Spacetime Constraints Attempted

Taylor Shaw

Friday September 14, 2001

1 Introduction

For my research project this summer with Professor Nancy Pollard, I attempted to implement a character animation system similar to the one described in Witkin and Kass' 1988 SIGGRAPH paper "Spacetime Constraints" [19]. My initial interest in this project arose while I was thinking about ways to ensure that character animations were physically correct. Typically, hand-animated character motion, and even most motion capture data, gives rise to animations which are physically impossible. That is, given the computer representation of the physical properties of the character, the animation does not satisfy Newton's laws of motion. This is a problem particularly when the animation is transformed by changing physical parameters because small deviations from physically correct motion could potentially be transformed into large, undesirable artifacts.

Spacetime Constraints is a method of animation in which the animator specifies certain constraints that a character's motion must satisfy and instructs the computer to solve for the correct motion. Typically one of the constraints that is enforced is that the motion be physically accurate. Thus, the spacetime constraints method seemed to me to be a good way to approach the problem of constraining an existing motion to be physically correct.

My project, then, was an attempt to implement a spacetime constraint animation system which could be used to generate character animation as well as modify existing animations. Unfortunately, this goal was quite a bit too lofty. The difficulties of implementing a working spacetime animation system proved much greater than I had anticipated, and ultimately I was unable to apply my efforts to characters of more than trivial complexity. However, I did learn a great deal about how to implement spacetime constraint animation, and have a much greater appreciation for why no commercial software packages currently support it. I hope that my work can at least provide some advice for future implementations.

The remainder of this paper is organized as follows. Section 2 presents a brief survey of the major papers related to spacetime constraints. Section 3 gives an overview of the spacetime problem formulation and Section 4 introduces the design of my program. Sections 5, 6 and 7 discuss implementation details of specific aspects of the design. Section 8 presents my results. Section 9 contains lessons I have learned from this project and other concluding remarks.

2 Related Work

There are two traditional approaches to computer animation. One approach uses the computer to enhance traditional animation tools, and leaves control of the animation largely up to the artist. This method of animation has the disadvantage that it does not fully utilize the power of the computer to automatically generate convincing animations and requires a skillful artist to get decent results. The second approach is that of physical simulation. In this approach the motion of objects is calculated based on their physical properties and the external forces acting upon them. While it creates physically accurate motion, the animator is left with very little control over the resulting animation.

Witkin and Kass [19] introduced the Spacetime Constraints method of animation as a compromise between these two extremes. They solve for a character's motion over the entire time range of the animation simultaneously. Thus, constraints imposed at particular times will have effects both forward and backward in time. The animator creates constraints on the character's motion to specify what tasks the motion must accomplish, as well as constraints to ensure that the motion is physically correct. In addition, the animator specifies an objective which determines how a motion should be performed. This provides the animator with significantly more control over an animation than pure simulation, yet does not sacrifice physical accuracy. Brotman and Netravali [2] have obtained a similar result using Optimal Control methods.

The problem with Spacetime Constraints is that solving for the optimal animation, as well as setting up the problem in the first place, is extremely difficult and time-consuming. The method has not yet been implemented in any commercial software (to my knowledge) precisely for this reason. Thus, most of the research done on spacetime constraints has revolved around making the method faster or more user-friendly. Cohen [4] describes a prototype implementation of a more usable spacetime animation system, using the concept of spacetime windows. Gleicher [9] [10] simplifies the spacetime problem by relaxing its requirement of physical accuracy, thus speeding up the solution process. Rose et al. [16] and Popovic and Witkin [13] simplify the problem by applying the spacetime method to very small intervals of time.

Similar problems have been solved through the use of simulated annealing, genetic algorithms [12] [18], and monte carlo methods [3]. For this project I briefly considered using a monte carlo method, and looked into genetic algorithms a little. My feeling, however, is that these methods are not particularly well suited to the spacetime animation problem because it is concerned with finding a single, optimal solution. Randomized methods are very good at generating large sequences of pretty good solutions, but are much slower than normal numerical optimization at finding the single, locally optimal solution of a problem.

3 Spacetime Constraints

The spacetime constraints approach is unique in that it formulates the process of generating an animation in terms of an optimization problem on the degrees of freedom of a character model. In my implementation, the degrees of freedom, S, of a model refer to the generalized coordinates that determine its position. That is, each character is modeled as a linked chain of rigid bodies, and its position is completely determined by the joint angles of this linkage. Each degree of freedom, S_j, varies over time and therefore is a function of a time variable, t. Following [4], I represent each S_j with a uniform cubic b-spline. Thus, every degree of freedom S_j is actually a function of a number of control points, X. Unlike the discrete method presented in [19], using b-splines has the advantage of guaranteeing that each degree of freedom will be twice-differentiable at every possible value of t. It also allows us to limit the number of independent variables in a problem by spacing the X farther apart in t.

A user specifies what an animation should do by creating a set of constraints, $C(S) = 0$, on the the degrees of freedom. That is, a constraint is any function on S which is equal to 0 at a given value of t. For example, constraints can be used to specify that a character be at a certain position at a certain time. Constraints are also used to ensure that an animation is physically possible. To do this we simply constrain the generalized force at the root of a character to always be equal to zero. This is sufficient to enforce the dynamic correctness of a motion. Constraints also arise as properties of a character model. For example, constraints may be imposed to limit the amount of force that can be applied at a particular joint. Constraints over a range of time are handled by creating multiple constraints at discrete time intervals. Currently, I do not support inequality constraints because they add additional complexity to the numerical optimization process.

