Practical Physics for Articulated Characters

Evangelos Kokkevis
vangelis_kokkevis@playstation.sony.com

Sony Computer Entertainment America
Research & Development

Abstract

This presentation describes a set of techniques for implementing a fast and stable dynamics simulator for articulated characters using an analytical constraint approach combined with Featherstone's linear-time forward dynamics algorithm. It presents an efficient method for enforcing multiple simultaneous constraints on an articulated character in order to control the character's motion and model its interactions with the environment. The technique described employs an optimized process derived from Featherstone's recursive algorithm to form a linear system representing the constraint conditions at each time instant of the simulation. The system is subsequently solved to compute the exact magnitude of the forces necessary to satisfy these constraints. This work demonstrates how a combination of unilateral and bilateral constraints can be used to model the effects of impacts and contacts, to enforce joint limits, and to accurately control limb motion through trajectory following. The algorithms are presented from a practical standpoint and pseudocode is provided to facilitate their implementation. The goal of this presentation is twofold: first, to convince developers that building a practical, stable, real-time physics simulator for articulated characters is a tractable problem, and second, to give them enough confidence and information to start building one for their game.

1 Introduction

The impressive capabilities of the latest generation of video game hardware have raised our expectations of not only how digital characters look, but also how they move. As a result, game developers are becoming increasingly interested in animation techniques to generate natural looking motion for the complex characters that populate the modern game worlds. These characters should be able to respond realistically to unpredictable user control and interact with their environments in a believable way, requirements that traditional hand animation and motion capture fail to adequately address.

Developers are beginning to look at physics as a way to address some of the shortcomings of keyframed motion. Until recently, physics was considered an off-line process, but faster hardware and improved algorithms have helped change that perception. As a matter of fact, it is quite common these days for games to use real-time dynamics solvers for particles, cloth, soft and rigid bodies. These techniques are becoming more mainstream, partly through the licensing of efficient physics middleware and partly via in-house development of such systems using the wealth of information available on the internet, in articles, tutorials, and books.

The next logical step, and one that is highly anticipated, is to use physics to improve the realism of articulated character movement. After tackling secondary motions such as those of bellies and ponytails, developers are turning their attention to full body motion for staging dramatic death scenes, modeling high impact collisions between football players, simulating reactions to kicks and punches, etc. Middleware is of some help here as well, and the first character physics engines are already available. However, in comparison to particle systems and rigid bodies, significantly less information is publicly available on how to build articulated body physics simulators, especially ones that can withstand the rigors of a user controlled video game environment and still run in real-time and provide adequate control.

This presentation will describe a set of techniques that we have used at the Sony Playstation R&D group to build such a simulator from the ground up. The articulated body physics simulation project is part of our ongoing research on advanced character animation techniques for the current and future generations of video games. The resulting simulator has proven to be stable, require minimal user tuning, and achieve real-time performance even for moderately complex characters. It supports common joint types in arbitrary skeleton hierarchies and offers an intuitive motion control interface.

When starting to look into dynamics formulations for articulated bodies, one can easily get overwhelmed by the number of available choices. For reasons that will be explained in the next section, we chose Featherstone's Articulated Body Method (ABM) as a base for the simulator. The ABM is an established algorithm that efficiently evaluates the motion equations of arbitrary articulated skeletons. As the ABM does not provide any means to directly handle collisions and joint limits, and

because we did not want to resort to using penalty forces, we implemented a constraint solver to deal with them. This solver is invoked at every frame of the simulation and computes a set of forces and torques that, when applied to the body, will result in a motion that satisfies the given constraint conditions. The heart of the solver is an efficient algorithm derived from the ABM. This algorithm procedurally creates a linear system representing the constraint conditions which is subsequently solved to produce the constraint forces.

Although the intent of this presentation is to address most of the basic components of the simulator, a large emphasis will be on motion constraints and the algorithms that we developed to resolve them. The presentation will start with a brief overview and comparison of the available methods for generating the equations of motion of articulated bodies. It will then proceed to cover the basic concepts of Featherstone's ABM, the method used by this simulator. Building on the ABM algorithm, the presentation will continue with an in-depth description of the constraint solver and explain how to resolve multiple simultaneous constraints on the character motion. The algorithms will be presented from a practical rather than a theoretical standpoint and pseudocode will be provided to facilitate their implementation by other developers. After the general constraint framework is described, details of how the impacts, contacts, joint limits, and motion control can be expressed as motion constraints will be presented. The presentation will close with a brief discussion on the remaining challenges in using physics for character simulation.

2 Background

2.1 Equations of Motion

An articulated body consists of rigid links connected by joints. In the absence of joints, each link would have six degrees of freedom (DOFs), three translational and three rotational. However, since joints restrict the relative motion of the links they connect, the total number of DOFs of an articulated body with m links is less than $6m$. The dynamic equations of motion of an articulated body describe the link accelerations with respect to the body's current state and the forces acting on it. These equations appear in the literature in a number of different forms. However, depending on how the connections between the links are treated, the methods for deriving them can broadly be categorized in one of two classes.

Maximal coordinate methods treat each link as a separate rigid body and use explicit constraints to remove the extraneous DOFs. An articulated body with m links and $n < 6m$ DOFs, uses $6m$ state variables and requires $6m - n$ constraint equations to represent the joints. These methods have become popular with computer graphics practitioners as they are a direct extension of well understood and documented rigid body techniques [Baraff et al. 1999; Hecker 1998; Eberly 2004]. The additional issues regarding the explicit joint constraints are covered in detail in [Shabana 1994; Baraff 1996]. An interesting variation of the standard maximal coordinates methods appears in [Jacobsen 2001] where constrained particles are used instead of rigid bodies to represent the body's links.

Reduced coordinate methods implicitly incorporate the joint constraints into the formulation of the equations of motion, and therefore use the n joint angles directly as state variables. The equations of motion for a specific articulated body can be derived in advance by hand (a tedious process for anything but the simplest hierarchies), or with the help of specialised software like [Hollars et al. 1991]. Alternatively, an appropriate algorithm like Featherstone's ABM [Featherstone 1987] can be used to evaluate them at run-time.

In terms of performance, there are implementations of both maximal and reduced coordinate methods that can run in $O(n)$ time and there has not been conclusive evidence that one method consistently outperforms the others. In practice, there are other factors to be considered when making a choice on what method to use for a character physics system. Maximal coordinate methods are credited with being modular and easy to understand and implement. However, their main drawback is that they operate in Cartesian space, dealing with link positions and orientations instead of the more intuitive joint space of reduced coordinates that deals directly with the joint angles. As a result, common tasks such as evaluating joint angles and velocities, enforcing joint limits, and even applying internal torques to achieve a specific motion require awkward conversions between the two spaces. Additionally, inaccuracy in numerical integration can cause links to drift apart leaving the articulated body in an invalid state. While maximal coordinate methods might be adequate for simple rag-doll type effects, we believe that they are less well suited for more complex character animation and especially motion control. For that reason we chose to use Featherstone's ABM an efficient, linear-time, reduced coordinate method that evaluates the equations of motion of arbitrary articulated bodies. The ABM supports the common joint types used by modelling packages, and if the short length of the pseudocode in section 3 is any indication, it is not very difficult to implement.

2.2 Constraints

The ABM algorithm simply computes the accelerations of the articulated body given its current state and any external forces and joint torques acting on it. Any additional restrictions on the motion such as those imposed by contacts and joint limits have to be dealt with separately through the application of appropriate forces and torques. Computing the magnitudes of the forces that simultaneously satisfy multiple constraint conditions is not trivial, especially for complex articulated bodies. By far the easiest to implement technique for constraining body motion is *penalty methods*. Penalty methods work by introducing restoring forces after the constraints are violated. These methods have been extensively used in computer graphics, especially for preventing interpenetration between bodies in contact. The restoring force approach is also the basis of Proportional Derivative (PD) controllers commonly used to generate joint torques for animated articulated characters. Unfortunately, these methods are not practical in real-time applications since they have a detrimental effect on the stability of the simulation unless either sophisticated (and computationally expensive) integration techniques are used or the integration timestep is significantly reduced.

