Efficient Factorization of the Joint Space Inertia Matrix for Branched Kinematic Trees

Roy Featherstone
Department of Information Engineering
Australian National University
Canberra ACT 0200, Australia

Abstract

This paper describes new factorization algorithms that exploit branch-induced sparsity in the joint-space inertia matrix (JSIM) of a kinematic tree. It also presents new formulas that show how the cost of calculating and factorizing the JSIM vary with the topology of the tree. These formulas show that the cost of calculating forward dynamics for a branched tree can be considerably less than the cost for an unbranched tree of the same size. Branches can also reduce complexity; and some examples are presented of kinematic trees for which the complexity of calculating and factorizing the JSIM are less than $O(n^2)$ and $O(n^3)$, respectively. Finally, a cost comparison is made between an $O(n)$ algorithm and an $O(n^3)$ algorithm, the latter incorporating one of the new factorization algorithms. It is shown that the $O(n^3)$ algorithm is only 15% slower than the $O(n)$ algorithm when applied to a 30-DoF humanoid, but is 2.6 times slower when applied to an equivalent unbranched chain. This is due mainly to the $O(n^3)$ algorithm running about 2.2 times faster on the humanoid than on the chain.

1 Introduction

Forward dynamics algorithms for kinematic trees can be classified broadly into two main types: propagation algorithms and inertia-matrix algorithms. Many examples of these can be found in the literature, e.g. [1, 2, 4, 6, 8, 9, 13, 17, 18, 19, 21, 22, 23, 24]. There are also a variety of algorithms that do not fit into either category.

Propagation algorithms work by propagating constraints between bodies in such a way that the joint accelerations can be calculated one at a time. They typically have a computational complexity that is stated either as $O(N)$ or $O(n)$, where N is the number of bodies and n is the number of joint variables; but there is no real difference between them, as $O(N) = O(n)$ for a kinematic tree. Due to their complexity, propagation algorithms are often simply called $O(n)$ algorithms.

[0]This paper has been accepted for publication in the International Journal of Robotics Research, and the final (edited, revised and typeset) version of this paper will be published in the International Journal of Robotics Research, vol. 24, no. 6, pp. 487–500, June 2005 by Sage Publications Ltd. All rights reserved. © Sage Publications Ltd.

Inertia-matrix algorithms work by formulating and solving an equation of the form

$$\mathbf{H}\,\ddot{\mathbf{q}} = \boldsymbol{\tau} - \mathbf{C}\,, \tag{1}$$

where $\mathbf{H}$ is the joint-space inertia matrix (JSIM), $\ddot{\mathbf{q}}$ is the vector of joint accelerations, $\boldsymbol{\tau}$ is a vector of joint forces, and $\mathbf{C}$ is a bias vector containing gravity terms, Coriolis and centrifugal terms, and so on. This equation is a set of n linear equations in n unknowns, where n is the number of joint variables. In general, the costs of calculating $\mathbf{H}$ and $\mathbf{C}$ are $O(n^2)$ and $O(n)$, respectively, and the cost of solving Eq. 1 is $O(n^3)$. Thus, inertia-matrix algorithms have an overall complexity of $O(n^3)$, and are often referred to simply as $O(n^3)$ algorithms.

If a kinematic tree contains branches, then certain elements of the JSIM will automatically be zero. Indeed, the number of such zeros can be a large fraction of the total. This phenomenon is called branch-induced sparsity. The exact number of branch-induced zeros in a JSIM depends only on the topology of the kinematic tree, and their locations depend only on the topology and the numbering scheme (which determines the order in which joint variables appear in $\ddot{\mathbf{q}}$).

Branch-induced sparsity has a profound effect on the efficiency of inertia-matrix algorithms. If a significant fraction of the JSIM's elements are zero, then fewer calculations are required to calculate the nonzero elements, and fewer calculations to factorize the matrix. It is therefore possible for an inertia-matrix algorithm to run significantly faster on a branched kinematic tree than on an unbranched tree having the same number of joints and joint variables.

This paper therefore makes the following contributions:

1. it describes new factorization algorithms that exploit branch-induced sparsity in the factorization process; and

2. it presents new cost formulas that show how the cost of calculating and factorizing the JSIM depend on the topology of the kinematic tree.

Two factorization algorithms are presented: one that performs an LTL factorization ($\mathbf{H} = \mathbf{L}^T\,\mathbf{L}$), and one that performs an LTDL factorization ($\mathbf{H} = \mathbf{L}^T\,\mathbf{D}\,\mathbf{L}$), where $\mathbf{L}$ is a lower-triangular matrix and $\mathbf{D}$ is diagonal. These factorizations have the following special property: if they are applied to a matrix with branch-induced sparsity, then the factorization proceeds without filling in any of the zeros. Such factorizations are described as optimal in [7], and they produce factors that are maximally sparse. In contrast, the factors produced by the standard Cholesky and LDLT factorizations ($\mathbf{H} = \mathbf{L}\,\mathbf{L}^T$ and $\mathbf{H} = \mathbf{L}\,\mathbf{D}\,\mathbf{L}^T$) are dense.

The cost formulas show that the complexity of an inertia-matrix algorithm can range between $O(n)$ and $O(n^3)$, depending on the topology of the tree. This result is illustrated with a few examples of trees for which the complexity is less than $O(n^3)$. They also show that calculation costs are lower for branched trees generally, even when the complexity remains at $O(n^3)$.

This paper also presents a detailed cost comparison between an inertia-matrix algorithm and a propagation algorithm, applied to two test cases: a 30-DoF humanoid (or quadruped) mechanism and an equivalent 30-DoF unbranched chain. The inertia-matrix algorithm comprises the fastest published algorithm for calculating $\mathbf{C}$ in Eq. 1 [3], the

fastest version of the composite rigid body algorithm (CRBA) to calculate $\mathbf{H}$, and the LTDL algorithm described in this paper. The propagation algorithm is the fastest published implementation of the articulated body algorithm [14, 15]. The results of this comparison can be summarized as follows: the $O(n)$ algorithm beats the $O(n^3)$ algorithm by a factor of 2.6 on the unbranched chain, but is only about 15% faster on the humanoid. This is mainly due to the $O(n^3)$ algorithm running approximately 2.2 times faster on the humanoid than on the unbranched chain.

This is not the first paper to advocate the use of sparse matrix algorithms for robot dynamics. In particular, Baraff has described an $O(n)$ algorithm based on sparse matrix techniques [5]. However, his algorithm exploits a different pattern of sparsity in a matrix that is very different from the JSIM. Sparse matrix techniques are also used in multibody dynamics simulators, e.g. [16].

This is also not the first paper to advocate the use of an LTDL factorization on the JSIM, since this has already been proposed by Saha [20] (who calls it a UDUT factorization). He describes various interesting properties of the factorization, but does not consider branch-induced sparsity.

The rest of this paper is organized as follows. Section 2 describes the sparse factorization algorithm, and Section 3 explains how it relates to existing sparse matrix theory. Sections 4 and 5 present computational cost and complexity analyses for the sparse factorization and the CRBA, respectively, and Section 6 compares costs for a 30-DoF humanoid and an equivalent unbranched chain.

2 Sparse Factorization Algorithm

This section describes an algorithm to factorize an arbitrary $n \times n$ symmetric, positive-definite matrix, while optimally exploiting any sparsity that fits the pattern of branch-induced sparsity in a JSIM.

Zeros can appear in a JSIM for four reasons:

1. coincidental cancellations at particular configurations,
2. special values of inertia parameters,
3. special values of kinematic parameters, and
4. branches in the kinematic tree.

Zeros in the first category are transitory, whereas those in the other three are permanent. Zeros in the second and third categories are desirable for their ability to simplify dynamics calculations, and their presence is usually the result of a deliberate design strategy. Zeros in the fourth category are the subject of this paper. They can be very numerous, and can account for a large fraction of all the elements in a JSIM.

From here on, we will assume that the kinematic tree has general inertia and kinematic parameters, and is currently in a general configuration. This rules out all except branch-induced zeros. We will therefore use the term 'zero' to refer to elements of the JSIM that are branch-induced zeros, and the term 'nonzero' to refer to all other elements.

Consider the rigid-body system shown in Figure 1, which we shall call Tree 1. It is a binary kinematic tree consisting of a single fixed body, seven mobile bodies and seven

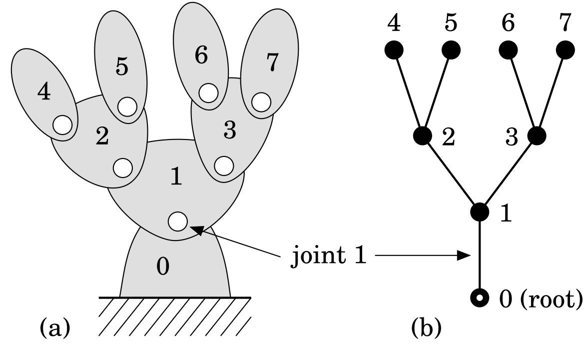

Figure 1: A binary kinematic tree (a) and its connectivity graph (b).

joints. The fixed body provides a fixed base, or fixed reference frame, for the rest of the mechanism.

