

Exploiting Sparsity in Operational-Space Dynamics

Roy Featherstone
Dept. Information Engineering
The Australian National University
Canberra, ACT 0200, Australia

2nd revised version, submitted to IJRR on November 15, 2009

Abstract

This paper presents a new method for calculating operational-space inertia matrices, and other related quantities, for branched kinematic trees. It is based on the exploitation of branch-induced sparsity in the joint-space inertia matrix and the task Jacobian. Detailed cost figures are given for the new method, and its efficacy is demonstrated by means of a realistic example based on the ASIMO Next-Generation humanoid robot. In this example, the new method is shown to be 6.7 times faster than the basic matrix method, and 1.6 times faster than the efficient low-order algorithm of Rodriguez et al. Furthermore, cost savings of more than 50,000 arithmetic operations are obtained in the calculation of the inertia-weighted pseudoinverse of the task Jacobian and its null-space projection matrix. Additional examples are considered briefly, in order to further compare the new method with the algorithm of Rodriguez et al.

1 Introduction

If a robot has a multiplicity of limbs, then it has a branched kinematic structure. An automatic consequence of these branches is that the robot's joint-space inertia matrix exhibits a sparsity pattern (a pattern of zero-valued elements) that can be exploited to reduce the cost of calculating the robot's forward dynamics (Featherstone, 2005). The present paper shows how to extend this idea to operational-space dynamics and control.

Starting with the equation $\mathbf{A}^{-1} = \mathbf{J}\mathbf{H}^{-1}\mathbf{J}^T$, where \mathbf{H} and \mathbf{A} are the joint-space and operational-space inertia matrices and \mathbf{J} is the task Jacobian, this paper shows how to exploit the sparsity in \mathbf{H} and \mathbf{J} so as to greatly accelerate the calculation of \mathbf{A}^{-1} . Cost figures are given for the new method, and its efficacy is demonstrated using an example based on the ASIMO Next-Generation humanoid robot. In this example, the new method is 6.7 times faster than the basic method (i.e., evaluating $\mathbf{J}\mathbf{H}^{-1}\mathbf{J}^T$ without exploiting sparsity), and it is 1.6 times faster than an optimized version of the recursive, low-order algorithm for \mathbf{A}^{-1} described in Rodriguez et al. (1992).

This paper also considers two other matrices: the dynamically-consistent generalized inverse Jacobian, $\bar{\mathbf{J}}$, which is the inertia-weighted pseudoinverse of \mathbf{J} , and the null-space

projection matrix, \mathbf{N} , that is derived from \mathbf{J} and $\bar{\mathbf{J}}$. Both can be calculated more efficiently by exploiting sparsity. In the ASIMO example, the costs of calculating these matrices are each reduced by more than 50,000 floating-point arithmetic operations, measured relative to the basic matrix method.

Operational-space dynamics was originally applied to serial robots having a single end-effector. For this kind of robot, the obvious choice of operational space is the six degrees of motion freedom of the end-effector. In this case, \mathbf{A} is only a 6×6 matrix, and it can be obtained via the formula $\mathbf{A}^{-1} = \mathbf{J}_e \mathbf{H}^{-1} \mathbf{J}_e^T$, where \mathbf{J}_e is the end-effector Jacobian. However, the calculation of \mathbf{A} by this method has a complexity of $O(n^3)$, and this observation has motivated the development of several $O(n)$ algorithms for this special case (Kreutz-Delgado et al., 1992; Lilly, 1993; Lilly and Orin, 1993).

If a robot has a branched kinematic structure, as is the case for a humanoid or legged robot, then it is necessary to use a more general operational space that depends on the motions of more than one body. In this case, the above formula still applies, provided one uses the task Jacobian instead of the end-effector Jacobian (Russakow et al., 1995). The complexity of calculating \mathbf{A} by this method is $O(n^3 + m^3)$, where m is the dimension of operational space. Alternatively, one can use the low-order recursive algorithm invented by Rodriguez et al. (1992) for calculating \mathbf{A}^{-1} . An optimized version of this algorithm, together with cost figures, appears in Appendix B of this paper. The complexity of this algorithm is $O(n + m^2d)$, where d is the depth of the connectivity tree. An extra $O(m^3)$ is then required to calculate \mathbf{A} from \mathbf{A}^{-1} . A very similar algorithm was subsequently developed by Chang and Khatib (1999, 2000, 2001), and the difference between these two algorithms is explained in Appendix B.

The approach taken in this paper is to accelerate the calculation of $\mathbf{JH}^{-1}\mathbf{J}^T$ by exploiting the sparsity in \mathbf{H} and \mathbf{J} . The computational complexity of the new method is $O(nd^2 + md^2 + m^2d)$, which is better than the basic method but worse than Rodriguez' algorithm. Thus, Rodriguez' algorithm will inevitably be faster for sufficiently large values of d . Nevertheless, the new method is easily the fastest on the ASIMO example, and also on several more examples considered at the end of Section 9. These results show that the new method is likely to be the best choice for most humanoids and similar robots.

The rest of this paper is organized as follows. Sections 2 and 3 explain how connectivity is described, and how branches cause branch-induced sparsity. Section 4 presents a brief summary of operational-space dynamics, sufficient to define the matrices \mathbf{A} , $\bar{\mathbf{J}}$ and \mathbf{N} . Section 5 describes the sparsity pattern in the task Jacobian. Section 6 shows how \mathbf{A}^{-1} can be factorized as $\mathbf{A}^{-1} = \mathbf{Y}\mathbf{Y}^T$ or $\mathbf{A}^{-1} = \mathbf{Y}_D\mathbf{Y}_D^T$, where \mathbf{Y} , \mathbf{Y}_d and \mathbf{Y}_D all have the same sparsity pattern as \mathbf{J} . Section 7 explains the close connections between Section 6 and the innovations factorization of Rodriguez et al. Finally, Section 8 presents the new sparse-matrix algorithms; and Section 9 presents a table of computational cost formulae, an analysis of computational complexity, and the actual costs incurred in the ASIMO example.

2 Describing Connectivity

A rigid-body system can be regarded as a collection of rigid bodies connected together by joints. The connectivity of such a system can be described by means of a graph in which the nodes represent the bodies and the arcs represent the joints. We use the term

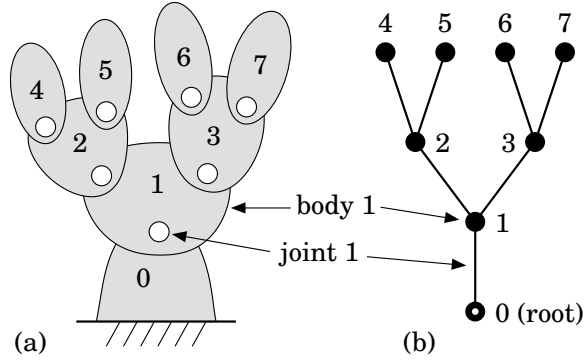


Figure 1: A kinematic tree (a) and its connectivity graph (b)

kinematic tree to describe any rigid-body system for which the connectivity graph is a tree. In practical terms, a kinematic tree is a mechanical system without kinematic loops. An example of a kinematic tree is shown in Figure 1(a), and its connectivity graph is shown in Figure 1(b).

In general, a kinematic tree will consist of N bodies, N joints and a fixed base. We treat the fixed base as a special body, so the connectivity graph will contain N arcs and $N + 1$ nodes. If the kinematic tree describes a mobile robot, then one body in the robot is identified as the *floating base*, and a six-degree-of-freedom (6-DoF) joint is inserted between the fixed and floating bases. This joint is not physically a part of the robot, but the six joint variables associated with it are necessary for describing the robot's location. If the kinematic tree describes a collection of mobile robots, then each one has its own floating base and 6-DoF joint.

The bodies, joints, nodes and arcs are numbered according to the following standard numbering scheme. First, the node representing the fixed base is assigned the number 0, and is regarded as the root node of the tree. Next, the remaining nodes are numbered consecutively from 1 in any order such that each node has a higher number than its parent. The arcs are then numbered such that arc i connects between node i and its parent. Finally, the bodies and joints are given the same numbers as their corresponding nodes and arcs.

Once the bodies have been numbered, the connectivity of a kinematic tree can be described by its parent array, λ . This is an N -element array such that $\lambda(i)$ is the body number of the parent of body i . The parent array for the example in Figure 1 is $\lambda = [0, 1, 1, 2, 2, 3, 3]$, meaning that $\lambda(1) = 0$, $\lambda(2) = 1$, and so on. The node-numbering rules ensure that λ has the property $0 \leq \lambda(i) < i$, which is exploited in many algorithms.

Given λ , the following sets can be defined, which describe various properties of the connectivity graph:

$\mu(i)$: the set of children of body i , defined by $\mu(i) = \{j | \lambda(j) = i\}$;

$\kappa(i)$: the set of joints that support body i , defined by $\kappa(i) = \{i\} \cup \kappa(\lambda(i))$ and $\kappa(0) = \emptyset$ (the empty set); and

$\nu(i)$: the set of bodies supported by joint i , defined by $\nu(i) = \{j | i \in \kappa(j)\}$.

A joint is said to support a body if it lies on the path between that body and the root node. Thus, $\kappa(i)$ is the set of all joints on the path between body i and the root, and $\nu(i)$

is the set of all bodies in the subtree starting at body i . These sets have various properties that follow from their definitions. For example, $j \in \kappa(i)$ implies $i \in \nu(j)$ and vice versa, and $\kappa(i) \cap \nu(j) \neq \emptyset$ if and only if $j \in \kappa(i)$. For the connectivity tree in Figure 1, we have $\mu(1) = \{2, 3\}$, $\nu(2) = \{2, 4, 5\}$, $\kappa(4) = \{1, 2, 4\}$, $\mu(5) = \emptyset$, and so on. A kinematic tree is branched if at least one set $\mu(i)$ contains more than one element.

Most mainstream dynamics algorithms can be couched in terms that use only λ . Nevertheless, $\mu(i)$, $\nu(i)$ and $\kappa(i)$ can be useful in the mathematical descriptions of these algorithms, and in the analysis of their properties.

3 Branch-Induced Sparsity

The phenomenon of branch-induced sparsity refers to a pattern of zeros appearing in certain important matrices as a direct consequence of branches in the kinematic tree. By exploiting these zeros, it is possible to reduce the cost, and even the computational complexity, of some dynamics calculations. For example, it is generally thought that the calculation of a robot's forward dynamics via the joint-space inertia matrix (i.e., via Eq. 1 below) is an $O(n^3)$ calculation, where n is the number of joint variables; but its true complexity is only $O(nd^2)$, where d is the depth of the connectivity tree. This lower complexity can be achieved by exploiting branch-induced sparsity.

Branch-induced sparsity was first studied in the joint-space inertia matrix, which is the coefficient matrix \mathbf{H} in the joint-space equation of motion

$$\boldsymbol{\tau} = \mathbf{H}\ddot{\mathbf{q}} + \mathbf{C}. \quad (1)$$

($\boldsymbol{\tau}$ and $\ddot{\mathbf{q}}$ are vectors of joint force and acceleration variables, and \mathbf{C} is a vector of Coriolis, centrifugal and gravity terms.) The formula for \mathbf{H} is

$$\mathbf{H}_{ij} = \begin{cases} \mathbf{S}_i^T \mathbf{I}_i^c \mathbf{S}_j & \text{if } i \in \nu(j) \\ \mathbf{S}_i^T \mathbf{I}_j^c \mathbf{S}_j & \text{if } j \in \nu(i) \\ \mathbf{0} & \text{otherwise,} \end{cases} \quad (2)$$

where \mathbf{I}_i^c is the composite-rigid-body inertia of the set of bodies in $\nu(i)$, and \mathbf{S}_i is the motion subspace matrix of joint i (also known as the free modes matrix of joint i). \mathbf{I}_i^c is a 6×6 matrix, and \mathbf{S}_i is a $6 \times n_i$ matrix, where n_i is the DoF of joint i . Given that the total number of joint variables is n , it follows that $\sum_{i=1}^N n_i = n$, and that \mathbf{H} is both an $n \times n$ matrix and an $N \times N$ block matrix composed of blocks \mathbf{H}_{ij} having dimensions $n_i \times n_j$. A derivation of Eq. 2 can be found in Featherstone (2008).