A more appropriate solution is provided by a different class of techniques, often called *analytical methods*, that work by computing the exact magnitude of forces that will satisfy the constraints down to the acceleration level in every step of the simulation. These methods have proven to be accurate, require no parameter tuning by the user, and have the ability to support relatively large integration steps without sacrificing stability. A common application of analytical methods is in resolving explicit joint constraints resulting from the use of maximal coordinates [Baraff 1996], but they have also been used in the context of constrained rigid body motion [Witkin et al. 1990; Barzel and Barr 1988; Gleicher 1994]. One of the most successful uses of analytical methods in real-time simulation is in computing contact forces between colliding rigid bodies [Baraff 1994]. Analytical methods work by constructing a linear system of the form:

$$\mathbf{Af} + \mathbf{b} = \mathbf{a}, \tag{1}$$

whose size is equal to the number of constraint conditions. An appropriate solver (as we will see later, the system might not be strictly a system of equations), solves for the magnitudes $\mathbf{f}$ of the constraint forces that are subsequently applied to the body. The matrix $\mathbf{A}$ and vectors $\mathbf{b}$ and $\mathbf{a}$ depend on the current state of the body and the constraint conditions, and need to be evaluated at every step of the simulation. Unfortunately, deriving exact expressions for the elements of $\mathbf{A}$ and $\mathbf{b}$ is not trivial, especially when complex articulated bodies are involved. This problem was solved with an easy-to-implement procedural algorithm presented first in [Kokkevis and Metaxas 1998] and repeated here in section 4.2. The algorithm uses test forces and repeated evaluations of the ABM equations to compute the elements of the system. A further performance improvement was achieved with a new algorithm, derived directly from Featherstone's recursive equations and presented in section 4.3. These two algorithms allowed us to take advantage of all the previous work done on analytic constraints techniques, including the fast contact force determination method of [Baraff 1994], and use them in conjunction with ABM.

3 Featherstone's Articulated Body Method

Character skeletons in commercial animation packages consist of multiple-DOF joints and the bones are arranged in a tree-like hierarchy with the root bone free to move in space. In contrast, the articulated body model in Featherstone's original algorithm is more restrictive and consists of n rigid links connected by n single-DOF joints in a branch-free chain. Links and joints are numbered from 1 to n such that joint h connects link h to its parent link $h-1$. Additionally, joint 1 connects link 1 to an immobile base. Fortunately, there is a straightforward technique for converting between the two representations. A general k-DOF character joint can be converted to an equivalent series of k links connected by single-DOF joints with the first $k-1$ links having zero length. The free moving root of the original character can be modeled by adding six zero-length links at the top of the articulated body chain, connected via three prismatic and three revolute joints. Alternatively, a free rigid body can be substituted for the root link. The equations appearing in this section depend on a consistent numbering scheme for the links: if link i is the parent of link j, then i will be less than j.

In the remainder of the paper, the term *articulated body* refers to a branched linked structure compliant with Featherstone's representation as described above, and the term *joint* refers to a single-DOF joint, which could be either prismatic or more commonly revolute.

The state of an n-DOF articulated body at any time instant is fully described by its joint variables q_i and their first derivative $\dot{q}_i$. For the sake of simplicity, the joint variables of both revolute and prismatic joints will be referred to as joint angles even

thought prismatic joint variables represent translations. A single n-dimensional vector $\mathbf{q}$ is used to collectively represent all the joint angles; their velocities and accelerations will be denoted by $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$ respectively.

Featherstone's algorithm computes the accelerations $\ddot{\mathbf{q}}$ of an articulated body given the body's current state and any external forces $\mathbf{f}^E$ and joint torques G acting on it. The computation takes place in three distinct loops, two outbound, from the root link to the leaves, and one inbound, from the leaves to the root. Every one of the n links is visited once in each of the three loops which explains the algorithm's $O(n)$ performance.

The following procedure is based on the original work of Featherstone [Featherstone 1983; Featherstone 1987], extended to branched chains. It is included here both for completeness and because it is subsequently used in section 4.3 to derive the new constraint matrix evaluation algorithm described in this work. A brief introduction to the six-dimensional spatial algebra used by these equations is included in Appendix A. A ground up derivation of the ABM algorithm that does not require a strong background in mechanics can be found in [Mirtich 1996].

```
1  proc ABM_Accelerations(q, q̇, F^E, G)
2      v̂_0 = 0̂
3      (* First outbound loop *)
4      for i = 1 to n
5          j ← index of parent of link i
6          v̂_i = _iX̂_j v̂_j + ŝ_i q̇_i
7          p̂_i^v = v̂_i ×̂ Î_i v̂_i − f̂_i^E
8          ĉ_i = v̂_i ×̂ ŝ_i q̇_i
9      (* Inbound loop *)
10     for i = n to 1
11         Î_i^A = Î_i
12         p̂_i = p̂_i^v
13         for j = 1 to numChildren(i)
14             k ← index of j^th child of i
15             Î_i^A += _iX̂_k (Î_k^A − (ĥ_k ĥ_k') / d_k) _kX̂_i
16             p̂_i += _iX̂_k (p̂_k + Î_k^A ĉ_k + (u_k / d_k) ĥ_k)
17         ĥ_i = Î_i^A ŝ_i
18         d_i = ŝ_i' ĥ_i
19         u_i = G_i − ĥ_i' ĉ_i − ŝ_i' p̂_i
20     (* Second outbound loop *)
21     â_0 = 0̂
22     for i = 1 to n
23         j ← index of parent of link i
24         q̈_i = (u_i − ĥ_i' _iX̂_j â_j) / d_i
25         â_i = _iX̂_j â_j + ĉ_i + ŝ_i q̈_i
```

In the above procedure, every link i has an associated spatial velocity, $\hat{\mathbf{v}}_i$, spatial acceleration, $\hat{\mathbf{a}}_i$, and joint acceleration, $\ddot{q}_i$. The articulated body inertia that represents the relationship between a force applied to the link and the resulting acceleration is denoted by $\hat{\mathbf{I}}_i^A$. The spatial transformation from the coordinate system of link j to the coordinate system of link i is represented by ${}_i\hat{\mathbf{X}}_j$. Finally, the variables $\hat{\mathbf{c}}_i$, $\hat{\mathbf{h}}_i$, d_i and u_i help eliminate repetitive evaluations of common subexpressions.

From the accelerations $\ddot{q}_i$ computed by this procedure, a *numerical integrator*, can evaluate the body's motion over time given a start state and the external forces and joint torques acting on the body. While there are many choices of integrators go, we have found that our constraint force evaluation algorithm avoids large spikes in the applied forces and as a result a simple Euler integrator with appropriate sub-frame stepping works well in most cases.

4 Acceleration Constraints

Most interesting physics based animations of articulated characters involve interaction with the environment. Characters are also expected to observe the same biomechanical limits in their range of motion as their real-life counterparts. The feet of the character cannot go below the ground, arms cannot swing through the torso, and elbows and knees cannot bend backwards. All these conditions impose certain natural constraints on the motion which should be enforced. In the context of computer animation, the character's limbs are often required to follow a certain motion path commonly described as a series of joint angles over time. A walk cycle, a jump, a punch are examples of such motions that impose additional constraints on the character.