In the connectivity graph of this mechanism, the bodies are represented by nodes, the joints by arcs, and the base body is the root node. The bodies are numbered according to the following rule: the base is numbered 0, and the other bodies are numbered consecutively from 1 in any order such that each has a higher number than its parent. This is called a regular numbering scheme. The joints are then numbered such that joint i connects body i to its parent.

The connectivity of a kinematic tree can be described by an array of integers called the parent array. It has one entry for each mobile body, which identifies the body number of its parent. Thus, if λ is the parent array for Tree 1, then $\lambda = (0, 1, 1, 2, 2, 3, 3)$, and $\lambda(i)$ is the parent of body i for any $i \in \{1 \dots 7\}$. Regular numbering ensures that λ has the following important property:

$$\forall i,\ 0 \leq \lambda(i) < i\,. \tag{2}$$

In the special case of an unbranched kinematic tree, $\lambda(i) = i - 1$.

If there are N mobile bodies in a kinematic tree then there are also N joints. However, the total number of joint variables can be larger than this, because individual joints can have more than one degree of freedom (DoF), hence more than one joint variable. Let n be the number of joint variables for the tree, and let n_i be the number of variables for joint i. n is then given by the formula

$$n = \sum_{i=1}^{N} n_i\,. \tag{3}$$

The joint acceleration and force vectors for the whole system will be n-dimensional vectors, and the JSIM will be an $n \times n$ matrix.

If a kinematic tree contains joints with more than one DoF then it is necessary to construct an expanded parent array for use by the factorization algorithm. This is because the parent array is one of the inputs to the algorithm, and it must have the same dimension as the matrix to be factorized. The expanded parent array is obtained from an expanded connectivity graph, which is constructed as follows:

> For each joint in turn, if $n_i > 1$ then replace joint i with a serial chain of $n_i - 1$ bodies and n_i joints. Number these extra bodies and joints consecutively, and then add $n_i - 1$ to each of the remaining body and joint numbers in the system.

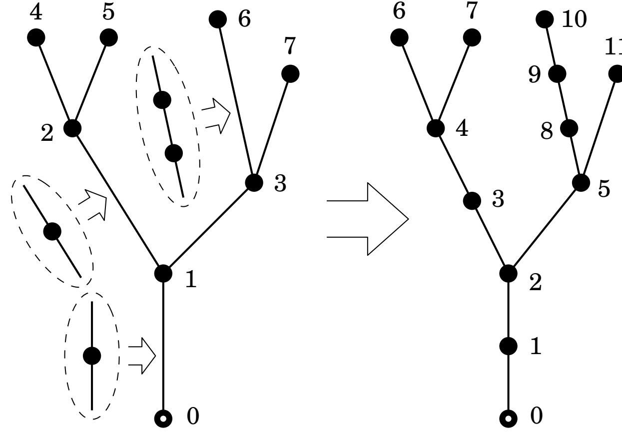

Figure 2: Expanding a connectivity graph to account for multi-DoF joints in the original mechanism.

The important property of the expanded graph is that the variable at index position i in the joint acceleration or force vector can be associated with joint i in the expanded graph. Thus, joint i in the expanded graph pertains to row and column i in the matrix.

The procedure is illustrated in Figure 2. This figure shows the connectivity graph and numbering scheme that would result if joints 1, 2 and 6 in Tree 1 were to have 2, 2 and 3 DoF, respectively. In this case, the JSIM would be an 11×11 matrix, and the factorization algorithm would need an 11-element parent array. The expanded parent array is $\lambda = (0, 1, 2, 3, 2, 4, 4, 5, 8, 9, 5)$.

Returning to Tree 1, let us assume that every joint in this mechanism is a 1-DoF joint, so that $n = N = 7$. The JSIM for this mechanism will then be a 7×7 matrix with the following sparsity pattern:

$$\mathbf{H} = \begin{bmatrix} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & & \times & \times & & \\ \times & & \times & & & \times & \times \\ \times & \times & & \times & & & \\ \times & \times & & & \times & & \\ \times & & \times & & & \times & \\ \times & & \times & & & & \times \end{bmatrix} \tag{4}$$

where '$\times$' denotes a nonzero entry in the matrix, and the zeros have been left blank. This pattern follows directly from the definition of the JSIM for a branched kinematic tree [9, 10]:

$$H_{ij} = \begin{cases} \mathbf{s}_i^T \, \mathbf{I}_i^c \, \mathbf{s}_j & \text{if } \ j \in \nu(i) \\ \mathbf{s}_i^T \, \mathbf{I}_j^c \, \mathbf{s}_j & \text{if } \ i \in \nu(j) \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where $\mathbf{s}_i$ is the motion axis of joint i, $\nu(i)$ is the set of bodies descended from body i, including body i itself, and $\mathbf{I}_i^c$ is the composite rigid-body inertia of all the bodies in $\nu(i)$. For Tree 1, $\nu(1) = \{1, 2, 3, 4, 5, 6, 7\}$, $\nu(2) = \{2, 4, 5\}$, and so on. Eq. 5 gives us the following general formula for the pattern of branch-induced sparsity in a JSIM:

$$i \notin \nu(j) \wedge j \notin \nu(i) \Rightarrow H_{ij} = 0 \,. \tag{6}$$

Another way to say this is that $H_{ij} = 0$ whenever i and j are on different branches.

If we factorize a JSIM into either $\mathbf{H} = \mathbf{L}\,\mathbf{L}^T$ or $\mathbf{H} = \mathbf{L}\,\mathbf{D}\,\mathbf{L}^T$ (standard Cholesky and LDLT factorizations, respectively), then the resulting triangular factors will be dense. However, if instead we factorize the JSIM into either $\mathbf{H} = \mathbf{L}^T\,\mathbf{L}$ or $\mathbf{H} = \mathbf{L}^T\,\mathbf{D}\,\mathbf{L}$ (LTL and LTDL factorizations, respectively), then the factorization proceeds without filling in any of the zeros in the JSIM, and produces a triangular factor that is maximally sparse for the given matrix. A factorization that accomplishes this is considered optimal [7]. The sparsity pattern for an optimally-sparse $\mathbf{L}$ is

$$i \notin \nu(j) \Rightarrow L_{ij} = 0\,,$$

and the pattern for Tree 1 is

$$\mathbf{L} = \begin{bmatrix} \times & & & & & & \\ \times & \times & & & & & \\ \times & & \times & & & & \\ \times & \times & & \times & & & \\ \times & \times & & & \times & & \\ \times & & \times & & & \times & \\ \times & & \times & & & & \times \end{bmatrix}.$$

Given any $n \times n$ symmetric, positive-definite matrix $\mathbf{H}$, and any n-element array λ, such that $\mathbf{H}$ satisfies Eq. 6 and λ satisfies Eq. 2, the following algorithm will perform an optimal, sparse LTL factorization on $\mathbf{H}$. Note that $\mathbf{H}$ need not be a JSIM, λ need not be a parent array, and $\mathbf{H}$ need not contain any branch-induced sparsity. In particular, if $\lambda(i) = i - 1$ for all i then the matrix is treated as dense. A quick way to verify that $\mathbf{H}$ satisfies Eq. 6 is to check, for each row i, that the nonzero elements below the main diagonal appear only in columns $\lambda(i)$, $\lambda(\lambda(i))$, and so on. This algorithm works in-situ, and it leaves $\mathbf{L}$ in the lower triangle of $\mathbf{H}$.

for $k = n$ **to** 1 **do**
    $H_{kk} = \sqrt{H_{kk}}$
    $j = \lambda(k)$
    while $j \neq 0$ **do**
        $H_{kj} = H_{kj} / H_{kk}$
        $j = \lambda(j)$
    end
    $i = \lambda(k)$
    while $i \neq 0$ **do**
        $j = i$
        while $j \neq 0$ **do**
            $H_{ij} = H_{ij} - H_{ki}\,H_{kj}$
            $j = \lambda(j)$
        end
        $i = \lambda(i)$
    end
end

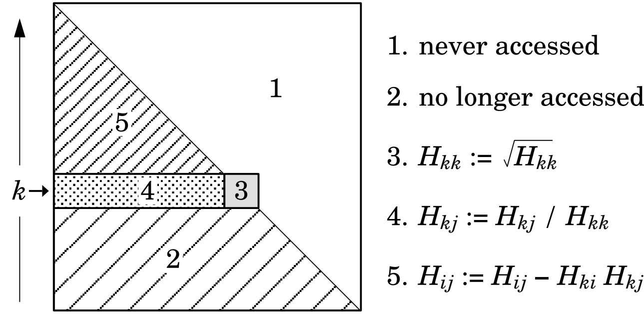

Figure 3: The factorization process.

The sparse LTDL algorithm differs only slightly from this, and is described in Appendix A. This appendix also describes algorithms for calculating the expressions $\mathbf{L}\,\mathbf{x}$, $\mathbf{L}^T\,\mathbf{x}$, $\mathbf{L}^{-1}\,\mathbf{x}$ and $\mathbf{L}^{-T}\,\mathbf{x}$, for arbitrary vectors $\mathbf{x}$, in a way that exploits the sparsity in $\mathbf{L}$.

