Linear-Time Dynamics using Lagrange Multipliers

David Baraff

CMU-RI-TR-95-44

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

January 1996

Contents

List of Figures

Abstract

Current linear-time simulation methods for articulated figures are based exclusively on reduced-coordinate formulations. This paper describes a general, non-iterative linear-time simulation method based instead on Lagrange multipliers. Lagrange multiplier methods are important for computer graphics applications because they bypass the difficult (and often intractable) problem of parameterizing a system's degrees of freedom. Given a loop-free set of n equality constraints acting between pairs of bodies, the method takes $O(n)$ time to compute the system's dynamics. The method does not rely on matrix bandwidth, so no assumptions about the constraints' topology are needed. Bodies need not be rigid, constraints can be of various dimensions, and unlike reduced-coordinate approaches, nonholonomic (e.g. velocity-dependent) constraints are allowed. An additional set of k one-dimensional constraints which induce loops and/or handle inequalities can be accommodated with cost $O(kn)$. This makes it practical to simulate complicated, closed-loop articulated figures with joint-limits and contact at interactive rates. A complete description of a sample implementation is provided in pseudocode.

1 Introduction

Forward simulation with constraints is a key problem in computer graphics. Typically, a system's constraints are *sparse*: each constraint directly affects only one or two bodies (for example, geometric connection constraints) and for a system with n bodies, there are only $O(n)$ constraints. In particular, the simulation of articulated figures and mechanisms falls into this category. Sparse constraint systems are also either nearly or completely acyclic: for example, robot arms are usually open-loop structures, as are animation models for humans and animals. Considerable effort has been directed toward efficiently simulating these types of systems.

Reading through the dynamics literature, a large variety of dynamics formulation can be found (Newton-Euler, Gibbs-Appel, D'Alembert, Gauss' Least Constraint Principle, etc.) but the details of these variations matter little; ultimately, we are faced with a basic choice. Either we model constraints by reducing the number of coordinates needed to describe the system's state, or we introduce additional forces into the system that maintain the constraints.

A *reduced-coordinate* formulation takes a system with m degrees of freedom (d.o.f.'s), a set of constraints that removes c of those d.o.f.'s, and parameterizes the remaining $n = m - c$ d.o.f.'s using a reduced set of n coordinates. A reduced set of n coordinates are called *generalized* coordinates; coordinates describing the original m-d.o.f. system are called *maximal* coordinates. For an arbitrary set of constraints, finding a parameterization for m maximal coordinates in terms of n generalized coordinates is arbitrarily hard; if such a parameterization can be found, $O(n^3)$ time is required to compute the acceleration of the n generalized coordinates at any instant. However, loop-free articulated rigid bodies are trivially parameterized, and methods for computing the n generalized coordinate accelerations in $O(n)$ time are well known [5].

In contrast, Lagrange multiplier methods express the system's state using the simpler set of m maximal coordinates. Constraints are enforced by introducing *constraint forces* into the system. At each instant, a basis for the constraint forces is known *a priori*; the Lagrange multipliers (which we must compute) are a vector of c scalar coordinates that describe the constraint force in terms of the basis. Lagrange multiplier approaches are extremely important for interactive computer graphics applications, because they allow an arbitrary set of constraints to be combined. This is difficult (often impossible) to achieve with a reduced-coordinate formulation. Additionally, Lagrange multiplier formulations allow (and frankly encourage) a highly modular knowledge/software design, in which bodies, constraints, and geometry regard each other as black-box entities (section 2 develops this further). Lagrange multipliers also allow us to handle nonholonomic constraints, such as velocity-dependent constraints; reduced-coordinate approaches inherently lack this capability.

For a system whose constraints remove c d.o.f.'s, the Lagrange multipliers are the c unknown variables of a set of c linear equations. If c is much greater than n, so that the constrained system possesses only a few d.o.f.'s, clearly the reduced-coordinate approach is preferred. However, for the case of open-loop articulated three-dimensional rigid bodies, $c = O(n)$, since c is at least $1/5n$ and at most $5n$. Even though n and c are linearly related for articulated figures, the current prevailing view is that articulated figures can be simulated in linear time *only* by using a reduced-coordinate formulation. The possibility of achieving $O(n)$ performance for Lagrange multiplier methods has been largely discounted, because the prospects for simply and easily solving the resulting $O(n) \times O(n)$ matrix system in $O(n)$ time have seemed dismal, at best. We show in this paper that a very simple direct (that is, non-iterative) $O(n)$ solution method exists for computing Lagrange multipliers for sparse acyclic constraint systems. The method itself arises from elementary sparse-matrix techniques, applied to a matrix equation that varies just slightly from the normal computer-graphics constrained dynamics formulation using Lagrange multipliers.

1.1 Specific Contributions

The results of this paper are the following. Consider a set of n bodies (not necessarily rigid) and a set of $n-1$ constraints, with each constraint enforcing a relationship of some dimension between two of the bodies. Assuming the constraint connectivity is acyclic (for example, a constraint between body 1 and 2, between body 2 and 3, and between body 3 and 1 forms a cycle), we describe a simple, direct $O(n)$ method for computing the Lagrange multipliers for these constraints. We will call this acyclic set of constraints the *primary constraints*. The primary constraints need not be holonomic, though they must be equality constraints. Nonholonomic velocity-based constraints—such as a relationship between rotational speeds of bodies—fit into this framework and are handled as primary constraints. Reduced-coordinates approaches are restricted to holonomic constraints.

In addition, a set of *auxiliary* constraints can also be accommodated. Closed loops are handled by designating constraints which cause cycles as auxiliary, rather than primary constraints. Similarly, constraints that act on only a single body, or on more than two bodies are designated as auxiliary constraints, as are inequality constraints, such as joint-angle limits or contact constraints. If the primary constraints partition the bodies into separate components (for example, two separate chains), then an inequality might involve only one of the primary constraint components (a chain colliding with itself); however, a constraint involving two or more components (two different chains colliding with each other) is handled just as easily. In addition to the $O(n)$ time required to deal with the primary constraints, k one-dimensional auxiliary constraints cost $O(nk)$ time to formulate a $k \times k$ matrix system and $O(k^3)$ time to solve the system. When k is small compared to n, the added cost is essentially just an additional $O(nk)$. The auxiliary constraint method described is particularly efficient in conjunction with our $O(n)$ primary constraint method. Our approach to auxiliary constraints could also be used to handle auxiliary constraints in linear-time reduced-coordinate formulations.

In our (biased) view, linear-time performance is achieved far more easily for Lagrange multiplier methods than for reduced-coordinate formulations. While $O(n)$ *inverse* reduced-coordinate approaches are easily understood, *forward* reduced-coordinate formulations with linear time complexity have an extremely steep learning curve, and make use of a formidable array of notational tools. The author admits (as do many practitioners the author has queried) to lacking a solid, intuitive understanding of these methods.[1] We believe that a reader who

[1] This is *not* an attack on such methods; rather it is an invitation and opportunity for someone to step forward and clearly and simply explain such methods to the rest of us!

already understands the standard $O(n^3)$ method for formulating and computing Lagrange multipliers should have no difficulty in implementing the $O(n)$ method presented in this paper. To back this point up, appendix A contains a complete (yet extremely short) pseudocode implementation. Given an existing $O(n^3)$ Lagrange multiplier based simulation system, converting to the required $O(n)$ datastructures is simply and easily accomplished.

2 Motivation

It is probably as important for us to stress what this paper does *not* say as to stress what this paper does say. The existence of a linear-time Lagrange multiplier method shows that the Lagrange multiplier approach can achieve the same asymptotic complexity results as reduced-coordinate formulations; this is of theoretical interest. However, in presenting a linear-time method for computing multipliers we are *not* asserting that such a method is faster on articulated figures than, say, Featherstone's $O(n)$ method. On the other hand, we are also not asserting it is necessarily slower. It used to be that one could attempt to discuss the running times of algorithms based on the number of multiplications and additions; today, when a memory access may be as costly as a multiplication, such analysis no longer holds true. In section 9, we will discuss and relate actual running times of our algorithm to the few published results with which we are familiar.

2.1 Why Reduced Coordinates?

There are certainly valid reasons for preferring a reduced-coordinate approach over a multiplier approach. In particular, if the n d.o.f.'s left to the system is very much smaller than the c d.o.f.'s removed by the constraints, a reduced-coordinate approach is clearly called for. Even if c and n are linearly related the use of generalized coordinates eliminates the "drifting" problem that multiplier methods have. (For example, two links which are supposed to remain connected will have a tendency to drift apart somewhat when a multiplier approach is used.) Such drift is partly a consequence of numerical errors in computing the multipliers, but mostly from the inevitable errors of numerical integration during the simulation. Constraint stabilization techniques [4, 3] are used to help combat this problem. The use of generalized coordinates eliminates this worry completely, since generalized coordinates only express configurations which exactly satisfy the constraints. There is anecdotal evidence that the use of generalized coordinates thus allows simulations to proceed faster, not because evaluation of the generalized coordinate accelerations is faster, but because larger timesteps can be taken by the integrator. This may well be true. More importantly, for the case of articulated figures, we know that with a reduced-coordinate approach, linear-time performance is achievable.