The animator also specifies how a motion should be performed by supplying an objective function, $R(S)$, which must be minimized. Typically this function minimizes the amount of force exerted by a character's joints, but in theory it can be any function of the degrees of freedom. Currently, my implementation just minimizes the sum of the squares of the generalized force at each joint of a character. I had wanted to explore more interesting measures of optimality, such as minimizing power or motion smoothness, but was not able to get to this.

The set of constraints and objective together constitute a nonlinearly constrained optimization problem which must be solved to yield the values of X which minimize R subject to C. The solution to this problem is the optimal character motion for the given situation.

4 Design Overview

Figure 1 illustrates the basic design of my system, which is written in C^{++} using a Qt GUI. The heart of the system is the `Expression Manager`. This class is

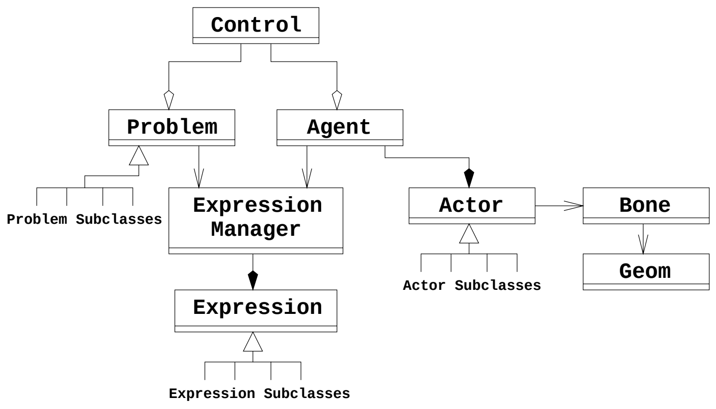

Figure 1: Overview of design

responsible for managing the creation and optimization of every symbolic math expression used in the program, and is referenced by nearly every other major class. It contains a list of all expressions, as well as various subsidiary lists of specific types of expressions, such as variables, which are maintained for efficiency reasons. The `Expression`s themselves are atomic units such as `Sin` or `Sum`, which can be combined in a tree structure to form larger expressions. They are capable of returning their own derivatives as well as optimizing themselves.

Above the expression manager, the system is conceptually divided between the two tasks of character specification and spacetime problem solution. The `Actor` is the abstract superclass of all characters, and contains a hierarchy of generic `Bone` objects stored in a tree structure which reflects their physical linkage. Each bone knows about its parent and children bones and is able to recursively generate symbolic expressions for any quantity necessary for a spacetime problem (such as its generalized position, velocity, force, etc.) using the equations of rigid body dynamics. Each bone is associated with a `Geom` object which encapsulates physical properties of the bone, such as its mass, inertia tensor and bounding box. This allows the logic of the `Bone` class to be used for bones of any size or shape. Each `Actor` subclass is responsible for creating and managing its bone tree as well as handling user requests for symbolic expressions describing properties of specific bones. All instantiated actors are managed by the `Agent` (get it?), which is responsible for generating any expressions involving multiple actors, as well as providing the user interface with information about the actors.

The `Problem` is the superclass for all methods of solving a given spacetime optimization problem and is essentially just a container for constraint and objective expressions. It has methods which the user interface calls to add constraints and objectives, as well as to indicate which variables should be treated as independent. `Problem` subclasses use the constraint and objective information in whatever way necessary to find the values of the given independent variables which minimize the objective function. The problem also provides various methods which allow the user to control the progress of the solution process.

The `Control` is currently a substitute for a fully realized user interface. In theory, the UI would allow the user to create actors by communicating with the agent. It would also let the user request expressions from an actor and add them to the problem as constraints or objectives. Because I did not have time to complete a UI, however, these operations are currently hard-coded in the control to set up specific test problems.

Other Elements of the user interface which I implemented but are not included in Figure 1 are an OpenGL window with movable camera and time slider for displaying animations, and a graphing widget which can display the value of any expression over time.

5 Symbolic Math Package

When solving a spacetime optimization problem it is necessary to be able to calculate the first, and sometimes the second, partial derivatives of the constraint and objective functions. Since these functions can be arbitrarily complex, and to avoid taking derivatives by hand, it is necessary to develop a system which can differentiate equations automatically. I therefore implemented a symbolic math package capable of calculating the value and gradient of complex multivariate functions. Previous spacetime approaches (e.g., [4]) have utilized existing symbolic math libraries such as Maple to this end. I decided, however, that for learning purposes it would be more interesting to try to write my own.

Every function or value in my program is composed of a group of atomic `Expression` objects, arranged in a directed acyclic graph structure (basically an expression tree in which repeated subtrees are shared by multiple parents). Even simple values like physical constants are represented by expressions. Thus, things like the radius of a sphere, or the mass of Luxo's base, which are usually given constant expressions for simplicity, could instead be assigned to be a function of any other variable in the program. This makes the system extremely adaptable and versatile. For reference, a simplified version of the header file for the `Expression` superclass is shown in Figure 2.