This section deals specifically with the problem of converting a set of such constraints into a linear system similar to that in equation 1 that can be solved to obtain the necessary constraint forces. For simplicity, in the remainder of the paper we will use the term *generalized forces* to refer to both the external forces and internal joint torques that are necessary to resolve the various constraint conditions. Two algorithms will be presented that procedurally fill in the entries of matrix $\mathbf{A}$ and vector $\mathbf{b}$. The first algorithm will make direct use of the ABM procedure detailed in section 3, while the second one will use an optimized variant of that procedure.

Common constraints in articulated body motion can be expressed in terms of the body's accelerations. For example, a point on the body whose velocity is currently zero can be prevented from moving by setting its acceleration to zero. In dynamics simulation, at any time instant the state of an articulated body is fully described by $\mathbf{q}$ and $\dot{\mathbf{q}}$, the joint angle and velocity vectors respectively. Because time is advanced in discrete steps in numerical integration, at a given instant, any force acting on the body affects only the joint accelerations $\ddot{\mathbf{q}}$. Joint angles and velocities will change only after the integrator is invoked to compute the state at the next instant. This property of numerical integration is explored by the technique described in this work to compute a set of constraint forces at every time instant that, when applied to the articulated body, will satisfy a set of acceleration constraints.

It can be shown that, at a given time instant, there is a linear relationship between the magnitude of a generalized force applied to the body and the magnitude of the acceleration of the body's joints or links. The proof is in Appendix A. This relationship can be described as follows:

$$a^f = kf + a^0, \tag{2}$$

where a^0 and a^f are the observed acceleration magnitude before and after the force is applied, respectively; f is the magnitude of the force and k is some scalar constant. The observed acceleration, a, can represent the acceleration, $\ddot{q}$, of any of the body's joints or the magnitude of the linear acceleration, $\mathbf{a}_P$, of some point P on a link of the articulated body. By extension, for any linear function of the joint accelerations, $h(\ddot{\mathbf{q}})$, the following relationship holds for some k:

$$h(\ddot{\mathbf{q}}^f) - h(\ddot{\mathbf{q}}^0) = kf, \tag{3}$$

where $\ddot{\mathbf{q}}^0$ and $\ddot{\mathbf{q}}^f$ are the vectors of joint accelerations before and after the force is applied. The vector $\ddot{\mathbf{q}}^0$ will be referred to as the *default acceleration* vector and the difference $h(\ddot{\mathbf{q}}^f) - h(\ddot{\mathbf{q}}^0)$ as the *net effect* of f on the function h.

Most common useful acceleration constraints, such as setting the acceleration of a joint or a point on a link to some desired value, can be expressed as a linear constraint function $h(\ddot{\mathbf{q}}) = a - a^d$. The constraint will be satisfied with an appropriate joint acceleration vector, $\ddot{\mathbf{q}}^c$ such that $h(\ddot{\mathbf{q}}^c) = 0$.

4.1 Single Constraint

To make things a bit more concrete, imagine that we would like to set the acceleration of some arbitrary joint l of the articulated body to some desired value $\ddot{q}_l^d$. In other words, we need to find a torque g^c that when applied at joint l will cause the acceleration of l to become $\ddot{q}_l^c = \ddot{q}_l^d$. An easy way to compute the desired torque, g^c, using the results from the previous section and the ABM algorithm would be the following: Making a call to `ABM_Accelerations()`, compute the default acceleration of joint l, $\ddot{q}_l^0$. This acceleration reflects the current state of the body and all the forces acting on it at this particular time instant. Next, apply a test torque of some known (non-zero) magnitude g^t on joint l, call `ABM_Accelerations()` and record the new joint acceleration $\ddot{q}_l^t$. According to equation 2 the following relationship must hold:

$$\ddot{q}_l^t = kg^t + \ddot{q}_l^0, \tag{4}$$

from which we can compute k as:

$$k = \frac{\ddot{q}_l^t - \ddot{q}_l^0}{g^t}. \tag{5}$$

Additionally, since according to equation 2 the following relationship holds:

$$\ddot{q}_l^c = kg^c + \ddot{q}_l^0, \tag{6}$$

we can combine equations 4 and 6 to compute the desired constraint torque g^c from:

$$g^c = g^t \frac{\ddot{q}_l^c - \ddot{q}_l^0}{\ddot{q}_l^t - \ddot{q}_l^0}. \tag{7}$$

The constraint expression $h(\ddot{\mathbf{q}})$ in this simple example would have been $h(\ddot{\mathbf{q}}) = \ddot{q}_l - \ddot{q}_l^d$. In general, for linear function $h(\ddot{\mathbf{q}})$, the constant k and the generalized constraint force f can be computed as follows [Kokkevis and Metaxas 1998]:

1. Before applying any force to the articulated body, compute the default accelerations $\ddot{\mathbf{q}}^0$ and the value of $h(\ddot{\mathbf{q}}^0)$.
2. By applying a test force of some known (non-zero) magnitude f^t, compute the resulting accelerations $\ddot{\mathbf{q}}^t$ and evaluate $h(\ddot{\mathbf{q}}^t)$.
3. Using (3) compute k as:

$$k = \frac{1}{f^t}(h(\ddot{\mathbf{q}}^t) - h(\ddot{\mathbf{q}}^0)). \tag{8}$$

The main advantage of this technique is that it reuses the ABM algorithm that you presumably already have, and therefore requires very little additional coding. As a matter of fact, any algorithm that evaluates the body's accelerations could be used instead of ABM. A call to the ABM procedure takes $O(n)$ time for an n-DOF articulated body and even though two calls to `ABM_Accelerations()`, one for $\ddot{\mathbf{q}}^0$ and one for $\ddot{\mathbf{q}}^t$, are required, the overall asymptotic complexity is still $O(n)$.

Setting the acceleration of a point on a link of the articulated body to some desired value is achieved in the same way. The goal when solving for such a constraint is to compute an appropriate external force, $\mathbf{f}^c$, which, when applied to a point P on the body, will make the acceleration $\mathbf{a}_P$ of P equal to some desired value $\mathbf{a}_P^d$. Since constraint expressions are scalar functions, a single constraint can only set the acceleration along one direction in space. Notice how equation (32) demonstrates that, at a given time instant, the acceleration of any point on the articulated body can be written as a linear function of the joint accelerations. Therefore a linear function $h(\ddot{\mathbf{q}}) = a_P(\ddot{\mathbf{q}}) - a_P^d$ can be defined, with $a_P(\ddot{\mathbf{q}})$ and a_P^d the magnitudes of the actual and the desired acceleration of P along some direction vector $\vec{\imath}$. The test force must be acting along the same direction to guarantee a non-zero net effect on the acceleration magnitude. As a result, the computed constraint force will also be along the direction of $\vec{\imath}$.

4.2 Multiple Constraints

So far the technique described computes a single constraint force to satisfy one scalar acceleration constraint. If multiple constraints are to be enforced simultaneously, the problem becomes harder to solve. A force applied on the body to help enforce one constraint will potentially alter the acceleration of multiple joints and therefore affect other constraints as well. The constraint equations must be solved simultaneously to achieve the desired results.