2.2 Why Lagrange Multipliers?

On the other hand, there are also strong motivations for preferring a multiplier approach. Work by Witkin *et al*. [14], Barzel and Barr [3], Baraff [1], and most recently and comprehensively Gleicher [6], present a variety of arguments in favor of multiplier methods. In particular, multiplier methods neatly compartmentalize knowledge, enabling strongly modular systems. For general-purpose, extensible simulation system, this is vital. Consider two bodies and a constraint that the world-space location of two points (each point having a fixed body-space location) be coincident. Parameterizing the freedoms of this system using generalized coordinates requires us to have symbolic knowledge of the body-space to world-space mapping for each body. This is obviously not a problem if we limit ourselves to rigid bodies, but suppose that one or both of the bodies can both rotate, translate, and scale (possibly among one or more axes). We must know the freedoms of the bodies, in order to form the generalized coordinates. Similarly, a constraint that depends upon surface geometry requires symbolic knowledge of the surface equation. From a software modularity standpoint, every combination of constraint, body, and geometry yields a new type of parameterization. This results in a quadratic explosion in the amount of code that must be generated.

In some cases it may be either to difficult, or even impossible, to derive the necessary generalized coordinate parameterizations. Once we move past rigid bodies to globally deformable frames, parameterization of the constraints becomes totally impractical. Even for rigid bodies, parameterization can be hard: imagine a tangency constraint between two rigid smooth surfaces, that requires that the bodies remain in tangential contact (thus allowing sliding motions). This constraint removes exactly one degree of freedom from the bodies' motions. For all but the simplest shapes, the required parameterization is extremely complicated (and closed-form solutions will not in general exist).

Finally, nonholonomic constraints cannot be expressed in terms of generalized coordinates. Consider a mechanical simulation, with an abstraction of a complicated gearing mechanism. We may have a simple constraint—for example, that the rotational speed a an object in three-dimensions be twice the speed of another—but be completely unable to express it in a reduced-coordinate formulation. In contrast, such velocity-based constraints are trivially handled using multiplier methods.

Suppose however that we are interest only in simulating articulated rigid bodies, so that none of the above issues apply. If the implementation of one of the $O(n)$ reduced-coordinate algorithms described in the literature is seen as manageable, quite possibly there is no gain to be realized from the algorithm described in this paper. If generalized coordinates are desired, but the effort to implement a linear-time reduced-coordinate approach is prohibitive, a middle ground exists: at each step of the simulation, translate the generalized coordinates and velocities to maximal coordinates and velocities, compute the Lagrange multipliers and thus the maximal coordinate accelerations, and translate these accelerations back into generalized coordinates. For all other cases (when a reduced-coordinate approach is infeasible because of the demands it places on software architecture, or because the necessary parameterization simply cannot be realized) the algorithm described in this paper yields a practical, simple linear-time alternative to traditional reduced-coordinate techniques.

3 Background

In this paper, the term simulation does not merely refer to dynamic, physical simulation: the use of constrained differential kinematic manipulation, as pioneered by Witkin *et al.* [14] and Gleicher [6] is also considered simulation. For dynamic, or "second-order" simulation, we relate acceleration to force according to Newton's law $f = ma$, while for kinematic manipulation we instantaneously relate velocity and "force" according to the first-order law $f = mv$. Similarly, in a dynamics simulation with collisions, the velocity discontinuity Δv caused by a collision is related to an impulsive force j according to the law $\Delta v = mj$. In all of the above cases, the problem is to compute the correct acceleration, velocity, or velocity change that satisfies the constraints of the system. We will not distinguish between any of these problems further; this paper deals with $f = ma$ problems, with the understanding that the results obtained obviously transfer to the other two problems.

Lagrange multipliers are usually computed by solving a matrix equation (which we describe in greater detail later)

$$\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T\boldsymbol{\lambda} = \mathbf{c}.$$

The elements of the vector $\boldsymbol{\lambda}$ are the multipliers we wish to solve for, while $\mathbf{M}$ is a block-diagonal matrix. The rows of $\mathbf{J}$ encode the constraints' connectivity in block-form: if the ith constraint affects only bodies p and q, then only the pth and qth blocks of $\mathbf{J}$'s ith row are nonzero. (We discuss the block structure of $\mathbf{J}$ and $\mathbf{M}$ more carefully in the next section.) Because of $\mathbf{J}$'s and $\mathbf{M}$'s structure, for some special cases it is obvious that $\boldsymbol{\lambda}$ can be computed in linear time.

For example, consider a serial chain (an unbranching sequence of links). The dynamics of serial chain robot arms were not generally known to be solvable in linear time until very recently, with the advent of Featherstone's [5] recursive articulated-body method.[2] This is a curious oversight, when one considers that linear-time simulation of serial chains with Lagrange multiplier methods is obvious and trivial, because $\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T$ is tightly banded (assuming an ordering so that body p is connected to body $p+1$ for all bodies).

Once we move past simple chains, the problem becomes more complicated. Depending on the structure of the constraints, exploiting bandedness is still a possibility. For example, Surles [12] exploited bandedness (by symmetrically permuting rows and columns of $\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T$) to achieve a direct, linear-time solution for the multipliers on systems that are very chain-like in their connectivity, but have some limited branching. As structures become less chainlike however, the bandwidth of the system increases, and his method reduces to a regular $O(n^3)$ dense solution method. Negrut *et al.* [10] describe a similar method. The method described in this paper does not attempt to exploit bandwidth, because for many structures, there is *no* permutation that yields a matrix system with reasonable bandwidth.

While sparse (but not necessarily acyclic) constraint systems always yield sparse matrices $\mathbf{J}$ and $\mathbf{M}^{-1}$, in more general problems the product $\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T$ (although usually sparse)

[2]Featherstone made this discovery independently of earlier work by Vereshchagin, in 1974. Vereshchagin [13] described a solution algorithm for serial chains which turned out to have linear time complexity, although the algorithm was not advertised as such.

need not be. One well-known approach to dealing with this kind of sparsity is the use of iterative methods, with time-complexity $O(n^2)$ (or lower, depending on convergence properties). Despite impressive recent results by Gleicher [6] in applying conjugate-gradient methods to compute multipliers, the prospect of computing multipliers in less than $O(n^3)$ still seems to be largely viewed as a theoretical result, but not a practical actuality. Similarly, many papers have pointed out that, in theory, the sparsity of $\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T$ can be exploited by direct, non-iterative methods in linear time by applying general sparse-matrix solvers to the problem. However, this first supposes that $\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T$ is sparse which is generally but not always true. Even if sparsity exists, solving such problems by employing a general-purpose sparse-matrix solver is, practically speaking, not something that most computer graphicists would approach with much enthusiasm.

To the best of our knowledge though, no one has made the observation that *any* pairwise, acyclic set of constraints results in a system that (when formulated correctly) is easily solved in linear time using rudimentary sparse-matrix principles. The next few sections simply elaborate on this basic observation. In section 8, we describe a practical method for dealing with loop-closing and inequality constraints that are not handled by the simpler sparse formulation we are about to describe.

4 The Lagrange Multiplier Formulation

Our goal is to treat bodies, forces, and constraints as "anonymously" as possible: we wish to assume the minimum possible structure. For example, we may have a mix of body types (e.g. rigid, rigid plus scalable, etc.) and constraints of various dimensions (e.g. a pinjoint of dimension three, a point-to-surface constraint with dimension one). This lead us to a formulation where matrices are composed of smaller, dense matrices; this is known as a *block-matrix* formulation [7]. The dimensions of an individual block are dictated by the dimensions of bodies and constraints. A body's dimension is the number of d.o.f.'s the body has when unconstrained, while a constraint's dimension is the number of d.o.f.'s the constraint removes from the system. If no body has a dimension greater than p, then no constraint will have a dimension greater than p. As a result, all blocks will be of size $p \times p$ or smaller. Regarding p as a constant for the simulation, an operation on a single block or pair of blocks (inversion, multiplication, addition) takes constant time.

Our assumptions about the constraints are made as weak as possible. At any instant, each constraint is specified as a linear condition on the acceleration of a pair of bodies. The mechanics of expressing various geometric and velocity-based constraints as conditions on bodies' accelerations has been extensively considered in past work [3, 1, 6, 11]; we therefore omit the details of particular constraints. Hopefully, this rather aggressive retreat into anonymous notation will both simplify the resulting discussion, and explicitly define the modular relationship between bodies, constraints, and the computation of the Lagrange multipliers.