The third case in Eq. 2 is the one that gives rise to branch-induced sparsity. It states that \mathbf{H}_{ij} will be zero for every i and j such that joint i does not support body j and joint j does not support body i . This condition will be true if and only if bodies i and j lie on separate branches of the connectivity graph, hence the name ‘branch-induced sparsity’. To give a pictorial example, Figure 2(a) shows the sparsity pattern (i.e., the pattern of zeros) that would appear in \mathbf{H} as a consequence of the branches in the kinematic tree in Figure 1; and Figure 2(b) shows a lower-triangular matrix having the same sparsity pattern below the main diagonal. In both cases, the grey areas denote nonzero elements¹ or submatrices. We regard both of these patterns as examples of branch-induced sparsity.

¹A nonzero element in a sparse matrix is one that is free to take any value, including zero. In effect, ‘nonzero’ means ‘not constrained to be zero’.

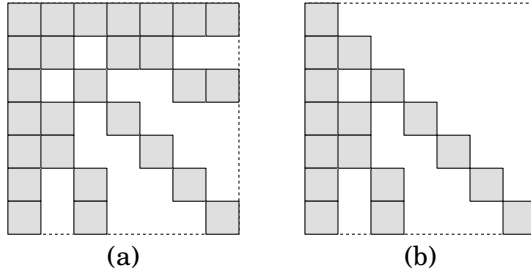


Figure 2: Branch-induced sparsity pattern in a symmetric matrix (a) and a lower-triangular matrix (b) for the kinematic tree in Figure 1

A formal definition of branch-induced sparsity can now be stated as follows. For a given λ , which can be any array of integers such that $0 \leq \lambda(i) < i$ for all i ,

1. a symmetric matrix \mathbf{H} , having elements H_{ij} , exhibits branch-induced sparsity if nonzero values appear only in those elements satisfying $j \in \kappa(i) \cup \nu(i)$; and
2. a lower-triangular matrix \mathbf{L} , having elements L_{ij} , exhibits branch-induced sparsity if nonzero values appear only in those elements satisfying $j \in \kappa(i)$.

The definition can be extended to block matrices by replacing ‘element’ with ‘submatrix’. For convenience, we define $\text{SPD}(\lambda)$ and $\text{L}(\lambda)$ to be the sets of symmetric positive-definite and nonsingular lower-triangular matrices, respectively, having the branch-induced sparsity pattern defined by λ . Note that the pattern shown in Figure 2(b), or permutations thereof, has been known for a long time in the dynamics literature. For example, it can be seen in the expression for ϕ on p. 35 of Rodriguez et al. (1992), and in Figure 5.9 in Wittenburg (1977).

It can be shown that $\text{L}(\lambda)$ forms a group under matrix multiplication, and a proof is given in Appendix A. One immediate consequence is that branch-induced sparsity patterns in lower-triangular matrices are preserved under both matrix multiplication and inversion. This property is exploited in the sequel.

4 Operational-Space Dynamics

The equation of motion for a robot mechanism, expressed in operational space, can be written in the form

$$\ddot{\mathbf{x}} = \mathbf{A}^{-1}\mathbf{F} + \boldsymbol{\beta}. \quad (3)$$

In this equation, $\ddot{\mathbf{x}}$ and \mathbf{F} are operational-space acceleration and force vectors, respectively, \mathbf{A}^{-1} is the operational-space inverse inertia matrix, and $\boldsymbol{\beta}$ is an acceleration bias vector, being the acceleration that would result if \mathbf{F} were zero. \mathbf{A}^{-1} is an $m \times m$ symmetric matrix, where m is the dimension of operational space, and the other quantities in this equation are m -dimensional vectors. If \mathbf{A}^{-1} is nonsingular then Eq. 3 can be inverted to obtain the equation

$$\mathbf{F} = \mathbf{A}\ddot{\mathbf{x}} + \boldsymbol{\mu} + \boldsymbol{\rho}, \quad (4)$$

where $\boldsymbol{\mu}$ is a vector of Coriolis and centrifugal force terms, $\boldsymbol{\rho}$ is a vector of gravitational force terms, and

$$\boldsymbol{\mu} + \boldsymbol{\rho} = -\mathbf{A}\boldsymbol{\beta}.$$

In early works on the subject, operational space was typically defined to be the motion space of the end-effector of a robot arm. However, in its general form, operational space can be a function of the motions of any number of bodies in a general robot mechanism.

The relationship between Eqs. 1 and 3 is established via a *task Jacobian*, \mathbf{J} , that maps joint-space velocities to operational-space velocities according to

$$\dot{\mathbf{x}} = \mathbf{J}\dot{\mathbf{q}}. \quad (5)$$

\mathbf{J} is therefore an $m \times n$ matrix. This Jacobian also maps operational-space forces to joint-space forces according to

$$\boldsymbol{\tau} = \mathbf{J}^T \mathbf{F}. \quad (6)$$

Given these equations, it can be seen that

$$\boldsymbol{\Lambda}^{-1} = \mathbf{J}\mathbf{H}^{-1}\mathbf{J}^T \quad (7)$$

and

$$\boldsymbol{\beta} = \dot{\mathbf{J}}\dot{\mathbf{q}} - \mathbf{J}\mathbf{H}^{-1}\mathbf{C}. \quad (8)$$

Another matrix of interest is Khatib's dynamically-consistent generalized inverse of \mathbf{J} , denoted $\bar{\mathbf{J}}$, which is given by

$$\bar{\mathbf{J}} = \mathbf{H}^{-1}\mathbf{J}^T\boldsymbol{\Lambda} \quad (9)$$

(assuming $\boldsymbol{\Lambda}^{-1}$ is invertible). This matrix is also the inertia-weighted pseudoinverse of \mathbf{J} , which means that $\dot{\mathbf{q}} = \bar{\mathbf{J}}\dot{\mathbf{x}}$ is the unique solution to Eq. 5 that minimizes $\dot{\mathbf{q}}^T \mathbf{H} \dot{\mathbf{q}}$. In addition to its use in operational-space control, this matrix has been used in redundancy-resolution schemes to obtain motions that instantaneously minimize a robot's kinetic energy (Khatib, 1987; Hollerbach and Suh, 1987).

Yet another matrix of interest is the null-space matrix derived from $\bar{\mathbf{J}}$, which defines the dynamically-consistent null space of the robot mechanism—the space of joint forces that cause no acceleration in operational space. This matrix is defined by

$$\mathbf{N} = \mathbf{1} - \bar{\mathbf{J}}\mathbf{J}, \quad (10)$$

and it is used in the equation of operational-space control,

$$\boldsymbol{\tau} = \mathbf{J}^T \mathbf{F} + \mathbf{N}^T \boldsymbol{\tau}_0, \quad (11)$$

which computes the joint force command, $\boldsymbol{\tau}$, as the sum of a force that will cause a desired acceleration in the operational space and a force that will cause an acceleration only in the null space. $\boldsymbol{\tau}_0$ can be any joint-space force vector, and \mathbf{N}^T projects it onto the null space. For more information on operational-space dynamics, see Khatib (1987, 1995); Khatib et al. (2004).

5 Sparsity in the Task Jacobian

In this section, we construct a formula for the Jacobian having the form

$$\mathbf{J} = \mathbf{R}\mathbf{S}, \quad (12)$$

where \mathbf{S} is a mapping from joint space to the space of spatial (6D) velocities of all N bodies (a $6N$ -dimensional vector space), and \mathbf{R} maps from this space to operational space. Thus, \mathbf{S} is a $6N \times n$ matrix and \mathbf{R} is $m \times 6N$. It will be shown that \mathbf{S} has the same sparsity pattern as an element of $L(\lambda)$, although it is not literally a member of this group because it is rectangular. By placing some restrictions on the definition of operational space, it can be arranged that \mathbf{R} preserves the sparsity in the rows of \mathbf{S} , so that each row in \mathbf{J} has the same sparsity pattern as a row appearing in \mathbf{S} .

Let \mathbf{v}_i be the spatial velocity of body i in the robot mechanism, and let \mathbf{v}_{J_i} be the spatial velocity across joint i , which is defined to be the velocity of body i relative to its parent (i.e., $\mathbf{v}_{J_i} = \mathbf{v}_i - \mathbf{v}_{\lambda(i)}$). By definition, \mathbf{v}_{J_i} is given by the formula $\mathbf{v}_{J_i} = \mathbf{S}_i \dot{\mathbf{q}}_i$, where $\dot{\mathbf{q}}_i$ is an n_i -dimensional vector containing the velocity variables of joint i , and \mathbf{S}_i is the joint's motion subspace matrix, as mentioned in Section 3. We therefore have

$$\begin{aligned} \mathbf{v}_i &= \mathbf{S}_i \dot{\mathbf{q}}_i + \mathbf{v}_{\lambda(i)} \\ &= \mathbf{S}_i \dot{\mathbf{q}}_i + \mathbf{S}_{\lambda(i)} \dot{\mathbf{q}}_{\lambda(i)} + \cdots \\ &= \sum_{j \in \kappa(i)} \mathbf{S}_j \dot{\mathbf{q}}_j, \end{aligned} \quad (13)$$

which simply says that the velocity of body i is the sum of the velocities across each of the joints on the path between the fixed base and body i . By collecting together these equations for the individual body velocities, we obtain the following composite equation describing the velocity of every body in the tree:

$$\mathbf{v} = \mathbf{S} \dot{\mathbf{q}}, \quad (14)$$

where

$$\mathbf{v} = \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_N \end{bmatrix}, \quad \dot{\mathbf{q}} = \begin{bmatrix} \dot{\mathbf{q}}_1 \\ \dot{\mathbf{q}}_2 \\ \vdots \\ \dot{\mathbf{q}}_N \end{bmatrix}, \quad \mathbf{S} = \begin{bmatrix} \mathbf{S}_{11} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{S}_{21} & \mathbf{S}_{22} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{S}_{N1} & \mathbf{S}_{N2} & \cdots & \mathbf{S}_{NN} \end{bmatrix}$$

and

$$\mathbf{S}_{ij} = \begin{cases} \mathbf{S}_j & \text{if } j \in \kappa(i) \\ \mathbf{0} & \text{otherwise.} \end{cases} \quad (15)$$

\mathbf{v} is therefore a $6N$ -dimensional vector, and \mathbf{S} is a $6N \times n$ matrix. \mathbf{S} is also an $N \times N$ block matrix composed of rectangular blocks \mathbf{S}_{ij} having dimensions $6 \times n_j$. If it is important to identify what coordinate system is being used, then add a leading superscript; thus, ${}^i\mathbf{S}_j$ is the motion subspace of joint j expressed in link i coordinates, and so on.