Each `Expression` subclass is capable of performing a single, simple computation. The system currently supports expressions for basic arithmetic operations ($+, -, \times$), sine, cosine, exponentiation, constants, variables, and b-splines. `Variable`s are expressions whose values can be modified by either the user or the spacetime problem. All expressions, with the exception of variables and constants, are associated with child expressions on which they operate. Thus, a complicated function built by compositing simpler expressions is represented as an expression tree, with variables and constants at the leaves. Some expressions (like `Sum` and `Product`) operate on lists of children for efficiency reasons.

The large number of expressions generated when setting up spacetime problems makes it essential to avoid the creation of duplicate expressions. Thus the `Expression Manager` handles the creation of new expressions and stores all

```
class Expression {
public:

    double          calcValue(); // functions for calculating values
    Expression*     calcPartial( Variable* var );
    double*         calcGradForward( int gradSize );
    void            calcGradReverse();

    Expression*     optimize(); // optimization functions
    bool            isZero();
    bool            isOne();

    // various accessors here

protected:
    double          _value; // the actual value of the function
    bool            _valueDirty;

    double*         _gradient; // for forward differentiation
    bool            _gradientDirty;

    double          _revVal; // for reverse differentiation
};
```

Figure 2: `Expression` superclass header

created expressions in a large STL `map` keyed on a unique expression hashcode[1]. When the manager receives a request to create a new expression, it looks for an existing expression with an identical hashcode. If one exists, it simply returns the old expression, otherwise it creates the new expression and stores it in its map. This system ensures that no duplicate expressions are ever created, and that common subexpressions are reused maximally. This is beneficial because the value of a common subexpression only needs to be computed once. Each expression class will only evaluate itself if it is "dirty." That is, if it has not yet been evaluated, or if the value of one of its child variables has been modified. The expression manager is responsible for setting an expression's dirty flag when appropriate. When an expression is evaluated, it simply recursively calculates the values of its children and returns the result of performing the appropriate operation on them.

Witkin and Kass [19] also implemented a symbolic math package similar to

[1]The "hashcode" of an expression is simply an STL string representation computed by recursively traversing the expression graph. This is definitely not the most memory- or speed-efficient hashcode to use. I only did it this way for simplicity.

the one I developed. However, their approach differed in that their symbolic expressions were capable of generating a compiled lisp code for evaluating the various functions needed for the optimization problem. I chose not to take this approach because I assumed that the speed of C++ on modern computer architectures was sufficient enough not to warrant precompiling the spacetime expressions. In retrospect, however, I think that Witkin and Kass' approach is still justified, and is perhaps the only means to get decently fast solutions to spacetime problems. Were I to attempt a new implementation, I would use my expression graphs to generate C code which could be compiled directly into the solution process.

5.1 Expression Optimization

Before the expression manager adds a new expression to its map, the new expression is symbolically optimized for best performance. Each `Expression` subclass has an optimize method which analyses the types and values of any children and returns an equivalent expression which has been algebraically manipulated into a simpler form. For example, the expression $x + 0 + 1$ simplifies to $x + 1$. Because this optimization is done whenever any expression is created, it will often simplify expressions recursively. For example, the expression $x \times (x + 1)$ will distribute to $(x \times x) + (x \times 1)$ which is then simplified to $(x \times x) + x$ when the expression $(x \times 1)$ is created. Note that we do not form $x^2 + x$ because in this case the power operator is computationally more inefficient than simple multiplication. In order to perform this optimization efficiently, each expression knows about its type, as well as whether its value is zero or one (which can only be true for `Constants`).

5.2 Differentiation

There are many methods of calculating the gradient of symbolic expressions. I implemented three of them with the aim of comparing the relative merits of each. The first, and simplest, is just to generate a new symbolic expression for each partial derivative. This is done in a straightforward manner by simply recursing through the expression graph and telling each expression to differentiate itself with respect to a given `Variable` expression. For example, if a `Product` expression with two child expressions, `f(x,y)` $\times$ `g(x,y)`, is differentiated with respect to the variable `x`, it returns the new expression `f(x,y)`$\times$`(g(x,y)->calcPartial(x))` + `(f(x,y)->calcPartial(x))`$\times$`g(x,y)`. This method has the advantage of being simple, but has the disadvantage of generating many new expressions which take up a lot of memory. It is also relatively slow to individually evaluate each partial expression in the gradient.

The other methods I implemented were originally proposed in [11] and recommended by [9]. They are variants of what is called *automatic differentiation* and are significant because they calculate the entire gradient of a function by only walking through the expression graph *once*. In theory, then, they provide a much faster method of taking derivatives than generating symbolic partial expressions.

The *forward* mode of automatic differentiation proceeds as follows. At each expression node, the complete gradient of the subgraph rooted at that node is stored. The expression calculates its gradient by looping through its children and, for each child, forming the scalar product of the child's gradient and the expression's partial derivative with respect to the child. This vector is then added to the expression's gradient value. Pseudocode for this process is given in Figure 3.

```
calcGradForward( numVars )
    gradient ← vector of size numVars
    for each child do
        part ← value of partial deriv wrt child
        childGrad ← child.calcGradForward( numVars )
        childGrad.scale( part )
        gradient.add( childGrad )
    end
    return gradient
end
```

Figure 3: Pseudocode executed when visiting an expression node during forward differentiation.