4.1 Notation

With the above in mind, we introduce a small amount of notation. The dimension of the ith body is denoted $\dim(i)$ and is the number of d.o.f.'s the body has when unconstrained. We describe the ith body's velocity as a vector $\mathbf{v}_i \in \mathbb{R}^{\dim(i)}$; a force $\mathbf{F}_i$ acting on the ith body is also a vector in $\mathbb{R}^{\dim(i)}$. The acceleration $\dot{\mathbf{v}}_i$ of the ith body in response to the force $\mathbf{F}_i$ is

$$\mathbf{M}_i \dot{\mathbf{v}}_i = \mathbf{F}_i$$

where $\mathbf{M}_i$ is a $\dim(i) \times \dim(i)$ symmetric positive definite matrix which describes the mass properties of body i. The matrix $\mathbf{M}_i$ may vary over time, according to the body's geometric state; however, $\mathbf{M}_i$ is independent of $\mathbf{v}_i$. For a system of n bodies, the vector $\mathbf{v} = (\mathbf{v}_1, \ldots, \mathbf{v}_n)$ denotes the velocity of the entire system, and similarly for $\dot{\mathbf{v}}$. (Note that $\mathbf{v}$ is described in block-fashion; $\mathbf{v}$'s ith element $\mathbf{v}_i$ is itself a vector, with dimension $\dim(i)$.) Similarly, a force $\mathbf{F} = (\mathbf{F}_1, \ldots, \mathbf{F}_n)$ acting on the system means that a force $\mathbf{F}_1$ act on body 1, and so on. Given such a force $\mathbf{F}$, the system's evolution over time is

$$\mathbf{M}\dot{\mathbf{v}} = \mathbf{F} \tag{1}$$

where $\mathbf{M}$ is the block-diagonal matrix

$$\mathbf{M} = \begin{pmatrix} \mathbf{M}_1 & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{M}_2 & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{M}_n \end{pmatrix}.$$

The dimension of a constraint is the number of d.o.f.'s the constraint removes from the system. As we said earlier, a constraint is expressed as a linear condition on bodies' accelerations. If the ith constraint has dimension m, then an expression for the constraint is an m-dimensional acceleration condition of the form

$$\mathbf{j}_{i1}\dot{\mathbf{v}}_1 + \cdots + \mathbf{j}_{ik}\dot{\mathbf{v}}_k + \cdots + \mathbf{j}_{in}\dot{\mathbf{v}}_n + \mathbf{c}_i = \mathbf{0}. \tag{2}$$

Each matrix $\mathbf{j}_{ik}$ has dimension $m \times \dim(k)$, $\mathbf{c}_i$ is an m-length column vector, and $\mathbf{0}$ is the zero vector of length m. The coefficients of this equation (the $\mathbf{j}_{ik}$ matrices and the vector $\mathbf{c}_i$) depend on the specifics of the bodies and the exact constraint being enforced, as well as the position and velocities of the bodies at the current instant. In the next section, we will require that each primary constraint affect only a pair of bodies; this means that for each value of i, all but two of the $\mathbf{j}_{ik}$ matrices will be zero. For now, this restriction is not important.

4.2 Constraint Forces

In order to enforce the acceleration conditions of the constraints, a constraint force must be added to the system. For the primary constraints of the system, we deal only with constraints that are maintained by *workless* constraint forces. A rigorously physical definition of workless constraints is difficult, because explicitly time-varying constraint functions (such as those in Barzel and Barr [3], which cause gradual assemblages of structures) can add energy into the system.[3] The most direct way to attack the problem is to say that by workless constraint forces, we really mean "constraint forces that are as lazy possible." Fortunately, this intuitive notion has a simple mathematical translation: the constraint force that maintains the ith constraint is workless only if the force it exerts on the bodies is of the form

$$\begin{pmatrix} \mathbf{j}_{i1}^T \\ \vdots \\ \mathbf{j}_{in}^T \end{pmatrix} \boldsymbol{\lambda}_i \tag{3}$$

where $\boldsymbol{\lambda}_i$ is a column vector of dimension m (the dimension of the ith constraint). We call the vector $\boldsymbol{\lambda}_i$ the Lagrange multiplier of the ith constraint. (If the ith constraint is not maintained by such a force, it must be treated as an auxiliary constraint.)

To talk about a total of q constraints, we switch to matrix notation. We can express these q multi-dimensional acceleration conditions in the form

$$\begin{aligned} \mathbf{j}_{11}\dot{\mathbf{v}}_1 + \cdots + \mathbf{j}_{1n}\dot{\mathbf{v}}_n + \mathbf{c}_1 &= \mathbf{0} \\ \mathbf{j}_{21}\dot{\mathbf{v}}_1 + \cdots + \mathbf{j}_{2n}\dot{\mathbf{v}}_n + \mathbf{c}_2 &= \mathbf{0} \\ &\vdots \\ \mathbf{j}_{q1}\dot{\mathbf{v}}_1 + \cdots + \mathbf{j}_{qn}\dot{\mathbf{v}}_n + \mathbf{c}_q &= \mathbf{0}. \end{aligned} \tag{4}$$

If we define

$$\mathbf{J} = \begin{pmatrix} \mathbf{j}_{11} & \mathbf{j}_{12} & \cdots & \mathbf{j}_{1n} \\ \vdots & \vdots & & \vdots \\ \mathbf{j}_{q1} & \mathbf{j}_{q2} & \cdots & \mathbf{j}_{qn} \end{pmatrix} \qquad \text{and} \qquad \mathbf{c} = \begin{pmatrix} \mathbf{c}_1 \\ \vdots \\ \mathbf{c}_q \end{pmatrix}$$

then we can replace equation (4) with simply

$$\mathbf{J}\dot{\mathbf{v}} + \mathbf{c} = \mathbf{0}. \tag{5}$$

In a similar fashion, we group the individual vectors $\boldsymbol{\lambda}_1$ through $\boldsymbol{\lambda}_n$ into one large vector $\boldsymbol{\lambda} = (\boldsymbol{\lambda}_1, \ldots, \boldsymbol{\lambda}_n)$.

From equation (3), we see that the vector being multiplied by $\boldsymbol{\lambda}_i$ forms the ith block-column of $\mathbf{J}^T$; accordingly, the sum of all the individual constraint forces must have therefor have the form $\mathbf{J}^T\boldsymbol{\lambda}$. The problem now is to find a vector $\boldsymbol{\lambda}$ so that the constraint force $\mathbf{J}^T\boldsymbol{\lambda}$, combined with any external forces (such as gravity), produces a motion of the system that satisfies the constraints; that is, $\mathbf{J}\dot{\mathbf{v}} + \mathbf{c} = \mathbf{0}$.

[3] A lengthy discussion on the topic of rheonomic, scleronomic, monogenic and polygenic constraints and forces, as in Lanczos [8], can pin down an exact definition, but offers little insight. We forego such a discussion here. A precise, but nonconstructive mathematical definition would be to say that workless constraint forces are those which maintain the system according to Gauss' "principle of least constraint."

5 The $\mathbf{JM}^{-1}\mathbf{J}^T$ Approach

The formulation most commonly used by the graphics community to compute $\boldsymbol{\lambda}$ is as follows. Given that an unknown constraint force $\mathbf{J}^T\boldsymbol{\lambda}$ acts upon the bodies, and letting $\mathbf{F}^{\text{ext}}$ represent the known net external force acting on the system (including all inertial velocity-dependent forces), from equation (1) we know that

$$\mathbf{M}\dot{\mathbf{v}} = \mathbf{J}^T\boldsymbol{\lambda} + \mathbf{F}^{\text{ext}}.$$

Solving for $\dot{\mathbf{v}}$, this yields

$$\dot{\mathbf{v}} = \mathbf{M}^{-1}\mathbf{J}^T\boldsymbol{\lambda} + \mathbf{M}^{-1}\mathbf{F}^{\text{ext}}. \tag{6}$$

Thus, once we compute $\boldsymbol{\lambda}$, we will be able to easily compute $\dot{\mathbf{v}}$. Since $\mathbf{M}$ is block diagonal, $\mathbf{M}^{-1}$ is as well. Substituting equation (6) into equation (5), we obtain

$$\mathbf{J}(\mathbf{M}^{-1}\mathbf{J}^T\boldsymbol{\lambda} + \mathbf{M}^{-1}\mathbf{F}^{\text{ext}}) + \mathbf{c} = \mathbf{0}.$$

If the matrix $\mathbf{A}$ and vector $\mathbf{b}$ are defined by

$$\mathbf{A} = \mathbf{JM}^{-1}\mathbf{J}^T \qquad \text{and} \qquad \mathbf{b} = -(\mathbf{JM}^{-1}\mathbf{F}^{\text{ext}} + \mathbf{c})$$

then we can express $\boldsymbol{\lambda}$ as the solution of the equation

$$\mathbf{A}\boldsymbol{\lambda} = \mathbf{b}. \tag{7}$$

This formulation has a number of desirable properties. First, assuming that $\mathbf{J}$ has full rank (equivalently, none of the imposed constraints are conflicting or redundant) then since $\mathbf{M}^{-1}$ is symmetric positive definite, $\mathbf{A}$ is as well. Note that for an articulated structure, $\mathbf{J}$ automatically has full rank, independent of the structure's current geometric configuration.[4]

As long as $\mathbf{A}$ is not too large, we can use direct methods to compute $\boldsymbol{\lambda}$. In particular, when $\mathbf{A}$ is nonsingular, the Choleski decomposition is an excellent method for computing $\boldsymbol{\lambda}$. As $\mathbf{A}$ becomes larger, iterative methods can be used to solve equation (7), either by explicitly forming the matrix $\mathbf{A}$ when it is sparse, or by using methods that work in terms of the (always) sparse factors $\mathbf{J}$ and $\mathbf{M}^{-1}$. In discussing the sparsity of $\mathbf{A}$, we regard $\mathbf{A}$ as a block matrix, with the blocks defined by the blocks of $\mathbf{M}$ and $\mathbf{J}$.