Consider m constraints, each with a corresponding scalar constraint function $h_i(\ddot{\mathbf{q}})$ such that $h_i(\ddot{\mathbf{q}}) = 0$ when constraint i is satisfied. The goal is to find an appropriate set of m forces, f_i^c, that will satisfy all the constraints simultaneously. According to equation 3, the net effect of a force f_i on a constraint expression $h_j(\ddot{\mathbf{q}})$ will be equal to $k_{ij} f_i$ for some constant k_{ij}. Adding up the effect of all the forces for each constraint results in the following system of equations:

$$\begin{aligned} h_1(\ddot{\mathbf{q}}) - h_1(\ddot{\mathbf{q}}^0) &= k_{11} f_1 + k_{12} f_2 + \cdots + k_{1m} f_m \\ h_2(\ddot{\mathbf{q}}) - h_2(\ddot{\mathbf{q}}^0) &= k_{21} f_1 + k_{22} f_2 + \cdots + k_{2m} f_m \\ &\vdots \\ h_m(\ddot{\mathbf{q}}) - h_m(\ddot{\mathbf{q}}^0) &= k_{m1} f_1 + k_{m2} f_2 + \cdots + k_{mm} f_m \end{aligned} \tag{9}$$

In matrix form, the system can be written as:

$$\mathbf{h} - \mathbf{h}^0 = \mathbf{K}\mathbf{f}, \tag{10}$$

where $\mathbf{h}$ and $\mathbf{h}^0$ are the $m \times 1$ vectors of $h_i(\ddot{\mathbf{q}})$ and $h_i(\ddot{\mathbf{q}}^0)$ respectively, $\mathbf{K}$ is the $m \times m$ constraint system matrix consisting of the elements k_{ij}, and $\mathbf{f}$ is the $m \times 1$ vector of forces f_i. Setting $\mathbf{h} = \mathbf{0}$ and solving for the force magnitudes $\mathbf{f}$ computes the constraint forces that will simultaneously satisfy all the acceleration constraints.

The elements k_{ij} of $\mathbf{K}$ can be computed using test forces with a process similar to the one described in the single constraint case:

1. Before applying any additional forces on the articulated body the default joint accelerations, $\ddot{\mathbf{q}}^0$, and the value of constraint expressions $h_i(\ddot{\mathbf{q}}^0)$ are computed.

2. For each constraint i, apply an appropriate test force of known magnitude f_i^t and compute the resulting joint accelerations, $\ddot{\mathbf{q}}_i^t$. Using these accelerations, compute the values of each constraint expression $h_j(\ddot{\mathbf{q}}_i^t)$. The value of k_{ji} for $1 \leq j \leq m$ will be:

$$k_{ji} = \frac{1}{f_i^t}(h_j(\ddot{\mathbf{q}}_i^t) - h_j(\ddot{\mathbf{q}}^0)). \tag{11}$$

It is worth noticing that the constraint system in equation 10 is equivalent to the canonical system of equation 1. Matrix $\mathbf{A}$ corresponds to the constraint matrix $\mathbf{K}$ and $\mathbf{a}$ and $\mathbf{b}$ correspond to $\mathbf{h}$ and $\mathbf{h}^0$ respectively.

Using Featherstone's algorithm, the above process computes all the elements of $\mathbf{K}$ with $m+1$ calls to `ABM_Accelerations()` for a total cost of $O(nm+m^2)$. While details on how to solve the system will be given in a later section, linear system solving is generally considered to be an $O(m^3)$ process, making the overall cost of computing the m constraint forces equal to $O(nm+m^3)$. In practice, the m^3 term does not dominate the performance of this method since m is typically a lot smaller than n. Additionally, since the $m+1$ calculations of accelerations are independent, they can be computed in parallel.

4.3 An Optimized Algorithm

The algorithm described in the previous section computes the elements of the constraint system matrix $\mathbf{K}$ by repeatedly evaluating the articulated body's accelerations with different applied forces. The performance of the algorithm depends on the efficiency of the underlying solver used to compute these accelerations. Even though Featherstone's ABM computes the body accelerations in linear time, repeated calls to `ABM_Accelerations()` result in unnecessary computation that can be avoided. This section will provide an alternative algorithm derived from the original equations of motions that directly computes the net effect of each applied force on the joint accelerations. Although the asymptotic complexity of this new algorithm is also $O(n)$ for each applied force, the constant involved is significantly smaller.

Since the constraint expressions $h(\ddot{\mathbf{q}})$ are linear functions of the joint accelerations $\ddot{\mathbf{q}}$, the following relationship is true:

$$h(\ddot{\mathbf{q}}^t) - h(\ddot{\mathbf{q}}^0) = h(\ddot{\mathbf{q}}^t - \ddot{\mathbf{q}}^0), \tag{12}$$

for any joint acceleration vector $\ddot{\mathbf{q}}^t$. In other words, in order to compute the net effect of a test force $\mathbf{f}^t$ on a given constraint it suffices to compute the net effect of the force on the joint accelerations, ${}^d\ddot{\mathbf{q}}^t = \ddot{\mathbf{q}}^t - \ddot{\mathbf{q}}^0$, and evaluate the constraint expression, h, using ${}^d\ddot{\mathbf{q}}^t$.

A closer look at the equations in `ABM_Accelerations()` reveals which quantities are affected by the application of an additional test force $\mathbf{f}_l^t$ on some link l, and how the effect propagates up and down the link chains. Since the calculation is performed at a given state of the articulated body, all the joint velocities, $\dot{q}$, and transformations, $\hat{\mathbf{X}}$, can be treated as constant. Additionally, any quantity that depends only on the state will also remain unchanged including the link velocities, $\hat{\mathbf{v}}$, the articulated body inertias, $\hat{\mathbf{I}}^A$, and the common subexpressions, $\hat{\mathbf{c}}$, $\hat{\mathbf{h}}$ and d. The additional force $\mathbf{f}_l^t$ will directly affect $\hat{\mathbf{p}}_l^v$. The effect will propagate via the inbound loop to the values of $\hat{\mathbf{p}}$ and u for all the ancestors of l. With the second outbound loop it will spread to the accelerations $\ddot{q}$ and $\hat{\mathbf{a}}$ of all the joints and links respectively. The complete procedure can be written as:

```
1  proc ABM_AccelerationDeltas( l , f̂_l^t )
2      for i = l + 1 to n
3          ^d p̂_i^v = ^d p̂_i = 0̂
4          ^d u_i = 0
5      (* Inbound loop *)
6      for i = l to 1
7          if (i == l)
8              ^d p̂_i^v = −f̂_l^t
9              ^d p̂_i = ^d p̂_i^v
10         else
11             ^d p̂_i^v = 0̂
12             ^d p̂_i = 0̂
13         for j = 1 to numChildren(i)
14             k ← index of j^th child of i
15             ^d p̂_i += _i X̂_k ( ^d p̂_k + (^d u_k / d_k) ĥ_k )
16             ^d u_i += −ŝ_i' ^d p̂_i
17     (* Outbound loop *)
18     ^d q̈_0 = 0
19     for i = 1 to n
20         j ← index of parent of link i
21         ^d q̈_i = ( ^d u_i − ĥ_i' _i X̂_j ^d â_j ) / d_i
22         ^d â_i = _i X̂_j ^d â_j + ŝ_i ^d q̈_i
```

In the above procedure, $\hat{\mathbf{f}}_l^t$ is the spatial equivalent of $\mathbf{f}_l^t$ expressed in the local coordinate frame of link l, and ${}^d\ddot{q}_i$ is the net effect of $\mathbf{f}_l^t$ on the acceleration of joint i which is equal to $\ddot{q}_i^t - \ddot{q}_i^0$.

For joint acceleration constraints, the code has to be slightly modified to compute the net effect of a test torque g_l^t on joint l, lines 7 through 12 are replaced by:

```
^d p̂_i^v = ^d p̂_i = 0̂
```

and:

```
if (i == l)
    ^d u_i += g_l^t
```

is inserted inside the inbound loop after line 16.