For this routine to work correctly, each `Variable` expression is assigned an index into the gradient. At the variable leaves, the calculated gradient is simply zero everywhere with a one in the gradient position corresponding to the variable's index. This approach allows us to calculate the gradient with respect to only a subset of the independent variables by simply not setting the index of the variables we do not care about.

The forward mode is able to take advantage of repeated subexpressions because the gradient at an expression node only needs to be calculated once. However, it has the disadvantage that at every node the entire gradient must be calculated and stored, which wastes memory and time if every expression depends on only a small subset of the independent variables.

The *reverse* mode of automatic differentiation, on the other hand, only requires that a single scalar quantity be maintained at each expression node. The algorithm executes a breadth-first, preorder traversal of the graph, and at each node executes the pseudocode shown in Figure 4. For each child of a node we multiply the stored value at the node, `revVal`, by the partial derivative with respect to the child and then add that value to the child's `revVal`. We can define `revVal` at the root to be any value by which we wish to scale the gradient (usually 1). When the traversal terminates, the stored scalar value at a variable leaf is equal to the partial derivative of the entire expression with respect to that

```
calcGradReverse()
    for each child do
        temp ← value of partial deriv wrt child
        child.revVal += ( this.revVal × temp )
    end
end
```

Figure 4: Routine executed at each expression node during reverse differentiation

variable. In theory this executes considerably faster than the forward mode because many fewer quantities must be updated at each node. The disadvantage of the reverse mode, however, is that because the `revVal` of a child expression depends on the `revVal` of the parent, the algorithm cannot take advantage of common subexpressions, as in the forward mode. In practice, which mode is better depends on the situation. I implemented a simple operation counting mechanism to help determine which mode was more appropriate for a given set of equations.

The forward and reverse modes are, in theory, capable of also calculating the value of second and higher-order derivatives in a similar manner. My implementation, however, only supports the calculation of first partial derivatives. When the Hessian of a function is needed, my approach is to symbolically calculate the gradient expression and then automatically differentiate the first partials. This approach works, but significant speedups can be achieved by supporting second-order automatic differentiation.

5.3 B-Splines

Cubic B-Splines are a special type of expression because the method in which they calculate their value is dependent upon the value of an external expression (i.e., the time variable). I struggled for a long time to figure out how to integrate b-splines with the rest of the symbolic math framework, and ultimately came up with a solution which is not particularly elegant, but works correctly.

Each `B-Spline` object is given a set of data points to interpolate, from which the values of the initail b-spline control points, X, are calculated. These control points are the main independent variables of a typical spacetime problem. The b-spline also contains subexpressions for each of its four basis functions, B. To calculate the value of the b-spline at time t, it is necessary to find the curve segment active at time t, and the the distance along that curve in t (i.e., a value between 0 and 1). Then the four control points active at t are found and the value of the b-spline is simply $S_j(t) = \sum_{l=0}^{4} X_{j,l} B_l(t)$, where $X_{j,l}$ are the values of the jth degree of freedom's active control points. To take the derivative of b-splines, we first check whether the variable in question is the time variable. The partial derivative of the spline with respect to the time variable is generated by simply returning a new b-spline with identical control points, but the derivatives of the basis functions as new bases. That is, $\dot{S}_j(t) = \sum_{l=0}^{4} X_{j,l}\dot{B}_l(t)$ and $\ddot{S}_j(t) = \sum_{l=0}^{4} X_{j,l}\ddot{B}_l(t)$. The partial derivatives of the b-spline with respect to any of the control points is simply the basis function active for that control point at time t, or 0.

5.4 Other Classes

In addition to the expression manager and various expression subclasses, the symbolic math package has some classes which function as expression containers, and allow expressions to be operated on in specific ways. Matrices and vectors are two examples of these. The `Matrix` class is an $n \times n$ matrix of expressions and supports operations like transpose and matrix multiplication. The `Vector` class is a subclass of `Matrix` and extends it to support inner product and vector product operations.

5.5 Problems

The major unexpected problem which arose from my symbolic math implementation was that the expression graphs generated by even modest spacetime problems were enormous. With very many (more than 5000 or so) expressions active for a given problem, traversing the graphs to calculate values and derivatives becomes very slow. While I was careful to cache values and avoid repetitive computations whenever possible, recursing through hundreds of functions calls is not fast. More significantly, with enormous expression graphs, memory usage becomes a major issue. I did not design my expression classes with compactness in mind, and the program frequently ate up more memory than was available. Swapping expression instances into and out of memory causes any function evaluation to quickly grind to a halt. Were I to reimplement my symbolic math package, minimizing memory usage would be my primary concern. This would most likely mean not using the STL for data structures, since the STL lists, maps and strings created for each expression were the main cause of memory consumption.