The sparse LTL algorithm differs from a standard Cholesky algorithm in only two respects: the outer loop runs backwards from n to 1; and the inner loops iterate over the ancestors of k, i.e., $\lambda(k)$, $\lambda(\lambda(k))$, and so on, back to the root. The reversal of the outer loop is what makes this algorithm perform an LTL factorization instead of Cholesky, and the behaviour of the inner loops is what exploits the sparsity.

Figure 3 illustrates the factorization process. At the point where the algorithm begins to process row k, it has already completed the processing of rows $k+1$ to n, and these rows now contain rows $k+1$ to n of $\mathbf{L}$. The processing of row k involves three steps:

1. Replace element H_{kk} with its square root.

2. Divide the elements in region 4 (the portion of row k below the diagonal) by H_{kk}.

3. Subtract from region 5 the outer product of region 4 with itself. Thus, each element H_{ij} within region 5 is replaced by $H_{ij} - H_{ki}\, H_{kj}$.

Furthermore, in the processing of row k, the inner loops iterate only over the ancestors of body k. This has the effect of visiting only the nonzero elements in region 4, and a subset of the nonzero elements in region 5. This is where the cost savings come from—the algorithm performs the minimum possible amount of work to accomplish the factorization, given the sparsity pattern of the matrix.

Consider what happens when the algorithm is applied to the JSIM of Tree 1, which has the sparsity pattern shown in Eq. 4. Starting at $k = 7$, the inner loops iterate over only the two values 3 and 1, because $\lambda(7) = 3$, $\lambda(\lambda(7)) = 1$ and $\lambda(\lambda(\lambda(7))) = 0$ (the exit condition for the inner loops). Thus, the algorithm updates only the two elements H_{73} and H_{71} at step 2, and the three elements H_{33}, H_{31} and H_{11} at step 3. In contrast, a dense matrix factorization would update 6 elements at step 2, and a further 21 elements at step 3.

In effect, the algorithm performs a stripped-down version of the LTL factorization of a dense matrix, in which it simply skips over all the entries that are known to be zero in the original matrix, and thereby omits every operation that involves a multiplication by zero. This strategy works because the factorization process preserves the sparsity pattern of the matrix: any element that starts out zero, remains zero throughout the factorization process.

We can prove this property by induction. First, assume that rows $k+1$ to n have already been processed, and that no fill-in has yet occurred as a result of this processing; so the pattern of zeros is still the same as in the original matrix. Now, it is impossible for fill-in to occur as a result of executing steps 1 and 2 on row k, so we focus on step 3. This step affects every element H_{ij} within region 5 for which $H_{ki}\,H_{kj} \neq 0$; but this expression can only be nonzero if both i and j are ancestors of k, which in turn implies that either $i \in \nu(j)$ or $j \in \nu(i)$, which is the condition for H_{ij} to be a nonzero element of $\mathbf{H}$. Thus, no fill-in occurs during the processing of row k.

The LTL and LTDL algorithms have been tested for numerical accuracy on a variety of JSIMs, with the following results. On matrices with a substantial amount of sparsity, they tend to be slightly more accurate than Matlab's native Cholesky factorization; but on matrices with little or no sparsity, they tend to be slightly less accurate.

Compact storage schemes for sparse matrices have not been investigated, on the grounds that there is little to be gained from them unless the JSIM contains thousands of zeros. After all, even a thousand zeros is still only eight kilobytes.

3 Context

This section puts the new algorithm into its correct context within existing sparse matrix theory as described in [11]. It is shown that the new algorithm is equivalent to a reordered Cholesky factorization, and that the JSIM of a kinematic tree belongs to a class of matrices which are known to be factorizable without fill-in. Thus, the new algorithm does not accomplish anything that could not have been done using existing methods. The novelty therefore lies only in the details of its principle of operation, which results in a particularly simple and easy factorization process for JSIMs.

Let $\mathbf{A}$ be a symmetric, positive-definite matrix, and let $\mathbf{P}$ be a permutation matrix. The matrix $\tilde{\mathbf{A}} = \mathbf{P}^T\,\mathbf{A}\,\mathbf{P}$ is then a symmetric permutation of $\mathbf{A}$, and is also a symmetric, positive-definite matrix. Permutation matrices are orthogonal (i.e., $\mathbf{P}^T = \mathbf{P}^{-1}$), so it follows that $\mathbf{A} = \mathbf{P}\,\tilde{\mathbf{A}}\,\mathbf{P}^T$.

A Cholesky factorization of $\tilde{\mathbf{A}}$ into $\tilde{\mathbf{A}} = \mathbf{G}\,\mathbf{G}^T$ is called a reordered Cholesky factorization of $\mathbf{A}$, the factors being $\mathbf{P}\,\mathbf{G}$ and $(\mathbf{P}\,\mathbf{G})^T$:

$$\mathbf{A} = \mathbf{P}\,\tilde{\mathbf{A}}\,\mathbf{P}^T = \mathbf{P}\,\mathbf{G}\,\mathbf{G}^T\,\mathbf{P}^T = (\mathbf{P}\,\mathbf{G})\,(\mathbf{P}\,\mathbf{G})^T\,.$$

To obtain the LTL factorization, we introduce a second permutation, $\mathbf{Q}$, and insert a factor $\mathbf{Q}\,\mathbf{Q}^T$ (which is the identity matrix) as follows:

$$\begin{aligned} \mathbf{A} &= \mathbf{P}\,\mathbf{G}\,\mathbf{Q}\,\mathbf{Q}^T\,\mathbf{G}^T\,\mathbf{P}^T \\ &= (\mathbf{P}\,\mathbf{G}\,\mathbf{Q})\,(\mathbf{P}\,\mathbf{G}\,\mathbf{Q})^T\,. \end{aligned}$$

We then set $\mathbf{P} = \mathbf{Q} = \mathbf{R}$, where $\mathbf{R}$ is the special permutation that reverses the order of the rows of a matrix. $\mathbf{R}$ has ones along its off-diagonal (top right to bottom left) and zeros elsewhere. With this substitution, the factorization becomes

$$\mathbf{A} = (\mathbf{R}\,\mathbf{G}\,\mathbf{R})\,(\mathbf{R}\,\mathbf{G}\,\mathbf{R})^T\,.$$

Now, $\mathbf{R}\,\mathbf{G}\,\mathbf{R}$ is an upper-triangular matrix, so we may equate it with a lower-triangular matrix, $\mathbf{L}$, as follows:

$$\mathbf{L}^T = \mathbf{R}\,\mathbf{G}\,\mathbf{R}\,.$$

Having established the correspondence, it follows that the LTL factorization has the same general mathematical and numerical properties as the Cholesky factorization.

Branch-induced sparsity has a pattern that is a symmetric permutation of a standard pattern known as nested, doubly-bordered block-diagonal. This pattern is defined recursively as follows: the matrix as a whole consists of a block-diagonal sub-matrix bordered below and to the right by one or more rows and columns of nonzero entries; and zero or more of the blocks within the block-diagonal sub-matrix have the same structure.

Any JSIM can be brought into this form by the following procedure. First, construct a new regular numbering (if necessary) that traverses the tree in depth-first order; then reorder the rows and columns of $\mathbf{H}$ according to the new regular numbering; then reverse the order of the rows and columns. If this procedure is applied to $\mathbf{H}$ in Eq. 4, to produce a permuted matrix $\tilde{\mathbf{H}}$, then

$$\tilde{\mathbf{H}} = \left[\begin{array}{ccc|ccc|c} \times & & \times & & & & \times \\ & \times & \times & & & & \times \\ \times & \times & \times & & & & \times \\ \hline & & & \times & & \times & \times \\ & & & & \times & \times & \times \\ & & & \times & \times & \times & \times \\ \hline \times & \times & \times & \times & \times & \times & \times \end{array}\right] . \tag{7}$$

In the sparse matrix literature, it is usually the case that a nested, doubly-bordered block-diagonal matrix has a substantial amount of additional sparsity in its borders and atomic blocks (the ones that are not themselves nested, doubly-bordered block-diagonal). The challenge is then to devise algorithms that exploit all of the sparsity. The JSIM is a special case in which there is no additional sparsity, which makes it amenable to simpler algorithms.

One important property of $\tilde{\mathbf{H}}$ is that it contains no zeros inside its envelope. The envelope of a sparse matrix is the set of elements below the main diagonal that are either nonzero elements themselves, or are located somewhere to the right of a nonzero element. The envelope of $\tilde{\mathbf{H}}$ consists of the elements $\tilde{H}_{31}$, $\tilde{H}_{32}$, $\tilde{H}_{64}$, $\tilde{H}_{65}$ and $\tilde{H}_{71} \dots \tilde{H}_{76}$. The importance of the envelope is that a standard Cholesky factorization preserves every zero outside the envelope. As $\tilde{\mathbf{H}}$ contains no zeros inside its envelope, it follows that a standard Cholesky factorization will proceed without filling in any of the zeros in this matrix.

