Fast Dynamic Simulation of Highly Articulated Robots with Contact via $\Theta(n^2)$ Time Dense Generalized Inertia Matrix Inversion

Evan Drumwright

The George Washington University, Washington, D.C. USA, `drum@gwu.edu`

Abstract. The generalized inertia matrix and its inverse are used extensively in robotics applications. While construction of the inertia matrix requires $\Theta(n^2)$ time, inverting it traditionally employs algorithms running in time $O(n^3)$. We describe an algorithm that reduces the asymptotic time complexity of this operation to the theoretical minimum: $\Theta(n^2)$. We also present simple modifications that reduce the number of arithmetic operations (and thereby the running time). We compare our approach against fast Cholesky factorization both theoretically (using number of arithmetic operations) and empirically (using running times). We demonstrate our method to dynamically simulate a highly articulated robot undergoing contact, yielding an order of magnitude decrease in running time over existing methods.

1 Introduction

The inverse of the generalized inertia matrix is used in numerous robotics applications such as computing the *contact space inertia matrix* and the *kinetic energy matrix* [1]. When mechanism dynamics are computed in absolute coordinates (*i.e.*, mass matrices are of size $6m \times 6m$, where m is the number of links in the mechanism), the inertia matrix is banded and can be trivially inverted in time $\Theta(m)$ (though constraint forces must be computed, typically requiring operations exhibiting cubic time complexity). For n degree of freedom mechanisms without kinematic loops, the $n \times n$ generalized inertia matrix (formulated in independent coordinates) is dense, symmetric, and positive definite (PD). This matrix can be constructed in $\Theta(n^2)$ time using the *Composite Rigid Body Algorithm* [2, 3] and is traditionally inverted using a combination of $O(n^3)$ Cholesky factorization and $\Theta(n^2)$ backsubstitution.
The key algorithms that make possible the results in this paper (the *Recursive Newton-Euler Algorithm* [4] and the *Articulated Body Algorithm* [5]) were developed over three decades ago; however, the community of robotics researchers is generally unaware of the straightforward implication of $\Theta(n^2)$ generalized inertia matrix inversion. This paper presents a modified version of the latter algorithm optimized toward that new purpose (efficient $\Theta(n^2)$ inversion of $n \times n$ generalized inertia matrices). We compare both operation counts and running times against existing methods for performing the factorization plus backsubstitution.

Before proceeding, observe that we use the concepts of matrix inversion and solving linear systems of equations interchangeably unless otherwise noted to simplify presentation. In the context of our presented algorithms, asymptotic time complexity for matrix inversion is $\Theta(n^2)$ and complexity for solving a system of linear equations with m right hand sides is $\Theta(mn)$.

2 Background

2.1 State of the art in robot dynamics computation

Robot dynamics equations are usually given in the form:

$$\mathbf{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\dot{\boldsymbol{q}}, \boldsymbol{q})\dot{\boldsymbol{q}} + \boldsymbol{G}(\boldsymbol{q}) = \boldsymbol{\tau} \tag{1}$$

where $\mathbf{M}(\boldsymbol{q})$ is the generalized inertia matrix, $\boldsymbol{C}(\dot{\boldsymbol{q}}, \boldsymbol{q})\dot{\boldsymbol{q}}$ is the combined vector of Coriolis and centrifugal forces acting on the robot, $\boldsymbol{G}(\boldsymbol{q})$ is the vector of gravity forces acting on the robot, and $\boldsymbol{\tau}$ is the vector of actuator forces. When the robot base is "floating" (not affixed to the environment), the equation maintains the same structure, but additional terms are added both for the base acceleration and external forces applied to the base.

State of the art algorithms for computing the joint accelerations as a function of actuator forces are compiled in works by Featherstone [3, 6] and Featherstone and Orin [7]. Featherstone groups these algorithms into two categories: $O(n^3)$ algorithms that directly solve the system of linear equations described by Equation 1 directly and an $\Theta(n)$ algorithm (the *Articulated Body Method*) that treats each link as a rigid body with a "handle". This latter algorithm is the key to achieving the $\Theta(n^2)$ inversion operation.

The former category is epitomized by the *Composite Rigid Body Algorithm*, which was studied extensively by Walker and Orin [2]. They claim $\Theta(n^2)$ running time for one of their algorithms (method 4); however, their analysis describes dense matrix-vector multiplication as an $O(n)$ operation (standard algorithms exhibit $O(n^2)$ complexity), and $O(n^3)$ is the true asymptotic behavior.[1] Featherstone [8] has reduced this $O(n^3)$ complexity to $O(n^2d)$, where d is the maximum depth of the kinematic "tree", by exploiting *branch induced sparsity*: mechanisms with branches (multiple child links emanating from a parent) induce sparsity in the generalized inertia matrix. Mechanisms composed of long chains will yield running times closer to $O(n^3)$ than $O(n^2)$, and it is even likely that Featherstone's specialized Choesky factorization and $LDL^\intercal$ algorithms [8] for exploiting branch induced sparsity are slower than specialized dense libraries like LAPACK for most cases (experiments described in Section 6 give credence to this hypothesis).