6 Spatial Rigid Body Dynamics

All the actors in the program are modeled as linked chains of rigid bodies. Because most of the research on efficient methods of representing linked rigid-body motion has come out of the robotics community, I use an actor description based on the Denavit-Hartenburg notation for robot manipulators [5]. Each bone in the linkage is assigned an index, i, such that the index of a bone is greater than that of its parent, and no bones have the same index. The ith joint, then, is the joint between the ith bone and its parent. In this paper parent and child bones are referred to as $i-1$ and $i+1$ respectively, though if the linkage is branched, this convention does not hold.

```
Step 0: Initialization
    $\hat{v}_0 = \hat{a}_0 = \hat{f}_{n+1} = \hat{0}$

Step 1: Forward Recursion
    $\hat{v}_i = \hat{X}^i_{i-1}\hat{v}_{i-1} + \hat{s}'_i\dot{q}_i$
    $\hat{a}_i = \hat{X}^i_{i-1}\hat{a}_{i-1} + \hat{v}\hat{\times}\hat{s}'_i\ddot{q}_i$
    $\hat{f}^*_i = \hat{I}'_i\hat{a}_i + \hat{v}_i\hat{\times}\hat{I}'_i\hat{v}_i$

Step 2: Backward Recursion
    $\hat{f}_i = \hat{X}^i_{i+1}\hat{f}_{i+1} + \hat{f}^*_i$
    $Q_i = \hat{s}'^S_i\hat{f}_i$

end
```

Figure 5: Featherstone's recursive inverse dynamics algorithm

Calculating the forces required at each of the joints in order to produce a desired actor motion is a type of inverse dynamics problem. The dynamics of a system of rigid bodies with n degrees of freedom is simply described by a system of n coupled differential equations. The problem is devising a representation of these equations which allows for efficient solution. The solution will be identical no matter which dynamics formulation is used (Newton-Euler, Euler-Lagrange, Kane, etc.), but the amount of computation done will vary between representations. For an excellent survey of approaches to this problem, see [1].

6.1 Featherstone's Formulation

I chose to represent the dynamics equations of the actors in terms of Featherstone's recursive formulation [6]. The main difference between this method and most other dynamics algorithms is that it uses 6-dimensional *spatial vectors* to represent the combined linear and angular components of physical quantities involved in rigid body dynamics. Inertia tensors are likewise represented by a 6×6 spatial matrix. Spatial vectors are defined in terms of Plücker coordinates. A line which passes through the point A in the direction a is represented as $(a, \overrightarrow{OA} \times a)^T$ where $\overrightarrow{OA}$ is the vector from the origin to A.

Featherstone's algorithm is shown in Figure 5. Spatial quantities are indicated with a hat $(\hat{})$ over them. In this figure $\hat{s}_i, \hat{v}_i, \hat{a}_i \hat{f}_i$ are the spatial position, velocity, acceleration and force of the ith link; $q_i, \dot{q}_i, \ddot{q}_i$ and Q_i are the generalized position, velocity, acceleration and force of the ith link; $\hat{I}_i$ is the inertia tensor of the ith link and $\hat{X}^j_i$ is the spatial transform from the ith link's frame to the jth link's frame. The spatial transpose and cross product, ^{S} and $\hat{\times}$, are defined in [6].

The only real advantage of Featherstone's method is that it reduces the number of quantities involved in solving various dynamics problems. The important thing to note is that by formulating the dynamics problem recursively and using a reasonably compact representation for linear and angular quantities, computing all of the generalized forces requires $130n - 68$ scalar multiplications and $101n - 56$ scalar additions. There are certainly other dynamics formulations which involve less computation (e.g., [1]), and even some faster algorithms which use spatial algebra similar to Featherstone's (e.g., [15]). Because evaluation of the dynamics equations is the major bottleneck during the solution process, using as fast an algorithm as possible is of the utmost importance. I used Featherstone's method, however, because it is decently fast, and simple to implement. Also, I had already implemented it once, so it was particularly easy to adapt it to use my symbolic math package.

6.2 Implementation

Actors in my system are modeled following Featherstone's model of a robot manipulator. They are a tree of rigid links (bones) starting with a fixed root and connected by prismatic, rotational or screw joints. Each bone is associated with the spatial representation of its parent joint, and knows about its parent and children bones. For simplicity, no cycles are allowed in the linkage. To create actors with mobile bases, we simply create bones for the x, y and z position of the base using `NullGeom` objects, which make these "virtual" bones invisible and have zero mass.

When a spacetime problem is initially set up, the user asks specific bones to return expressions for various dynamics quantities. Each bone knows how to assemble all types of dynamics expressions using Featherstone's recursive formulas. For example, if the bone receives a request for its spatial velocity expression, it executes the algorithm shown in Figure 6. In this code, `_v` is the

```
Bone::spatialVel()
{
    if( _parent == NULL ){ // we're the root
        _v.setZero();
    } else {
        SpatialVector parentVel = _parent.spatialVel();
        _parentXform.transform( parentVel, _v );
    }
    SpatialVector temp;
    temp.scale( this.generalizedVel(), this.spatialPos() );
    _v.add( temp );

    return _v;
}
```

Figure 6: The routine for calculating the spatial velocity of a bone

bone's spatial velocity vector and `_parentXform` is the spatial transformation from the bone's parent's frame to the bone's frame. The code in Figure 6 is functionally identical to Featherstone's expression for spatial velocity. In this way expressions for dynamic quantities are generated exactly according to Featherstone's equations.