At this point, we restrict ourselves to constraints that act between a pair of bodies. Referring to equation (2), this means that for a given value i, only two elements of the ith block-row of $\mathbf{J}$ are nonzero. If constraint i acts on bodies r and s, then only $\mathbf{j}_{ir}$ and $\mathbf{j}_{is}$ will be nonzero. How does this translate to sparsity on the matrix $\mathbf{A}$? From the definition of $\mathbf{A}$, the ijth block of $\mathbf{A}$ is

$$\mathbf{A}_{ij} = \sum_{k=1}^{n} (\mathbf{j}_{ik})(\mathbf{M}_k)\left(\mathbf{j}_{jk}^T\right).$$

[4]The *inverse dynamics* of a straight chain are singular; however, the forward dynamics are always well defined. Contrary to popular belief, $\mathbf{A}$ remains nonsingular for articulated figures unless one accidentally repeats some of the articulation constraints in forming $\mathbf{J}$. However, a perfectly straight chain that has *both* its endpoints constrained does result in a singular matrix $\mathbf{A}$.

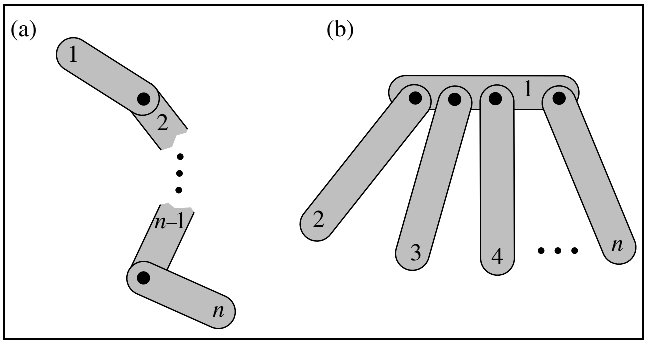

Figure 1: (a) A serial chain. (b) A branched object yielding a complete dense matrix $\mathbf{A} = \mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T$.

When is $\mathbf{A}_{ij}$ nonzero? Since each $\mathbf{M}_k$ is nonzero, $\mathbf{A}_{ij}$ is nonzero only if there exists k such that $\mathbf{j}_{ik}\mathbf{j}_{jk}^T \neq \mathbf{0}$. From equation (2), this means that there must exist a body k that both the ith and jth constraint affect.

As was previously pointed out, serial chains yield tightly banded matrix system. Assuming a chain of n links ordered so that body i connects to body $i+1$ (figure 1a) we see that $\mathbf{A}_{ij}$ is zero if $|i-j| > 2$. Thus, we can trivially solve $\mathbf{A}\boldsymbol{\lambda} = \mathbf{b}$ in $O(n)$ time using a banded solution method (e.g. banded Cholesky decomposition). However, if we have instead a branching structure, so that neither $\mathbf{A}$ (nor any permutation of $\mathbf{A}$) is banded, can we find some general way to exploit the sparsity of $\mathbf{A}$? The answer to this is "no," because $\mathbf{A}$ is not necessarily sparse at all!

Consider a structure where constraint 1 acts between body 1 and 2, constraint 2 acts between body 1 and 3, and so on (figure 1b). The matrix $\mathbf{A}$ for this structure is not sparse at all: in fact, $\mathbf{A}$ is completely dense, because every constraint has body 1 in common (i.e. the product $(\mathbf{j}_{i1})\mathbf{M}_1(\mathbf{j}_{j1}^T)$ is nonzero for all pairs i and j). To exploit sparsity we must abandon the approach of computing $\boldsymbol{\lambda}$ in terms of the matrix $\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T$.

6 An (Always) Sparse Formulation

The matrix $\mathbf{A}$ is square and has dimension $N_c \times N_c$ where N_c is the sum of all the constraint's dimensions. Instead of computing $\boldsymbol{\lambda}$ in terms of $\mathbf{A}$, consider the matrix equation

$$\begin{pmatrix} \mathbf{M} & -\mathbf{J}^T \\ -\mathbf{J} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{y} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \mathbf{0} \\ -\mathbf{b} \end{pmatrix}. \tag{8}$$

The top row yields $\mathbf{M}\mathbf{y} - \mathbf{J}^T\boldsymbol{\lambda} = \mathbf{0}$, or equivalently, $\mathbf{y} = \mathbf{M}^{-1}\mathbf{J}^T\boldsymbol{\lambda}$. Substituting this into the bottom row and multiplying by -1 yields

$$\mathbf{J}\mathbf{y} = \mathbf{J}(\mathbf{M}^{-1}\mathbf{J}^T\boldsymbol{\lambda}) = \mathbf{b}$$

which is equation (7). Thus, we can compute $\boldsymbol{\lambda}$ by solving equation (8) for $\boldsymbol{\lambda}$ and $\mathbf{y}$ (although $\mathbf{y}$ is an unneeded byproduct).

Let us define the matrix of equation (8) by writing

$$\mathbf{H} = \begin{pmatrix} \mathbf{M} & -\mathbf{J}^T \\ -\mathbf{J} & \mathbf{0} \end{pmatrix}.$$

This formulation is commonly seen in the robotics and mechanical-engineering literature. While some see the use of **H** as helping to explicitly separate the equations of motion (the top row of the matrix) from the constraint conditions (the bottom row of the matrix), it is clear that actually computing $\boldsymbol{\lambda}$ directly from equation (8) is a very foolish thing to do, *using dense matrix methods*. Using an $O(n^3)$ technique, equation (7) is easier to solve because **A** is much smaller than **H** and also because **A** is positive definite, while **H** is not. However, when we consider the problem from a sparse viewpoint, it becomes apparent that equation (8) is superior to equation (7), because **H** is *always* sparse. In the next section, we describe a simple $O(n)$ solution procedure for solving equation (8).

7 A Sparse Solution Method

Our $O(n)$ algorithm is based solely on the properties of the graph of **H**. The *graph* of a square symmetric s block by s block matrix **H** is an undirected graph with s nodes. For $i \neq j$, there is an edge between nodes i and j if $\mathbf{H}_{ij}$ is nonzero. (The diagonal elements of **H** are always regarded as nonzero elements, but they do not contribute edges to the graph.) Because the connectivity of the primary constraint is acyclic, the graph of **H** is also acyclic; hence, **H**'s graph is a tree.[5] For example, consider the structure shown in figure 2a: the matrix **J** associated with this set of constraints has the form

$$\mathbf{J} = \begin{pmatrix} \mathbf{j}_{11} & \mathbf{j}_{12} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{j}_{22} & \mathbf{j}_{23} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{j}_{32} & \mathbf{0} & \mathbf{j}_{34} & \mathbf{0} & \mathbf{0} \\ \mathbf{j}_{41} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{j}_{45} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{j}_{55} & \mathbf{j}_{56} \end{pmatrix}$$

and thus yields the matrix

$$\mathbf{H} = \begin{pmatrix} \mathbf{M}_1 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{j}_{11}^T & \mathbf{0} & \mathbf{0} & \mathbf{j}_{41}^T & \mathbf{0} \\ \mathbf{0} & \mathbf{M}_2 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{j}_{12}^T & \mathbf{j}_{22}^T & \mathbf{j}_{32}^T & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{M}_3 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{j}_{23}^T & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{M}_4 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{j}_{34}^T & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{M}_5 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{j}_{45}^T & \mathbf{j}_{55}^T \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{M}_6 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{j}_{56}^T \\ \mathbf{j}_{11} & \mathbf{j}_{12} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{j}_{22} & \mathbf{j}_{23} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{j}_{32} & \mathbf{0} & \mathbf{j}_{34} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{j}_{41} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{j}_{45} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{j}_{55} & \mathbf{j}_{56} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{pmatrix} \tag{9}$$

The graph defined by **H** is shown in figure 2b.