[1]Walker and Orin's algorithm is a minor adaptation of the iterative conjugate gradient method for solving symmetric PD systems and is known to exhibit $O(n^3)$ complexity.

Finally, we note that Baraff also provides a linear time dynamics algorithm [9] that could be employed toward our purpose instead of Featherstone's algorithm. However, the constant factor for Baraff's algorithm (in absolute coordinates) is dependent upon $6n - r$, where n is the number of joints in the system and r is the number of joint degrees-of-freedom, while the constant factor for Featherstone's algorithm (in independent coordinates) is dependent only upon r; thus, for typical applications with single degree-of-freedom robot joints, Featherstone's algorithm should be significantly faster.

3 Overview of Spatial Algebra

This section presents an overview of Spatial Algebra, which permits dynamics algorithms to be described clearly and succinctly. Extensive tutorials of this subject are contained in [3, 6]; this paper employs the system described in [3].
Spatial vectors are composed of two stacked three-dimensional vectors (a "line vector" and a "free vector"). For example, the spatial velocity of a rigid body is represented by the vector $\hat{\boldsymbol{v}} = \begin{bmatrix}\boldsymbol{\omega} \ \dot{\boldsymbol{x}}\end{bmatrix}^{\mathsf{T}}$. All Spatial Algebra operations can be performed using standard matrix and vector arithmetic except the spatial transpose operation. The spatial transpose operation is denoted using the superscript $^{\mathsf{S}}$ and yields $\hat{\boldsymbol{v}}^{\mathsf{S}} = \begin{bmatrix}\dot{\boldsymbol{x}}^{\mathsf{T}} \ \boldsymbol{\omega}^{\mathsf{T}}\end{bmatrix}$ when applied to the vector above.

3.1 Spatial transformations

Spatial transformations take the form:

$$ {}_j\hat{\mathbf{X}}_i = \begin{bmatrix} \mathbf{E} & \mathbf{0} \\ -\tilde{\boldsymbol{r}}\mathbf{E} & \mathbf{E} \end{bmatrix} \tag{2}$$

where ${}_j\hat{\mathbf{X}}_i$ is the spatial transform from frame i (defined by rotation matrix $\mathbf{R}_i$ and offset $\boldsymbol{x}_i$) to frame j (defined by rotation matrix $\mathbf{R}_j$ and offset $\boldsymbol{x}_j$) and $\mathbf{E} = \mathbf{R}_j^{\mathsf{T}}\mathbf{R}_i$ and $\boldsymbol{r} = \mathbf{R}_j^{\mathsf{T}}(\boldsymbol{x}_j - \boldsymbol{x}_i)$. The skew-symmetric operator $\tilde{}$ is defined on vector $\boldsymbol{r} = \begin{bmatrix}r_x \ r_y \ r_z\end{bmatrix}^{\mathsf{T}}$ below:

$$\tilde{\boldsymbol{r}} = \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}.$$

A spatial vector $\hat{\boldsymbol{v}}_i$ can be transformed from frame i to frame j by:

$$\hat{\boldsymbol{v}}_j = {}_j\hat{\mathbf{X}}_i \ \hat{\boldsymbol{v}}_i. \tag{3}$$

while a spatial inertia matrix (defined below) $\hat{\mathbf{I}}_i$ can be transformed from frame i to frame j via two matrix-matrix multiplications:

$$\hat{\mathbf{I}}_j = {}_j\hat{\mathbf{X}}_i \ \hat{\mathbf{I}}_i \ {}_i\hat{\mathbf{X}}_j. \tag{4}$$

3.2 Spatial rigid body and articulated body inertias

The spatial inertia of a rigid body (in its local frame) is:

$$\hat{\mathbf{I}}' = \begin{bmatrix} \mathbf{0} & m\mathbf{1} \\ \mathbf{J} & \mathbf{0} \end{bmatrix} \tag{5}$$

where the mass of the rigid body is m ($\mathbf{1}$ is the identity matrix) and its 3×3 moment of inertia matrix is denoted $\mathbf{J}$. The spatial rigid body inertia transformed into the "global" reference frame (*i.e.*, via the spatial transformation in Section 3.1) is denoted $\hat{\mathbf{I}}$ (rather than $\hat{\mathbf{I}}'$). We call the sum of the spatial rigid body inertia of link i and the composite inertias of all of link i's children (successors in the kinematic chain) $\hat{\mathbf{I}}_{c_i}$, the *composite inertia*:

$$\hat{\mathbf{I}}_{c_i} = \hat{\mathbf{I}}_i + \sum_{j \in \text{children}(i)} \hat{\mathbf{I}}_{c_j}. \tag{6}$$