If any external forces will ever need to be applied to a bone, the actor must create a spatial vector variable for that force and tell the bone about it. The bone then uses this variable when creating its expressions for spatial force. Any time an independent variable is changed during the solution process, the actor is responsible for calculating the current value of the external force and setting the force variable appropriately. If the external force does not exist (e.g., if it is a force for handling collisions, and there are no current collisions), the force variable is simply set to zero. I am sure that this is not the most elegant way to handle external forces, but I had a lot of trouble devising a better solution.

7 Numerical Optimization

Once the spacetime problem constraints and objective function are fully specified, all that remains is to solve the nonlinearly constrained optimization problem. There is no general, guaranteed solution method for problems of this type. Finding an optimization method which is suited to a particular problem is something of an art. Some researchers have emphasized simplifying the spacetime problem to such a degree that the solution can be found by using relatively simple methods. For example [16] is able to simply use the BFGS method described in [8]. Most spacetime work, however, has focused on using variants of *Sequential Quadratic Programming* as a solution method (e.g., [4], [9], [13], [19]). This is the method which I chose to employ.

An SQP algorithm proceeds by solving a sequence of solvable subproblems. At each iteration, a quadratic program (i.e., an optimization problem with linear constraints and a quadratic objective, usually in the form of a linear system) is created which approximates the nonlinear problem at the current values of the independent variables. The solution to this subproblem determines a direction in which to step for the next iteration. Typically a line search is used to determine the step length to take in the calculated direction. A solution is found when all the $C = 0$ and the gradient of R is equal to some linear combination of the gradients of the C.

I considered two methods of solution, that of Cohen [4] and that of Witkin and Kass [19]. I implemented Cohen's method first because it seemed simpler than that of Witkin and Kass. Whereas the Witkin and Kass approach involves solving two linear systems in sequence, Cohen only solves one linear system at each iteration. In addition, the matrix involved in Cohen's linear system is guaranteed to be symmetric. These features made it seem more promising initially. However, I came to believe that Cohen's method is more inefficient than Witkin and Kass' because it has a higher likelihood of generating ill-conditioned matrices, and because it required the use of a line search. Ultimately, I implemented Witkin and Kass' method as well. While I still was unable to solve large optimization problems with it, it solved small problems much faster. These differences could simply result from the details of my implementations so I cannot make any conclusive claims about the relative merits of either approach. My feeling is, however, that Witkin and Kass' solution method is more versatile and applicable.

7.1 Cohen's Approach

Cohen suggests minimizing the Lagrangian of the constrained optimization problem, which creates an equivalent unconstrained problem. The Lagrangian of the spacetime problem is

$$L(X, \lambda) = R(X) + \sum_{i=1}^{m} \lambda_i C_i(X)$$

where the λ_i are Lagrange multipliers which roughly represent the influence of the constraints on the objective. The optimization proceeds by solving

$$\nabla^2 L \begin{bmatrix} \partial X \\ \partial \lambda \end{bmatrix} = -\nabla L^T$$

to yield a step direction in X and λ. Because

$$\nabla^2 L = \begin{bmatrix} \nabla^2_{XX} L & \nabla^2_{X\lambda} L \\ \nabla^2_{\lambda X} L & \nabla^2_{\lambda\lambda} L \end{bmatrix} = \begin{bmatrix} \nabla^2 R + \lambda^T \nabla^2 C & \nabla C^T \\ \nabla C & 0 \end{bmatrix}$$

the Hessian $\nabla^2 L$ will always be symmetric, extremely sparse, and easy to calculate. In theory, this simplifies the solution somewhat. To solve this system of linear equations, I adapted a conjugate gradient method for sparse, symmetric linear systems described in [14] which minimizes the residual, $\frac{1}{2}|\nabla^2 L \cdot X + \nabla L^T|^2$. Briefly, the conjugate gradient method for linear systems is similar to the method of steepest descent, except that it chooses the steepest descent direction which is also *conjugate* to all n previously chosen directions. One main advantage of conjugate gradient methods which solve $Mx = b$ is that they can be written such that the matrix M is only used to transform vectors. Thus, special optimized routines to perform this transformation can be written which take advantage of the sparsity of the matrix M.

Cohen's method worked reasonably well for small particle problems. However, it would often blow up if a line search was not utilized. Because evaluation of the objective function is very slow, and the accuracy of a line search depends on how many evaluations of the objective are performed, this is a big disadvantage.

I was not able to get Cohen's method to work for large problems. My feeling is that by introducing extra independent variables (the λ) and creating one huge system which minimizes both the constraints and the objective, the possibility for instability is increased dramatically. Also, the condition number of the Hessian matrix often becomes extremely high.