[5] If the primary constraints partition the bodies into discrete components, **H**'s graph is a forest (i.e. a set of trees). For simplicity, assume the primary constraints do not partition the bodies into more than one component.

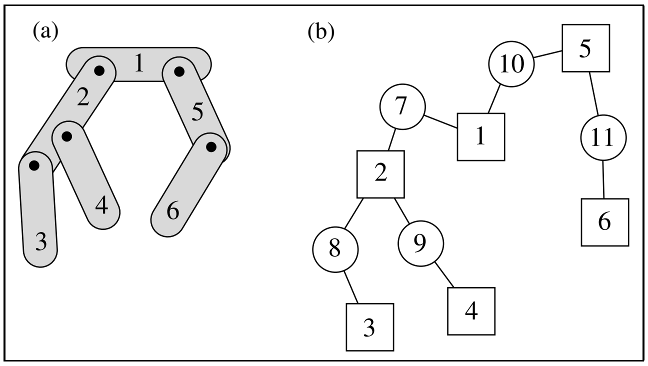

Figure 2: (a) An articulated object. (b) The graph of the matrix H corresponding to the object. Nodes corresponding to bodies are squares; circles indicate constraint nodes. For clarity, constraints are numbered beginning with 7.

Our first thought was to solve equation (8) by computing the Choleski decomposition $\mathbf{H} = \mathbf{L}\mathbf{L}^T$ where $\mathbf{L}$ is lower triangular. Unfortunately, this does not work because the lower-right corner of $\mathbf{H}$ is zero, making $\mathbf{H}$ indefinite. Instead, we factor $\mathbf{H}$ as $\mathbf{H} = \mathbf{L}\mathbf{D}\mathbf{L}^T$ where $\mathbf{L}$ is a lower-triangular block matrix whose diagonal entries are identity matrices, and $\mathbf{D}$ is a block-diagonal matrix. We then solve the system $\mathbf{L}\mathbf{D}\mathbf{L}^T\mathbf{x} = \left(\begin{smallmatrix}\mathbf{0}\\ -\mathbf{b}\end{smallmatrix}\right)$ and extract the portion of $\mathbf{x}$ which corresponds to $\boldsymbol{\lambda}$. Although $\mathbf{H}$ is always sparse, we must permute $\mathbf{H}$ to exploit this sparsity.

7.1 Elimination Order

A fundamental fact of sparse-matrix theory is that a matrix whose graph is acyclic possesses a *perfect elimination order*; this means that $\mathbf{H}$ can be reordered so that when factored, the matrix factor $\mathbf{L}$ will be *just as sparse as* $\mathbf{H}$. As a result $\mathbf{L}$ can be computed in $O(n)$ time (and stored in $O(n)$ space) and then $\mathbf{L}\mathbf{D}\mathbf{L}^T\mathbf{x} = \left(\begin{smallmatrix}\mathbf{0}\\ -\mathbf{b}\end{smallmatrix}\right)$ can be solved in $O(n)$ time.

The matrix $\mathbf{H}$ is correctly ordered if it satisfies the following property. Let us view $\mathbf{H}$'s graph as a rooted tree, with node n being the root. This defines a parent/child relationship between every pair of nodes connected by an edge. The matrix $\mathbf{H}$ must be ordered so that the each node's index is greater than the indices of its children. When $\mathbf{H}$ is ordered so that the tree has this property, then the factor $\mathbf{L}$ will have its nonzero entries only where $\mathbf{H}$ has nonzero entries. An ordering with this property is trivially found by perfoming a depth-first search on the original $\mathbf{H}$'s graph (see appendix A). The reordered matrix $\mathbf{H}$, when factored, is said to have no "fill-in"; in other words, factoring methods such as Gaussian elimination (or the $\mathbf{L}\mathbf{D}\mathbf{L}^T$ decomposition we will use) do not introduce new nonzero elements during the factoring process.

As an example, a proper reordering of the matrix in equation (9) would be

$$\mathbf{H} = \begin{pmatrix}
\mathbf{M}_3 & \mathbf{j}_{23}^T & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{j}_{23} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{j}_{22} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{M}_4 & \mathbf{j}_{34}^T & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{j}_{34} & \mathbf{0} & \mathbf{j}_{32} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{j}_{22}^T & \mathbf{0} & \mathbf{j}_{32}^T & \mathbf{M}_2 & \mathbf{j}_{12}^T & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{j}_{12} & \mathbf{0} & \mathbf{j}_{11} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{j}_{11}^T & \mathbf{M}_1 & \mathbf{j}_{41}^T & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{j}_{41} & \mathbf{0} & \mathbf{j}_{45} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{j}_{45}^T & \mathbf{M}_5 & \mathbf{0} & \mathbf{j}_{55}^T \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{M}_6 & \mathbf{j}_{56}^T \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{j}_{55} & \mathbf{j}_{56} & \mathbf{0}
\end{pmatrix}.$$

In actual practice, **H** is not actually changed; rather the rows and columns are processed in a particular order. The bookkeeping associated with this is very simple and is given in appendix A.

7.2 An $O(n^3)$ Factorization Method

If we treat **H** as dense, then an $O(n^3)$ solution method is as follows. First, the upper triangular portion of **H** is overwritten with the entries of $\mathbf{L}^T$, and the diagonal of **H** is overwritten with the entries of **D**. (The diagonal entries of **L** are identity matrices so there is no reason to keep track of them.) The code for this is short, and requires $O(n^3)$ time:

procedure densefactor
for $i = 1$ **to** n
  for $k = i-1$ **to** 1
    $\mathbf{H}_{ii} = \mathbf{H}_{ii} - \mathbf{H}_{ki}^T \mathbf{H}_{ii} \mathbf{H}_{ki}$
  for $j = i+1$ **to** n
    for $k = i-1$ **to** 1
      $\mathbf{H}_{ij} = \mathbf{H}_{ij} - \mathbf{H}_{ki}^T \mathbf{H}_{kk} \mathbf{H}_{kj}$
    $\mathbf{H}_{ij} = \mathbf{H}_{ii}^{-1} \mathbf{H}_{ij}$

Then, defining $\mathbf{z} = \binom{\mathbf{0}}{-\mathbf{b}}$, we solve $\mathbf{L}\mathbf{x}^{(1)} = \mathbf{z}$, followed by $\mathbf{D}\mathbf{x}^{(2)} = \mathbf{x}^{(1)}$ and finally $\mathbf{L}^T\mathbf{x} = \mathbf{x}^{(2)}$, which yields a solution to $\mathbf{H}\mathbf{x} = \mathbf{z}$, with the lower portion of **x** containing $\boldsymbol{\lambda}$. This can be done (successively overwriting **x** at each step) in $O(n^2)$ time:

procedure densesolve
for $i = 1$ **to** n
  $\mathbf{x}_i = \mathbf{z}_i$
  for $j = 1$ **to** $i-1$
    $\mathbf{x}_i = \mathbf{x}_i - \mathbf{H}_{ij}^T \mathbf{x}_j$
for $i = n$ **to** 1
  $\mathbf{x}_i = \mathbf{H}_{ii}^{-1} \mathbf{x}_i$
  for $j = i+1$ **to** n
    $\mathbf{x}_i = \mathbf{x}_i - \mathbf{H}_{ij} \mathbf{x}_j$

7.3 An $O(n)$ Factorization Method

Now let us treat $\mathbf{H}$ as sparse. To simplify our discussion of the solution procedure below, assume we are dealing with a matrix $\mathbf{H}$ which has been reordered as described in section 7.1. To make the two previous procedures run in linear time, we need a small amount of book-keeping. Let us define $\text{par}(i) = j$ to denote that in $\mathbf{H}$'s graph, node j is node i's parent. Conversely, define $\text{child}(j) = \{i \mid \text{par}(i) = j\}$ and note that

- if $i < j$ then $\mathbf{H}_{ij}$ is nonzero only if $\text{par}(i) = j$ which means that $i \in \text{child}(j)$ and
- if $i > j$, then $\mathbf{H}_{ij}$ is nonzero only if $\text{par}(j) = i$, which means that $j \in \text{child}(i)$.

Since every node in the graph has at most one parent, $\mathbf{H}$ has the property that in each row, only *one* nonzero block ever occurs to the *right* of the diagonal. We can store the upper triangular portion of $\mathbf{H}$ by row, with each row having only two entries (one entry for the diagonal, and one entry for the single nonzero element to the right of the diagonal). As we overwrite $\mathbf{H}$ with $\mathbf{L}^T$, this structure is preserved. The pseudocode in appendix A gives specific implementation details.