The process for computing entries of the constraint matrix $\mathbf{K}$ presented in section 4.2 now becomes:

1. Use `ABM_Accelerations()` to compute the default accelerations $\ddot{\mathbf{q}}^0$ and the values of the constraint expressions $h_i(\ddot{\mathbf{q}}^0)$.
2. For each constraint i, use an appropriate test force $\hat{\mathbf{f}}_l^t$ of known magnitude $\mathbf{f}^t$, call `ABM_AccelerationDeltas()` to obtain ${}^d\ddot{\mathbf{q}}_i^t$ and compute the values of all constraint expressions, $h_j({}^d\ddot{\mathbf{q}}_i^t)$. Evaluate k_{ji} for $1 \leq j \leq m$ as:

$$k_{ji} = \frac{1}{f_i^t} h_j({}^d\ddot{\mathbf{q}}_i^t) \tag{13}$$

For a constraint system of size m, step 2 requires m calls to `ABM_AccelerationDeltas()`. Notice how these calls use the values of d and $\hat{\mathbf{h}}$ computed by `ABM_Accelerations()` in step 1.

`ABM_AccelerationDeltas()` is an $O(n)$ algorithm since it makes two linear passes through the n links of the body. Therefore, the asymptotic complexity of the process outlined above to compute the elements of $\mathbf{K}$ is still $O(nm + m^3)$. However, this new method uses a significantly smaller number of math operations and therefore offers an important practical performance advantage. The exact number of operations required by `ABM_AccelerationDeltas()` depends on where the link l is in the

joint hierarchy. Even in the worst case, where l is a leaf link, the algorithm requires three matrix-vector multiplications, and three vector-vector additions or subtractions per link. In comparison, `ABM_Accelerations()` requires nine matrix-vector multiplications, two matrix-matrix multiplications and ten vector-vector additions or subtractions. Code profiling shows that one call to `ABM_AccelerationDeltas()` is in general about three times faster than one call to `ABM_Accelerations()`.

In all fairness, if `ABM_Accelerations()` is implemented carefully, it can be optimized for the m repeated calls required by the algorithm of section 4.2. Link quantities that depend only on the state of the body can be calculated once and reused in subsequent calls. In this case the performance advantage of the method described in this section is smaller, in the order of fifty to sixty percent, but still significant.

5 Impact Constraints

The acceleration constraint methods discussed so far dealt with enforcing specific accelerations to various parts of the articulated body. This section describes a different type of constraints, the impact constraints, that use impulses to instantaneously change the body's velocities. It turns out that the constraint solvers for acceleration constraints can be used for impacts as well. Impact constraints deal with instantaneous changes of the articulated body velocities and work through impulse forces. During the course of a simulation, sudden velocity changes are required when links collide with other objects or when joints reach their limit angles. Simulating the effects of such impacts as an instantaneous change in the joint velocities of the articulated body that happens between integration steps improves the stability of the simulation. It avoids the use of unreasonably large forces over the course of the simulation to maintain physical consistency, which helps the integration safely proceed at a faster pace.

When solving for impact constraints, the magnitude of the impulse forces that satisfy the constraints is not itself of particular interest. Rather, what needs to be determined is the effect of the impulse on the body velocities as it propagates across the joints.

As an example, consider that during the course of a simulation, because of an impact, the magnitude of the velocity of a certain point Q on the body needs to change instantaneously from its current value, v_Q^-, to some known value, v_Q^+. The velocity change is the result of an impulse force acting on Q which will affect the velocities of the body, changing them from $\dot{\mathbf{q}}^-$ right before to $\dot{\mathbf{q}}^+$ right after the impact. The goal is to compute $\dot{\mathbf{q}}^+$ in order to update the state of the body with the new velocities and continue with the simulation. The impulse force can be considered as a large force acting over an infinitesimally short length of time δt. During that time, the magnitude of the acceleration of Q will be:

$$a_Q^I = (v_Q^+ - v_Q^-)/\delta t. \tag{14}$$

Similarly, the joint accelerations will be $\ddot{\mathbf{q}}^I = (\dot{\mathbf{q}}^+ - \dot{\mathbf{q}}^-)/\delta t$ or:

$$\dot{\mathbf{q}}^+ = \dot{\mathbf{q}}^- + \ddot{\mathbf{q}}^I \delta t. \tag{15}$$

Without loss of generality, δt can be set to 1 and therefore $a_Q^I = (v_Q^+ - v_Q^-)$. Using a_Q^I as the desired acceleration of Q and solving an acceleration constraint problem with $h(\ddot{\mathbf{q}}) = a_Q(\ddot{\mathbf{q}}) - a_Q^I$, the appropriate constraint force magnitude f_Q that will enforce $a_Q = a_Q^I$ can be computed with the method described in section 4.1. The net effect of f_Q on the joint accelerations can be computed by evaluating the default accelerations, $\ddot{\mathbf{q}}^0$, and the accelerations, $\ddot{\mathbf{q}}^Q$, after f_Q is applied, and subtracting the two. The joint velocities after the impact, $\dot{\mathbf{q}}^+$, are evaluated from equation (15) using $\ddot{\mathbf{q}}^I = \ddot{\mathbf{q}}^Q - \ddot{\mathbf{q}}^0$. Notice that due to way joint velocities are computed, the result is independent of the value chosen for δt.

This technique can be extended to simulate multiple simultaneous impacts and compute their effect on the joint velocities of the articulated body. Given the velocity change at m impact points, a system of multiple acceleration constraints can be formed using the values derived from equation (14) as desired accelerations. Once the corresponding constraint forces are computed, they can be applied to the articulated body to evaluate their net effect on the joint accelerations. Using that result, the post-impact joint velocities are computed from equation (15). Clearly, the complexity of this method is identical to the complexity of the acceleration constraint solver and therefore the cost of resolving m simultaneous impacts is also $O(mn + m^3)$.

6 Unilateral Constraints

There exists an important class of constraint problems in dynamic simulation that impose additional restrictions on the direction of the allowable constraint force and the resulting acceleration. Contact forces between bodies are a typical source of such

constraints. At the point of contact, P, the relative acceleration of the bodies along the contact normal should be either zero, in which case bodies remain in contact, or positive, where bodies separate. Additionally, when the acceleration is positive the force acting on each body has to be zero, while when the acceleration is zero the force has to be either zero or of repulsive nature. These conditions are satisfied by a *unilateral constraint* that is described by the following expressions:

$$a_P \geq 0 \ , \ f_P^c \geq 0 \ , \text{ and } a_P f_P^c = 0. \tag{16}$$

For rigid body simulations [Baraff 1994] presented a practical and efficient algorithm for solving the unilateral constraints resulting from rigid body contacts using a Linear Complementarity Problem (LCP) solver.

An LCP solver takes a linear system in the familiar form,

$$\mathbf{Mz} + \mathbf{q} = \mathbf{w}, \tag{17}$$

and solves for $\mathbf{z}$ such that for every i that corresponds to a unilateral constraint: $z_i \geq 0$, $w_i \geq 0$ and $z_i w_i = 0$. These conditions are often written as:

$$z_i \geq 0 \qquad \text{complementary to} \qquad w_i \geq 0. \tag{18}$$

With appropriate bookkeeping, unilateral and bilateral constraints can be mixed in the same system of equations.

The exact same technique can be applied to unilateral articulated body constraints using the algorithms from the previous section to compute the matrix $\mathbf{M}$ (our matrix $\mathbf{K}$) and vectors $\mathbf{z}$ ($\mathbf{h}^0$) and $\mathbf{w}$ ($\mathbf{h}$). It is worth mentioning that in articulated body simulation another common source of unilateral constraints besides contacts is joint limits.

Algorithms that solve LCPs can be broadly classified into two categories: *pivoting* methods and *iterative* methods. Pivoting methods use a finite number of steps and require the recursive solution of systems of linear equations. In contrast, iterative methods do not terminate finitely but rather converge to the solution. Dantzig's algorithm used in Baraff's paper and Lemke's algorithm described in [Eberly 2004] and by [Hecker 2004] are common examples of pivoting methods. Pivoting methods can be fast, but it has been our experience that for larger constraint systems they tend to suffer from round-off errors that cause them to produce incorrect results or find no results in perfectly solvable problems.