Featherstone's Articulated Body Algorithm uses a special inertia matrix, $\hat{\mathbf{I}}^A$—known as the *spatial articulated body inertia*—and defined below:

$$\hat{\mathbf{I}}_i^A = \hat{\mathbf{I}}_i + \sum_{j \in \text{children}(i)} [\hat{\mathbf{I}}_j^A - \frac{\hat{\mathbf{I}}_j^A \hat{\boldsymbol{s}}_j \hat{\boldsymbol{s}}_j^{\mathsf{S}} \hat{\mathbf{I}}_j^A}{\hat{\boldsymbol{s}}_j^{\mathsf{S}} \hat{\mathbf{I}}_j^A \hat{\boldsymbol{s}}_j}] \tag{7}$$

3.3 Spatial axes

The spatial axis for a link transforms its inner joint velocity to the change in spatial velocity of that link. Thus, a collection of spatial axes are analogous to the Jacobian that transforms joint velocities to end effector velocity. Spatial axes for common single degree-of-freedom joints—for simplify of presentation and without loss of generality, only single degree-of-freedom joints are considered in this paper—are given below:

$$\hat{\boldsymbol{s}}_i' = \begin{cases} \begin{bmatrix} \boldsymbol{u}_i \\ \mathbf{0} \end{bmatrix} & \text{if } i \text{ revolute,} \\ \\ \begin{bmatrix} \mathbf{0} \\ \boldsymbol{u}_i \end{bmatrix} & \text{if } i \text{ prismatic.} \end{cases} \tag{8}$$

where $\boldsymbol{u}_i$ is the unit three-dimensional vector pointing along the joint axis and $\mathbf{0}$ is the three-dimensional zero vector. Note that the spatial axes above are computed in a frame with origin at the joint's Cartesian position (denoted by the "prime" applied to $\hat{\boldsymbol{s}}_i$).

4 Notation and conventions

We adopt the following conventions/notation from [3, 6]:

- $\hat{\boldsymbol{u}}$ indicates that $\boldsymbol{u}$ is a $6 \times m$ spatial vector or matrix ($m \leq 6$)
- $\hat{\boldsymbol{r}}^S$ indicates the spatial transpose operation is applied to $\hat{\boldsymbol{r}}$
- ${}_j\hat{\mathbf{X}}_k$ denotes the spatial transformation from frame k to frame j
- $\hat{\boldsymbol{J}}^A$ indicates that $\hat{\boldsymbol{J}}$ is an *articulated body* vector or matrix (*i.e.*, it is used in the context of Featherstone's Articulated Body Algorithm)

– $\lambda : \mathbb{N} \rightarrow \mathbb{N}$ maps the index of a link to the index of that link's parent

The joints and links of an n joint, n link (excluding the base link) mechanism are indexed in the following manner: (1) the base link (fixed or "floating") assumes index 0; (2) the inner joint for the link with index i assumes joint index i as well; (3) every link and joint assumes a unique index; and (4) no "ancestor" to a link can possess a link index greater than its descendant.

5 $\Theta(n^2)$ inverse inertia matrix computation

Our simplified version of Featherstone's Articulated Body Algorithm is based on the classical mechanics equation relating change in linear momentum to applied impulses (abstracted to generalized coordinates):

$$\mathbf{M}\Delta\boldsymbol{v} = \boldsymbol{j} \tag{9}$$

where $\mathbf{M}$ is a $n \times n$ generalized inertia matrix, $\boldsymbol{v}$ is an n-dimensional vector of generalized velocities, and $\boldsymbol{j}$ is an n-dimensional vector of generalized impulses. We wish to find the inverse of $\mathbf{M}$:

$$\Delta\boldsymbol{v} = \mathbf{M}^{-1}\boldsymbol{j}. \tag{10}$$

This equation indicates that $\mathbf{M}^{-1}$ is equal to the changes in velocity due to applying n unit-vector impulses (represented by the $n \times n$ identity matrix) to the system. Using impulses and velocity changes in place of forces and accelerations allows us to avoid determination of gravity, centrifugal, and Coriolis forces inherent in forward dynamics computation. Algorithm 1 implements this strategy with focus on practical (*i.e.*, numerically stable) implementation: the inverse is not constructed explicitly.