Given these relations, we can simplify the $O(n^3)$ method as follows. In lines 3 and 4 of **densefactor**, k is less than i, which means $\mathbf{H}_{ki}$ is nonzero only for $k \in \text{child}(i)$. Lines 6 and 7 can be omitted entirely, because $k < i < j$, so that the product $\mathbf{H}_{ki}^T\mathbf{H}_{kk}\mathbf{H}_{kj}$ is always zero (since k cannot be i's child *and* j's child). Finally, since $i < j$ in line 8 and $\mathbf{H}_{ij}$ is nonzero only when $j = \text{par}(i)$, the factoring step reduces to simply

procedure sparsefactor
for $i = 1$ **to** n
  for $k \in \text{child}(i)$
    $\mathbf{H}_{ii} = \mathbf{H}_{ii} - \mathbf{H}_{ki}^T\mathbf{H}_{ii}\mathbf{H}_{ki}$
  if $i \neq n$
    $\mathbf{H}_{i,\text{par}(i)} = \mathbf{H}_{ii}^{-1}\mathbf{H}_{i,\text{par}(i)}$

Note that assignment to $\mathbf{H}_{ii}$ is executed once for each child node of another node, which means that **sparsefactor** takes time $O(n)$. Employing a similar strategy, we solve $\mathbf{Hx} = \binom{\mathbf{0}}{-\mathbf{b}} = \mathbf{z}$ in $O(n)$ time:

procedure sparsesolve
for $i = 1$ **to** n
  $\mathbf{x}_i = \mathbf{z}_i$
  for $j \in \text{child}(i)$
    $\mathbf{x}_i = \mathbf{x}_i - \mathbf{H}_{ij}^T\mathbf{x}_j$
for $i = n$ **to** 1
  $\mathbf{x}_i = \mathbf{H}_{ii}^{-1}\mathbf{x}_i$
  if $i \neq n$
    $\mathbf{x}_i = \mathbf{x}_i - \mathbf{H}_{i,\text{par}(i)}\mathbf{x}_{\text{par}(i)}$

After computing $\mathbf{x}$, we extract the appropriate elements to form the vector $\boldsymbol{\lambda}$, and then perform two (sparse!) multiplications to compute

$$\dot{\mathbf{v}} = \mathbf{M}^{-1}(\mathbf{J}^T\boldsymbol{\lambda} + \mathbf{F}^{\text{ext}}).$$

Thus, we can compute an acceleration $\dot{\mathbf{v}}$ that satisfies the primary constraints in only $O(n)$ time. A complete (yet surprisingly short) pseudocode implementation of both **sparsefactor** and **sparsesolve**, using sparse datastructures, is presented in appendix A.

8 Auxiliary Constraints

Now that we know how to quickly compute the multipliers for the primary constraints, we can turn our attention to handling the auxiliary constraints (such as loop-closure or contact) which cannot be formulated as part of the primary constraints. In this section, it is best to internalize the results of the last few sections as the statement "we can quickly determine the primary constraint force $\mathbf{J}^T\boldsymbol{\lambda}$ that would arise in response to an external force $\mathbf{F}^{\text{ext}}$."

8.1 Constraint Anticipation

Our approach to computing the multipliers for the secondary constraints is as follows. We will begin by first computing the multipliers for the *auxiliary* constraints; however, in doing so, we will anticipate the response of the primary constraints due to the auxiliary constraint forces. Once we have computed the auxiliary constraint forces, we then go back and compute the primary constraint forces; but since we have already anticipated their effects, adding the primary constraint forces into the system will not violate the conditions of the auxiliary constraints.

This "anticipation" of the primary constraints effects is as follows. Consider a force $\mathbf{F}$ acting on the system. If not for the primary constraints, the accelerational response of the system in reaction to the force $\mathbf{F}$ would be $\dot{\mathbf{v}} = \mathbf{M}^{-1}\mathbf{F}$. However, because of the primary constraints, the response is quite different. What we would like to do is compute a new mass matrix $\widehat{\mathbf{M}}$ which reflects how the system behaves as a result of the primary constraints. That is, we would like to be able to write that the response of the system due to a force $\mathbf{F}$ is, taking into account the primary constraints, $\widehat{\mathbf{M}}^{-1}\mathbf{F}$. We will not compute either the actual matrix $\widehat{\mathbf{M}}$ or its inverse $\widehat{\mathbf{M}}^{-1}$; we will use the $O(n)$ method of section 7.3 to compute vectors $\widehat{\mathbf{M}}^{-1}\mathbf{F}$ for a variety of forces $\mathbf{F}$.

In describing the k auxiliary constraints, we will regard each constraint as a separate, one-dimensional constraint. This means the matrix system we build will not have any block structure: this is appropriate, because the matrix system will be in general completely dense. For each constraint, we will produce a scalar expression a_i which is a measure of acceleration; each a_i will have an associated scalar multiplier μ_i. The relation between the vector of a's and the vector of μ's is, as always, a linear relation. Our goal here is to show how we can efficiently compute the $k \times k$ coefficient matrix that relates the a_i's to the μ_i's in $O(kn) + O(k^2)$ time, where n is the number of primary constraints to be maintained by the system.[6] Once we have computed this coefficient matrix, we can use known techniques to

[6] If each auxiliary constraint acts on only one or two bodies, the time required to formulate the system is $O(kn) + O(k^2)$. If auxiliary constraints constrain n bodies at a time, (which is rare), the time becomes $O(kn) + O(nk^2)$ In either case, it is the $O(kn)$ term which dominates; the constant in front of the $O(k^2)$ term or $O(nk^2)$ term is small.

compute the μ_i multipliers. For an equality constraint, a_i is forced to be zero, and μ_i can have any sign. If all the constraints are equality constraints, we can solve for the μ_i using standard matrix techniques in time $O(k^3)$. Going beyond this, simple workless inequality constraints, such as contact or joint-angle limits require that $a_i \geq 0$ and $a_i \mu_i = 0$. Methods for handling a mix of equality, inequality and frictional conditions are described by Baraff [2] and have time complexity $O(k^3)$ for a system of k constraints. As long as k is small compared to n, it is the computation of the $k \times k$ matrix of coefficients which dominates the running time, and not the computation of the μ_i multipliers. Thus, our focus here is on computing the coefficient matrix as opposed to the multipliers themselves.

The auxiliary constraints are described in a form similar to that of equation (2). Let the vector $\mathbf{a}$ of the k auxiliary a_i variables be expressed in the form

$$\mathbf{a} = \begin{pmatrix} a_i \\ \vdots \\ a_k \end{pmatrix} = \mathbf{J}^a \dot{\mathbf{v}} + \mathbf{c}^a \tag{10}$$

where $\mathbf{J}^a$ has k rows and $\mathbf{c}^a \in \mathbb{R}^k$. Since the auxiliary constraint forces do not have to be workless, let the constraint force acting on the system due to the ith constraint have the form

$$\mathbf{k}_i \mu_i$$

where $\mathbf{k}_i$ is a column vector of the same dimension as $\mathbf{v}$ (that is, $\mathbf{k}$'s dimension is the sum of all the bodies' dimensions). Defining $\mathbf{K}$ as the k-column matrix

$$\mathbf{K} = [\, \mathbf{k}_1 \quad \mathbf{k}_2 \quad \dots \quad \mathbf{k}_k \,] \tag{11}$$

the constraint force due to all k constraints has the form

$$\mathbf{k}_1 \mu_i + \cdots + \mathbf{k}_k \mu_k = \mathbf{K}\boldsymbol{\mu}.$$

The process of computing both the primary and auxiliary multipliers is as follows. First, we compute what $\dot{\mathbf{v}}$ would be *without* the auxiliary constraints. That is, given an external force $\mathbf{F}^{\text{ext}}$, we solve equation (7) for $\boldsymbol{\lambda}$ (using **sparsefactor** and **sparsesolve**). We then define

$$\widehat{\mathbf{F}}^{\text{ext}} = \mathbf{J}^T \boldsymbol{\lambda} + \mathbf{F}^{\text{ext}}.$$

The force $\widehat{\mathbf{F}}^{\text{ext}}$ is the external force *as seen by the auxiliary constraints*. (Remember, the auxiliary constraints are formulated so as to anticipate the response of the primary constraints. The first step in this anticipation is to know what the primary constraint force would have been in the absence of any auxiliary constraint forces.) Having computed $\widehat{\mathbf{F}}^{\text{ext}}$, we know that the system's acceleration without the auxiliary constraints is $\mathbf{M}^{-1}\widehat{\mathbf{F}}^{\text{ext}}$. Let us write

$$\dot{\mathbf{v}}^{\text{aux}} = \mathbf{M}^{-1}\widehat{\mathbf{F}}^{\text{ext}}$$

to express this. The auxiliary constraint forces must now “kick in” to the extent that the acceleration $\dot{\mathbf{v}}^{\text{aux}}$ violates the auxiliary constraints.