In our simulator, we have chosen to use a Projected Gauss-Seidel iterative solver, which works admirably well. Iterative LCP solvers come closer to a solution with every iteration, meaning that even if they are interrupted early, an intermediate result can be good enough for the simulation to continue. In other words, to a certain extent, one can trade off speed of execution for accuracy with these methods, which is a very valuable feature for real-time simulations. There are two additional advantages to iterative solvers: They can gracefully deal with singular systems (which are common in simulation in the presence of friction or contradicting user constraints), and they are very easy to implement. A complete coverage of the LCP and the associated algorithms for solving it can be found in [Cottle et al. 1992].

7 The simulation step

The solvers for the acceleration and the impact constraints are invoked at different stages of the simulation step. The following procedure outlines the order of operations in a typical simulation step which advances the state of an articulated body from time t to time $t+h$.

```
 1 proc Step(q_t, q̇_t, C, h)
 2     while (h > 0)
 3         f^c ← ComputeConstraintForces(C)
 4         q̈_t ← ComputeAccelerations(q_t, q̇_t, f^c)
 5         hTry = h ; repeat = true
 6         while (repeat)
 7             [ q_new, q̇_new ] ← Integrate(q_t, q̇_t, q̈_t, hTry)
 8             [ C, needImpact, needBacktrack ] ← UpdateConstraintSet(q_new, q̇_new)
 9             if (needBacktrack)
10                 hTry = hTry * 0.5
11                 continue
12             if (needImpact)
13                 q̇_new ← ResolveImpacts(C)
14             h = h - hTry ; repeat = false
15             [ q_t, q̇_t ] ← [ q_new, q̇_new ]
16     return (q_t, q̇_t, C)
```

Each step starts by resolving the current acceleration constraints (from the elements of constraint set **C**), and evaluating the appropriate constraint forces using the algorithm in section 4. The constraint forces are supplied to the ABM algorithm in line 4 which in turn evaluates the equations of motion and computes the joint accelerations. In line 7, an attempt is made to take a step forward using the current state and the accelerations from line 4. After the new, tentative, state is computed, `UpdateConstraintSet()` is called to update the elements of **C** by adding any new constraints (such as new contacts) and removing ones that are no longer valid. Important events like collisions need to be detected early enough to avoid interpenetrations. If the integration step just taken was too large and objects have interpenetrated already the *needBacktrack* flag is set to signal that a smaller step must be attempted (the code above halves the stepsize but more sophisticated step prediction can be used as well). Certain events like new collisions might require a velocity adjustment which is taken care of in line 13 using the impact constraints described in section 5. If the last integration step was successful and no backtracking was required, the current state is updated (line 15). The code in lines 5 to 15 is executed repeatedly until the state of the body is advanced by the whole time interval h. Notice that the constraint set **C** at the end of each step can be used as the starting constraint set for the following step.

8 Constraint Examples

This section describes three examples of practical applications of the acceleration, impact, and unilateral constraints in the simulation of articulated body dynamics.

8.1 Collision Response

Our simulator relies on a collision detection module that uses simple geometric primitives (boxes, spheres and capsules) directly parented under the link transforms of the articulated body to determine whether body parts come in contact with the rest of the environment. The collision detection results appear as a list of discrete contact point pairs. Each contact point pair has an associated normal direction which defines the plane of contact. A relative velocity along the contact normal is defined such that a positive sign signifies that the contact points are moving away from each other and a negative sign that they are moving towards each other.

During each step of the simulation, the list of current contact points is updated within `UpdateConstraintSet()`. Each contact point corresponds to one or more constraint conditions in the constraint set **C**. Contact constraints are dealt with in two

separate parts of the simulation step. When the relative normal velocity at a contact point is negative, the *needImpact* flag is set in order to trigger an impact constraint resolution (line 13) for all the current constraints. After impact, the relative velocity at each contact point will either be zero (*resting contact*) or positive (contact points will tend to separate). In either case, an acceleration constraint resolved by `ComputeConstraintForces()` in line 3 will prevent interpenetration.

In the presence of friction, the force at each contact point can be broken into two components: The *normal* component that acts in a direction perpendicular to the contact plane and prevents interpenetration, and the *tangent* component that corresponds to the friction force which opposes any relative motion on the contact plane. The magnitude of both force components is computed by resolving three constraint conditions. The first condition deals with the acceleration of the point along the contact normal, and the second with the acceleration on the contact plane. The third condition prevents the ratio of the tangent force to the normal force from exceeding the predefined friction coefficient constant. All three conditions are expressed as unilateral constraints in the following way:

$$a_n - a_n^d \geq 0 \quad \text{complementary to} \quad f_n \geq 0 \tag{19}$$

$$(a_t - a_t^d) + \lambda \geq 0 \quad \text{complementary to} \quad f_t \geq 0 \tag{20}$$

$$\mu f_n - f_t \geq 0 \quad \text{complementary to} \quad \lambda \geq 0 \tag{21}$$

In the above constraint expressions, a_n, a_t and a_n^d, a_t^d are the actual and desired acceleration magnitudes along the contact normal and tangent respectively, f_n and f_t are the magnitudes of the normal and tangent force components, and μ is the friction coefficient. λ is an additional constraint variable that guarantees that the friction force will do its best to satisfy the tangent acceleration condition. In other words, f_t can be less than μf_n only when $a_t = a_t^d$. The LCP solver will determine values for f_n, f_t and λ, although the actual value of λ is of no practical use.

Friction is classified as *dynamic* or *static* depending on whether there is sliding at the contact or not. In the case of dynamic friction (objects are sliding), the tangent force acts in direction opposite to the direction of the relative tangent velocity. In the case of static friction, the direction of the tangent force cannot be determined beforehand and the number of required constraint conditions grows from three to six. To avoid such bloating of the resulting constraint system one can pick a random tangent direction for new contacts and the last valid tangent for existing contacts that have stopped sliding. This trick works fairly well in our experience.

When a new contact is detected, an impact constraint might be required to instantaneously change the normal velocity at the contact point to prevent interpenetration at the next simulation step. Assuming a simple elastic collision model, the normal velocity after the collision, v_n^+, can be written in terms of the coefficient of elasticity, e, and the normal velocity before the collision, v_n^-, as: $v_n^+ = -ev_n^-$. The three impact constraint conditions for this point will use $a_n^d = v_n^+ - v_n^-$ and $a_t^d = -v_t^-$. As described in section 5, impact resolution will cause an instantaneous velocity change that will set the normal velocity at this contact point to v_n^+ and the tangent velocity to a value between 0 and v_t^-, depending on how much friction force can be applied.

For resting contact, the desired normal acceleration, a_n^d is set to 0 to prevent movement along the contact normal. The friction component of the force should attempt to cancel any relative velocity on the contact plane within the next timestep, h. If the current tangent velocity is v_t^- then a desired acceleration $a_t^d = -v_t^-/h$ will achieve just that. Of course, the friction force is limited by the magnitude of the normal force and therefore the tangent velocity might not actually reach zero.

8.2 Joint Limits

The process of dealing with joint limits is very similar to the process outlined above for dealing with collisions. Detecting joint limit violations is simply a matter of comparing the current joint angle against the preset limit angles. This process takes place within `UpdateConstraintSet()` and results in one constraint condition added to the set $\mathbf{C}$ for every joint reaching a limit. A joint velocity in a direction that violates the joint limit is neutralized using an impact constraint. Subsequently, an acceleration constraint that sets the desired joint acceleration to zero will prevent a joint resting at a limit angle from moving against the limit. Once again, unilateral constraints are required since the torque generated by solving the constraint system should not pull the joint towards its limit.