Equation 15 implies that \mathbf{S} has the same sparsity pattern, expressed at the block-matrix level, as an element of $L(\lambda)$. For example, the value of \mathbf{S} for the kinematic tree in Figure 1 is

$$\mathbf{S} = \begin{bmatrix} \mathbf{S}_1 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{S}_1 & \mathbf{S}_2 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{S}_1 & \mathbf{0} & \mathbf{S}_3 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{S}_1 & \mathbf{S}_2 & \mathbf{0} & \mathbf{S}_4 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{S}_1 & \mathbf{S}_2 & \mathbf{0} & \mathbf{0} & \mathbf{S}_5 & \mathbf{0} & \mathbf{0} \\ \mathbf{S}_1 & \mathbf{0} & \mathbf{S}_3 & \mathbf{0} & \mathbf{0} & \mathbf{S}_6 & \mathbf{0} \\ \mathbf{S}_1 & \mathbf{0} & \mathbf{S}_3 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{S}_7 \end{bmatrix}. \quad (16)$$

Having obtained a formula for \mathbf{S} in Eq. 12, the next step is to find a formula for \mathbf{R} . In order to preserve as much sparsity as possible, we assume that each element in $\dot{\mathbf{x}}$ depends on the velocity of a single body in the robot mechanism, and introduce an m -element array of body numbers, b , such that $b(i)$ is the number of the body upon which \dot{x}_i depends. As the relationships between velocities are linear, it follows that \dot{x}_i can be expressed in the form

$$\dot{x}_i = \mathbf{R}_i \mathbf{v}_{b(i)}, \quad (17)$$

where \mathbf{R}_i is a 1×6 matrix. Collecting the equations for each variable, we have

$$\dot{\mathbf{x}} = \mathbf{R} \mathbf{v}, \quad (18)$$

where

$$\mathbf{R} = \begin{bmatrix} \mathbf{R}_{11} & \cdots & \mathbf{R}_{1N} \\ \vdots & \ddots & \vdots \\ \mathbf{R}_{m1} & \cdots & \mathbf{R}_{mN} \end{bmatrix}$$

and

$$\mathbf{R}_{ij} = \begin{cases} \mathbf{R}_i & \text{if } j = b(i) \\ \mathbf{0} & \text{otherwise.} \end{cases} \quad (19)$$

Thus, \mathbf{R} is a sparse $m \times N$ block matrix composed of submatrices \mathbf{R}_{ij} having dimensions 1×6 , with the property that exactly one block on each row is nonzero.

To give a concrete example of the sparsity patterns that arise, suppose that an operational space is defined for the kinematic tree in Figure 1 such that $m = 2$ and $b = [4, 7]$. In this case, we will have

$$\mathbf{R} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{R}_1 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{R}_2 \end{bmatrix}$$

and

$$\mathbf{J} = \begin{bmatrix} \mathbf{R}_1 \mathbf{S}_1 & \mathbf{R}_1 \mathbf{S}_2 & \mathbf{0} & \mathbf{R}_1 \mathbf{S}_4 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{R}_2 \mathbf{S}_1 & \mathbf{0} & \mathbf{R}_2 \mathbf{S}_3 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{R}_2 \mathbf{S}_7 \end{bmatrix}.$$

Observe that the sparsity patterns in the two rows of this matrix are the same as the patterns in rows 4 and 7 of Eq. 16. In general, submatrix \mathbf{J}_{ij} of the task Jacobian is nonzero if and only if $j \in \kappa(b(i))$.

6 Sparse Factorization of \mathbf{A}^{-1}

It was shown in Featherstone (2005) that any $\mathbf{H} \in \text{SPD}(\lambda)$ can be factorized into $\mathbf{H} = \mathbf{L}^T \mathbf{L}$ using the *LTL* factorization, or into $\mathbf{H} = \mathbf{L}_d^T \mathbf{D} \mathbf{L}_d$ using the *LTDL* factorization, where \mathbf{L} is lower-triangular, \mathbf{D} is diagonal, \mathbf{L}_d is unit-lower-triangular, and $\mathbf{L}, \mathbf{L}_d \in \text{L}(\lambda)$. Combining these factorizations with the formula in Eq. 7 yields the following factorizations of \mathbf{A}^{-1} :

$$\begin{aligned} \mathbf{A}^{-1} &= \mathbf{J}(\mathbf{L}^T \mathbf{L})^{-1} \mathbf{J}^T \\ &= \mathbf{J} \mathbf{L}^{-1} (\mathbf{J} \mathbf{L}^{-1})^T \\ &= \mathbf{Y} \mathbf{Y}^T, \end{aligned} \quad (20)$$

where

$$\mathbf{Y} = \mathbf{J}\mathbf{L}^{-1}, \quad (21)$$

and

$$\begin{aligned} \mathbf{A}^{-1} &= \mathbf{J}(\mathbf{L}_d^T \mathbf{D} \mathbf{L}_d)^{-1} \mathbf{J}^T \\ &= \mathbf{J}\mathbf{L}_d^{-1} \mathbf{D}^{-1} (\mathbf{J}\mathbf{L}_d^{-1})^T \\ &= \mathbf{Y}_d \mathbf{D}^{-1} \mathbf{Y}_d^T \\ &= \mathbf{Y}_D \mathbf{Y}_d^T, \end{aligned} \quad (22)$$

where

$$\mathbf{Y}_d = \mathbf{J}\mathbf{L}_d^{-1}, \quad \mathbf{Y}_D = \mathbf{Y}_d \mathbf{D}^{-1}. \quad (23)$$

These factorizations allow a more efficient calculation of \mathbf{A}^{-1} than is possible using Eq. 7 and standard matrix methods. The reasons for the improved efficiency are:

1. the *LTL* and *LTDL* factorizations exploit the sparsity in \mathbf{H} ,
2. $\mathbf{L}, \mathbf{L}_d \in \mathbf{L}(\lambda)$,
3. \mathbf{Y}, \mathbf{Y}_d and \mathbf{Y}_D have the same sparsity pattern as \mathbf{J} ,
4. the calculation of \mathbf{Y}, \mathbf{Y}_d and \mathbf{Y}_D can be accelerated by exploiting sparsity, and
5. the calculation of \mathbf{A}^{-1} from either \mathbf{Y} or \mathbf{Y}_d and \mathbf{Y}_D can be accelerated by exploiting sparsity.

Items 1 and 2 are covered in Featherstone (2005), while items 3 to 5 are the subject of this paper. Item 3 is a more general version of a result implicit in Rodriguez et al. (1992), and is therefore proved below.² Note that the presence of sparsity in \mathbf{Y}, \mathbf{Y}_d and \mathbf{Y}_D means that Eqs. 20 and 22 can be regarded as sparse factorizations of \mathbf{A}^{-1} .

Sparsity pattern of \mathbf{Y}

For any $\mathbf{K}, \mathbf{L} \in \mathbf{L}(\lambda)$, the product $\mathbf{K}\mathbf{L}^{-1}$ must also be an element of $\mathbf{L}(\lambda)$ because $\mathbf{L}(\lambda)$ is a group. So $\mathbf{K}\mathbf{L}^{-1}$ has the same sparsity pattern as \mathbf{K} , which implies that the operation of post-multiplication by \mathbf{L}^{-1} has preserved the sparsity pattern in each individual row of \mathbf{K} . Now, each row in \mathbf{S} has the same sparsity pattern as a row of an element of $\mathbf{L}(\lambda)$, so the post-multiplication of \mathbf{S} by \mathbf{L}^{-1} will preserve the sparsity pattern in each individual row, and therefore also in the matrix as a whole. Thus, the matrix $\mathbf{S}\mathbf{L}^{-1}$ has the same sparsity pattern as \mathbf{S} . Finally, the sparsity pattern in a product of two sparse matrices is determined by the sparsity patterns in the two multiplicands. As \mathbf{J} is the product of \mathbf{R} and \mathbf{S} , while \mathbf{Y} is the product of \mathbf{R} and $\mathbf{S}\mathbf{L}^{-1}$, it follows that \mathbf{J} and \mathbf{Y} have the same sparsity pattern. The argument extends trivially to \mathbf{Y}_d and \mathbf{Y}_D .

Observe that this proof did not rely on any special property of \mathbf{R} , and is therefore valid for any operational space. However, the utility of this result depends on the amount of sparsity in \mathbf{Y} , and that does depend on the properties of \mathbf{R} . The purpose of the restrictions described in Section 5 is partly to maximize the total amount of sparsity, and partly to give \mathbf{J} and \mathbf{Y} a sparsity pattern that is easily exploited using simple software, such as the algorithms to be described in Section 8.

²It is more general because it applies to a larger class of operational spaces, and because $\text{SPD}(\lambda)$ and $\mathbf{L}(\lambda)$ are larger than the sets of all possible joint-space inertia matrices and their triangular factors.

7 Innovations Factorization

In this section, we temporarily adopt the symbol \mathcal{M} for the joint-space inertia matrix, and let \mathbf{H} stand for the block-diagonal matrix $\text{diag}(\mathbf{S}_i^T)$. In a series of ground-breaking papers, Rodriguez and his co-workers showed that the joint-space inertia matrix of a robot mechanism could be expressed in the following form, which they called the innovations factorization:

$$\mathcal{M} = (\mathbf{1} + \mathbf{H}\phi\mathbf{K})\mathbf{D}(\mathbf{1} + \mathbf{H}\phi\mathbf{K})^T. \quad (24)$$

They also showed that

$$(\mathbf{1} + \mathbf{H}\phi\mathbf{K})^{-1} = \mathbf{1} - \mathbf{H}\psi\mathbf{K}, \quad (25)$$

which implies

$$\mathcal{M}^{-1} = (\mathbf{1} - \mathbf{H}\psi\mathbf{K})^T\mathbf{D}^{-1}(\mathbf{1} - \mathbf{H}\psi\mathbf{K}). \quad (26)$$

Although their first papers dealt only with unbranched chains, the theory was extended to general kinematic trees in Rodriguez et al. (1992). The equations above are Eqs. 28 to 30 in that paper. The matrix \mathbf{D} is $n \times n$ and block diagonal, having one block per joint; $\mathbf{H}\phi$ is the transpose of the matrix \mathbf{S} in Eq. 14; and \mathbf{K} is a $6N \times n$ block-diagonal matrix.

The Rodriguez team routinely numbered their bodies from tip to base, which is the reverse of the usual numbering order. Using their numbering scheme, the factor $\mathbf{1} + \mathbf{H}\phi\mathbf{K}$ is block lower triangular. However, if we use the numbering scheme described in Section 2 then this matrix is upper triangular. Thus, the right-hand side of Eq. 24 is the product of an upper-triangular matrix with a diagonal matrix and a lower-triangular matrix. It is therefore very closely related to the *LTDL* factorization. In fact, the relationship is so close that if every joint has only a single degree of freedom then the two factorizations are numerically identical and we can equate \mathbf{L}_d with $(\mathbf{1} + \mathbf{H}\phi\mathbf{K})^T$.

The sparsity pattern in $\mathbf{1} + \mathbf{H}\phi\mathbf{K}$ is the same as the pattern in ϕ , which in turn is the same as the pattern in \mathbf{S}^T ; so the factor $\mathbf{1} + \mathbf{H}\phi\mathbf{K}$ clearly exhibits branch-induced sparsity. Furthermore, the sparsity pattern in $\mathbf{1} - \mathbf{H}\psi\mathbf{K}$ is the same as the pattern in ψ , and it is shown in Rodriguez et al. (1992) that ψ has the same sparsity pattern as ϕ . Thus, Rodriguez et al. have already shown that the sparsity in their triangular factor is preserved under inversion—a result that mirrors the one in Appendix A.