Algorithm 1 mMultInv(.) multiplies the inverse of the generalized inertia matrix for a fixed-base mechanism with n joints by an $n \times m$ matrix ($\mathbf{R}$) in $\Theta(nm)$ time.

```
Require: articulated body inertia matrices in global frame ($\hat{\mathbf{I}}^A$), spatial axes in global
    frame ($\hat{\boldsymbol{s}}$)
 1: for $k = 1 \ldots m$ do
 2:     for $i \leftarrow 1 \ldots n$ do {Initialize articulated body impulses}
 3:         $\hat{\mathbf{Y}}_i^A \leftarrow \hat{\mathbf{0}}$
 4:     for $i = n \ldots 1$ do {Propagate impulses}
 5:         $\hat{\mathbf{Y}}_{\lambda(i)}^A \leftarrow \hat{\mathbf{Y}}_{\lambda(i)}^A + \left[\frac{\hat{\mathbf{I}}_i^A \hat{\boldsymbol{s}}_i (R_{ik} - \hat{\boldsymbol{s}}_i^S \hat{\mathbf{Y}}_i^A)}{\hat{\boldsymbol{s}}_i^S \hat{\mathbf{I}}_i^A \hat{\boldsymbol{s}}_i}\right]$
 6:     $\Delta\hat{\boldsymbol{v}}_0 \leftarrow \hat{\mathbf{0}}$
 7:     for $i = 1 \ldots n$ do {Compute velocity updates}
 8:         $\Delta\dot{q}_i \leftarrow \frac{\delta_{ik} - \hat{\boldsymbol{s}}_i^S \left[\hat{\mathbf{I}}_i^A \Delta\hat{\boldsymbol{v}}_{\lambda(i)} + \hat{\mathbf{Y}}_i^A\right]}{\hat{\boldsymbol{s}}^S \hat{\mathbf{I}}_i^A \hat{\boldsymbol{s}}_i}$
 9:         $\Delta\hat{\boldsymbol{v}}_i \leftarrow \Delta\hat{\boldsymbol{v}}_{\lambda(i)} + \hat{\boldsymbol{s}}_i \Delta\dot{q}_i$
10:         $M_{ik} \leftarrow \Delta\dot{q}_i$
```

All calculations are computed in the global frame rather than using link frames: the former is more efficient for our approach due to the $\Theta(nm)$ spatial transformations between link frames that would otherwise be required (Featherstone [7] shows that computation is more efficient in the link frame than the global frame for the unmodified Articulated Body Algorithm, in general). We also present Algorithm 2, which contains necessary modifications to handle mechanisms with floating bases.

Algorithm 2 mMultInvFB(.) multiplies the inverse of the generalized inertia matrix for a floating-base mechanism with n joints by an $n \times m$ matrix ($\mathbf{R}$) in $\Theta(nm)$ time.

```
Require: articulated body inertia matrices in global frame ($\hat{\mathbf{I}}^A$), spatial axes in global
    frame ($\hat{\boldsymbol{s}}$)
 1: for $k = 1 \ldots m$ do
 2:    for $i \leftarrow 0 \ldots n$ do {Initialize articulated body impulses}
 3:       $\hat{\mathbf{Y}}_i^A \leftarrow \hat{\mathbf{0}}$
 4:    for $i = n \ldots 1$ do {Propagate impulses}
 5:       $\hat{\mathbf{Y}}_{\lambda(i)}^A \leftarrow \hat{\mathbf{Y}}_{\lambda(i)}^A + \left[\frac{\hat{\mathbf{I}}_i^A \hat{\boldsymbol{s}}_i (R_{ik} - \hat{\boldsymbol{s}}_i^S \hat{\mathbf{Y}}_i^A)}{\hat{\boldsymbol{s}}_i^S \hat{\mathbf{I}}_i^A \hat{\boldsymbol{s}}_i}\right]$
 6:    if $k \leq 6$ then {Compute floating base velocity change}
 7:       $\Delta\hat{\boldsymbol{v}}_0 \leftarrow -\hat{\mathbf{I}}_0^{A^{-1}} (\hat{\boldsymbol{Y}}_0^A + \boldsymbol{I}_k)$ {$\boldsymbol{I}_k$ is the $k^{\text{th}}$ column of the $6 \times 6$ identity matrix}
 8:    else
 9:       $\Delta\hat{\boldsymbol{v}}_0 \leftarrow -\hat{\mathbf{I}}_0^{A^{-1}} (\hat{\boldsymbol{Y}}_0^A)$
10:    $\boldsymbol{u}_k \leftarrow \Delta\hat{\boldsymbol{v}}_0$
11:    for $i = 1 \ldots n$ do {Compute velocity updates}
12:       $\Delta\dot{q}_i \leftarrow \frac{\delta_{ik} - \hat{\boldsymbol{s}}_i^S \left[\hat{\mathbf{I}}_i^A \Delta\hat{\boldsymbol{v}}_{\lambda(i)} + \hat{\mathbf{Y}}_i^A\right]}{\hat{\boldsymbol{s}}^S \hat{\mathbf{I}}_i^A \hat{\boldsymbol{s}}_i}$
13:       $\Delta\hat{\boldsymbol{v}}_i \leftarrow \Delta\hat{\boldsymbol{v}}_{\lambda(i)} + \hat{\boldsymbol{s}}_i \Delta\dot{q}_i$
14:       $A_{ik} \leftarrow \Delta\dot{q}_i$
15: return $\begin{bmatrix} \boldsymbol{u}_1 \ \ldots \ \boldsymbol{u}_n \\ \mathbf{A} \end{bmatrix}$
```