This is sufficient to demonstrate that an optimal factorization of a JSIM with branch-induced sparsity can be accomplished using standard techniques from sparse matrix theory. In comparison, the only advantages offered by the LTL algorithm are matters of convenience: it is easy to implement and retro-fit into existing code; it works directly on the original JSIM; and it produces factors that are literally triangular, rather than permutations of a triangular matrix.

4 Factorization Cost Analysis

This section presents general formulas for the computational cost of the sparse LTL and LTDL factorizations, and for multiplication and back-substitution operations involving the sparse factors. It also presents cost formulas for three different families of tree, to show how different topologies affect the cost.

	LTL	LTDL
factorize	$n\sqrt{} + D_1\, div + D_2\,(m+a)$	$D_1\, div + D_2\,(m+a)$
back-subst	$2n\, div + 2D_1\,(m+a)$	$n\, div + 2D_1\,(m+a)$
$\mathbf{L}\,\mathbf{x}$, $\mathbf{L}^T\,\mathbf{x}$	$n\, m + D_1\,(m+a)$	$D_1\,(m+a)$
$\mathbf{L}^{-1}\,\mathbf{x}$, $\mathbf{L}^{-T}\,\mathbf{x}$	$n\, div + D_1\,(m+a)$	$D_1\,(m+a)$

Table 1: The Cost of factorization, back-substitution and multiplying a vector by a sparse triangular factor, for the sparse LTL and LTDT algorithms.

4.1 Sparse Factorization

Let d_i be the distance between node i and the root node in the connectivity graph of a kinematic tree, measured as the number of intervening arcs (joints). The regular numbering scheme ensures that $d_i \le i$ for all i; so, for any node other than the root, $1 \le d_i \le i$. These numbers play the following role in the cost analysis. Referring back to Figure 3, the number of nonzero elements in region 4 is $d_k - 1$; so step 2 in the factorization process for row k involves $d_k - 1$ divisions, and step 3 involves making updates to $d_k(d_k - 1)/2$ elements in region 5.

Let us define the quantities

$$D_1 = \sum_{i=1}^{n} (d_i - 1) \tag{8}$$

and

$$D_2 = \sum_{i=1}^{n} \frac{d_i(d_i - 1)}{2}\,. \tag{9}$$

D_1 is the total number of step-2 operations performed by the factorization algorithm. It is also the total number of nonzero elements below the diagonal in the JSIM. D_2 is the total number of step-3 operations performed the algorithm, each such operation involving one multiplication and one subtraction. The total cost of the sparse LTL factorization is therefore

$$n\sqrt{} + D_1\, div + D_2\,(m+a)\,,$$

where the symbols $\sqrt{}$, div, m and a stand for square-root calculations, divisions, multiplications and additions, respectively, with subtractions counting as additions for cost purposes.

Table 1 presents a summary of the computational costs of factorization, back-substitution, and multiplying a vector by a sparse triangular factor or its inverse, for both the LTL and LTDT factorizations. As you can see, the LTDL algorithm comes out slightly ahead, beating the LTL algorithm by n square-root operations in the factorization process and n divisions in the back-substitution process.

The quantities D_1 and D_2 are bounded by

$$0 \le D_1 \le (n^2 - n)/2 \tag{10}$$

and

$$0 \le D_2 \le (n^3 - n)/6\,. \tag{11}$$

Thus, the asymptotic complexity of factorization can vary between $O(1)$ and $O(n^3)$, but the asymptotic complexity of back-substitution can vary only between $O(n)$ and $O(n^2)$.

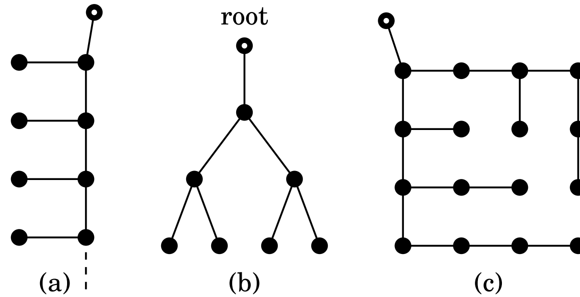

Figure 4: A kinematic chain with short side branches (a), a balanced binary tree (b) and a spanning tree for a square grid (c).

The lower limit occurs when every body is connected directly to the root (i.e., $d_i = 1$ for all i). In this case, the matrix is diagonal. Although the LTDL factorization can, in theory, be performed without any cost, the algorithm listed in Appendix A contains a loop that will iterate over all n rows, and will therefore execute $O(n)$ instructions.

The upper limit occurs when there are no branches in the kinematic tree ($d_i = i$ for all i). In this case, the matrix is dense, and the factorization costs for LTL and LTDL are identical to the costs for standard Cholesky and LDLT factorizations, respectively. However, there is a slight overhead in the inner loops of the sparse algorithms, in that assignment statements like $i = \lambda(i)$ typically take slightly longer to execute than incrementing (or decrementing) a variable. Nevertheless, the overhead is sufficiently small that there is almost nothing to lose, and potentially a lot to gain, by simply replacing Cholesky and LDLT factorizations with LTL and LTDL wherever JSIMs get factorized.

The depth of the tree has a major influence on complexity. For example, if d_i is subject to an upper limit, d_{max}, such that $d_i \leq d_{max}$ for all i, then D_1 and D_2 are bounded by

$$0 \leq D_1 \leq n\,(d_{max} - 1) \tag{12}$$

and

$$0 \leq D_2 \leq n\,d_{max}\,(d_{max} - 1)/2\,. \tag{13}$$

If d_{max} is a constant, then both D_1 and D_2 are $O(n)$. Systems with this property do occur in practice; for example, a swarm of identical mobile robots.

The exact cost of a sparse factorization or back-substitution must be calculated via D_1 and D_2; but a reasonable estimate can be obtained via the following rule of thumb. Let α be a number between 0 and 1 representing the density of the matrix to be factorized, i.e., the ratio of nonzero elements to the total number of elements in the matrix. The costs of sparse factorization and back-substitution will then be approximately α^2 and α times the costs of dense factorization and back-substitution, respectively. Thus, if 50% of the elements of a JSIM are nonzero then the sparse factorization will be about four times faster than a dense factorization, and so on.

4.2 Complexity Examples

This section examines the computational cost and complexity of the LTDL algorithm for kinematic trees with four different topologies: an unbranched chain, a chain with short

Topology	D_1	D_2	where	Order
unbranched	$(n^2-n)/2$	$(n^3-n)/6$		n^3
short s.b.	m^2	$(2m^3+3m^2+m)/6$	$n=2m$	n^3
bal. tree	$\sum_{i=1}^{m-1} i\cdot 2^i$	$\sum_{i=1}^{m-1} i\,(i+1)\cdot 2^{i-1}$	$n=2^m-1$	$n(\log(n))^2$
span grid	m^3-m^2	$(7m^4-6m^3-m^2)/12$	$n=m^2$	n^2

Table 2: Formulas for D_1 and D_2 for various tree topologies.

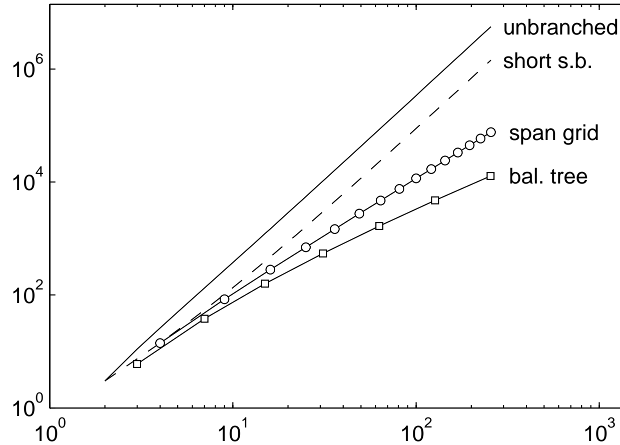

Figure 5: Comparison of factorization cost (operations count) versus n for the tree topologies in Table 2.

side branches, a balanced binary tree and a spanning tree for a square grid. The latter three are illustrated in Figure 4.

Table 2 presents formulas for D_1 and D_2 that allow their values to be calculated as functions of n. The formulas for the unbranched tree are valid for all n, and are therefore expressed directly in terms of n. The other formulas are not valid for all n, so they are expressed in terms of a second integer, m, and an expression is given in the 'where' column to indicate how n is related to m. The expression $n = 2m$, for example, implies that n must be an even number.

Figure 5 plots the factorization cost of each topology against n. The cost figures in this graph, and those quoted in the rest of this section, are based on a simple operations count; i.e., the cost of an LTDL factorization is simply the number D_1+2D_2. Cost figures in Section 6 are computed differently.

For the chain with short side branches, D_1 and D_2 converge to one half and one quarter, respectively, of their values for the unbranched case. As D_2 dominates, the cost of factorization converges to one quarter of the cost for the unbranched chain. This is a good example of the rule of thumb mentioned previously: about half the elements of the JSIM are zeros, and the factorization cost is four times less than the unbranched case.