Using the anticipated response matrix $\widehat{\mathbf{M}}^{-1}$, the acceleration $\dot{\mathbf{v}}$ of the system in response to an auxiliary constraint force $\mathbf{K}\boldsymbol{\mu}$ is the system's acceleration without the auxiliary constraint force, $\dot{\mathbf{v}}^{\text{aux}}$, plus the response to $\mathbf{K}\boldsymbol{\mu}$:

$$\dot{\mathbf{v}} = \widehat{\mathbf{M}}^{-1}\mathbf{K}\boldsymbol{\mu} + \dot{\mathbf{v}}^{\text{aux}}.$$

If we actually had access to the matrix $\widehat{\mathbf{M}}^{-1}$, we could stop at this point: from equation (10), we obtain

$$\mathbf{a} = \mathbf{J}^a\dot{\mathbf{v}} + \mathbf{c}^a = \mathbf{J}^a\widehat{\mathbf{M}}^{-1}\mathbf{K}\boldsymbol{\mu} + (\mathbf{J}^a\dot{\mathbf{v}}^{\text{aux}} + \mathbf{c}^a) \tag{12}$$

which gives the desired relation between $\mathbf{a}$ and $\boldsymbol{\mu}$. (At this point, we can easily evaluate $\mathbf{J}^a\dot{\mathbf{v}}^{\text{aux}} + \mathbf{c}^a$, since we have actually computed $\dot{\mathbf{v}}^{\text{aux}}$.) The real trick then is to compute the coefficient matrix $\widehat{\mathbf{M}}^{-1}\mathbf{K}$.

Remember that equation (11) defines $\mathbf{K}$ in terms of columns $\mathbf{k}_i$, and that $\mathbf{k}_i$ is the direction that the ith auxiliary constraint force acts in. We cannot (nor do we wish to) formulate $\widehat{\mathbf{M}}^{-1}$ directly; instead, we wish to compute $\widehat{\mathbf{M}}^{-1}\mathbf{K}$ column by column. Since $\widehat{\mathbf{M}}^{-1}$ encapsulates the response of the system to a force, given a vector $\mathbf{k}_i$, we compute $\widehat{\mathbf{M}}^{-1}\mathbf{k}_i$ as follows. The primary constraints, in reaction to a force $\mathbf{k}_i$, generate a response force $\mathbf{F}^{\text{resp}} = \mathbf{J}^T\boldsymbol{\lambda}$ where $\mathbf{A}\boldsymbol{\lambda} = \mathbf{J}\mathbf{M}^{-1}\mathbf{k}_i$. As a result, the system's response to a force $\mathbf{k}_i$, is not $\mathbf{M}^{-1}\mathbf{k}_i$, but rather

$$\mathbf{M}^{-1}(\mathbf{F}^{\text{resp}} + \mathbf{k}_i).$$

This gives us a computational definition of $\widehat{\mathbf{M}}^{-1}$: we can now write that the system's response to the force $\mathbf{k}_i$ is

$$\widehat{\mathbf{M}}^{-1}\mathbf{k}_i = \mathbf{M}^{-1}(\mathbf{F}^{\text{resp}} + \mathbf{k}_i)$$

where $\mathbf{F}^{\text{resp}} = \mathbf{J}^T\boldsymbol{\lambda}$ and $\boldsymbol{\lambda}$ is computed by solving $\mathbf{A}\boldsymbol{\lambda} = \mathbf{J}\mathbf{M}^{-1}\mathbf{k}_i$. The cost to compute $\widehat{\mathbf{M}}^{-1}\mathbf{k}_i$ is thus $O(n)$. Given equation (11), we can express $\widehat{\mathbf{M}}^{-1}\mathbf{K}$ column-wise as

$$\widehat{\mathbf{M}}^{-1}\mathbf{K} = [\,\widehat{\mathbf{M}}^{-1}\mathbf{k}_1 \quad \widehat{\mathbf{M}}^{-1}\mathbf{k}_2 \quad \dots \quad \widehat{\mathbf{M}}^{-1}\mathbf{k}_k\,]$$

where each column $\widehat{\mathbf{M}}^{-1}\mathbf{k}_i$ is computed according to the above procedure. The cost to do this $O(nk)$, since we have k columns, and each column requires $O(n)$ work. Having computed $\widehat{\mathbf{M}}^{-1}\mathbf{K}$, we can easily compute the coefficient matrix $\mathbf{J}^a\widehat{\mathbf{M}}^{-1}\mathbf{K}$ of equation (12). If $\mathbf{J}^a$ is sparse, the $k \times k$ matrix $\mathbf{J}^a\widehat{\mathbf{M}}^{-1}\mathbf{K}$ is computed in $O(k^2)$ time while a dense matrix $\mathbf{J}^a$ takes $O(nk^2)$ time.

8.2 Computing the Net Constraint Force

It is extremely important to note that although we must compute a total of $k+2$ different $\boldsymbol{\lambda}$'s during the solution process (see below), each $\boldsymbol{\lambda}$ is actually computed by solving a system of the form $\mathbf{Hx} = \begin{pmatrix} \mathbf{0} \\ -\mathbf{b} \end{pmatrix}$ and then extracting $\boldsymbol{\lambda}$ from $\mathbf{x}$. The significance of this is that what is changing each time is *not* the matrix $\mathbf{H}$, but $\mathbf{b}$. This means that we call the procedure **sparsefactor** of section 7.3 only once during the entire constraint force computation described below; for each different vector $\mathbf{b}$, we only need to perform the second step **sparsesolve**. Although both steps take $O(n)$ time, **sparsefactor** is approximately four times as expensive as **sparsesolve**. Thus, refactoring each time would still yield an $O(n)$ algorithm, but would needlessly repeat computation.

At this point, the entire sequence of steps required may sound complicated, but again, the implementation is straightforward. In the description below, whenever we solve an equation $\mathbf{A}\boldsymbol{\lambda} = \mathbf{b}$ we do so in terms of the associated equation $\mathbf{Hx} = \begin{pmatrix} \mathbf{0} \\ -\mathbf{b} \end{pmatrix}$ of the previous section. The steps for the entire solution process are as follows.

1. Formulate the sparse matrix $\mathbf{H}$ for the primary constraints, and run **sparsefactor** to factor $\mathbf{H}$.

2. Given the external force $\mathbf{F}^{\text{ext}}$, compute the primary constraint force $\mathbf{J}^T\boldsymbol{\lambda}$ due to $\mathbf{F}^{\text{ext}}$ by solving $\mathbf{A}\boldsymbol{\lambda} = -(\mathbf{JM}^{-1}\mathbf{F}^{\text{ext}} + \mathbf{c})$. This requires one call to **sparsesolve**. Once $\boldsymbol{\lambda}$ has been computed, set $\dot{\mathbf{v}}^{\text{aux}} = \mathbf{M}^{-1}(\mathbf{J}^T\boldsymbol{\lambda} + \mathbf{F}^{\text{ext}})$.

3. For j from 1 to k, compute the response force $\mathbf{F}^{\text{resp}} = \mathbf{J}^T\boldsymbol{\lambda}$ by solving $\mathbf{A}\boldsymbol{\lambda} = -\mathbf{JM}^{-1}\mathbf{k}_j$. This requires k calls to **sparsesolve**. Forming the product $\widehat{\mathbf{M}}^{-1}(\mathbf{F}^{\text{resp}} + \mathbf{k}_j)$ yields the jth column of $\widehat{\mathbf{M}}^{-1}\mathbf{K}$. Multiplying the ith row of $\mathbf{J}^a$ with this jth column yields the ijth entry in the coefficient matrix $\mathbf{J}^a\widehat{\mathbf{M}}^{-1}\mathbf{K}$. Computing these k^2 different products takes either $O(k^2)$ or $O(nk^2)$ time, depending on the sparsity of $\mathbf{J}^a$.

4. Now that the coefficients of equation (12) have been determined, compute the multipliers $\boldsymbol{\mu}$, employing either a standard linear solution method (for example, Gaussian elimination) or the method for contact constraints and friction described by Baraff [2]. This takes approximately $O(k^3)$ time.

5. Given the auxiliary constraint force $\mathbf{K}\boldsymbol{\mu}$, compute the primary constraint's response to the force $\mathbf{F}^{\text{ext}} + \mathbf{K}\boldsymbol{\mu}$; that is, solve $\mathbf{A}\boldsymbol{\lambda} = -\mathbf{JM}^{-1}(\mathbf{K}\boldsymbol{\mu} + \mathbf{F}^{\text{ext}} + \mathbf{c})$. The final constraint force due to both the primary and auxiliary constraints is $\mathbf{K}\boldsymbol{\mu} + \mathbf{J}^T\boldsymbol{\lambda}$; adding to this the external force $\mathbf{F}^{\text{ext}}$ yields the net force acting on the system.