5.1 Complexity Analysis

The operations on lines 5, 8, and 9 of Algorithm 1 each require constant time ($\hat{\boldsymbol{Y}}^A$, $\hat{\mathbf{I}}^A$, $\hat{\boldsymbol{s}}$, and $\Delta\hat{\boldsymbol{v}}$ are of fixed size), so the nested **for** loops result in $\Theta(nm)$ time complexity.

5.2 Arithmetic Analysis

We can utilize a few optimizations not shown in Algorithm 1 to decrease the number of arithmetic operations. Vectors and quantities $\hat{\mathbf{I}}^A\hat{\boldsymbol{s}}$ and $\hat{\boldsymbol{s}}^{\mathrm{S}}\hat{\mathbf{I}}^A\hat{\boldsymbol{s}}$ are computed only once, rather than separately for each iteration ($\hat{\boldsymbol{s}}$ and $\hat{\mathbf{I}}^A$ are dependent upon only the coordinates of the mechanism and not its velocity). Additionally, the impulse propagation process (Lines 4–5 of Algorithm 1) needs to start only at the single joint at which δ_{ik} is

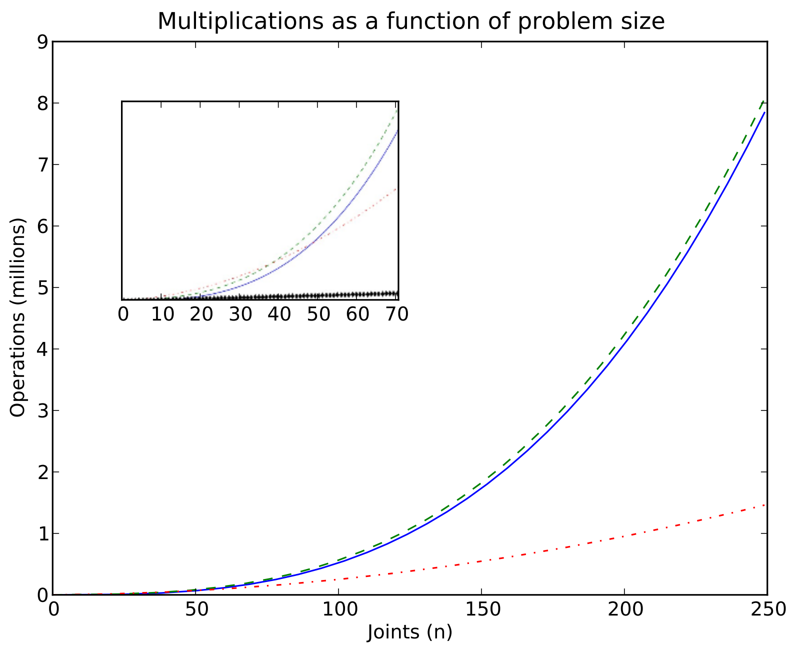

Fig. 1: Number of multiplication operations for generalized inertia inversion/factorization as a function of multibody joints for four methods: Cholesky factorization only (blue/solid), both formation and Cholesky factorization (green/dashed), the $\Theta(n^2)$ method presented in this paper (red/dash-dot), and the $\Theta(n^2)$ parallel method using n processors described in Section 5.2 (black/diamonds). Numbers of arithmetic operations are computed as described in Section 5.2.

nonzero. Table 1 shows the multiplication and addition counts for the inversion algorithm.

One exciting potential of this work is its utilization in massively parallel computation. Although computing articulated spatial inertias must be done sequentially, the spatial axes ($\hat{\boldsymbol{s}}$) and the isolated spatial inertias ($\hat{\mathbf{I}}$) can be computed in parallel. Each column of $\mathbf{R}$ can be solved (or, equivalently, each column of $\mathbf{M}^{-1}$ can be determined) in parallel as well. Thus, if one employs an n multiprocessor system for solving, only $261n/2 + 108$ multiplications and $133n + 63$ additions are required (specific operation counts are given in Table 3 for an n-processor system). Cholesky factorization benefits little from massively parallel or SIMD computation.