If we increase the number of bodies in each side branch from one to two, then D_1 and D_2 converge to one third and one ninth, respectively, of their values for the unbranched case. The same would be true if we kept the branches at their present size, but had two branches per node on the main chain instead of only one. More generally, if β is the ratio of the length of the main chain to the total number of bodies, then D_1 and D_2 converge to β and β^2 times their values for the unbranched case.

The balanced binary tree shows a very different picture. In a tree containing $n = 2^m - 1$ nodes, excluding the root, there are 2^{k-1} nodes for which $d_i = k$, for $k = 1 \dots m$. So D_1 and D_2 are bounded by

$$(m-1) \cdot 2^{m-1} \leq D_1 \leq (m-1) \cdot 2^m$$

and

$$m(m-1) \cdot 2^{m-2} \leq D_2 \leq m(m-1) \cdot 2^{m-1} .$$

Thus, D_1 is $O(mn)$ and D_2 is $O(m^2 n)$, where $m \simeq \log_2 n$, and the cost of factorization is therefore $O(n(\log(n))^2)$. Compared with the cost of factorizing a dense matrix, the cost of factorizing the JSIM of a balanced binary tree is 7.8 times less at $n = 15$, and 430 times less at $n = 255$.

Figure 4(c) shows one of several possible designs of spanning tree for a square grid of nodes. Any spanning tree is acceptable, provided it connects every node in the grid to the root via a minimum-length path. The formulas in Table 2 apply to all such spanning trees. For this kind of tree, D_1 is $O(n^{1.5})$ and D_2 is $O(n^2)$. Compared with the cost of factorizing a dense matrix, the cost of factorizing a JSIM for this kind of tree is 5.3 times less at $n = 16$, and 74 times less at $n = 256$.

5 CRBA Cost Analysis

This section presents a cost formula for the CRBA, for the case of a branched kinematic tree having general geometry, general inertia parameters, revolute joints, and an optional floating base. A 6-DoF joint connects a floating base to the fixed base.

Consider the following implementation of the CRBA for a kinematic tree, which calculates the JSIM as defined in Eq. 5. It is, essentially, the algorithm described in [9, §7.2], and it differs from the version in [10] only in the order in which the calculations are performed. This implementation assumes 1-DoF joints.

for $i = 1$ **to** n **do**
  $\mathbf{I}_i^c = \mathbf{I}_i$
end
for $i = n$ **to** 1 **do**
  $\mathbf{f} = \mathbf{I}_i^c \, \mathbf{s}_i$
  $H_{ii} = \mathbf{s}_i^T \, \mathbf{f}$
  if $\lambda(i) \neq 0$ **then**
    $\mathbf{I}_{\lambda(i)}^c = \mathbf{I}_{\lambda(i)}^c + {}^{\lambda(i)}\mathbf{X}_i^F \, \mathbf{I}_i^c \, {}^i\mathbf{X}_{\lambda(i)}^M$
  end
  $j = i$
  while $\lambda(j) \neq 0$ **do**
    $\mathbf{f} = {}^{\lambda(j)}\mathbf{X}_j^F \, \mathbf{f}$
    $j = \lambda(j)$
    $H_{ij} = H_{ji} = \mathbf{s}_j^T \, \mathbf{f}$
  end
end

This algorithm calculates every nonzero element of the JSIM. It does not initialize or access the zeros. If the JSIM is to be accessed by other software that is not aware of its sparsity structure, then the zero elements must be initialized to zero at least once. This can be done when the matrix is created or allocated, or before its first use. If other software fills in the zeros (e.g. by using a dense factorization algorithm) then the zero elements must be initialized each time the JSIM is calculated.

The quantities appearing in this algorithm are as follows. All are expressed in link coordinates. $\mathbf{s}_i$ is a 6-D vector representing the axis of joint i; and $\mathbf{I}_i$ and $\mathbf{I}_i^c$ are 6×6 matrices representing the rigid-body inertias of link i and body C_i, respectively, where C_i is the composite rigid body formed by the rigid assembly of all the links in $\nu(i)$. $\mathbf{s}_i$ and $\mathbf{I}_i$ are constants in link-i coordinates. ${}^{\lambda(i)}\mathbf{X}_i^F$ and ${}^i\mathbf{X}_{\lambda(i)}^M$ are coordinate transformation matrices that transform a force vector or a motion vector, respectively, from the coordinate system indicated in the subscript to the coordinate system indicated in the leading superscript. They are related by $({}^{\lambda(i)}\mathbf{X}_i^F)^T = {}^i\mathbf{X}_{\lambda(i)}^M$. Finally, $\mathbf{f}$ is a vector representing the force required to impart an acceleration of $\mathbf{s}_i$ to C_i. This force is first calculated in link-i coordinates, and is then transformed successively to the coordinate systems of link $\lambda(i)$, link $\lambda(\lambda(i))$, and so on.

For cost calculation purposes, it is assumed that the coordinates of $\mathbf{s}_i$ consist of five zeros and a one, so that the multiplications $\mathbf{I}_i^c\,\mathbf{s}_i$ and $\mathbf{s}_i^T\,\mathbf{f}$ simplify to selecting a column from $\mathbf{I}_i^c$ and an element from $\mathbf{f}$, respectively. Given these assumptions, the computational cost of the above algorithm can be expressed as

$$D_0\,(ra + rx) + D_1\,vx\,, \tag{14}$$

where the symbols ra, rx and vx stand for the operations 'rigid-body add', 'rigid-body transform' and 'vector transform', respectively. D_0 is the number of mobile bodies in the system that are not connected directly to the base; i.e., the number of bodies for which $\lambda(i) \neq 0$, or the number of bodies for which $d_i > 1$. D_0 can be expressed as

$$D_0 = \sum_{i=1}^{n} \min(1, d_i - 1) = \sum_{i=1}^{n} \min(1, \lambda(i))\,, \tag{15}$$

and its value lies in the range

$$0 \leq D_0 \leq n - 1\,. \tag{16}$$

The extreme case of $D_0 = 0$ occurs when every mobile body is connected directly to the base. In this case, $d_i = 1$ for all i, $D_0 = D_1 = 0$, the JSIM is a diagonal matrix, and its value is constant. The run-time cost of calculating this matrix is therefore zero, and so the theoretical minimum complexity of the CRBA is $O(1)$; but the algorithm above does not reach this theoretical minimum because it contains loops with an execution cost of $O(n)$.

A system like this is highly unusual, and mainly of theoretical interest. Most practical systems will have a value of D_0 that is either equal to or slightly less than the maximum possible value. Examples of systems in which $D_0 < n - 1$ include systems representing multiple independent robots, and the spanning trees of closed-loop mechanisms with multiple connections to the base (such as a typical parallel robot).

Equation 14 clearly shows that the cost of the CRBA depends on D_0 and D_1, rather than directly on n. For an unbranched kinematic chain, both D_0 and D_1 take their maximum possible values, and the asymptotic complexity will be $O(n^2)$. If the tree contains

branches, then D_1 (at least) will be smaller, and the computational cost correspondingly less. From the data in Table 2, the cost of the CRBA for a chain with short side branches should converge to half the cost of the unbranched case for the same number of bodies; and the asymptotic complexity of the CRBA is $O(n \log(n))$ for a balanced binary tree and $O(n^{1.5})$ for the spanning tree of a square grid.

Much effort has gone into finding minimum-cost implementations for operations like ra, rx and vx. The minimum cost for ra is $10a$, but the minimum costs for the other two depend on how the link coordinate frames are defined. This is where the situation becomes more complicated for a branched tree than for an unbranched tree.

The most efficient implementations of rx and vx require that the coordinate frames be located in accordance with a set of DH parameters [9, 12], in which case the coordinate transformations implied by ${}^{\lambda(i)}\mathbf{X}_i^F$ and ${}^{i}\mathbf{X}_{\lambda(i)}^M$ can be accomplished via the successive application of two axial screw transforms: one aligned with the x axis, and one aligned with the z axis [9, 14]. The current best figures for these operations are $32m + 33a$ for rx, and $20m + 12a$ for vx (Table II in [14]).[1]

However, as explained in [12], if a node has two or more children, then only one child can have the benefit of DH parameters, while the others must use a general coordinate transformation instead (unless the mechanism has a special geometry). We shall use the terms 'DH node' and 'non-DH node' to refer to those nodes that do have the benefit of DH parameters, and those that do not, respectively. The root node has no parent, and is therefore excluded from this classification.

The number of non-DH nodes is determined by the connectivity graph. If $c(i)$ is the number of children of node i, then the number of non-DH nodes is given by the formula

$$\textit{non-DH} = \sum_{i=0}^{N} \max(0, c(i) - 1)\,, \tag{17}$$

where N is the number of mobile bodies in the tree. In an unbranched chain, $c(i) \leq 1$ for all i, so every node is a DH-node; but Tree 1 has three nodes with two children each, and therefore has three non-DH nodes. Referring back to Figure 1(b), one child of node 1, one child of node 2 and one child of node 3 must be non-DH nodes, but we are free to choose which child in each case.

