymbol{s}_i^{\mathrm{T}}\boldsymbol{I}_k\boldsymbol{s}_j\dot{\boldsymbol{q}}_i\dot{\boldsymbol{q}}_j\,. \tag{41}$$

On comparing (41) with (38), if we take into account that both equations must be true for all $\dot{\boldsymbol{q}}$, and also that $n = N$ for the class of robots we are considering, then it follows that

$$H_{ij} = \sum_{k\in\nu(i)\cap\nu(j)}\boldsymbol{s}_i^{\mathrm{T}}\boldsymbol{I}_k\boldsymbol{s}_j. \tag{42}$$

There are two simplifications we can make to this equation. The first is that

$$\nu(i)\cap\nu(j) = \begin{cases} \nu(i) & \text{if } i \in \nu(j) \\ \nu(j) & \text{if } j \in \nu(i) \\ \emptyset & \text{otherwise.} \end{cases} \tag{43}$$

The second is that we can define a composite rigid-body inertia, $\boldsymbol{I}_i^c$, which is the inertia of all the bodies in the subtree $\nu(i)$ treated as a single composite rigid body. This inertia is given by

$$\boldsymbol{I}_i^c = \sum_{j\in\nu(i)}\boldsymbol{I}_j,$$

but the best way to calculate it is via the recursive formula

$$\boldsymbol{I}_i^c = \boldsymbol{I}_i + \sum_{j\in\mu(i)}\boldsymbol{I}_j^c. \tag{44}$$

With these two simplifications, we can rewrite (42) as

$$H_{ij} = \begin{cases} \boldsymbol{s}_i^T\boldsymbol{I}_i^c\boldsymbol{s}_j & \text{if } i \in \nu(j) \\ \boldsymbol{s}_i^T\boldsymbol{I}_j^c\boldsymbol{s}_j & \text{if } j \in \nu(i) \\ 0 & \text{otherwise.} \end{cases} \tag{45}$$

Equations (44) and (45) together define the composite-rigid-body algorithm.

Before moving on, let us review what we have just achieved. Using only a small amount of algebra, we have obtained a very compact expression for H_{ij} in (42) and a compact description of the composite-rigid-body algorithm in (44) and (45). Along the way, we have not had to worry about whether joint i is revolute, prismatic, or helical and write different equations for each case; nor have we written separate expressions for the linear and angular components of kinetic energy; nor have we defined a point in each body, expressed equations at that point, and transferred them from one point to another; and nor have we written equations to calculate the center of mass of a composite body or use the

parallel-axes theorem to calculate a rotational inertia about a new center of mass. In short, we have benefited considerably from the use of spatial-vector notation. Readers may wish to compare this derivation with the original 3-D-vector derivation in [10], bearing in mind that the original applied only to unbranched chains with revolute and prismatic joints.

Equations (44) and (45) provide us with a basic mathematical description of the algorithm. If we want to implement it on a computer, then we must first decide what coordinate systems to use. The best choice, for all but the largest rigid-body systems, is to use link coordinates. We can express the algorithm in link coordinates as follows:

$$\boldsymbol{I}_i^c = \boldsymbol{I}_i + \sum_{j \in \mu(i)} {}^i\boldsymbol{X}_j^* \boldsymbol{I}_j^{c\,j}\boldsymbol{X}_i, \tag{46}$$

$${}^{\lambda(j)}\boldsymbol{f}_i = {}^{\lambda(j)}\boldsymbol{X}_j^{*\,j}\boldsymbol{f}_i, \qquad ({}^i\boldsymbol{f}_i = \boldsymbol{I}_i^c \boldsymbol{s}_i) \tag{47}$$

$$H_{ij} = \begin{cases} {}^j\boldsymbol{f}_i^{\mathrm{T}}\boldsymbol{s}_j & \text{if } i \in v(j) \\ H_{ji} & \text{if } j \in v(i) \\ 0 & \text{otherwise.} \end{cases} \tag{48}$$

These equations show explicitly where the coordinate transforms are performed. Note that the quantities $\boldsymbol{s}_i$, $\boldsymbol{I}_i$, and $\boldsymbol{I}_i^c$ appearing in these equations are expressed in link-i coordinates, whereas the same symbols in previous equations were tacitly assumed to be expressed in a single unidentified common coordinate system.