7.2 Witkin and Kass' approach

Witkin and Kass' approach is preferable to Cohen's method because it breaks the solution process up into the solution of two smaller systems of linear equations. First a step $\hat{X}$ is calculated which minimizes a quadratic approximation to the objective, without taking the constraints into consideration, by solving:

$$-\frac{\partial R}{\partial X_i} = \sum_j H_{ji} \hat{X}_j$$

where H_{ji} is the Hessian of the objective, $\nabla^2 R$. Then this step is projected onto the null space of the constraint Jacobian, J_{ij}, by solving:

$$-C_i = \sum_j J_{ij} (\hat{X}_j + \tilde{X}_j)$$

The final step is $\Delta X = \hat{X} + \tilde{X}$. This method is advantageous because after each iteration the constraints will be very nearly satisfied, so the resulting animation will look better after fewer iterations. It also has the advantage that a line search is usually unnecessary. The main problem with this approach is that the Jacobian matrix, J, is almost never symmetrical, or even square. To solve this system then, we need to adapt a conjugate gradient solver to handle non-square matrices. Witkin and Kass recommend adapting the conjugate gradient method mentioned in Section 7.1 to use the pseudo-inverse of the matrix J. [14] suggests using the generalized minimum residual method described in [17]. I chose, however, to implement the LSQR method presented in [7] mainly because Fortran code was available.

The Witkin and Kass approach worked better than Cohen's method but also was unable to solve large problems. Mostly this was because the Hessian matrix H was often extremely ill-conditioned. I attempted various techniques to fix this such as normalizing the rows of H, damping the solution process, and using a preconditioner matrix, but could not get anything to work.

7.3 Miscellaneous Details

When evaluating the values and gradients of the all expressions used with a given optimization method, one of the easiest things to forget was to correctly set the time variable t for the quantity being evaluated. In my implementation, all constraints are defined only at a specific time. In order to correctly evaluate the constraint expression, it is necessary to set the time variable to the constraint time prior to performing any calculations. Similarly, the objective function is defined as an integral over the entire time range of the animation. I evaluate it by looping through the time range and calculating the value of the objective function at discrete sample times.

Also, all the spacetime papers recommend taking advantage of the sparsity of the matrices which arise during the solution process. Thus, I implemented a sparse matrix class as described in [14]. Based on a trial optimization step with random values of the independent variables, it conservatively estimates the sparsity of a given $n \times n$ matrix of doubles, and creates a sparse matrix of doubles with a fixed sparsity pattern. The only operations supported by this class are multiplication of a vector, and multiplication of a vector by the transpose, as required by the CG solver algorithms described above.

8 Results

I created two main types of spacetime problems to test the performance of my system. First I got everything working really well for a simple particle, and then immediately tried to apply my program to a 4-bone jumping Luxo lamp. The particle typically worked very well and its results were always encouraging. It demonstrated that all the various parts of my program worked in the way I thought they should. Luxo, on the other hand, was a problem of such greater magnitude than the particle that I never was able to get it working correctly. In retrospect, I probably should have progressed through a sequence of simpler test actors before attempting to implement Luxo. A gradual approach would have allowed me to work out bugs in a more systematic way. But I was impatient, so I pressed onward.

8.1 The Particle

The test particle consists of a small sphere object attached to the end of a chain of three "virtual" bones, which allow motion in all three spatial dimensions. The particle has a constant mass and inertia tensor. Gravity can be applied to it, as well as a force resulting from contact with the ground plane. Typical optimization problems consist of finding the motion which interpolates some number of test points with minimum force. Particle problems generally consist of about 300–500 expressions.

Results from these tests confirmed that the numerical optimization, symbolic expression framework and rigid body dynamics were all working correctly, at least for simple problems. The particle always successfully moved through the trial points in an increasingly smooth manner. Turning on gravity caused the particle's path to visibly sag between test points, as was expected.

Figure 7 illustrates the starting and ending configurations of a sample particle problem. The solution process is initialized with all control points of the particle's b-splines set to zero. After a single iteration, the animation satisfies the constraints but contains a lot of wasteful motion. Only the control points which directly affect the constraints have been significantly modified. In the converged solution, however, nearly all the control points have been modified to create a much smoother motion. The force exerted by the particle in the final solution, then, is successfully minimized.

Both numerical optimizers successfully solved the particle problem. Cohen's method, however, was significantly slower than Witkin and Kass' method. This is most likely because the line search used in Cohen's method was ex-