The best figures for rx and vx for a non-DH node are $47m + 48a$ and $24m + 18a$, respectively. The former is the cost of three successive axial screws, according to the figures in [14], and the latter comes from Table 8-3 in [9].

To account for these differences in cost, it is necessary to separate D_0 and D_1 each into two components: one to count how many operations are performed at the DH nodes, and the other to count operations at the non-DH nodes. Thus, we seek an expanded cost formula of the form

$$\begin{aligned} &D_{0a}\,(ra + rx_a) + D_{1a}\,vx_a \\ &+ D_{0b}\,(ra + rx_b) + D_{1b}\,vx_b\,, \end{aligned} \tag{18}$$

where the subscripts a and b refer to the DH and non-DH nodes, respectively. We already know the cost figures for ra, rx and vx, so we just need expressions for $D_{0a} \ldots D_{1b}$.

[1] After discussing the matter with Prof. Orin, I have used $15m + 15a$ as the base cost of a screw transform, instead of the $15m + 16a$ that appears in this table.

D_{0a} counts how many times $ra+rx$ is performed at a DH node; D_{0b} counts how many times $ra+rx$ is performed at a non-DH node; and so on. An inspection of the CRBA implementation listed above reveals that it performs $ra+rx+|\nu(i)|\,vx$ at each node i satisfying $\lambda(i)\neq 0$, where $|\nu(i)|$ is the number of elements in $\nu(i)$. So let us define the sets π_a and π_b to be the set of all DH nodes, and the set of all non-DH nodes, respectively, that are not directly connected to the root. It follows immediately from these definitions that

$$\begin{aligned} D_{0a} &= |\pi_a|\,, \qquad & D_{1a} &= \textstyle\sum_{i\in\pi_a} |\nu(i)|\,, \\ D_{0b} &= |\pi_b|\,, \qquad & D_{1b} &= \textstyle\sum_{i\in\pi_b} |\nu(i)|\,. \end{aligned} \tag{19}$$

Before moving on, let us tie up one loose end. It is clearly necessary that $D_{0a}+D_{0b}=D_0$ and $D_{1a}+D_{1b}=D_1$. The first equation follows directly from the definitions of D_0, π_a and π_b. The second equation can be proved as follows. Starting from Eq. 8,

$$\begin{aligned} D_1 &= \sum_{i=1}^{n} (d_i - 1) \\ &= \sum_{i=1}^{n} (|\nu(i)| - 1) \\ &= \sum_{i=1}^{n} |\nu(i)| - n \\ &= \sum_{i\in\pi_a} |\nu(i)| + \sum_{i\in\pi_b} |\nu(i)| + \sum_{i\notin(\pi_a\cup\pi_b)} |\nu(i)| - n \\ &= \sum_{i\in\pi_a} |\nu(i)| + \sum_{i\in\pi_b} |\nu(i)|\,. \end{aligned}$$

To follow the first step, observe that d_i-1 and $|\nu(i)|-1$ are the numbers of nonzero elements on row i below and above the main diagonal, respectively. Thus, the summations on the first and second lines count the total number of nonzero elements below and above the main diagonal, respectively. As the matrix is symmetrical, the two are the same. The final step uses the fact that $\sum_{i\notin(\pi_a\cup\pi_b)} |\nu(i)|$ is just a count of all the descendants of the root node, and therefore evaluates to n.

Given $D_{0a}\dots D_{1b}$ and the previously-mentioned costs for ra, rx and vx, the computational cost of the CRBA for a branched kinematic tree is

$$\begin{aligned} &D_{0a}\,(32m+43a) + D_{1a}\,(20m+12a) \\ &+ D_{0b}\,(47m+58a) + D_{1b}\,(24m+18a)\,. \end{aligned} \tag{20}$$

When choosing the DH nodes, any choice that maximizes D_{1a} is optimal.

Equation 20 gives the cost of the CRBA for a fixed-base system; but there is one more optimization one can make for a floating-base system, which exploits the fact that three of the DH parameters between links i and $\lambda(i)$ can be set to zero if link $\lambda(i)$ happens to be a floating base [15]. This allows a saving of $18m+21a$ on each affected rx operation, and $12m+8a$ on each affected vx operation. To incorporate this optimization into the cost formula, we define a third set, π_c, which is the set of DH nodes that are children of a floating base. We then define the numbers $D_{0c}=|\pi_c|$ and $D_{1c}=\sum_{i\in\pi_c}|\nu(i)|$, which count the number of occurrences of the rx and vx savings, respectively. The final cost

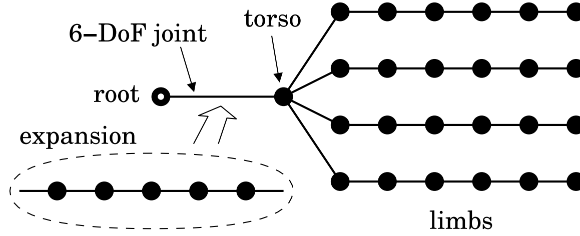

Figure 6: Connectivity graph of a 30-DoF humanoid or quadruped mechanism consisting of a torso and four 6-DoF limbs.

formula is then

$$\begin{aligned} & D_{0a}\,(32m+43a)+D_{1a}\,(20m+12a) \\ & +D_{0b}\,(47m+58a)+D_{1b}\,(24m+18a) \\ & -D_{0c}\,(18m+21a)-D_{1c}\,(12m+8a)\,. \end{aligned} \tag{21}$$

6 Practical Example

The purpose of this section is to illustrate the effect of branches on the cost of forward dynamics calculations. To this end, it presents cost figures for two rigid-body systems: a simple humanoid (or quadruped) consisting of a single rigid torso and four 6-DoF limbs, and an unbranched chain with a floating base at one end. Both systems have 25 bodies connected together by 24 revolute joints; both have a floating base, hence 30 DoF in total; and both have general geometry and general inertias. Thus, the only relevant difference between them is that one is branched and the other is not.

The figures presented here support the following two statements:

1. an $O(n^3)$ algorithm based on the CRBA and the LTDL factorization can calculate the dynamics of the humanoid more than twice as quickly as it can calculate the dynamics of the equivalent unbranched chain; and

2. this $O(n^3)$ algorithm can calculate the dynamics of the humanoid almost as quickly as the fastest $O(n)$ algorithm.

The obvious conclusion is that $O(n^3)$ algorithms are still competitive with $O(n)$ algorithms, even at relatively high values of n like $n = 30$, provided there is a sufficient amount of branching in the kinematic tree.

In the paragraphs that follow, we will be comparing the costs of various calculations by quoting cost ratios. Unfortunately, these numbers depend on the relative cost of an addition compared to a multiplication, and there is no agreed value for this ratio. We can overcome this difficulty by quoting two figures: one based on the assumption that additions are free (i.e., $a = 0$), and the other based on the assumption that an addition costs the same as a multiplication (i.e., $a = m$). These represent two extreme cases, and any realistic assumption about the cost of an addition will lie somewhere in between. Thus, if we say that calculation X is 2.5 times faster than calculation Y if $a = 0$, and 2.6 times faster if $a = m$, then the realistic cost ratio will lie somewhere in between. Divisions will be counted as multiplications for cost calculation purposes.

The first step is to calculate the various D quantities for the humanoid. These are obtained from the connectivity graph, which is shown in Figure 6. As the torso is connected to the fixed base via a 6-DoF joint, we will actually be using two connectivity graphs: the original graph and the expanded graph, where the latter is obtained from the former by replacing the 6-DoF joint with the expansion shown inset in the diagram. To avoid possible confusion, we shall apply a superscript e to the D quantities obtained from the expanded graph.

The factorization and back-substitution costs depend on D_1^e and D_2^e, which are obtained from the expanded graph as follows. From Figure 6, one can see that there is one body in the expanded graph for which $d_i = 1$, one body for which $d_i = 2$, and so on, up to $d_i = 6$, this body being the torso. Thereafter, there are four bodies for which $d_i = 7$, four for which $d_i = 8$, and so on, up to $d_i = 12$. Thus, from Eqs. 8 and 9,

$$D_1^e = \sum_{i=1}^{6} (i-1) + 4 \times \sum_{i=7}^{12} (i-1) = 219$$

and

$$D_2^e = \sum_{i=1}^{6} \frac{i\,(i-1)}{2} + 4 \times \sum_{i=7}^{12} \frac{i\,(i-1)}{2} = 1039\,.$$

These numbers imply that 48% of the elements in the humanoid's JSIM (which is a 30×30 matrix) are branch-induced zeros. To see this, recall that D_i^e is the number of nonzero elements below the main diagonal, as mentioned in Section 4.1. The total number of nonzero elements is therefore $2\,D_1^e + 30 = 468$, hence the number of zeros is $900 - 468 = 432$.

Using the formulas in Table 1, the cost of factorizing the humanoid's JSIM (setting $div = m$ for cost purposes) is $1258m + 1039a$, and the cost of back-substitution is $468m + 438a$ per vector. For comparison, the cost of factorizing a dense 30×30 matrix is $4930m + 4495a$, and the cost of back-substitution is $900m + 870a$ per vector.