The symbol ${}^j\boldsymbol{f}_i$ in (47) is the spatial force, expressed in link-j coordinates, that imparts an acceleration of $\boldsymbol{s}_i$ (i.e., a unit acceleration about the axis of joint i) to a composite rigid body comprising all of the bodies in subtree $v(i)$. The algorithm requires the calculation of ${}^i\boldsymbol{f}_i$ for every i and ${}^j\boldsymbol{f}_i$ for every $j \in \kappa(i) \backslash \{i\}$.

The pseudocode for this algorithm is shown in Figure 7. It employs the same tactic as was used in the recursive Newton–Euler algorithm to convert the summation over $\mu(i)$ in (46) into code that uses only λ: each variable $\boldsymbol{I}_i^c$ is initialized to $\boldsymbol{I}_i$ in the first loop, and then each $\boldsymbol{I}_i^c$ is added to its parent in the second loop. By the time $\boldsymbol{I}_i^c$ is used on line 9 to calculate ${}^i\boldsymbol{f}_i$, which is stored in the local variable $\boldsymbol{f}$, it has already received the contributions from all of its children and, therefore, has the correct final value.

The statement $\boldsymbol{H} = \boldsymbol{0}$ on line 1 is necessary, because the remaining code will initialize only the nonzero elements of $\boldsymbol{H}$. Certain elements of $\boldsymbol{H}$ will automatically be zero, simply because of the connectivity of the robot. This phenomenon is called branch-induced sparsity, and it arises from the third case in (48). This phenomenon is discussed in detail in [2] and [3] along with methods to greatly accelerate the solution of (36) by exploiting the sparsity.

Articulated-Body Algorithm

The articulated-body algorithm is an $O(N)$ algorithm that solves the forward-dynamics problem by the following strategy: at the outset, we know neither the acceleration of body i nor the force transmitted across joint i; however, we do know that the relationship between them must be linear. It must therefore be possible to express the relationship between these two vectors in an equation of the form

$$\boldsymbol{f}_i = \boldsymbol{I}_i^A \boldsymbol{a}_i + \boldsymbol{p}_i^A. \tag{49}$$

The two coefficients in this equation, $\boldsymbol{I}_i^A$ and $\boldsymbol{p}_i^A$, are called the articulated-body inertia and bias force, respectively, of body i; they describe the acceleration response of body i to an applied spatial force, taking into account the influence of all the other bodies in the subtree $v(i)$. These coefficients have two special properties that form the basis of the articulated-body algorithm. The special properties are that

1) they can be calculated recursively from the tips of the tree to the base, and
2) once they have been calculated, they allow the accelerations of the bodies and joints to be calculated recursively from the base to the tips.

The calculation of $\boldsymbol{I}_i^A$ and $\boldsymbol{p}_i^A$ closely resembles the two-body example presented in Part 1 [6]. Referring to Figure 8, we initially assume that body i has only one child, which is labeled body j. The relevant equations for body i are then

$$\boldsymbol{f}_i - \boldsymbol{f}_j = \boldsymbol{I}_i \boldsymbol{a}_i + \boldsymbol{p}_i, \tag{50}$$

$$\boldsymbol{f}_j = \boldsymbol{I}_j^A \boldsymbol{a}_j + \boldsymbol{p}_j^A, \tag{51}$$

$$\boldsymbol{a}_j = \boldsymbol{a}_i + \boldsymbol{c}_j + \boldsymbol{s}_j \ddot{\boldsymbol{q}}_j \tag{52}$$

```
1  H = 0
2  for i = 1 to N do
3     I_i^c = I_i
4  end
5  for i = N to 1 do
6     If λ(i) ≠ 0 then
7        I_λ(i)^c = I_λ(i)^c + λ(i)X_i^* I_i^c iX_λ(i)
8     end
9     f = I_i^c s_i
10    H_ii = f^T s_i
11    j = i
12    While λ(j) ≠ 0 do
13       f = λ(j)X_j^* f
14       j = λ(j)
15       H_ij = f^T s_j
16       H_ji = H_ij
17    end
18 end
```

Figure 7. *The composite-rigid-body algorithm.*

Spatial vectors can be used both for positional kinematics and for instantaneous kinematics.