Yet another result obtained by Rodriguez et al. is that the inverse of the operational-space inertia matrix can be expressed as

$$\mathbf{A}^{-1} = \mathbf{B}^T\psi^T\mathbf{H}^T\mathbf{D}^{-1}\mathbf{H}\psi\mathbf{B}, \quad (27)$$

where \mathbf{B}^T is the pick-off matrix, which is a special case of \mathbf{R} in Eq. 12. (See Eq. 34 in Rodriguez et al. (1992), and bear in mind that the quantity we call \mathbf{A}^{-1} is called \mathbf{A} in their paper.) This, in turn, implies the following equations for \mathbf{Y} and \mathbf{Y}_d in Eqs. 20 and 22:

$$\mathbf{Y}_d \approx \mathbf{B}^T\psi^T\mathbf{H}^T \quad (28)$$

$$\mathbf{Y} \approx \mathbf{B}^T\psi^T\mathbf{H}^T\mathbf{D}^{-1/2}. \quad (29)$$

These equations will be exact if every joint has only a single degree of freedom. Otherwise, they represent only an approximate correspondence, since the left-hand sides are triangular but the right-hand sides are only block-triangular. Nevertheless, the sparsity

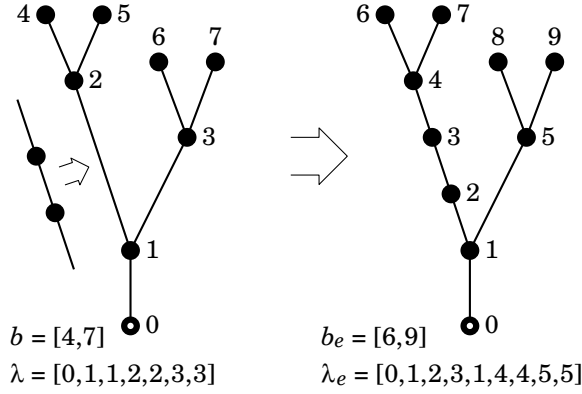


Figure 3: Expanding a connectivity graph

pattern of \mathbf{Y} and \mathbf{Y}_d can be deduced directly from these equations, since \mathbf{B} , \mathbf{H} and \mathbf{D} are all block-diagonal, and $\boldsymbol{\psi}^T$ has the same sparsity pattern as $\boldsymbol{\phi}^T$, which is the same pattern as \mathbf{S} .

From a mathematical point of view, the relationship between the innovations factorization and the *LTDL* factorization is this: the latter is a numerical procedure that can be applied to any symmetric, positive-definite matrix, whereas the former provides us with a deeper and more powerful symbolic result, but is applicable only to joint-space inertia matrices and other matrices having the same special structure.

From a computational point of view, the *LTDL* factorization is applied directly to a known matrix in order to factorize it, but the innovations factorization provides us with matrix-operator expressions for an unknown matrix, or the product of an unknown matrix with a known vector, from which a recursive algorithm can be deduced for the purpose of calculating the desired unknown quantity.

8 Algorithms

This section presents algorithms to calculate $\mathbf{J}\mathbf{x}$ and $\mathbf{J}^T\mathbf{x}$ for arbitrary vectors \mathbf{x} , and algorithms to calculate \mathbf{Y} , \mathbf{Y}_d , \mathbf{Y}_D , $\mathbf{Y}\mathbf{Y}^T$ and $\mathbf{Y}_D\mathbf{Y}_d^T$. It also shows how to calculate $\bar{\mathbf{J}}$ and \mathbf{N} efficiently using these algorithms. The *LTL* and *LTDL* algorithms are explained in Featherstone (2005, 2008) and are not repeated here.

Expanded connectivity graph

The algorithms in this section operate directly on the elements of \mathbf{J} , \mathbf{L} , etc., rather than on their submatrices. They therefore require element-oriented versions of λ and b , which we shall call λ_e and b_e . (b is defined just before Eq. 17.) Conceptually, λ_e and b_e can be regarded as the parent and body-number arrays for an expanded version of the original connectivity graph, in which each joint having more than one DoF is replaced by a chain of single-DoF joints. Another useful quantity is $\kappa_e(i)$, which is the support set for node i in the expanded connectivity graph. These sets are used in the analysis and cost figures, but not directly in the algorithms.

The idea of an expanded connectivity graph is illustrated in Figure 3. In this example, joint 2 in the mechanism from Figure 1 has 3 DoF, while the other joints each have only

```

for  $i = 1$  to  $n$  do
     $\lambda_e(i) = i - 1$ 
end
 $map(0) = 0$ 
for  $i = 1$  to  $N$  do
     $map(i) = map(i - 1) + n_i$ 
end
for  $i = 1$  to  $N$  do
     $\lambda_e(map(i - 1) + 1) = map(\lambda(i))$ 
end
for  $i = 1$  to  $m$  do
     $b_e(i) = map(b(i))$ 
end

```

Figure 4: Algorithm to calculate λ_e and b_e from λ , b and n_i

```

for  $i = 1$  to  $m$  do
     $j = b_e(i)$ 
     $y_i = J_{ij} x_j$ 
     $j = \lambda_e(j)$ 
    while  $j \neq 0$  do
         $y_i = y_i + J_{ij} x_j$ 
         $j = \lambda_e(j)$ 
    end
end

```

Figure 5: Algorithm to calculate $\mathbf{y} = \mathbf{J}\mathbf{x}$ from \mathbf{J} and \mathbf{x}

a single DoF. The expanded connectivity graph is therefore obtained from the original by replacing arc 2 with a chain of three arcs. This alteration implies the addition of two new nodes, and it necessitates a renumbering of the nodes and arcs. The latter is performed in such a manner that arc i in the expanded graph refers specifically to element i in $\dot{\mathbf{q}}$ (or $\ddot{\mathbf{q}}$ or $\boldsymbol{\tau}$).

An algorithm to calculate λ_e and b_e from λ and b is presented in Figure 4. It is an extension of an algorithm appearing in Featherstone (2008) that calculates only λ_e . Note that λ_e is a constant, and b_e changes only when the task space is redefined, so the calculations in Figure 4 need to be performed only infrequently. More on this topic can be found in Featherstone (2005, 2008).

Calculation of $\mathbf{J}\mathbf{x}$ and $\mathbf{J}^T\mathbf{x}$

Figures 5 and 6 show a pair of algorithms to calculate $\mathbf{J}\mathbf{x}$ and $\mathbf{J}^T\mathbf{x}$, respectively, for arbitrary vectors \mathbf{x} . (\mathbf{x} here is just the name of the vector being multiplied—it should not be confused with the vector of operational-space coordinates.) Note that these algorithms can also calculate $\mathbf{Y}\mathbf{x}$, $\mathbf{Y}^T\mathbf{x}$, $\mathbf{Y}_d\mathbf{x}$, and so on, because \mathbf{Y} , \mathbf{Y}_d and \mathbf{Y}_D all have the same sparsity pattern as \mathbf{J} . Also, these algorithms can easily be extended to calculate $\mathbf{J}\mathbf{X}$ or $\mathbf{J}^T\mathbf{X}$, where \mathbf{X} is a dense matrix, by replacing each instance of x_i or x_j with row i or j

```

y = 0
for  $i = 1$  to  $m$  do
     $j = b_e(i)$ 
    while  $j \neq 0$  do
         $y_j = y_j + J_{ij} x_i$ 
         $j = \lambda_e(j)$ 
    end
end

```

Figure 6: Algorithm to calculate $\mathbf{y} = \mathbf{J}^T \mathbf{x}$ from \mathbf{J} and \mathbf{x}

```

y = 0
for  $i = 1$  to  $m$  do
    for  $j = 1$  to  $n$  do
        if  $A_{ij} \neq 0$  then
             $y_i = y_i + A_{ij} x_j$ 
        end
    end
end

```

Figure 7: Basic algorithm to calculate $\mathbf{y} = \mathbf{A} \mathbf{x}$ where \mathbf{A} is sparse

of \mathbf{X} , and each instance of y_i or y_j with row i or j of the result matrix.

Both algorithms use the same basic strategy, which is to iterate over only the nonzero elements of \mathbf{J} . This strategy works as follows. If $\mathbf{y} = \mathbf{J} \mathbf{x}$ then the general formula for an element of \mathbf{y} is

$$y_i = \sum_{j=1}^n J_{ij} x_j.$$

However, we know that $J_{ij} \neq 0$ only if $j \in \kappa_e(b_e(i))$, so the formula for y_i can be simplified to

$$y_i = \sum_{j \in \kappa_e(b_e(i))} J_{ij} x_j. \quad (30)$$

The algorithm for $\mathbf{J} \mathbf{x}$ implements this formula directly, and the algorithm for $\mathbf{J}^T \mathbf{x}$ implements a very similar formula in which y_j and x_i replace y_i and x_j . In both cases, the inner **while** loop is iterating backwards over the elements of $\kappa_e(b_e(i))$.

In the interests of balance, the algorithms in Figures 5 and 6 should be compared with the well-known algorithm for dealing with sparse matrices, which is to test each element against zero before using it. Figure 7 shows an implementation of this algorithm. The advantages of this algorithm, relative to those in Figures 5 and 6, are that it works with any sparsity pattern, that it does not need to be told what the pattern is, and that incrementing a loop variable is cheaper than calculating $j = \lambda_e(j)$. Its disadvantage is that it accesses and tests every element of the given matrix, rather than accessing only the nonzero elements. If this algorithm is applied to the Jacobian in the ASIMO example given later in this paper, then it will perform exactly the same number of floating-point arithmetic operations as the algorithm in Figure 6, but it will access all 960 elements of the Jacobian rather than only the 312 nonzero elements.

```

Y = J
for  $k = 1$  to  $m$  do
   $i = b_e(k)$ 
  while  $i \neq 0$  do
     $Y_{ki} = Y_{ki}/L_{ii}$ 
     $j = \lambda_e(i)$ 
    while  $j \neq 0$  do
       $Y_{kj} = Y_{kj} - Y_{ki} L_{ij}$ 
       $j = \lambda_e(j)$ 
    end
     $i = \lambda_e(i)$ 
  end
end

```

Figure 8: Algorithm to calculate $\mathbf{Y} = \mathbf{J}\mathbf{L}^{-1}$ from \mathbf{J} and \mathbf{L}

```

Yd = J
YD = 0
for  $k = 1$  to  $m$  do
   $i = b_e(k)$ 
  while  $i \neq 0$  do
     $j = \lambda_e(i)$ 
    while  $j \neq 0$  do
       $Y_{dkj} = Y_{dkj} - Y_{dki} L_{dij}$ 
       $j = \lambda_e(j)$ 
    end
     $Y_{Dki} = Y_{dki}/D_{ii}$ 
     $i = \lambda_e(i)$ 
  end
end

```

Figure 9: Algorithm to calculate $\mathbf{Y}_d = \mathbf{J}\mathbf{L}_d^{-1}$ and $\mathbf{Y}_D = \mathbf{Y}_d \mathbf{D}^{-1}$ from \mathbf{J} , \mathbf{L}_d and \mathbf{D}

Calculation of \mathbf{Y} , \mathbf{Y}_d and \mathbf{Y}_D

An algorithm to calculate $\mathbf{Y} = \mathbf{J}\mathbf{L}^{-1}$ directly from \mathbf{J} and \mathbf{L} is shown in Figure 8. It consists of a standard back-substitution algorithm that has been modified to exploit the sparsity in both \mathbf{J} and \mathbf{L} . The first step is to copy \mathbf{J} to \mathbf{Y} . This is necessary because the rest of the algorithm works *in situ*. The index k iterates over the rows of \mathbf{Y} ; for each k , index i iterates over the elements of $\kappa_e(b_e(k))$; and, for each i , index j iterates over the elements of $\kappa_e(\lambda_e(i))$. In both **while** loops, the iteration proceeds in reverse numerical order, but this is only important for the outer **while** loop. In effect, the outer **while** loop is skipping over the zeros on row k of \mathbf{Y} , while the inner **while** loop is skipping over the zeros on row i of \mathbf{L} .