To calculate the cost of the CRBA, we need the quantities $D_{0a} \ldots D_{1c}$ pertaining to the original connectivity graph. In this graph, the torso is the only body for which $d_i = 1$; then there are four bodies for which $d_i = 2$, four for which $d_i = 3$, and so on, up to $d_i = 7$. Thus, from Eqs. 15 and 8,

$$D_0 = \sum_{i=1}^{25} \min(1, d_i - 1) = 24$$

and

$$D_1 = 0 + 4 \times \sum_{i=2}^{7} (i-1) = 84\,.$$

The torso is the only body with more than one child. Only one child can be a DH node, so the other three are non-DH nodes. These are the only three non-DH nodes in the graph. Each non-DH node is the root of a subtree containing six nodes, so $|\nu(i)| = 6$ for each one. Thus, from Eq. 19, we have

$$D_{0b} = 3\,, \qquad D_{1b} = 3 \times 6 = 18\,,$$
$$D_{0a} = 21\,, \qquad D_{1a} = D_1 - D_{1b} = 66\,.$$

And finally, there is only one DH node that is the child of a floating base, and this node also has $|\nu(i)| = 6$, so

$$D_{0c} = 1\,, \qquad D_{1c} = 6\,.$$

For comparison, the corresponding figures for the equivalent unbranched chain are:

$$\begin{aligned} D_{0a} &= 24\,, \qquad & D_{1a} &= 300\,, \\ D_{0b} &= 0\,, & D_{1b} &= 0\,, \\ D_{0c} &= 1\,, & D_{1c} &= 24\,. \end{aligned}$$

Plugging these values into Eq. 21 gives the cost of calculating the JSIM of the humanoid as $2475m+2124a$, and the cost of calculating the JSIM of the equivalent unbranched chain as $6462m + 4419a$. Thus, the CRBA runs more than twice as quickly on the humanoid than on the equivalent unbranched chain. The cost ratio is 2.37 if we assume $a = m$, and 2.61 if we assume $a = 0$.

Let us now look at the cost of the complete forward dynamics calculation for both the humanoid mechanism and its equivalent unbranched chain; and let us compare these figures with the cost of calculating the forward dynamics via an efficient $O(n)$ algorithm. For this comparison, we need a set of cost figures for an inverse dynamics algorithm (to calculate $\mathbf{C}$ in Eq. 1), and for an $O(n)$ forward dynamics algorithm.

For the inverse dynamics, we shall use the efficient implementation of the recursive Newton-Euler algorithm (RNEA) described in [3] as Algorithm 3. The cost formula for this algorithm is $(93n-69)m+(81n-66)a$. However, these figures refer to an unbranched kinematic chain with a fixed base. To account for a floating base, we first increase n to 25, which accounts for the extra joint, but costs it as revolute; then we add a correction term, $7m + 13a$, which is the difference between the cost of a 6-DoF joint and a revolute joint if both are connected to the root node. This gives us a cost figure of $2263m + 1972a$ for the equivalent chain. To get a figure for the humanoid, we add a further correction of $3 \times (4m + 8a)$, which accounts for the extra transformation costs at the three non-DH nodes, giving a cost figure of $2275m + 1996a$. These correction terms are specific to this algorithm.

For the $O(n)$ forward dynamics, we shall use the figures in Table II of [15], which pertain to a very efficient implementation of the articulated body algorithm (ABA). According to this table, the cost formula for an unbranched chain with a floating base is $(224n-30)m+(205n-37)a$. This immediately gives us a figure of $5346m+4883a$ for the cost of the ABA on the equivalent unbranched chain. To get a figure for the humanoid, we must add a correction of $3 \times (66m + 57a)$ to account for the additional transformation costs incurred at the three non-DH nodes, resulting in a total cost of $5544m+5054a$. This correction term is specific to the algorithm described in [15], and it was obtained with the aid of data in Table II of [14].

Based on these figures, the cost of the complete forward dynamics calculation via the CRBA is $6476m + 5597a$ for the humanoid, and $14555m + 11756a$ for the equivalent unbranched chain. As a result of the branches in the kinematic tree, the dynamics calculation for the humanoid is 2.18 times faster than for the unbranched chain assuming $a = m$, or 2.25 times faster assuming $a = 0$. It is also only about 14% slower than the ABA assuming $a = m$, or 17% slower assuming $a = 0$. These results are summarized in Figure 7. Observe that the ABA's speed advantage is almost completely wiped out by

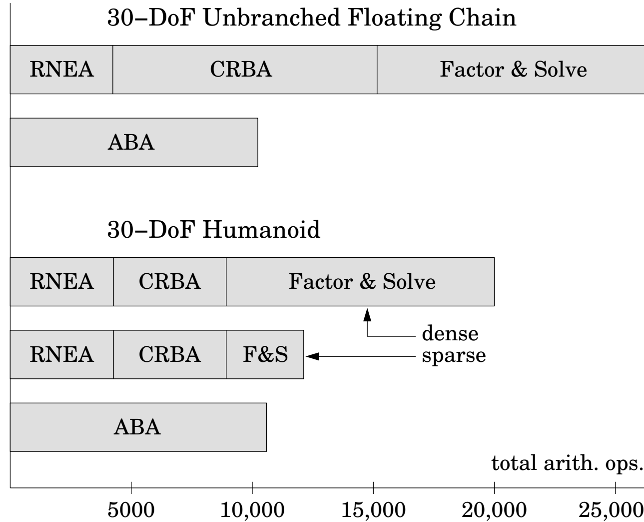

Figure 7: Computational cost of forward dynamics, via CRBA and ABA, for the humanoid mechanism shown in Figure 6 and an equivalent unbranched chain.

the cost reduction due to branch-induced sparsity. In round numbers, the ABA is faster by a factor of 2.6 on the unbranched chain, but only 15% faster on the humanoid. One may therefore conclude that $O(n^3)$ algorithms are still competitive with $O(n)$ algorithms, even at quite high values of n like $n = 30$, provided there is sufficient branching in the kinematic tree.

7 Conclusion

This paper has presented a new factorization algorithm that fully exploits branch-induced sparsity in the joint-space inertia matrix (JSIM). It is simple to implement and use, and it incurs almost no overhead compared with standard algorithms; yet it can deliver large reductions in the cost of factorizing a JSIM, and in the cost of using the resulting sparse factors. The complexity of factorization depends on the number of nonzero elements, rather than the size of the matrix, and the theoretical lower limit is $O(1)$. Some examples are presented of kinematic trees with factorization complexities ranging from $O(n(\log(n))^2)$ to $O(n^3)$.

This paper also presented a cost and complexity analysis for the composite rigid body algorithm (CRBA) for the case of a branched kinematic tree. It is shown that the cost of this algorithm can be considerably less for a branched kinematic tree than for an equivalent unbranched chain, and that the theoretical lower limit on the complexity of the CRBA is $O(1)$.

Finally, this paper presented the results of a detailed costing of the forward dynamics of a 30-DoF branched kinematic tree that could represent either a simple humanoid robot or a quadruped, and an equivalent 30-DoF unbranched chain. It was shown that an $O(n^3)$ algorithm incorporating the CRBA and a sparse factorization algorithm runs 2.6 times slower than an efficient $O(n)$ algorithm (the articulated-body algorithm) on the unbranched chain, but only 15% slower on the humanoid. This is mainly due to the $O(n^3)$ algorithm running 2.2 times faster on the humanoid than on the unbranched chain.

Thus, an $O(n^3)$ algorithm can be competitive with an $O(n)$ algorithm, even at $n = 30$, if there are enough branches in the tree, and if the branch-induced sparsity is fully exploited.

This paper did not consider systems with kinematic loops; but the results are relevant to any dynamics algorithm that solves closed-loop dynamics via the JSIM of the spanning tree.

References

[1] Angeles, J., and Ma, O. 1988. Dynamic Simulation of n-Axis Serial Robotic Manipulators Using a Natural Orthogonal Complement. *Int. J. Robotics Research*, vol. 7, no. 5, pp. 32–47.

[2] Bae, D. S., and Haug, E. J. 1987. A Recursive Formulation for Constrained Mechanical System Dynamics: Part I: Open Loop Systems. *Mechanics of Structures and Machines*, vol. 15, no. 3, pp. 359–382.

[3] Balafoutis, C. A., Patel, R. V., and Misra, P. 1988. Efficient Modeling and Computation of Manipulator Dynamics Using Orthogonal Cartesian Tensors. *IEEE J. Robotics & Automation*, vol. 4, no. 6, pp. 665–676.

[4] Balafoutis, C. A., and Patel, R. V. 1989. Efficient Computation of Manipulator Inertia Matrices and the Direct Dynamics Problem. *IEEE Trans. Systems, Man & Cybernetics*, vol. 19, no. 5, pp. 1313–1321.

[5] Baraff, D. 1996. Linear-Time Dynamics using Lagrange Multipliers. Proc. SIGGRAPH '96, New Orleans, August 4–9, pp. 137–146.