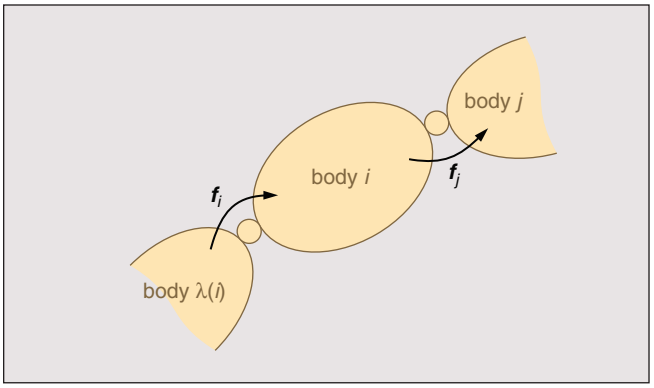

Figure 8. *Calculating articulated-body inertias.*

and

$$\tau_j = \boldsymbol{s}_j^{\mathrm{T}} \boldsymbol{f}_j, \tag{53}$$

where

$$\boldsymbol{p}_i = \boldsymbol{v}_i \times^* \boldsymbol{I}_i \boldsymbol{v}_i \tag{54}$$

and

$$\boldsymbol{c}_j = \boldsymbol{v}_j \times \boldsymbol{s}_j \dot{\boldsymbol{q}}_j. \tag{55}$$

Equation (50) is the equation of motion for body i, which we have written in terms of the rigid-body inertia and bias force, $\boldsymbol{I}_i$ and $\boldsymbol{p}_i$, to make it obvious that the rigid-body and articulated-body equations of motion have the same algebraic form. Equation (51) is the articulated-body equation of motion for body j, which describes the relationship between $\boldsymbol{f}_j$ and $\boldsymbol{a}_j$, taking into account the dynamics of every body and joint in the subtree $\nu(j)$. We assume that $\boldsymbol{I}_j^A$ and $\boldsymbol{p}_j^A$ are known. Equations (52) and (53) are the acceleration and force constraint equations for joint j.

The objective is to solve (50)–(53) to obtain an equation having the same form as (49), which is an equation involving only the two unknowns $\boldsymbol{f}_i$ and $\boldsymbol{a}_i$. To obtain this result, the first step is to solve (51)–(53) for the unknown acceleration $\ddot{\boldsymbol{q}}_j$. We can do this by substituting (51) and (52) into (53) as follows:

$$\begin{aligned}\tau_j &= \boldsymbol{s}_j^{\mathrm{T}} (\boldsymbol{I}_j^A \boldsymbol{a}_j + \boldsymbol{p}_j^A) \\ &= \boldsymbol{s}_j^{\mathrm{T}} (\boldsymbol{I}_j^A (\boldsymbol{a}_i + \boldsymbol{c}_j + \boldsymbol{s}_j \ddot{\boldsymbol{q}}_j) + \boldsymbol{p}_j^A),\end{aligned}$$

which yields the following equation for $\ddot{\boldsymbol{q}}_j$:

$$\ddot{\boldsymbol{q}}_j = \frac{\tau_j - \boldsymbol{s}_j^{\mathrm{T}} (\boldsymbol{I}_j^A (\boldsymbol{a}_i + \boldsymbol{c}_j) + \boldsymbol{p}_j^A)}{\boldsymbol{s}_j^{\mathrm{T}} \boldsymbol{I}_j^A \boldsymbol{s}_j}. \tag{56}$$