An algorithm to calculate $\mathbf{Y}_d = \mathbf{J}\mathbf{L}_d^{-1}$ and $\mathbf{Y}_D = \mathbf{Y}_d \mathbf{D}^{-1}$ directly from \mathbf{J} , \mathbf{L}_d and \mathbf{D} is shown in Figure 9. This algorithm differs from the one in Figure 8 in only three places. First, a line has been added near the top to initialize \mathbf{Y}_D to zero. This line is only necessary

```

for  $i = 1$  to  $m$  do
  for  $j = 1$  to  $i$  do
     $k = \text{ancest}(b_e(i), b_e(j), \lambda_e)$ 
    if  $k = 0$  then
       $\Lambda_{ij}^{-1} = 0$ 
    else
       $\Lambda_{ij}^{-1} = Y_{ik} Y_{jk}$ 
       $k = \lambda_e(k)$ 
      while  $k \neq 0$  do
         $\Lambda_{ij}^{-1} = \Lambda_{ij}^{-1} + Y_{ik} Y_{jk}$ 
         $k = \lambda_e(k)$ 
      end
    end
  end
end

```

Figure 10: Algorithm to calculate the lower triangle of $\mathbf{A}^{-1} = \mathbf{Y} \mathbf{Y}^T$

if \mathbf{Y}_D is to be accessed by software that does not know about its sparsity pattern, as the code in the **for** loop will only set the nonzero elements. Second, a line has been added near the bottom of the outer **while** loop to calculate each nonzero element of \mathbf{Y}_D as soon as the corresponding element of \mathbf{Y}_d has been calculated. Third, the line at the top of the outer **while** loop in Figure 8, which divides Y_{ki} by L_{ii} , has no counterpart in Figure 9 because $L_{dii} = 1$ for all i . It just so happens that the number of calculations saved by not having to divide Y_{dki} by L_{dii} exactly equals the number of extra calculations needed to calculate \mathbf{Y}_D , so the cost of this algorithm is identical to the cost of the algorithm in Figure 8.

Calculation of \mathbf{A}^{-1}

An algorithm to calculate $\mathbf{A}^{-1} = \mathbf{Y} \mathbf{Y}^T$ is shown in Figure 10. It can be adapted to calculate $\mathbf{A}^{-1} = \mathbf{Y}_D \mathbf{Y}_d^T$ simply by replacing each instance of $Y_{ik} Y_{jk}$ with $Y_{Dik} Y_{djk}$. This implies that the cost of calculating \mathbf{A}^{-1} from \mathbf{Y}_D and \mathbf{Y}_d is the same as the cost of calculating it from \mathbf{Y} . As \mathbf{A}^{-1} is symmetric, only the lower triangle is calculated. If the upper triangle is needed then it can be copied from the lower triangle.

This algorithm is just a standard matrix multiplication that has been adapted to skip over the zeros in \mathbf{Y} . In general, the formula for an element of \mathbf{A}^{-1} is

$$\Lambda_{ij}^{-1} = \sum_{k=1}^n Y_{ik} Y_{jk}.$$

However, the term $Y_{ik} Y_{jk}$ will only be nonzero for those values of k that are elements of both $\kappa_e(b_e(i))$ and $\kappa_e(b_e(j))$. Now, it can be shown that $\kappa_e(b_e(i)) \cap \kappa_e(b_e(j)) = \kappa_e(a)$, where a is the nearest common ancestor of $b_e(i)$ and $b_e(j)$; so the calculation of Λ_{ij}^{-1} simplifies to

$$\Lambda_{ij}^{-1} = \sum_{k \in \kappa_e(a)} Y_{ik} Y_{jk}. \quad (31)$$

The algorithm performs this calculation by first setting $k = a$ and then iterating backwards over the elements of $\kappa_e(a)$.

Line 3 of this algorithm calls a function, `ancest`, to calculate the nearest common ancestor of $b_e(i)$ and $b_e(j)$. This function can be implemented using the following code fragment, which calculates the nearest common ancestor to bodies p and q in a connectivity tree defined by the parent array λ :

```

while  $p \neq q$  do
  if  $p > q$  then
     $p = \lambda(p)$ 
  else
     $q = \lambda(q)$ 
  end
end

```

When the loop terminates, both p and q contain the result. As the values calculated by `ancest` depend only on the connectivity, it is possible to calculate them in advance, store them in a table, and replace line 3 with a table-lookup operation.

The case $k = 0$ in the `if` statement is only possible if the system described by λ_e consists of multiple independent robots. In this case, \mathbf{A}^{-1} will be block-diagonal with one block per robot. If it is known in advance that λ_e describes a single robot then this case can be ignored.

Calculation of $\bar{\mathbf{J}}$ and \mathbf{N}

Assuming that \mathbf{A} has already been calculated, $\bar{\mathbf{J}}$ can be calculated efficiently either from the formula

$$\bar{\mathbf{J}} = \mathbf{L}^{-1} \mathbf{Y}^T \mathbf{A}, \quad (32)$$

or from

$$\bar{\mathbf{J}} = \mathbf{L}_d^{-1} \mathbf{Y}_D^T \mathbf{A}, \quad (33)$$

depending on which factorization has been used. In both cases, the first step is to calculate $\mathbf{Y}^T \mathbf{A}$ (or $\mathbf{Y}_D^T \mathbf{A}$) using the algorithm for $\mathbf{J}^T \mathbf{x}$, and the second step is to pre-multiply the result by \mathbf{L}^{-1} (or \mathbf{L}_d^{-1}) using the back-substitution algorithms described in Featherstone (2005, 2008). The cost of the first step is the same for both formulae, but it is slightly cheaper to pre-multiply by \mathbf{L}_d^{-1} than by \mathbf{L}^{-1} .

\mathbf{N} can be calculated directly from the formula in Eq. 10, but using the algorithm for $\mathbf{J}^T \mathbf{x}$ to calculate $\mathbf{J}^T \bar{\mathbf{J}}^T$, which is the transpose of $\bar{\mathbf{J}} \mathbf{J}$.

9 Computational Costs

Table 1 shows the computational cost formulae for the algorithms in Figures 5, 6, 8 and 10, as well as the costs of calculating $\bar{\mathbf{J}}$ via Eqs. 32 and 33, and \mathbf{N} via Eq. 10. In each case, the cost formula for performing the same calculation using standard matrix arithmetic routines is also shown.

The symbols \mathbf{m} , \mathbf{a} and \mathbf{d} in this table represent the costs of performing a single floating-point multiplication, addition or division, respectively. Subtractions count as additions

operation	cost (dense)	cost (sparse)
$\mathbf{J}\mathbf{x}$	$mnm + m(n-1)\mathbf{a}$	$\sum_{i=1}^m d_i \mathbf{m} + (d_i - 1)\mathbf{a}$
$\mathbf{J}^T \mathbf{x}$	$mnm + (m-1)n\mathbf{a}$	$\sum_{i=1}^m d_i (\mathbf{m} + \mathbf{a})$
$\mathbf{J}\mathbf{L}^{-1}$	$mnd + \frac{1}{2}mn(n-1)(\mathbf{m} + \mathbf{a})$	$\sum_{i=1}^m d_i \mathbf{d} + \frac{1}{2}d_i(d_i - 1)(\mathbf{m} + \mathbf{a})$
$\mathbf{J}\mathbf{L}_d^{-1}, \mathbf{Y}_d \mathbf{D}^{-1}$	same as $\mathbf{J}\mathbf{L}^{-1}$	same as $\mathbf{J}\mathbf{L}^{-1}$
$\mathbf{Y}\mathbf{Y}^T$	$\frac{1}{2}m(m+1)(n\mathbf{m} + (n-1)\mathbf{a})$	$\sum_{i=1}^m \sum_{j=1}^i d_{ij} \mathbf{m} + \max(0, d_{ij} - 1)\mathbf{a}$
$\mathbf{Y}_d \mathbf{Y}_d^T$	same as $\mathbf{Y}\mathbf{Y}^T$	same as $\mathbf{Y}\mathbf{Y}^T$
$\mathbf{L}^{-1}(\mathbf{Y}^T \mathbf{A})$	$mn((m + \frac{1}{2}(n-1))(\mathbf{m} + \mathbf{a}) - \mathbf{a} + \mathbf{d})$	$m(D_1 + \sum_{i=1}^m d_i)(\mathbf{m} + \mathbf{a}) + mnd$
$\mathbf{L}_d^{-1}(\mathbf{Y}_d^T \mathbf{A})$	$mn((m + \frac{1}{2}(n-1))(\mathbf{m} + \mathbf{a}) - \mathbf{a})$	$m(D_1 + \sum_{i=1}^m d_i)(\mathbf{m} + \mathbf{a})$
$\mathbf{1} - \bar{\mathbf{J}}\mathbf{J}$	$mn^2\mathbf{m} + ((m-1)n^2 + n)\mathbf{a}$	$n\mathbf{a} + n(\sum_{i=1}^m d_i(\mathbf{m} + \mathbf{a}))$

where $d_i = |\kappa_e(b_e(i))|$, $d_{ij} = |\kappa_e(\text{ancest}(b_e(i), b_e(j), \lambda_e))|$, $D_1 = \sum_{i=1}^n (|\kappa_e(i)| - 1)$

Table 1: Computational cost formulae for the new algorithms

for cost purposes; and the negations needed to calculate $\mathbf{1} - \bar{\mathbf{J}}\mathbf{J}$ have been ignored. The expression $|\kappa_e(\dots)|$ means the number of elements in the set $\kappa_e(\dots)$. For the calculation of $\mathbf{Y}\mathbf{Y}^T$, both formulae present the cost of calculating only the lower triangle of the result.

Using standard matrix methods, the computational complexity of factorizing an $n \times n$ matrix is $O(n^3)$, and the complexity of calculating \mathbf{A}^{-1} , either from Eq. 7 or from an equation resembling Eq. 20 or 22, is $O(mn^2 + m^2n)$. However, by exploiting the sparsity in \mathbf{H} and \mathbf{J} , we obtain a computational complexity of $O(nd^2)$ for the *LTL* and *LTDL* factorizations, a complexity of $O(md^2)$ for calculating \mathbf{Y} , and a complexity of $O(m^2d)$ for calculating \mathbf{A}^{-1} from \mathbf{Y} , where d is the depth of the connectivity tree. The first figure is proved in Featherstone (2005), and the other two follow immediately from the formulae in Table 1 and the fact that $d_i \leq d$ and $d_{ij} \leq d$ for all i and j .

The numbers of nonzero elements in \mathbf{H} and \mathbf{J} are $O(nd)$ and $O(md)$, respectively. The cost of calculating \mathbf{H} via the composite-rigid-body algorithm is $O(nd)$ (Featherstone, 2005, 2008); and the cost of calculating \mathbf{J} , as defined in Section 5, can be seen to be $O(md)$ (at most $O(m)$ bodies, with at most $O(d)$ coordinate transform and joint motion subspace matrices to calculate per body), unless there is hidden complexity in calculating \mathbf{R} . Thus, if d is subject to a fixed upper limit then the complexity of calculating \mathbf{A}^{-1} from scratch is $O(n + m^2)$.