8.3 Motion control

Much like humans and animals that use muscles to move their limbs, digital articulated characters have to use internal torques to flex and extend their joints to perform a desired motion.

A desired motion trajectory described as a series of joint angles and velocities over time is typically the output of a high-level controller that is designed to make the articulated character perform a certain task. Acceleration constraints can be used to derive the appropriate torques that will make joints follow that desired trajectory.

If the target angle and velocity of a joint at a particular time instant are q^d and $\dot{q}^d$, then the following expression can be used to compute a value of joint acceleration $\ddot{q}^d$ necessary to move the joint towards its target:

$$\ddot{q}^d = K_p(q^d - q) + K_d(\dot{q}^d - \dot{q}), \tag{22}$$

where q and $\dot{q}$ are the current joint angle and velocity and K_p and K_d are two constants better known as the *controller gains*. An expression constraint using $\ddot{q}^d$ as the desired joint acceleration will compute the appropriate joint torque. The joint is part of a physical system and therefore has to follow a smooth motion; it cannot jump to the target angle instantaneously. The time it takes for the joint to reach the target and the path it follows to get there depend on the controller gains. A large value of K_p combined with a small value of K_d will result in an underdamped motion where the joint will overshoot and oscillate about its target. A large value of K_d will result in an overdamped motion that will delay reaching the target. For a critically damped motion that reaches within 10% of the target angle in t_s time one can set $K_p = 1/t_s^2$ and $K_d = \sqrt{2}/t_s$. Adjusting the values of K_p and K_d provides an interesting control knob for the user to get different looking motion behaviors from the same desired trajectory.

While there is a similarity between equation (22) and the Proportional Derivative (PD) control equation used extensively in the robotics and computer graphics literature, it is important to note that PD controllers affect the simulation by supplying a torque magnitude to the character's joint actuators. This torque will only attempt to move the joint towards its target but its success is determined by the current state and inertial properties of the system, and any internal or external forces acting on the joint. In contrast, acceleration constraint based controllers directly set the joint acceleration to a desired value, guaranteeing that the joint will reach its target in a predefined manner.

9 Conclusions

As the speed of the video game hardware increases and the algorithms get refined, we will hopefully start to see character physics playing a more prominent role in video games. There is a number of open issues that still need to be addressed especially regarding the integration of physics algorithms with the rest of the animation system in a game engine. Another area that needs improvement is the design of high-level dynamics controllers capable of making a character perform tasks autonomously. Even seemingly simple tasks such as walking down a straight line are prohibitively difficult to achieve robustly using only physics. Hopefully this work will inspire other developers to look closer into articulated body physics and help them develop their own simulators to experiment with. The end result might be a more rapid advancement of this technology and a more wide-spread adoption of physics based simulation in character animation.

Appendix

A Spatial notation

Spatial notation was introduced by Featherstone in [Featherstone 1987] as an elegant way to condense the equations describing a physical system. By essentially combining linear and angular quantities, two ordinary three-dimensional vectors are replaced by a single six-dimensional spatial vector. For example, the spatial velocity, $\hat{\mathbf{v}}$, of a rigid body is described in terms of the body's angular velocity, ω, and linear velocity, $\mathbf{v}$, as

$$\hat{\mathbf{v}} = \begin{bmatrix} \omega \\ \mathbf{v} \end{bmatrix} \equiv [\omega^T \mathbf{v}^T]^T.$$

Similarly, the spatial acceleration of a body can be written as

$$\hat{\mathbf{a}} = [\alpha^T \mathbf{a}^T]^T,$$

where α and $\mathbf{a}$ are the angular and linear acceleration of the body respectively. Along the same lines, the spatial representation of a force acting on a rigid body is

$$\hat{\mathbf{f}} = [\mathbf{f}^T \tau^T]^T,$$

where $\mathbf{f}$ is the three dimensional force vector and τ the corresponding three dimensional torque vector.

The spatial analog of an ordinary transformation matrix is the 6×6 spatial transformation matrix. If F and G are two coordinate frames, the spatial transformation matrix from F to G is given by

$${}_G\hat{\mathbf{X}}_F = \begin{bmatrix} \mathbf{R} & \mathbf{0} \\ \mathbf{r}\times\mathbf{R} & \mathbf{R} \end{bmatrix}.$$

In the above definition, matrix $\mathbf{R}$ is the 3×3 rotation matrix transforming vectors from F to G and vector $\mathbf{r}$ is the offset from the origin of F to the origin of G expressed in G's coordinates. Matrix $\mathbf{0}$ is the 3×3 zero matrix and $\mathbf{r}\times$ is the antisymmetric matrix corresponding to $\mathbf{r}$ defined as:

$$\mathbf{r} = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix}, \quad \mathbf{r}\times = \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}$$

Note that the product of matrix $\mathbf{r}\times$ with any vector $\mathbf{q}$ is equal to the vector cross product $\mathbf{r} \times \mathbf{q}$. The transformation of a spatial vector quantity from F to G is accomplished by the following spatial multiplication:

$$\hat{\mathbf{p}}_G = {}_G\hat{\mathbf{X}}_F \hat{\mathbf{p}}_F.$$

Like regular transformations, spatial transformations are combined by multiplying the associated matrices:

$${}_G\hat{\mathbf{X}}_K = {}_G\hat{\mathbf{X}}_F {}_F\hat{\mathbf{X}}_K$$

For efficiency purposes, spatial transformation matrices can be stored in a short form as a matrix-vector pair, $spx(\mathbf{R}, \mathbf{r})$. Using this form, the transformation of a spatial vector is computed as:

$$spx(\mathbf{R}, \mathbf{r}) \cdot \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix} = \begin{bmatrix} \mathbf{Ra} \\ \mathbf{r} \times (\mathbf{Ra}) + \mathbf{Rb} \end{bmatrix}.$$

Combining spatial transforms using their short form has a significant performance advantage over multiplying their corresponding 6×6 matrices. Using the following property:

$$(\mathbf{Rv})\times = \mathbf{R}\,\mathbf{v}\times\,\mathbf{R}^{-1},$$

where $\mathbf{R}$ is a 3×3 orthogonal matrix, spatial transform multiplication using the short form becomes:

$$spx(\mathbf{R_1}, \mathbf{r_1}) \cdot spx(\mathbf{R_2}, \mathbf{r_2}) = spx(\mathbf{R_1 R_2}, \mathbf{r_1} + \mathbf{R_1 r_2}).$$

The spatial inner product of two spatial vectors $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ is given by $\hat{\mathbf{x}}'\hat{\mathbf{y}}$ where $\hat{\mathbf{x}}'$ is the spatial transpose of $\hat{\mathbf{x}}$ defined as:

$$\hat{\mathbf{x}} = \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix}, \quad \hat{\mathbf{x}}' = [\mathbf{b}^T \mathbf{a}^T].$$

Finally, the spatial cross operator, $\hat{\times}$, is defined by:

$$\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix} \hat{\times} = \begin{bmatrix} \mathbf{a}\times & \mathbf{0} \\ \mathbf{b}\times & \mathbf{a}\times \end{bmatrix}.$$

B Force-Acceleration Relationship

Note: In order to prove the linear relationship between applied forces or torques and the resulting accelerations of an articulated body, it is more convenient to manipulate the body's dynamic equations in matrix form. Note however that the matrix form of the equations is equivalent to Featherstone's recursive form and therefore the results from this section apply to both.