At this point, we can simplify (56) a little by introducing the quantity

$$u_j = \tau_j - \boldsymbol{s}_j^{\mathrm{T}} \boldsymbol{p}_j^A. \tag{57}$$

Substituting (57) in (56) gives

$$\ddot{\boldsymbol{q}}_j = \frac{u_j - \boldsymbol{s}_j^{\mathrm{T}} \boldsymbol{I}_j^A (\boldsymbol{a}_i + \boldsymbol{c}_j)}{\boldsymbol{s}_j^{\mathrm{T}} \boldsymbol{I}_j^A \boldsymbol{s}_j}. \tag{58}$$

Having found an expression for $\ddot{\boldsymbol{q}}_j$, the remainder of the problem is solved by substituting (51), (52), and (58) back into (50) as follows:

$$\begin{aligned}\boldsymbol{f}_i &= \boldsymbol{I}_i \boldsymbol{a}_i + \boldsymbol{p}_i + \boldsymbol{f}_j \\ &= \boldsymbol{I}_i \boldsymbol{a}_i + \boldsymbol{I}_j^A \boldsymbol{a}_j + \boldsymbol{p}_i + \boldsymbol{p}_j^A \\ &= \boldsymbol{I}_i \boldsymbol{a}_i + \boldsymbol{I}_j^A (\boldsymbol{a}_i + \boldsymbol{c}_j + \boldsymbol{s}_j \ddot{\boldsymbol{q}}_j) + \boldsymbol{p}_i + \boldsymbol{p}_j^A \\ &= \boldsymbol{I}_i \boldsymbol{a}_i + \boldsymbol{I}_j^A \left(\boldsymbol{a}_i + \boldsymbol{c}_j + \frac{\boldsymbol{s}_j (u_j - \boldsymbol{s}_j^{\mathrm{T}} \boldsymbol{I}_j^A (\boldsymbol{a}_i + \boldsymbol{c}_j))}{\boldsymbol{s}_j^{\mathrm{T}} \boldsymbol{I}_j^A \boldsymbol{s}_j} \right) + \boldsymbol{p}_i + \boldsymbol{p}_j^A.\end{aligned} \tag{59}$$

On comparing this equation with (49), we get the following expressions for $\boldsymbol{I}_i^A$ and $\boldsymbol{p}_i^A$:

$$\boldsymbol{I}_i^A = \boldsymbol{I}_i + \boldsymbol{I}_j^a \tag{60}$$

and

$$\boldsymbol{p}_i^A = \boldsymbol{p}_i + \boldsymbol{p}_j^a, \tag{61}$$

where

$$\boldsymbol{I}_j^a = \boldsymbol{I}_j^A - \frac{\boldsymbol{I}_j^A \boldsymbol{s}_j \boldsymbol{s}_j^{\mathrm{T}} \boldsymbol{I}_j^A}{\boldsymbol{s}_j^{\mathrm{T}} \boldsymbol{I}_j^A \boldsymbol{s}_j} \tag{62}$$

and

$$\boldsymbol{p}_j^a = \boldsymbol{I}_j^a \boldsymbol{c}_j + \frac{\boldsymbol{I}_j^A \boldsymbol{s}_j u_j}{\boldsymbol{s}_j^{\mathrm{T}} \boldsymbol{I}_j^A \boldsymbol{s}_j} + \boldsymbol{p}_j^A. \tag{63}$$

The next step is to drop the assumption that body i has only one child. If body i has multiple children, then it is possible to process them one at a time using the above procedure. This works because spatial inertias are additive and rigid-body and articulated-body equations have the same algebraic form. In processing the rth child, we simply replace $\boldsymbol{I}_i$ and $\boldsymbol{p}_i$ in (50) with the articulated-body inertia and bias force that account for the first $r-1$ children. The end result is the following pair of equations, which replace (60) and (61):

$$\boldsymbol{I}_i^A = \boldsymbol{I}_i + \sum_{j \in \mu(i)} \boldsymbol{I}_j^a \tag{64}$$

and

$$\boldsymbol{p}_i^A = \boldsymbol{p}_i + \sum_{j \in \mu(i)} \boldsymbol{p}_j^a. \tag{65}$$

The definitions of $\boldsymbol{I}_j^a$ and $\boldsymbol{p}_j^a$ remain unchanged.