To see how the cost figures in Table 1 work out in practice, let us consider the case of an ASIMO Next-Generation humanoid robot (Honda, 2004) with an operational space consisting of the positions and orientations of the hands and feet. In this example, we have $n = 40$ and $m = 24$. Figure 11 shows the connectivity graph of the robot, taking the torso as the floating base; Table 2 shows the cost figures for this example; and Figure 12 displays the cost figures in the form of a bar graph. The labels $\bar{\mathbf{J}}$ and \mathbf{N} in the bar graph refer to the last two rows of Table 2. For comparison, Table 2 and Figure 12 also show the total cost of calculating \mathbf{H} and \mathbf{C} in Eq. 1, and the cost of performing the *LTDL* factorization on \mathbf{H} (i.e., the cost of calculating \mathbf{L}_d and \mathbf{D}). These figures are obtained

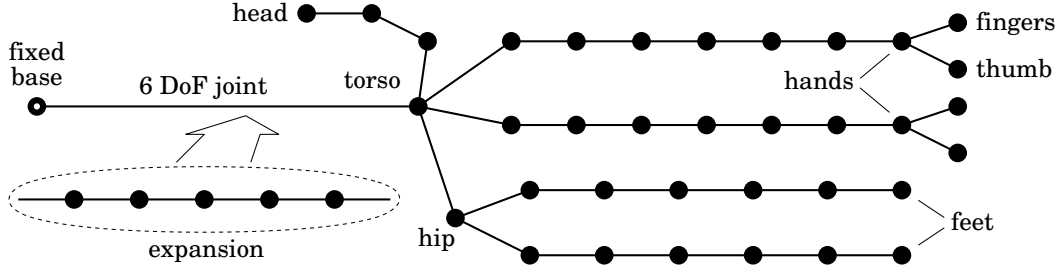


Figure 11: Connectivity graph of the ASIMO Next-Generation robot

operation	cost (dense)	cost (sparse)
H, C	$7317m + 6219a$	
$LTDL$	$780d + 10660m$ $+ 10660a$	$334d + 1779m$ $+ 1779a$
Jx	$960m + 936a$	$312m + 288a$
$J^T x$	$960m + 920a$	$312m + 312a$
JL^{-1}	$960d + 18720m$ $+ 18720a$	$312d + 1872m$ $+ 1872a$
YY^T	$12000m + 11700a$	$2424m + 2124a$
$L_d^{-1}(Y_D^T A)$	$41760m + 40800a$	$15504m + 15504a$
$1 - \bar{J}J$	$38400m + 36840a$	$12480m + 12520a$

Table 2: Computational costs for the ASIMO example

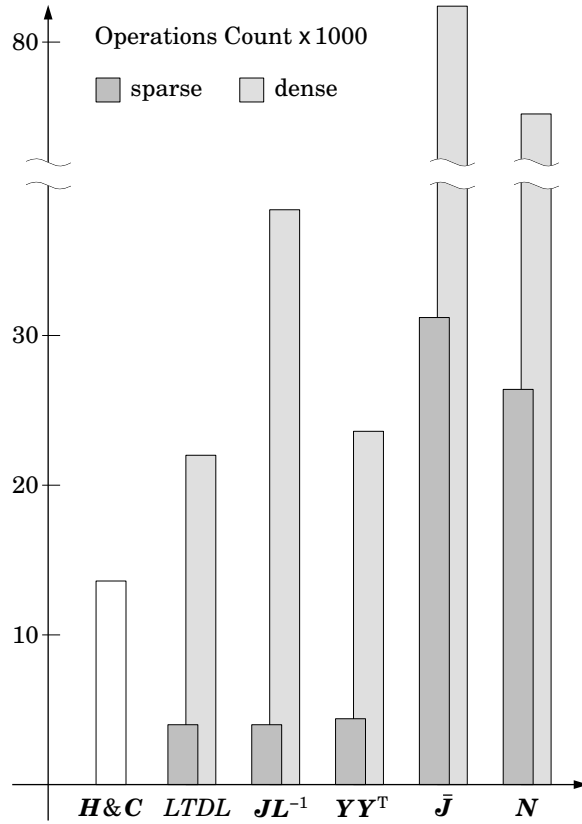


Figure 12: A comparison of operations counts

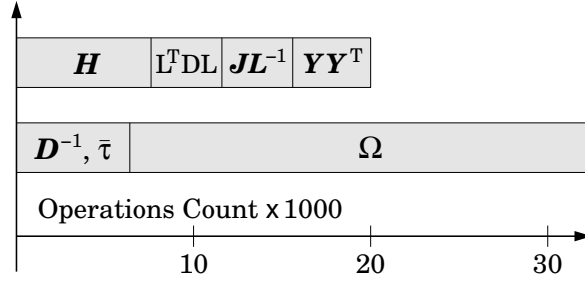


Figure 13: Sparse factorization versus the algorithm of Rodriguez et al.

using the formulae in Featherstone (2005). The lengths of the bars in Figure 12 represent the total number of floating-point operations performed in each calculation.

As can be seen, the exploitation of sparsity results in substantial reductions in computational cost. In absolute terms, the largest cost reductions occur in the calculation of $\bar{\mathbf{J}}$ and \mathbf{N} , where more than 50,000 floating-point operations are saved in each case. In relative terms, the largest cost reduction occurs in the calculation of $\mathbf{Y} = \mathbf{J}\mathbf{L}^{-1}$, where the operations count is reduced by a factor of 9.5. The total cost of calculating \mathbf{A}^{-1} from \mathbf{H} and \mathbf{J} is $646d + 6075m + 5775a$ using the new algorithms, compared with $1740d + 41380m + 41080a$ using standard matrix methods. This is a factor of 6.7 reduction in the total operations count.

A few more numbers may be of interest at this point: \mathbf{J} has 960 elements in total, of which 312 are nonzero; \mathbf{H} has 1600 elements in total, of which 708 are nonzero; and \mathbf{L} also has 1600 elements in total, of which 820 would be nonzero in a dense lower-triangular matrix, but only 374 elements of \mathbf{L} are actually nonzero. Approximately 67% of the elements of \mathbf{J} and 56% of the elements of \mathbf{H} are zeros.

Finally, let us compare the sparse factorization method with the recursive, low-order algorithm for calculating \mathbf{A}^{-1} that was invented by Rodriguez et al. (1992). To achieve a fair comparison, this algorithm has been extensively optimized; and both the optimized version and its associated cost figures are presented in Appendix B. Figure 13 shows the costs incurred by both algorithms when applied to the ASIMO example. The cost of calculating \mathbf{C} has been omitted because it is a common precursor to both algorithms. It can be seen that the cost of the low-order method is about 60% higher than the cost of the sparse factorization method on this particular example, due mainly to the high cost of calculating the matrix $\mathbf{\Omega}$. However, as the complexity of the low-order method is $O(N + m^2d)$, compared with $O(nd^2 + md^2 + m^2d)$ for the sparse factorization method, it follows that the low-order method will be faster on systems with sufficiently large d .

Clearly, the cost ratio of the two algorithms will vary as a function of the connectivity of the robot mechanism and the choice of operational space. To investigate this effect, Table 3 lists several variants of the ASIMO example along with their associated cost ratios, the latter being the cost of calculating \mathbf{D}^{-1} , $\bar{\boldsymbol{\tau}}$ and $\mathbf{\Omega}$ divided by the cost of calculating \mathbf{H} , $\mathbf{J}\mathbf{L}_d^{-1}$ and $\mathbf{Y}_D\mathbf{Y}_d^T$ and performing the *LTDL* factorization (cf. Figure 13).

The first line in Table 3 lists the data for the ASIMO example. The next line shows what happens if we increase the dimension of operational space by adding another body (the head). In this case, the cost ratio goes up to 1.923, meaning that the sparse factorization method is now more than 90% faster than the low-order algorithm. The reason for this is that the cost of calculating $\mathbf{\Omega}$ has grown by more than the cost of calculating

robot	oper. space	n	d	m	cost ratio
ASIMO	HF	40	14	24	1.618
ASIMO	HFH	40	14	30	1.923
+ 6 joints per limb	HF	64	20	24	1.369
+ 12 joints per limb	HF	88	26	24	1.095
4 arms, 4 legs	HF	70	14	48	2.496
+ 10 DoF per hand	HF	60	16	24	1.122
+ 10 DoF per hand	FTF	60	16	36	1.277

HF = hands and feet; HFH = hands, feet and head
FTF = forefingers, thumbs and feet

Table 3: Cost ratios for various robot mechanisms and operational spaces

$\mathbf{J}\mathbf{L}_d^{-1}$ and $\mathbf{Y}_D\mathbf{Y}_d^T$. Of course, the converse is also true: if we reduce the dimension of operational space by removing bodies then the cost ratio will go down.

In the next two lines, the robot mechanism is altered by adding first 6 and then 12 extra joints to each limb. The idea is to show what happens as we increase d without altering the operational space or adding any new branches to the mechanism. As expected, the cost ratio goes down. The low-order algorithm beats the sparse factorization algorithm once 15 extra joints have been added to each limb.

The next line shows what happens if we double the number of limbs. This has the effect of doubling m (twice as many hands and feet) and increasing n , but leaving d unaltered. In this case, the cost ratio has risen to almost 2.5, meaning that the sparse factorization method is almost 2.5 times faster than the low-order algorithm. This is partly because of the large increase in the cost of calculating $\mathbf{\Omega}$, and partly because the extra branches increase the sparsity of \mathbf{H} . (73% of the elements of \mathbf{H} are zero in this example.) Observe that this increase in relative efficiency has occurred despite the increase in n from 40 to 70. Thus, the exploitation of branch-induced sparsity has accelerated what would otherwise be an $O(n^3)$ algorithm to such an extent that it substantially out-performs an $O(n)$ algorithm at $n = 70$.

In the two last lines of Table 3, ASIMO’s 2-DoF hands have been replaced with 12-DoF hands comprising three fingers and a thumb, each with three independent degrees of freedom. These two examples show that the effect of branching at the hands is different from branching at the torso, since the cost ratio has gone down rather than up. There are two reasons for this. The first is that branches near the tips of a tree cause less branch-induced sparsity than branches near the root. In fact, only 64% of the elements of \mathbf{H} are zero in this case, compared with 73% for the 8-limb example; and the cost of factorizing \mathbf{H} is actually slightly higher than for the 8-limb example, despite it being a smaller matrix. The second reason, which applies only to the final example, is that the cost of calculating an off-diagonal submatrix $\mathbf{\Omega}_{ij}$ grows with the length of the path between i and j in the connectivity graph; so the total cost of calculating $\mathbf{\Omega}$ will be reduced because some submatrices $\mathbf{\Omega}_{ij}$ refer to a finger and thumb on the same hand.

10 Conclusion

This paper has shown how to exploit branch-induced sparsity when calculating the equations of operational-space dynamics for a robot mechanism having a branched kinematic structure. It is shown that the operational-space inverse inertia matrix can be expressed in the form $\mathbf{A}^{-1} = \mathbf{Y}\mathbf{Y}^T$, or $\mathbf{A}^{-1} = \mathbf{Y}_D\mathbf{Y}_d^T$, where \mathbf{Y} , \mathbf{Y}_d and \mathbf{Y}_D all have the same sparsity pattern as the task Jacobian, \mathbf{J} ; and a new method is proposed for calculating \mathbf{A}^{-1} via these expressions. The expressions themselves are not new, being implicit in the results of Rodriguez et al. (1992).