Table 1: Arithmetic operation counts for $\Theta(n^2)$ inversion algorithm (Algorithm 1)

Operation	Multiplications	Additions
Computing spatial axes in global frame ($\hat{\boldsymbol{s}}$)	$24n$	$6n$
Computing isolated spatial inertias ($\hat{\mathbf{I}}$)	$84n$	$57n$
Computing articulated spatial inertias ($\hat{\mathbf{I}}^A$)	$108n$	$107n$
Impulse propagation ($\hat{\boldsymbol{Y}}$)	$13n^2$	$13n^2$
Link and joint velocity updates ($\Delta\hat{\boldsymbol{v}}, \Delta\dot{q}$)	$19n^2/2$	$26n^2/2$
Total	$45n^2/2 + 216n$	$26n^2 + 170n$

Table 2: Arithmetic operation counts for Composite Rigid Body Algorithm (n joint, fixed base mechanism, global frame)

Operation	Multiplications	Additions
Computing spatial axes in global frame ($\hat{\boldsymbol{s}}$)	$24n$	$6n$
Computing isolated spatial inertias ($\hat{\mathbf{I}}$)	$84n$	$57n$
Computing composite inertia matrices ($\mathbf{I}_c$)	0	$13n - 13$
Computing $\hat{\mathbf{I}}\hat{\boldsymbol{s}}$ vectors	$36n$	$24n$
Computing $\hat{\boldsymbol{s}}_i^{\mathsf{S}}\hat{\mathbf{I}}_j\hat{\boldsymbol{s}}_j$ (elements of $\mathbf{M}$)	$3n^2$	$5n^2/2$
Cholesky factorization (naive)	$n^3/2 + n^2/2 + n$	$n^3/2$
Total	$n^3/2 + 7n^2/2 + 145n$	$n^3/2 + 5n^2/2 + 100n - 13$

6 Experiments

Numerical experiments were conducted with the *Moby* robot dynamics library. Vector arithmetic utilized tuned BLAS libraries (ATLAS). Cholesky factorization was performed using LAPACK, our implementation of Featherstone's branch induced sparsity (BIS) factorization [8], or both. Experiments were conducted on a dual core 2.8GHz Intel Xeon W3530 processor (four virtual cores using HyperThreading$^{\text{TM}}$) running Ubuntu Linux. for Cholesky factorization as well as our implementation of Featherstone's branch induced sparsity (BIS) Cholesky factorization.

6.1 Single-threaded inversion experiments

The experiment described in this section uses a single-threaded version of the *Moby* library. The articulated bodies used in these experiments are fully serial (*i.e.*, the kinematic tree is of depth n) to obtain the most conservative timings for our method. However, we also wished to compare our solving algorithm against BIS Cholesky factorization. As indicated by Figure 2, we tested branched bodies of expected depth $n/2$ *only* for the BIS method: with uniform probability 0.5, we added a link as a sibling to another rather than adding that link to the end of the chain (no link was permitted more than two children).

Table 3: Arithmetic operation counts per processor for $\Theta(n^2)$ inversion algorithm (n joint, fixed base mechanism) on n processors

Operation	Multiplications	Additions
Computing spatial axes in global frame ($\hat{\boldsymbol{s}}$)	24	6
Computing isolated spatial inertias ($\hat{\mathbf{I}}$)	84	57
Computing articulated spatial inertias ($\hat{\mathbf{I}}^A$)	$108n$	$107n$
Impulse propagation ($\hat{\boldsymbol{Y}}$)	$13n$	$13n$
Link and joint velocity updates ($\Delta\hat{\boldsymbol{v}}, \Delta\dot{q}$)	$19n/2$	$13n$
Total	$261n/2 + 108$	$133n + 63$

6.2 Multi-threaded inversion experiment

This experiment used OpenMP and a multi-threaded version of *Moby* to compute the inverse of the generalized inertia matrix; threads were limited to four (the Intel Xeon presents four virtual cores via HyperThreading$^{\text{TM}}$). Unlike the previous experiment, which timed only CPU operations, this experiment timed all operations (including I/O and time waiting for the OS's scheduler) in order to assess efficiency gains via parallelism.

6.3 Simulation experiments

We tested the effectiveness of our method on a "real world" problem: dynamic simulation of a centipede walking on a planar surface. The centipede was simulated using increasing numbers of body segments (the maximum number of body segments was 250). Each body segment was connected to two upper legs via spherical joints; each upper leg was connected to a lower leg via a revolute joint. The simulation used explicit Euler integration with a step size of $1e^{-5}$ for one thousand steps. The software setup used in the previous experiment was employed in this experiment as well. Timings were conducted over all operations: dynamics computation, collision detection, contact resolution, *etc.*; however, only CPU timings were used (as in the first experiment). Contact resolution used the method described in [10], which requires computing the contact space inertia matrices $\mathbf{N}^{\mathsf{T}}\mathbf{M}^{-1}\mathbf{N}$, $\mathbf{N}^{\mathsf{T}}\mathbf{M}^{-1}\mathbf{D}$, and $\mathbf{D}^{\mathsf{T}}\mathbf{M}^{-1}\mathbf{D}$ ($\mathbf{N}$ and $\mathbf{D}$ are contact Jacobians for the normal and tangent directions, respectively); this was the only code that required the inverse inertia matrix and is solely responsible for the timing differences in Figure 4.