The dynamic equations of motion of any articulated body with n DOFs can be written in matrix form as:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{V}(\mathbf{q},\dot{\mathbf{q}}) + \mathbf{G}(\mathbf{q}) = \mathbf{T}, \tag{23}$$

where $\mathbf{M}(\mathbf{q})$ is the $n \times n$ positive definite mass matrix, $\mathbf{V}(\mathbf{q},\dot{\mathbf{q}})$ is an $n \times 1$ vector of centrifugal and Coriolis terms, $\mathbf{G}(\mathbf{q})$ is an $n \times 1$ vector of gravity terms and $\mathbf{T}$ an $n \times 1$ vector of generalized forces and torques acting on the body [Craig 1986]. While each element of $\mathbf{M}(\mathbf{q})$, $\mathbf{V}(\mathbf{q},\dot{\mathbf{q}})$ and $\mathbf{G}(\mathbf{q})$ is a complex function of the state variables, at a given time instant the elements of $\mathbf{q}$ and $\dot{\mathbf{q}}$ are known and (23) can be simply written as:

$$\mathbf{M}\ddot{\mathbf{q}}^0 = \mathbf{F}, \tag{24}$$

where $\mathbf{F} = \mathbf{T} - \mathbf{V}(\mathbf{q},\dot{\mathbf{q}}) - \mathbf{G}(\mathbf{q})$ is an $n \times 1$ vector representing all the internal and external forces acting on the articulated body. Multiplying both sides of (24) by the inverse of $\vec{M}$ gives us a direct expression for the joint accelerations:

$$\ddot{\mathbf{q}}^0 = \mathbf{M}^{-1}\mathbf{F}. \tag{25}$$

Any additional generalized force τ applied to the articulated body at this time instant will affect only the joint accelerations $\ddot{\mathbf{q}}$ and (25) will take the form:

$$\ddot{\mathbf{q}} = \mathbf{M}^{-1}(\mathbf{F} + \tau). \tag{26}$$

In practice, the two ways of applying a generalized force on the articulated body are through joint torques or external forces acting on the links of the body. The generalized torque vector corresponding to any additional torque g_i applied to joint i of the articulated body is an $n \times 1$ vector τ with zeros for all its elements except for element i whose value is g_i:

$$\tau = [0 \ldots g_i \ldots 0]^T. \tag{27}$$

An external force acting at some point P on a link of the body can be converted from a cartesian vector, $\mathbf{f}$, to a generalized force vector, τ, using:

$$\tau = \mathbf{J}_P^T(\mathbf{q})\mathbf{f}, \tag{28}$$

where $\mathbf{J}_P(\mathbf{q})$ is the $3 \times n$ Jacobian matrix corresponding to P. Although the elements of the Jacobian matrix are a complex function of the joint angles $\mathbf{q}$, at the given time instant the Jacobian matrix for P can be considered to be a constant matrix $\mathbf{J}_P$. Equations (27) and (28) demonstrate that there is a linear relationship between the elements of τ and the magnitude of corresponding the torque or force τ is derived from.

Subtracting (25) from (26) provides an expression for ${}^d\ddot{\mathbf{q}}$, the *net effect* of generalized force τ on the joint accelerations:

$${}^d\ddot{\mathbf{q}} = \ddot{\mathbf{q}} - \ddot{\mathbf{q}}^0 = \mathbf{M}^{-1}\tau. \tag{29}$$

The velocity, $\mathbf{v}_Q$ of any point Q on a link of the articulated body can be written in terms of the Jacobian matrix, $\mathbf{J}_Q(\mathbf{q})$ of Q as:

$$\mathbf{v}_Q = \mathbf{J}_Q(\mathbf{q})\dot{\mathbf{q}}. \tag{30}$$

Differentiating the above equation with respect to time gives us the acceleration, $\mathbf{a}_Q$, of Q:

$$\mathbf{a}_Q = \dot{\mathbf{J}}_Q(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{J}_Q(\mathbf{q})\ddot{\mathbf{q}}. \tag{31}$$

Once again, at a given time instant $\dot{\mathbf{J}}_Q(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}}$ can be treated as some known constant vector, $\mathbf{k}_Q$, and therefore (31) can be simply written as:

$$\mathbf{a}_Q = \mathbf{k}_Q + \mathbf{J}_Q\ddot{\mathbf{q}}. \tag{32}$$

Using the default joint accelerations, $\ddot{\mathbf{q}}^0$, we compute the default acceleration of Q, $\mathbf{a}_Q^0$ as:

$$\mathbf{a}_Q^0 = \mathbf{k}_Q + \mathbf{J}_Q\ddot{\mathbf{q}}^0. \tag{33}$$

Subtracting (33) from (32) gives an expression for $\mathbf{e}_Q$, the net effect on the acceleration of point Q:

$$\mathbf{e}_Q = \mathbf{a}_Q - \mathbf{a}_Q^0 = \mathbf{J}_Q(\ddot{\mathbf{q}} - \ddot{\mathbf{q}}^0). \tag{34}$$

If $\ddot{\mathbf{q}}$ is the vector of joint accelerations when a given force (or torque) is acting on the figure, since $\ddot{\mathbf{q}} - \ddot{\mathbf{q}}^0$ is proportional to the force (or torque) magnitude, $\mathbf{e}_Q$ will also be also proportional to the force (or torque) magnitude.

To summarize the results from equations (28) and (34), any force (or torque) applied to an articulated body has a net effect on the body's joint and link accelerations that is proportional to the force (or torque) magnitude.

References

BARAFF, D., KASS, M., AND WITKIN, A. 1999. *Physically Based Modeling, Course Notes*. ACM Siggraph.

BARAFF, D. 1994. Fast contact force computation for nonpenetrating rigid bodies. In *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, ACM Press, A. Glassner, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, 23–34. ISBN 0-89791-667-0.

BARAFF, D. 1996. Linear-time dynamics using Lagrange multipliers. In *Proceedings of the ACM Conference on Computer Graphics*, ACM, New York, 137–146.

BARZEL, R., AND BARR, A. H. 1988. A modeling system based on dynamics constraings. *Computer Graphics (Proc. of SIGGRAPH '88) 22*, 4, 179–188.

COTTLE, R. W., PANG, J.-S., AND STONE, R. E. 1992. *The Linear Complementarity Problem*. Academic Press.

CRAIG, J. J. 1986. *Introduction to Robotics: Mechanics and Control*. Addison Wesley, Reading, MA.

EBERLY, D. 2004. *Game Physics*. Morgan Kaufmann.

FEATHERSTONE, R. 1983. The calculation of robot dynamics using articulated-body inertias. *International J. of Robotics, 2(1)*, 13–29.

FEATHERSTONE, R. 1987. *Robot dynamics algorithms*. Kluwer Academic Publishers.

GLEICHER, M. 1994. *A Differential Approach to Graphical Interaction*. PhD thesis, Carnegie Mellon University.

HECKER, C. 1998. Rigid body dynamics. `http://www.d6.com/users/checker/dynamics.htm` .

HECKER, C. 2004. Lemke's algorithm, the hammer in your toolbox? In *Game Developers Conference*.

HOLLARS, M., ROSENTHAL, D., AND SHERMAN, M. 1991. *SD-Fast User's Manual*. Symbolic Dynamics, Mountain View, CA.

JACOBSEN, T. 2001. Advanced character physics. In *Game Developers Conference*.

KOKKEVIS, E., AND METAXAS, D. 1998. Efficient dynamic constraints for animating articulated figures. *Multibody System Dynamics*, 2, 89–114.

MIRTICH, B. 1996. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California, Berkeley.

SHABANA, A. 1994. *Computational Dynamics*. John Wiley and Sons, Inc.

WITKIN, A., GLEICHER, M., AND WELCH, W. 1990. Interactive dynamics. In *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, vol. 24, 11–21.