This paper also presents a collection of algorithms for multiplying vectors by the Jacobian, and for calculating the matrices \mathbf{A}^{-1} , $\bar{\mathbf{J}}$ and \mathbf{N} , the latter two being the dynamically-consistent generalized inverse and null-space projection matrix, respectively, of \mathbf{J} . These algorithms exploit the sparsity in \mathbf{J} , \mathbf{Y} , and other matrices, to achieve considerable cost savings relative to standard matrix arithmetic functions. A table of cost formulae is given, and the computational complexity of calculating \mathbf{A}^{-1} is shown to be $O(nd^2 + md^2 + m^2d)$, where m and n are the dimensions of operational space and joint space, respectively, and d is the depth of the robot's connectivity tree. The magnitude of the cost savings is illustrated by an example involving the control of the hands and feet of an ASIMO Next-Generation humanoid robot. In this example, the exploitation of sparsity cuts the cost of calculating \mathbf{A}^{-1} by a factor of 6.7, and cuts the cost of calculating $\bar{\mathbf{J}}$ and \mathbf{N} by more than 50,000 arithmetic operations each.

The new method is also compared against a highly-optimized version of the low-order algorithm for calculating \mathbf{A}^{-1} invented by Rodriguez et al. (1992). This algorithm, and its associated cost formulae, are presented in Appendix B. It is shown that the new method is 1.6 times faster on the ASIMO example; and several variants of this example are examined briefly in order to show how the cost ratio varies as a function of the robot's connectivity and the choice of operational space.

One obvious item for future work is to extend the theory to more general operational spaces in which some coordinates are functions of the relative motion of two bodies or the motion of the system center of mass.

Acknowledgments

The author wishes to thank Dr. Abhi Jain for promptly supplying corrections to the algorithm for calculating \mathbf{A}^{-1} in Rodriguez et al. (1992). The author also wishes to thank the anonymous reviewer who spotted serious omissions in the first version of this paper and made helpful suggestions for improvement.

References

- Chang, K.-S., and Khatib, O. (1999). Efficient Algorithm for Extended Operational Space Inertia Matrix. *Proc. IEEE/RSJ Int. Conf. Intelligent Robots & Systems*, Kyongju, Korea, Oct. 17–21, pp. 350–355.
- Chang, K.-S., and Khatib, O. (2000). Operational Space Dynamics: Efficient Algorithms

- for Modeling and Control of Branching Mechanisms. *Proc. IEEE Int. Conf. Robotics & Automation*, San Francisco, CA, April 24–28, pp. 850–856.
- Chang, K.-S., and Khatib, O. (2001). Efficient Recursive Algorithm for the Operational Space Inertia Matrix of Branching Mechanisms. *Advanced Robotics*, 4(8):703–715.
- Featherstone, R. (2005). Efficient Factorization of the Joint Space Inertia Matrix for Branched Kinematic Chains. *Int. J. Robotics Research*, 24(6):487–500.
- Featherstone, R. (2008). *Rigid Body Dynamics Algorithms*. New York: Springer.
- Hollerbach, J. M., and Suh, K. C. (1987). Redundancy Resolution of Manipulators through Torque Optimization. *IEEE J. Robotics & Automation*, 3(4):308–316.
- Honda Motor Co. Ltd. (2004). ASIMO: The Next Generation. <http://world.honda.com/ASIMO/next-generation/index.html>
- Jain, A. (2009). *Correction to* Rodriguez, G., Jain, A., and Kreutz-Delgado, K. (1992), Spatial Operator Algebra for Multibody System Dynamics, *J. Astronautical Sciences*, 40(1):27–50. Available at <http://dartslab.jpl.nasa.gov/References/>
- Khatib, O. (1987). A Unified Approach for Motion and Force Control of Robot Manipulators: The Operational Space Formulation. *IEEE J. Robotics & Automation*, 3(1)43–53.
- Khatib, O. (1995). Inertial Properties in Robotic Manipulation: An Object-Level Framework. *Int. J. Robotics Research*, 14(1):19–36.
- Khatib, O., Sentis, L., Park, J. and Warren, J. (2004). Whole-Body Dynamic Behavior and Control of Human-Like Robots. *Int. J. Humanoid Robotics*, 1(1):29–43.
- Kreutz-Delgado, K., Jain, A., and Rodriguez, G. (1992). Recursive Formulation of Operational Space Control. *Int. J. Robotics Research*, 11(4):320–328.
- Lilly, K. W. (1993). *Efficient Dynamic Simulation of Robotic Mechanisms*. Boston: Kluwer Academic Publishers.
- Lilly, K. W., and Orin, D. E. (1993). Efficient $O(N)$ Recursive Computation of the Operational Space Inertia Matrix. *IEEE Trans. Systems, Man & Cybernetics*, 23(5)1384–1391.
- McMillan, S., and Orin, D. E. (1995). Efficient Computation of Articulated-Body Inertias Using Successive Axial Screws. *IEEE Trans. Robotics & Automation*, 11(4):606–611.
- Rodriguez, G., Jain, A., and Kreutz-Delgado, K. (1992). Spatial Operator Algebra for Multibody System Dynamics. *J. Astronautical Sciences*, 40(1):27–50.
- Russakow, J., Khatib, O., and Rock, S. M. (1995). Extended Operational Space Formulation for Serial-to-Parallel Chain (Branching) Manipulators. *Proc. IEEE Int. Conf. Robotics & Automation*, Nagoya, Japan, May 21–27, pp. 1056–1061.
- Wittenburg, J. (1977). *Dynamics of Systems of Rigid Bodies*. Stuttgart: B. G. Teubner.

A Proof that $L(\lambda)$ is a Group

It is well known that the set of nonsingular, lower-triangular, $n \times n$ matrices forms a group under matrix multiplication. $L(\lambda)$ is clearly a subset of this group. Therefore, in order to prove that $L(\lambda)$ is a group, it is sufficient to show the following:

1. $L(\lambda)$ contains the identity matrix,
2. multiplication is closed over $L(\lambda)$, and
3. $\mathbf{L} \in L(\lambda)$ implies $\mathbf{L}^{-1} \in L(\lambda)$.

Item 1 follows from the definition of $L(\lambda)$. Each possible value of λ identifies a set of elements below the main diagonal that must be zero, but places no constraints on other elements. As the identity matrix does not contain any nonzero elements below the main diagonal, it automatically qualifies as a member of $L(\lambda)$ for any value of λ .

Item 2 can be proved as follows. Let $\mathbf{A}, \mathbf{B} \in L(\lambda)$, and let $\mathbf{C} = \mathbf{AB}$. The elements of \mathbf{C} are given by the formula

$$C_{ij} = \sum_{k=1}^n A_{ik} B_{kj}. \quad (34)$$

Now, A_{ik} can be nonzero only if $k \in \kappa(i)$, and B_{kj} can be nonzero only if $j \in \kappa(k)$, which is equivalent to $k \in \nu(j)$, so Eq. 34 simplifies to

$$C_{ij} = \sum_{k \in \kappa(i) \cap \nu(j)} A_{ik} B_{kj}. \quad (35)$$

However, the set $\kappa(i) \cap \nu(j)$ will be nonempty only if $j \in \kappa(i)$; and so C_{ij} can have a nonzero value only if $j \in \kappa(i)$, which implies $\mathbf{C} \in L(\lambda)$.

Item 3 can be proved by induction. First, we define \bar{n} to mean ‘all elements except n ’, so that $\lambda(\bar{n})$ is the subarray of λ consisting of the first $n - 1$ elements, $\mathbf{L}_{\bar{n}\bar{n}}$ is the leading $(n - 1) \times (n - 1)$ submatrix of \mathbf{L} , and so on. The proof strategy is as follows: by assuming that Item 3 holds for $L(\lambda(\bar{n}))$, we can show that $\mathbf{L} \in L(\lambda)$ implies $\mathbf{L}^{-1} \in L(\lambda)$. As the assumption is trivially true for $n = 2$, the proof follows by induction for all $n > 2$.

Given any $\mathbf{L} \in L(\lambda)$, its inverse can be written in partitioned form as

$$\mathbf{L}^{-1} = \begin{bmatrix} \mathbf{L}_{\bar{n}\bar{n}}^{-1} & \mathbf{0} \\ -\mathbf{L}_{n\bar{n}} \mathbf{L}_{\bar{n}\bar{n}}^{-1} / L_{nn} & 1 / L_{nn} \end{bmatrix}, \quad (36)$$

where $\mathbf{L}_{\bar{n}\bar{n}}^{-1}$ means $(\mathbf{L}_{\bar{n}\bar{n}})^{-1}$. Now, $\mathbf{L} \in L(\lambda)$ implies $\mathbf{L}_{\bar{n}\bar{n}} \in L(\lambda(\bar{n}))$, so we assume $\mathbf{L}_{\bar{n}\bar{n}}^{-1} \in L(\lambda(\bar{n}))$. This assumption implies that the top $n - 1$ rows of \mathbf{L}^{-1} have the correct sparsity pattern for \mathbf{L}^{-1} to be a member of $L(\lambda)$, so the only question is whether or not the bottom row also has the correct sparsity pattern. The elements on this row, lying in columns 1 to $n - 1$, are given by

$$(\mathbf{L}^{-1})_{nj} = \frac{-1}{L_{nn}} \sum_{k=1}^{n-1} L_{nk} (\mathbf{L}_{\bar{n}\bar{n}}^{-1})_{kj}, \quad (j < n) \quad (37)$$

where $(\dots)_{ij}$ means element ij of the matrix expression inside the parentheses. Now, L_{nk} can only be nonzero if $k \in \kappa(n)$, and $(\mathbf{L}_{\bar{n}\bar{n}}^{-1})_{kj}$ can only be nonzero if $j \in \kappa(k)$, which is

equivalent to the condition $k \in \nu(j) \setminus \{n\}$ (the set difference of $\nu(j)$ and $\{n\}$), so Eq. 37 simplifies to

$$(\mathbf{L}^{-1})_{nj} = \frac{-1}{L_{nn}} \sum_{k \in \kappa(n) \cap \nu(j) \setminus \{n\}} L_{nk} (\mathbf{L}_{\bar{n}\bar{n}}^{-1})_{kj}. \quad (j < n) \quad (38)$$

Once again, the set $\kappa(n) \cap \nu(j) \setminus \{n\}$ can only be nonempty if $j \in \kappa(n)$, and so the bottom row of \mathbf{L}^{-1} does have the correct sparsity pattern for \mathbf{L}^{-1} to be a member of $L(\lambda)$.

B Optimized RJK Algorithm

This appendix presents an optimized version of a recursive, low-order algorithm for calculating \mathbf{A}^{-1} that was originally described in Rodriguez et al. (1992) (especially pages 46 and 47), with corrections appearing in Jain (2009). It will be referred to here as the RJK algorithm. The purpose of this appendix is partly to present an efficient version of this algorithm, and partly to present computational cost figures for this algorithm so that they can be compared with the algorithm in the main text. (Computational cost figures for the original RJK algorithm are not available.)