7 Discussion

Our $\Theta(n^2)$ inversion method is amenable to both symbolic computation (which could significantly reduce the number of arithmetic computations) and parallelization (especially in the context of SIMD/GPU processing).
Figures 2, 3, and 4 from the experiments in the previous section show that our algorithm is competitive with BLAS/LAPACK even for bodies with relatively few degrees-of-freedom (fewer than 100); for bodies

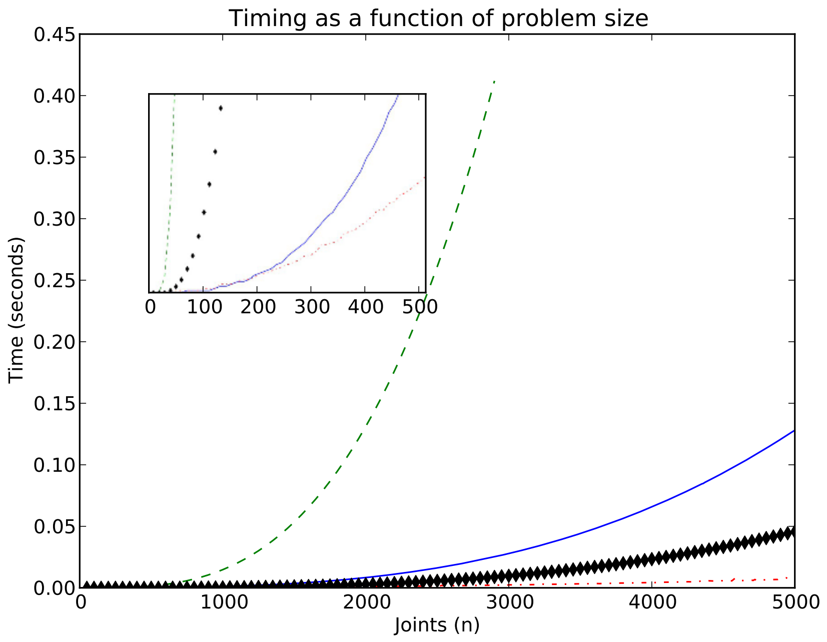

Fig. 2: Times required to compute the inverse of the generalized inertia matrix using the presented $\Theta(n^2)$ method (red/dash-dot) and the construction and Cholesky factorization of the generalized inertia matrix using (1) LAPACK (blue/solid) and the branch induced sparsity method for (2) a serial body of depth n (green/dashed) and for (3) a branched body of expected depth $n/2$ (black diamond).

with greater degrees-of-freedom, the asymptotic advantage of our approach becomes evident quickly. The experiment using multi-threading in Section 6.2 illustrates the potential gains in performance from multiprocessing: using two threads increases performance over one thread by 34%, and using four threads increases performance by 58% and 117%, respectively, over two threads and one thread. Larger scale SIMD parallelism (via GPU processing, for example) should yield further large increases in performance.

The experiments in the previous section illustrate the power of tuned BLAS and LAPACK libraries for vector arithmetic and linear algebra: Cholesky factorization yielded superior performance for $n < 130$ in the first experiment (which used tuned libraries) and inferior performance in the third experiment. The second experiment illustrates the potential gains in performance from multiprocessing: using two threads increases performance over one thread by 34%, and using four threads increases

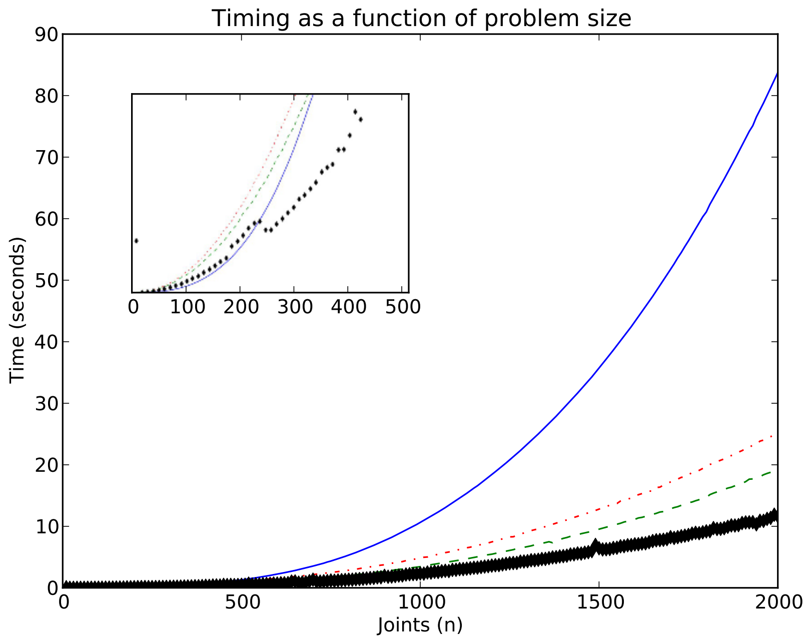

Fig. 3: Times required to compute the inverse of the generalized inertia matrix using LAPACK's Cholesky factorization (blue/solid) and the presented $\Theta(n^2)$ method with (1) one thread (red/dash-dot), (2) two threads (green/dashed), and (3) four threads (black/diamond).