[6] Brandl, H., Johanni, R., and Otter, M. 1988. A Very Efficient Algorithm for the Simulation of Robots and Similar Multibody Systems Without Inversion of the Mass Matrix. *Theory of Robots*, P. Kopacek, I. Troch & K. Desoyer (eds.), Oxford: Pergamon Press, pp. 95–100.

[7] Duff, I.S., Erisman, A. M., and Reid, J. K. 1986. *Direct Methods for Sparse Matrices.* Oxford: Clarendon Press.

[8] Featherstone, R. 1983. The Calculation of Robot Dynamics Using Articulated-Body Inertias. *Int. J. Robotics Research*, vol. 2, no. 1, pp. 13–30.

[9] Featherstone, R. 1987. *Robot Dynamics Algorithms.* Boston: Kluwer Academic Publishers.

[10] Featherstone, R., and Orin, D. E. 2000. Robot Dynamics: Equations and Algorithms. Proc. IEEE Int. Conf. Robotics & Automation, San Francisco, April, pp. 826–834.

[11] George, A., and Liu. J. W. H. 1981. *Computer Solution of Large Sparse Positive Definite Systems.* Englewood Cliffs, N.J.: Prentice Hall.

[12] Khalil, W., and Dombre, E. 2002. *Modeling, Identification and Control of Robots.* New York: Taylor & Francis Books.

[13] Lilly, K. W., and Orin, D. E. 1991. Alternate Formulations for the Manipulator Inertia Matrix. *Int. J. Robotics Research*, vol. 10, no. 1, pp. 64–74.

[14] McMillan, S., and Orin, D. E. 1995. Efficient Computation of Articulated-Body Inertias Using Successive Axial Screws. *IEEE Trans. Robotics & Automation*, vol. 11, no. 4, pp. 606–611.

[15] McMillan, S., and Orin, D. E. 1995. Efficient Dynamic Simulation of an Underwater Vehicle with a Robotic Manipulator. *IEEE Trans. Systems, Man & Cybernetics*, vol. 25, no. 8, pp. 1194–1206.

[16] Orlandea, N., Chace, M. A., and Calahan, D. A. 1977. A Sparsity-Oriented Approach to the Dynamic Analysis and Design of Mechanical Systems—Part 1. *Trans. ASME J. Engineering for Industry*, vol. 99, no. 3, pp. 773–779.

[17] Rodriguez, G. 1987. Kalman Filtering, Smoothing, and Recursive Robot Arm Forward and Inverse Dynamics. *IEEE J. Robotics & Automation*, vol. RA-3, no. 6, pp. 624–639.

[18] Rodriguez, G., Jain, A., and Kreutz-Delgado, K. 1991. A Spatial Operator Algebra for Manipulator Modelling and Control. *Int. J. Robotics Research*, vol. 10, no. 4, pp. 371–381.

[19] Rosenthal, D. E. 1990. An Order n Formulation for Robotic Systems. *J. Astronautical Sciences*, vol. 38, no. 4, pp. 511–529.

[20] Saha, S. K. 1997. A Decomposition of the Manipulator Inertia Matrix. *IEEE Trans. Robotics & Automation*, vol. 13, no. 2, pp. 301–304.

[21] Saha, S. K. 1999. Dynamics of Serial Multibody Systems Using the Decoupled Natural Orthogonal Complement Matrices. *Trans. ASME, J. Applied Mechanics*, vol. 66, no. 4, pp. 986–996.

[22] Stejskal, V., and Valášek, M. 1996. *Kinematics and Dynamics of Machinery*. New York: Marcel Dekker.

[23] Vereshchagin, A. F. 1974. Computer Simulation of the Dynamics of Complicated Mechanisms of Robot Manipulators. *Engineering Cybernetics*, no. 6, pp. 65–70.

[24] Walker, M. W., and Orin, D. E. 1982. Efficient Dynamic Computer Simulation of Robotic Mechanisms. *Trans. ASME, J. Dynamic Systems, Measurement & Control*, vol. 104, no. 3, pp. 205–211.

A Algorithms

This appendix lists an algorithm for the sparse LTDL factorization, and algorithms for the four multiplications $\mathbf{L}\,\mathbf{x}$, $\mathbf{L}^T\,\mathbf{x}$, $\mathbf{L}^{-1}\,\mathbf{x}$ and $\mathbf{L}^{-T}\,\mathbf{x}$, where $\mathbf{L}$ is a sparse triangular factor and $\mathbf{x}$ is an arbitrary vector or rectangular matrix. If $\mathbf{x}$ is a vector then the symbols x_i and y_i refer to element i of vectors $\mathbf{x}$ and $\mathbf{y}$; otherwise they refer to row i of matrices $\mathbf{x}$ and $\mathbf{y}$. The multiplication algorithms assume that $\mathbf{L}$ is a non-unit triangular factor, as produced by the LTL factorization, in which $L_{ii} \neq 1$. They can be modified to work with the unit triangular factors produced by the LTDL factorization simply by replacing occurrences of 'L_{ii}' with '1' and making any appropriate simplifications.

LTDL factorization

The algorithm below is the LTDL equivalent of the LTL algorithm described in Section 2. It expects the same inputs as the LTL algorithm, and requires them to meet the same conditions; i.e., an $n \times n$ symmetric, positive-definite matrix $\mathbf{A}$, and an n-element array λ, such that $\mathbf{A}$ satisfies Eq. 6 and λ satisfies Eq. 2. This algorithm works in situ, and it accesses only the lower triangle of its matrix argument. The computed factors $\mathbf{D}$ and $\mathbf{L}$ are returned in this triangle. As the diagonal elements of $\mathbf{L}$ are known to be 1, only the off-diagonal elements are returned.

for $k = n$ **to** 1 **do**
    $i = \lambda(k)$
    while $i \neq 0$ **do**
        $a = A_{ki}/A_{kk}$
        $j = i$
        while $j \neq 0$ **do**
            $A_{ij} = A_{ij} - A_{kj}\,a$
            $j = \lambda(j)$
        end
        $A_{ki} = a$
        $i = \lambda(i)$
    end
end

Algorithm for $\mathbf{y} = \mathbf{L}\,\mathbf{x}$

for $i = n$ **to** 1 **do**
    $y_i = L_{ii}\,x_i$
    $j = \lambda(i)$
    while $j \neq 0$ **do**
        $y_i = y_i + L_{ij}\,x_j$
        $j = \lambda(j)$
    end
end

If $\mathbf{y}$ and $\mathbf{x}$ are different vectors then this algorithm assigns the value $\mathbf{L}\,\mathbf{x}$ to $\mathbf{y}$, leaving $\mathbf{x}$ unaltered. If $\mathbf{y}$ and $\mathbf{x}$ refer to the same vector then this algorithm performs an in-situ multiplication on $\mathbf{x}$, overwriting it with $\mathbf{L}\,\mathbf{x}$.

Algorithm for $\mathbf{y} = \mathbf{L}^T\,\mathbf{x}$

```
for i = 1 to n do
    y_i = L_ii x_i
    j = λ(i)
    while j ≠ 0 do
        y_j = y_j + L_ij x_i
        j = λ(j)
    end
end
```

This algorithm assigns the value $\mathbf{L}^T\,\mathbf{x}$ to $\mathbf{y}$, leaving $\mathbf{x}$ unaltered. It does not work in situ, so $\mathbf{x}$ and $\mathbf{y}$ must be different. An in-situ version would be inefficient.

Algorithm for $\mathbf{x} = \mathbf{L}^{-1}\,\mathbf{x}$

```
for i = 1 to n do
    j = λ(i)
    while j ≠ 0 do
        x_i = x_i − L_ij x_j
        j = λ(j)
    end
    x_i = x_i / L_ii
end
```

This algorithm works in situ on the vector $\mathbf{x}$, replacing it with $\mathbf{L}^{-1}\,\mathbf{x}$. To implement $\mathbf{y} = \mathbf{L}^{-1}\,\mathbf{x}$, the algorithm can be modified by inserting the line '$y_i = x_i$' between lines 1 and 2, and replacing lines 4 and 7 with '$y_i = y_i - L_{ij}\,y_j$' and '$y_i = y_i/L_{ii}$', respectively. Alternatively, one can simply copy $\mathbf{x}$ to $\mathbf{y}$ and apply the above algorithm to $\mathbf{y}$.

Algorithm for $\mathbf{x} = \mathbf{L}^{-T}\,\mathbf{x}$

```
for i = n to 1 do
    x_i = x_i / L_ii
    j = λ(i)
    while j ≠ 0 do
        x_j = x_j − L_ij x_i
        j = λ(j)
    end
end
```

This algorithm works in situ on the vector $\mathbf{x}$, replacing it with $\mathbf{L}^{-T}\,\mathbf{x}$. To implement $\mathbf{y} = \mathbf{L}^{-T}\,\mathbf{x}$, one must copy $\mathbf{x}$ to $\mathbf{y}$ and apply the algorithm to $\mathbf{y}$.