6. Compute the net acceleration of the system and move on to the next timestep.

9 Results

We have implemented the described system, and used it for a number of simulations. Simulations were run on an SGI Indigo^2 workstation, with a 250 Mhz R4400 processor. The 108 multipliers for a system of 2D rigid bodies with 54 two-dimensional primary constraints required 6.2 milliseconds to compute. Approximately 2.2 milliseconds of that time was spent computing the entries of $\mathbf{J}$. When the connectivity was changed so that there were 96 primary multipliers and 12 auxiliary multipliers, the computation time increased by about 14 milliseconds. Virtually all of this increase was due to the $O(nk)$ computation of the auxiliary constraint coefficient matrix $\widehat{\mathbf{M}}^{-1}\mathbf{K}$. The $O(k^3)$ time spent actually computing the 12 auxiliary constraint multipliers was too small to notice.

A 3D rigid body system with 96 primary multipliers and 3 auxiliary multipliers due to 3 frictionless contacts required 14.4 milliseconds. Approximately 3.5 milliseconds of that time was spent computing the entries of $\mathbf{J}$. A larger 3D system with 127 constraints resulting in 381 primary multipliers required 35.7 milliseconds, with approximately 11.2 milliseconds spent evaluating $\mathbf{J}$. It is worth pointing out that on the first problem, with 99 multipliers, the $O(n)$ method yields only a factor of two speedup over Baraff [2]'s $O(n^3)$ method for equality and inequality constraints. However, for the larger problem, the speedup is close to a factor of forty.

Schröder [11] discusses an implementation of a linear-time reduced-coordinate scheme due to Lathrop [9], and reports some running times. Adjusting for machine speeds, our results appear to be competitive with the figures reported by Schröder (but we had to guess about a number of parameters, so it is hard to say for sure). We do note that Schröder discusses a number of numerical difficulties with the algorithm; in fact, the use of a singular-value decomposition is required, which is always a sign of ill-conditioning. We were pleasantly surprised to find that the sparse methods described in this paper required *no* numerical adjustments, even on large examples—glancing at the pseudocode in appendix A, there are no numerical tolerance values to be found.[7]

We were able to run Gleicher's "Bramble" system on our 2D example. Bramble uses a Lagrange multiplier formulation, and exploits sparsity to compute multipliers by using a conjugate gradient method [6]. Comparing relative performance is still difficult, since the performance of any iterative method can vary greatly based on the desired accuracy of the answer; on the other hand, the ability to compute results to a lower (but acceptable) precision is one of the great strengths of iterative methods. For the 2D problem with 108 primary multipliers and no auxiliary multipliers, our method was about three times faster than Bramble at computing the multipliers; however, when we induced loops, changing 12 of the primary multipliers to auxiliary multipliers, both simulation systems ran at approximately the same same speed. Thus, for problems of this size, an $O(n^2)$ conjugate gradient method is competitive with the presented method. As problems grow larger (for example, the 3D

[7]The algorithm as described requires the inversion of small matrices (for rigid bodies, these matrices are at most of size 6×6). Since these matrices are always either positive or negative definite, a Cholesky decomposition can be used to simply and stably perform the inversion. The Cholesky decomposition has no numerical tolerance values in it either.

example with 381 multipliers) our $O(n)$ method enjoys a significant advantage. On today's machines, examples fast enough to run at interactive speeds enjoy modest speed gains using our linear-time algorithm; however, as machine speeds increase, allowing larger interactive-rate simulations, the difference between $O(n)$, (n^2), and $O(n^3)$ methods will only become more pronounced.

A Pseudocode

This appendix gives an implementation of the bookkeeping and datastructures needed to perform the computations described by procedures **sparsefactor** and **sparsesolve** in section 7.3. As you can see, the code is extremely short, and thus easily implementable. Each body and constraint is represented by a node structure; a node also stores a row of the upper triangular portion of **H**. Recall that rows of the upper triangular portion of the (properly ordered) matrix **H** only have two nonzero elements: the diagonal element itself (denoted D below), and one off-diagonal element (denoted J below). Each node also stores space for a portion of the solution vector **x**.

```
struct node {
    boolean   isbody;
    int       i;
    matrix    D, Dinv, J;
    vector    x;
}
```

A node corresponding to a body has isconstraint set false, and the index field i set to the index of the body the node represents. Both D and Dinv are square matrices of size dim(i). If a node corresponds to a constraint, then D and Dinv are square with size equal to the dimension of the constraint, and isconstraint is set true. The variable i is the index of the constraint; constraints are numbered starting from 1, because $\boldsymbol{\lambda}_1$ is the multiplier for the first constraint. We assume that for a node n, the function **parent**(n) yields n's parent, or **NULL** if n is the root. Similarly, **children**(n) yields the set of nodes that are children of n, or the empty set if n is a leaf. (Obviously, this can be done in terms of extra pointers stored within a **node** structure.)

The global variables Forward and Backward are lists of nodes, with Forward ordered so that parent nodes occur earlier in the list than their children, and Backward being the reverse of Forward. Thus, the notation "**for** n ∈ Forward" indicates processing nodes from the root down, while "**for** n ∈ Backward" indicates processing nodes from the leaves up. The following routine, called once with the root of the tree, initializes the two lists (assuming that Forward and Backward are initially empty):

```
procedure ordermatrix(n)
for c ∈ children(n)
    ordermatrix(c)
Forward = [ Forward   n ]
Backward = [ n   Backward ]
```

Assuming that we have procedures which compute the blocks $\mathbf{M}_i$ and $\mathbf{j}_{pq}$ (with $\mathbf{j}_{pq}$ defined as in section 4), we store and factor **H** as follows:

```
procedure factor
for n ∈ Forward
   if n.isconstraint
      n.D = 0
   else
      n.D = M_n.i
   if parent(n) ≠ NULL
      int p = n.i, q = parent(n).i
      n.J = n.isconstraint ? J_pq : J_pq^T
for n ∈ Forward
   for c ∈ children(n)
      n.D −= (c.J^T)(n.D)(c.J)
   n.Dinv = n.D^−1
   if parent(n) ≠ NULL
      parent(n).J = (n.Dinv)(parent(n).J)
```

As previously mentioned, after we have called **factor**, we can solve the system $\mathbf{Hx} = \begin{pmatrix} \mathbf{0} \\ -\mathbf{b} \end{pmatrix}$ (extracting $\boldsymbol{\lambda}$ from $\mathbf{x}$) as many times as we wish. The solution process computes $\boldsymbol{\lambda}$ as follows:

```
procedure solve(b)
for n ∈ Forward
   n.soln = n.isconstraint ? b_n.i : 0
   for c ∈ children(n)
      n.soln −= c.J^T c.soln
for n ∈ Backward
   n.soln = (n.Dinv)(n.soln)
   if parent(n) ≠ NULL
      n.soln −= (parent(n).J)(parent(n).soln)
   if n.isconstraint
      λ_n.i = n.soln
```

References

[1] D. Baraff. Issues in computing contact forces for non-penetrating rigid bodies. *Algorithmica*, 10:292–352, 1993.

[2] D. Baraff. Fast contact force computation for nonpenetrating rigid bodies. *Computer Graphics* (*Proc. SIGGRAPH*), 28:23–34, 1994.

[3] R. Barzel and A.H. Barr. A modeling system based on dynamic constraints. In *Computer Graphics* (*Proc. SIGGRAPH*), volume 22, pages 179–188. ACM, July 1988.

[4] J. Baumgarte. Stabilization of constraints and integrals of motion in dynamical systems. *Computer Methods in Applied Mechanics*, pages 1–36, 1972.

[5] R. Featherstone. *Robot Dynamics Algorithms*. Kluwer, 1987.

[6] M. Gleicher. *A Differential Approach to Graphical Manipulation*. PhD thesis, Carnegie Mellon University, 1994.

[7] G. Golub and C. Van Loan. *Matrix Computations*. John Hopkins University Press, 1983.

[8] C. Lanczos. *The Variational Principles of Mechanics*. Dover Publications, Inc., 1970.

[9] R.H. Lathrop. Constrained (closed-loop) robot simulation by local constraint propagation. In *International Conference on Robotics and Automation*, pages 689–694. IEEE, 1986.

[10] D. Negrut, R. Serban, and F.A. Potra. A topology based approach for exploiting the sparsity of multibody dynamics. Technical Report 84, Department of Mathematics, University of Iowa, December 1995.

[11] P. Schröder and D. Zeltzer. The virtual erector set: Dynamic simulation with linear recursive constraint propagation. In *Proceedings 1990 Symposium on Interactive 3d Graphics*, volume 24, pages 23–31, March 1990.

[12] M.C. Surles. An algorithm with linear complexity for interactive, physically-based modeling of large proteins. *Computer Graphics* (*Proc. SIGGRAPH*), 26:221–230, 1992.

[13] A.F. Vereshchagin. Computer simulation of the dynamics of complicated mechansisms of robot manipulators. *Engineering Cybernetics*, 6:65–70, 1974.

[14] A. Witkin, M. Gleicher, and W. Welch. Interactive dynamics. In *Proceedings 1990 Symposium on Interactive 3d Graphics*, volume 24, pages 11–21, March 1990.