performance by 58% and 117%, respectively, over two threads and one thread.

All experiments clearly show that for applications that require inverting the generalized inertia matrix of highly articulated robots, our method yields significant performance increases. Further optimizations should yield additional increases.

References

1. Khatib, O.: A unified approach to motion and force control of robot manipulators: The operational space formulation. IEEE Journal on Robotics and Automation **3**(1) (Feb 1987) 43–53
2. Walker, M.W., Orin, D.E.: Efficient dynamic computer simulation of robotic mechanisms. ASME J. Dynamic Systems, Measurement, and Control **104** (1982) 205–211
3. Featherstone, R.: Robot Dynamics Algorithms. Kluwer (1987)
4. Hollerbach, J.M.: A recursive lagrangian formulation of manipulator dynamics and a comparative study of dynamics formulation com-

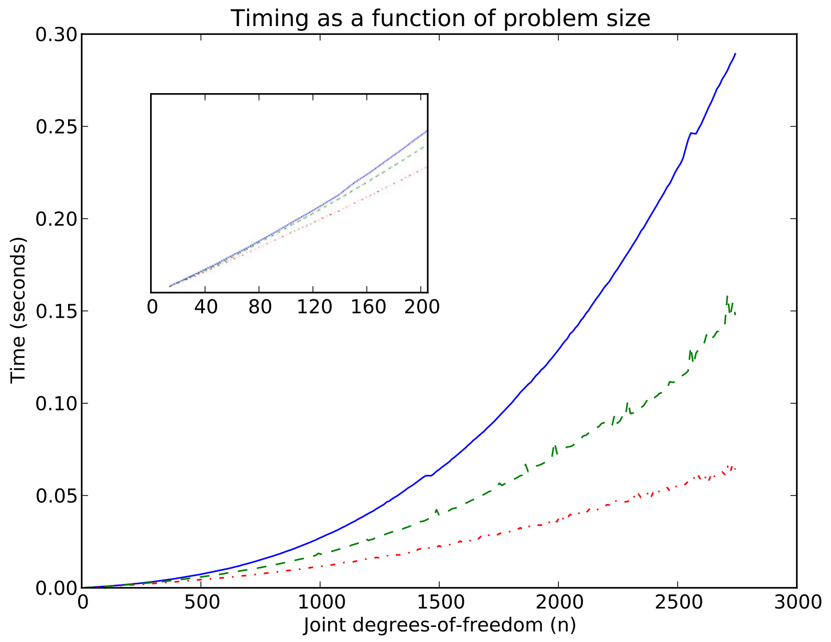

Fig. 4: Timings for dynamically simulating centipedes with varying numbers of legs walking on a planar surface. The contact space inertia matrix—$\mathbf{N}^{\mathsf{T}}\mathbf{M}^{-1}\mathbf{N}$—is computed using the presented $\Theta(n^2)$ method (red/dash-dot), Cholesky factorization and backsubstitution using LAPACK (blue/solid), and Cholesky factorization—using branch induced sparsity—plus backsubstitution (green/dashed).

plexity. IEEE Trans. Systems, Man, and Cybernetics **SMC-10**(11) (1980) 730–736

5. Featherstone, R.: The calculation of robot dynamics using articulated body inertias. Intl. J. Robotics Research **2**(1) (1983) 13–30
6. Featherstone, R.: Rigid Body Dynamics Algorithms. Springer (2008)
7. Featherstone, R., Orin, D.E.: Robot dynamics: Equations and algorithms. In: Proc. of IEEE Intl. Conf. on Robotics and Automation, San Francisco, CA (April 2000)
8. Featherstone, R.: Efficient factorization of the joint space inertia matrix for branched kinematic trees. Intl. J. Robotics Research **24**(6) (2005) 487–500
9. Baraff, D.: Linear-time dynamics using lagrange multipliers. In: Proc. of Computer Graphics, New Orleans, LA (Aug 1996)
10. Drumwright, E., Shell, D.A.: Modeling contact friction and joint friction in dynamic robotic simulation using the principle of maximum dissipation. In: Proc. of Workshop on the Algorithmic Foundations of Robotics (WAFR). (2010)