The final step is to calculate the accelerations. We already have the necessary equations, being (52) and (58), but the calculation can be performed slightly more efficiently as follows:

$$\boldsymbol{a}_i' = \boldsymbol{a}_{\lambda(i)} + \boldsymbol{c}_i, \qquad (\boldsymbol{a}_0 = -\boldsymbol{a}_g) \tag{66}$$

$$\ddot{\boldsymbol{q}}_i = \frac{u_i - \boldsymbol{s}_i^{\mathrm{T}} \boldsymbol{I}_i^A \boldsymbol{a}_i'}{\boldsymbol{s}_i^{\mathrm{T}} \boldsymbol{I}_i^A \boldsymbol{s}_i}, \tag{67}$$

Pass 1

$$\boldsymbol{v}_0 = \boldsymbol{0}$$
$$\boldsymbol{v}_i = {}^i\boldsymbol{X}_{\lambda(i)}\,\boldsymbol{v}_{\lambda(i)} + \boldsymbol{s}_i\dot{q}_i$$
$$\boldsymbol{c}_i = \boldsymbol{v}_i \times \boldsymbol{s}_i\dot{q}_i$$
$$\boldsymbol{p}_i = \boldsymbol{v}_i \times^* \boldsymbol{I}_i\,\boldsymbol{v}_i$$

Pass 2

$$\boldsymbol{I}_i^A = \boldsymbol{I}_i + \sum_{j\in\mu(i)} {}^i\boldsymbol{X}_j^*\,\boldsymbol{I}_j^a\,{}^j\boldsymbol{X}_i$$
$$\boldsymbol{p}_i^A = \boldsymbol{p}_i + \sum_{j\in\mu(i)} {}^i\boldsymbol{X}_j^*\,\boldsymbol{p}_j^a$$
$$\boldsymbol{h}_i = \boldsymbol{I}_i^A\boldsymbol{s}_i$$
$$d_i = \boldsymbol{s}_i^{\mathrm{T}}\boldsymbol{h}_i$$
$$u_i = \tau_i - \boldsymbol{s}_i^{\mathrm{T}}\boldsymbol{p}_i^A$$
$$\boldsymbol{I}_i^a = \boldsymbol{I}_i^A - \boldsymbol{h}_i\boldsymbol{h}_i^{\mathrm{T}}/d_i$$
$$\boldsymbol{p}_i^a = \boldsymbol{p}_i^A + \boldsymbol{I}_i^a\boldsymbol{c}_i + \boldsymbol{h}_i u_i/d_i$$

Pass 3

$$\boldsymbol{a}_0 = -\boldsymbol{a}_g$$
$$\boldsymbol{a}_i' = {}^i\boldsymbol{X}_{\lambda(i)}\,\boldsymbol{a}_{\lambda(i)} + \boldsymbol{c}_i$$
$$\ddot{q}_i = (u_i - \boldsymbol{h}_i^{\mathrm{T}}\boldsymbol{a}_i')/d_i$$
$$\boldsymbol{a}_i = \boldsymbol{a}_i' + \boldsymbol{s}_i\ddot{q}_i$$

Figure 9. *Equations of the articulated-body algorithm.*

$$\boldsymbol{a}_i = \boldsymbol{a}_i' + \boldsymbol{s}_i\ddot{\boldsymbol{q}}_i. \tag{68}$$

The complete equations for the articulated-body algorithm, expressed in link coordinates, are shown in Figure 9. The algorithm makes a total of three passes through the tree: an outward pass (base to tips) to calculate the velocity terms $\boldsymbol{c}_i$ and $\boldsymbol{p}_i$, an inward pass (tips to base) to calculate $\boldsymbol{I}_i^A$, $\boldsymbol{p}_i^A$, and related terms, and a second outward pass to calculate the accelerations. A more detailed description of this algorithm can be found in [3], and source code can be obtained from [5].