The optimized version of the RJK algorithm is shown in Figure 14. The notation used here resembles the original, so that the correspondence between this algorithm and the original can easily be seen. The quantities \mathbf{H}_i here equal ${}^i\mathbf{S}_i^T$ in the main text. If every joint is revolute then $\mathbf{H}_i = [0 \ 0 \ 1 \ 0 \ 0 \ 0]$ for all i . The quantities \mathbf{M}_i are the spatial inertias of the individual bodies in the robot mechanism, expressed in link coordinates, and they are known constants. The quantities \mathbf{P}_i are articulated-body inertias, and the second loop in Figure 14 is a stripped-down version of the main pass in the standard articulated-body algorithm which has been modified to calculate the quantities \mathbf{D}_i^{-1} and $\bar{\boldsymbol{\tau}}_i$ for subsequent use. $\bar{\boldsymbol{\tau}}_i$ are force propagators, and they have the special form

$$\bar{\boldsymbol{\tau}}_i = \begin{bmatrix} 1 & 0 & \tau_{i1} & 0 & 0 & 0 \\ 0 & 1 & \tau_{i2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \tau_{i4} & 1 & 0 & 0 \\ 0 & 0 & \tau_{i5} & 0 & 1 & 0 \\ 0 & 0 & \tau_{i6} & 0 & 0 & 1 \end{bmatrix}$$

if the joints are revolute. The quantities $\boldsymbol{\phi}_i$ are the coordinate transforms that map spatial force vectors from link i coordinates to link $\lambda(i)$ coordinates. They are therefore a little different from the corresponding quantities in Rodriguez et al. (1992), which implement only a shift of origin. In the original RJK algorithm, the matrices $\boldsymbol{\phi}_i$ and $\bar{\boldsymbol{\tau}}_i$ are multiplied together to produce a set of matrices $\boldsymbol{\psi}_i = \boldsymbol{\phi}_i \bar{\boldsymbol{\tau}}_i$, which are then used in later parts of the algorithm. This is not done here because it is more efficient to keep $\boldsymbol{\phi}_i$ and $\bar{\boldsymbol{\tau}}_i$ separate.

The matrix $\boldsymbol{\Omega}$ equals $\mathbf{S} \mathbf{H}^{-1} \mathbf{S}^T$ in the main text, and it is an $N \times N$ block matrix of 6×6 submatrices $\boldsymbol{\Omega}_{ij}$. The matrix \mathbf{B} corresponds to \mathbf{R}^T in the main text, although its definition in Rodriguez et al. (1992) is more restrictive than the definition of \mathbf{R} . To facilitate the comparison of computational costs, we shall assume that a task-space velocity vector is the concatenation of one or more spatial velocity vectors, each expressed in its local link coordinate system. This assumption simplifies \mathbf{B} to such an extent that

```

for  $i = 1$  to  $N$  do
   $P_i = M_i$ 
end
for  $i = N$  down to  $1$  do
   $D_i^{-1} = (H_i P_i H_i^T)^{-1}$ 
  if  $\lambda(i) \neq 0$  then
     $\bar{\tau}_i = 1 - P_i H_i^T D_i^{-1} H_i$ 
     $P_{\lambda(i)} = P_{\lambda(i)} + \phi_i \bar{\tau}_i P_i \phi_i^T$ 
  end
end
for  $i = 1$  to  $N$  do
  if  $need(i, i)$  then
     $\Omega_{ii} = H_i^T D_i^{-1} H_i$ 
    if  $\lambda(i) \neq 0$  then
       $\Omega_{ii} = \Omega_{ii} + \bar{\tau}_i^T \phi_i^T \Omega_{\lambda(i)\lambda(i)} \phi_i \bar{\tau}_i$ 
    end
  end
end
for  $i = 1$  to  $N - 1$  do
  for  $j = i + 1$  to  $N$  do
    if  $need(i, j)$  then
      if  $ancest(i, j, \lambda) = i$  then
         $\Omega_{ij} = \Omega_{i\lambda(j)} \phi_j \bar{\tau}_j$ 
      elseif  $ancest(i, j, \lambda) > 0$  then
         $\Omega_{ij} = \bar{\tau}_i^T \phi_i^T \Omega_{\lambda(i)\lambda(j)} \phi_j \bar{\tau}_j$ 
      else
         $\Omega_{ij} = 0$ 
      end
       $\Omega_{ji} = \Omega_{ij}^T$ 
    end
  end
end
 $\Lambda^{-1} = B^T \Omega B$ 

```

Figure 14: Optimized RJK Algorithm for calculating Λ^{-1}

the final line in Figure 14 amounts to selecting a subset of $\mathbf{\Omega}$, which has a computational cost of zero. Finally, note that the quantity \mathbf{A}^{-1} is called \mathbf{A} in Rodriguez et al. (1992).

The algorithm proposed by Chang (Chang and Khatib, 1999, 2000, 2001) differs from the RJK algorithm in only one detail: it calculates $\mathbf{\Omega}_{ij}$ from $\mathbf{\Omega}_{j\lambda(i)}^T$ and $\mathbf{\Omega}_{i\lambda(j)}$ instead of $\mathbf{\Omega}_{i\lambda(j)}$ and $\mathbf{\Omega}_{\lambda(i)\lambda(j)}$. This arrangement appears to require the computation of both the upper and lower triangles of $\mathbf{\Omega}$, and it is not clear whether the symmetry of $\mathbf{\Omega}$ is to be exploited. Without a more precise description, it is not clear whether Chang's algorithm offers any consistent advantage over the RJK algorithm.

The algorithm presented in Figure 14 differs from the original RJK algorithm in the following respects.

1. The bodies are numbered from the base to the tips.
2. The connectivity is described by the parent array, λ .
3. The calculations are performed explicitly in link coordinates.
4. Only the smallest necessary subset of $\mathbf{\Omega}$ is calculated.
5. The component calculations exploit numerous efficiency tricks.

Items 1 and 2 serve to make the algorithm simpler and easier to follow. Item 3 makes explicit an implementation detail that was left unspecified in the original RJK algorithm. (The algorithm is most efficient when performed in link coordinates.) The explicit use of link coordinates in the modified algorithm accounts for the different definitions of ϕ_i here and in Rodriguez et al. (1992). Items 4 and 5 account for the efficiency improvements. Item 4 is accomplished via the quantities $need(i, j)$, while the effect of item 5 can be seen in the figures in Table 4.

The quantities $need(i, j)$ are the elements of a square array of boolean values called *need*. This array is defined such that $need(i, j)$ is true if $\mathbf{\Omega}_{ij}$ needs to be calculated, given the particular set of bodies listed in the task-space bodies array, b . An algorithm for calculating *need* is shown in Figure 15. The first line sets the whole array to 0 (=false), and the rest of the code sets some individual elements to 1 (=true). It can be seen that *need* depends only on λ and b ; so its value needs to be recalculated only relatively infrequently. In practice, the use of $need(i, j)$ greatly improves the efficiency of the RJK algorithm.

Table 4 shows the computational costs of the various individual calculations appearing in Figure 14. The multiplications $\phi_i^T(\dots)$ and $\bar{\tau}_i^T(\dots)$ cost the same as $\mathbf{\Omega}_{i\lambda(j)} \phi_j$ and $(\dots) \bar{\tau}_j$, respectively, and have therefore not been included in the table. The figures in this table include a high degree of optimization, and exploit efficiency tricks such as those described in Featherstone (2008); McMillan and Orin (1995). In particular, they assume that every joint is revolute, and that $\mathbf{H}_i = [0 \ 0 \ 1 \ 0 \ 0 \ 0]$ in link coordinates for all i .

Another optimization concerns the use of axial-screw transforms (Featherstone, 2005, 2008; McMillan and Orin, 1995). When the idea behind Denavit-Hartenberg parameters is generalized to a branched kinematic tree, we find that exactly one child of each non-terminal link can have its coordinate frame positioned such that the coordinate transform between it and its parent can be accomplished by means of two axial-screw transforms: one about the x axis and one about the z axis. All other link-to-link coordinate transforms

```

need = 0
for  $i = N$  down to 1 do
  for  $j = N$  down to  $i$  do
    if  $i \in b$  and  $j \in b$  then
       $need(i, j) = 1$ 
    end
    if  $need(i, j)$  then
      if  $i = j$  then
        if  $\lambda(i) \neq 0$  then
           $need(\lambda(i), \lambda(i)) = 1$ 
        end
        elseif  $ancest(i, j, \lambda) = i$  then
           $need(i, \lambda(j)) = 1$ 
        elseif  $ancest(i, j, \lambda) > 0$  then
           $need(\lambda(i), \lambda(j)) = 1$ 
        end
      end
    end
  end
end

```

Figure 15: Algorithm for calculating *need*

calculation	screws	cost
D_i^{-1}		1d
$\bar{\tau}_i$		5m
$\bar{\tau}_i P_i$		15m + 15a
$\phi_i(\dots) \phi_i^T$	2	60m + 62a
$P_{\lambda(i)} + (\dots)$	3	98m + 102a
$P_{\lambda(i)} + (\dots)$		21a
$\phi_i^T \Omega_{\lambda(i)\lambda(i)} \phi_i$	2	76m + 80a
$\phi_i^T \Omega_{\lambda(i)\lambda(i)} \phi_i$	3	114m + 120a
$\bar{\tau}_i^T(\dots) \bar{\tau}_i$		35m + 28a
$\Omega_{ii} + (\dots)$		1a
$\Omega_{i\lambda(j)} \phi_j$	2	120m + 72a
$\Omega_{i\lambda(j)} \phi_j$	3	144m + 108a
$(\dots) \bar{\tau}_j$		30m + 24a

Table 4: Computational costs of component calculations, assuming all joints are revolute

are general, and are therefore equivalent to three axial screws. As a 2-screw transform is significantly cheaper than a 3-screw transform, it is assumed that this cost saving is exploited wherever possible. For each coordinate transform operation in Figure 14, Table 4 lists both the 2-screw cost and the 3-screw cost.

When the modified RJK algorithm is applied to the ASIMO example, we find that the second loop in Figure 14 is executed 35 times ($N = 35$), and that $\lambda(i) \neq 0$ is true on all but one iteration. Of the 34 coordinate transforms performed by this loop, 28 are 2-screw and 6 are 3-screw transforms. The total cost of calculating \mathbf{D}_i^{-1} and $\bar{\boldsymbol{\tau}}_i$ is therefore

$$\text{cost of } \mathbf{D}_i^{-1}, \bar{\boldsymbol{\tau}}_i : \quad 2948\text{m} + 3572\text{a} + 35\text{d} . \quad (39)$$

In the third loop, we find that $need(i, i)$ is true on 28 occasions, and that $\lambda(i) \neq 0$ is true on all but one of those occasions. Of the 27 coordinate transforms performed by this loop, 24 are 2-screw and 3 are 3-screw transforms. In the final loop, $need(i, j)$ is true on 39 occasions, and every one of those is an instance of the case $ancest(i, j, \lambda) > 0$. This loop therefore performs a total of 78 coordinate transforms, of which 71 are 2-screw transforms and 7 are 3-screw transforms. From these figures, the total cost of calculating $\boldsymbol{\Omega}$ is

$$\text{cost of } \boldsymbol{\Omega} : \quad 14979\text{m} + 10803\text{a} . \quad (40)$$

In the ASIMO example, $\boldsymbol{\Omega}$ is a 35×35 matrix of 6×6 blocks, so a 210×210 matrix overall. If we were to exploit only the symmetry of $\boldsymbol{\Omega}$, then it would be necessary to calculate 630 blocks. However, only 67 elements of $need$ are true, so the optimized RJK algorithm actually calculates only 67 blocks. Given that the off-diagonal blocks are more expensive than the diagonal blocks, the use of $need$ has reduced the cost of calculating $\boldsymbol{\Omega}$ by more than a factor of 10. It can be shown that the computational complexity of this algorithm (counting only the floating-point arithmetic) is $O(N + dm^2)$.