Deriving the articulated-body algorithm is an example of a dynamics problem that would be forbiddingly difficult to attempt using 3-D vectors. Whereas other algorithms described in this tutorial were invented using 3-D vectors, the articulated-body algorithm was invented using spatial vectors. In fact, spatial vectors themselves were invented as a side effect of trying to invent the articulated-body algorithm. This algorithm, and many others that have followed it, make an important statement about spatial vectors: they are a tool for discovery; they let you go beyond what is feasible to attempt using 3-D vectors.

Conclusion

This tutorial has demonstrated the use of spatial vectors in a variety of kinematics and dynamics calculations. A model-based approach was adopted in which a description of the robot mechanism is stored in a model data structure, and the various equations and algorithms are designed to use this data in their calculations. The class of robots considered was the class of general kinematic trees having revolute, prismatic, and helical joints; the idea being to show how easily spatial vectors cope with a high degree of generality. The focus of this tutorial has ranged from mathematics to computer code to make the point that spatial vectors are both an analytical tool and a computational tool. In both this tutorial and Part 1, the emphasis has been on human productivity: fewer equations, simpler problem solving, and shorter code. If your application also needs high computational efficiency, then see Appendix A of [3]. A mastery of spatial vectors gives you a different perspective on rigid-body kinematics and dynamics and is a worthwhile skill for a roboticist.

Keywords

Dynamics, kinematics, spatial vectors, dynamics algorithms, software.

References

[1] J. J. Craig, *Introduction to Robotics: Mechanics and Control*, 3rd ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2005.

[2] R. Featherstone, "Efficient factorization of the joint space inertia matrix for branched kinematic trees," *Int. J. Robot. Res.*, vol. 24, no. 6, pp. 487–500, 2005.

[3] R. Featherstone, *Rigid Body Dynamics Algorithms*. New York: Springer-Verlag, 2008.

[4] R. Featherstone and D. E. Orin, "Dynamics," *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Berlin: Springer-Verlag, 2008, pp. 35–65.

[5] R. Featherstone. (2010). "Spatial vector algebra," [Online]. Available: http://users.cecs.anu.edu.au/~roy/spatial/

[6] R. Featherstone, "A beginner's guide to 6-D vectors (part 1)," *IEEE Robot. Automat. Mag.*, vol. 17, no. 3, pp. 83–94, 2010.

[7] J. Y. S. Luh, M. W. Walker, and R. P. C. Paul, "On-line computational scheme for mechanical manipulators," *Trans. ASME J. Dyn. Syst., Meas. Control*, vol. 102, no. 2, pp. 69–76, 1980.

[8] R. E. Roberson and R. Schwertassek, *Dynamics of Multibody Systems*. Berlin: Springer-Verlag, 1988.

[9] K. Waldron and J. Schmiedeler, "Kinematics," *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Berlin: Springer-Verlag, 2008, pp. 9–33.

[10] M. W. Walker and D. E. Orin, "Efficient dynamic computer simulation of robotic mechanisms," *Trans. ASME J. Dyn. Syst., Meas. Control*, vol. 104, no. 3, pp. 205–211, 1982.

Roy Featherstone received his Ph.D. degree from Edinburgh University in 1984. He spent approximately seven years working in industry before moving to Oxford University in 1992 to take up an EPSRC Advanced Research Fellowship. He currently works for the Australian National University, which he joined in 2001. His research interests include robot motion, in particular, the kinematics, dynamics, control and enabling technology of complex and energetic robot motion. He is also interested in dynamics algorithms and simulation. He is a Fellow of the IEEE.

Address for Correspondence: Roy Featherstone, School of Engineering, RSISE Building 115, The Australian National University, Canberra, ACT 0200, Australia. E-mail: Roy.Featherstone@anu.edu.au.