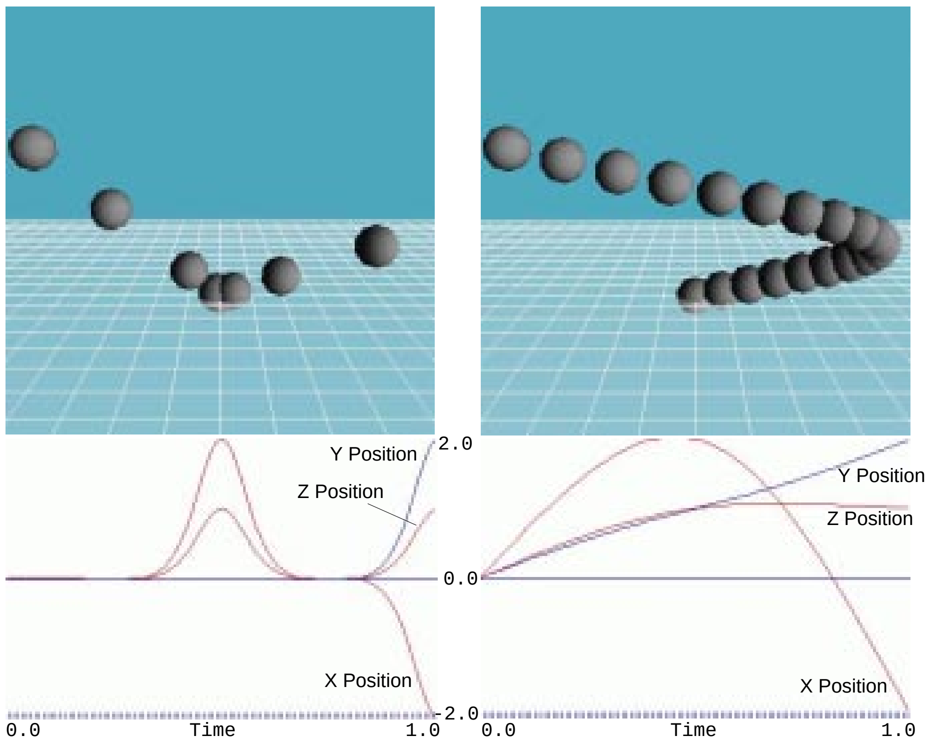

Figure 7: An example of a particle problem. The position of the particle is constrained to be $(0, 0, 0)$ at time $t = 0$, $(2, 1, 1)$ at $t = \frac{1}{2}$ and $(-2, 2, 1)$ at $t = 1$. The objective function is the force needed to move the particle. On the left is the solution obtained after one iteration of Witkin and Kass' optimization method. The final solution is shown on the right. The particle is displayed at intervals of $\Delta t = \frac{1}{20}$. The camera is positioned looking roughly along the $-Z$ axis.

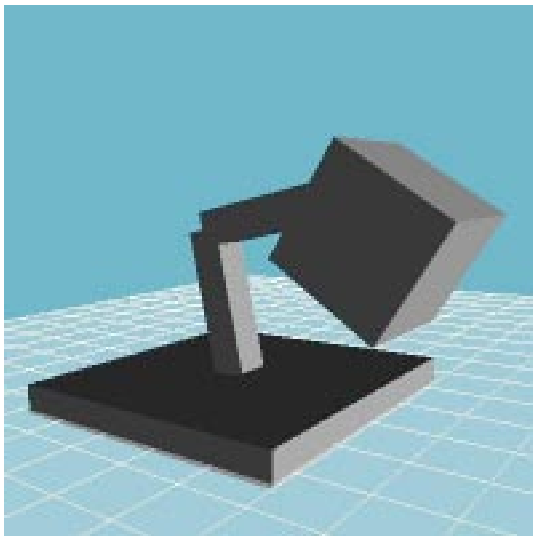

Figure 8: Luxo

tremely inexact, so convergence towards the solution progressed more slowly. In both methods, however, all linear systems were solved in comparable time with equally acceptable error estimates.

8.2 Luxo

Luxo is a significantly more complicated problem than the particle. Luxo is an animate lamp consisting of a base, two "arm" bones and a head. It also has x and y "virtual" bones which allow the base to move freely in the xy plane. Each real bone has an associated constant mass and inertia tensor. Gravity is always applied to it, and there is an additional upwards external force exerted on the base when it is resting on (or accelerating downwards into) the ground.

A typical Luxo problem is to get it to move from $(-1, 0, 0)$ to $(1, 0, 0)$ using as little force as possible. This is the same problem that Witkin and Kass were able to solve. In principle, the correct solution is for Luxo to perform a jumping motion from the starting position to the ending position. Unfortunately, my system is currently unable to solve this problem.

Luxo problems generally involve around 20,000 expressions. Because this is orders of magnitude larger than the simple particle problem, there are many, many more sources of numerical instability. In particular, the objective and constraint functions are extremely nonlinear, and it is very easy for an SQP solver to quickly blow up if nearly-exact line searches are not used. Also, the matrices formed in the various linear systems solved during the SQP solution process are often very ill-conditioned, so it is difficult for a CG method to determine the correct step direction. Even if the solution process works correctly, the number of iterations required to converge to a solution will be drastically higher than in the particle case. Because the larger expressions involved in a Luxo problem also take longer to evaluate, the solution process proceeds very slowly, making it difficult to test and debug. I did obtain a few solutions to the Luxo problem which looked promising, and which hinted that I was very close to getting it working. If I had more time to work on this problem, I am convinced that I could solve it eventually.

9 Conclusion

Currently my spacetime constraints implementation is not completely functional. While it works well for small systems, it is not capable of solving large spacetime problems. This is mainly due to instabilities in the various numerical optimization procedures which I implemented. Because the optimization was the last part of the system which I wrote, I did not have sufficient time to get it completely debugged. I do think, however, that the system I have implemented would be capable of solving problems of Luxo's complexity if I spent longer working on it. Everything besides the numerical optimization is functional. Therefore, were I to continue working on it, my first step would be try using third-party software to perform the optimization. Hopefully a well-tested optimization package such as Matlab or Maple would perform better than my own code.

Even if I never complete the project, though, I have learned a great deal about how to correctly implement spacetime constraints. I made a few bad decisions early on which I would not make again were I to attempt to write a new implementation from scratch. For example, I greatly underestimated the size of the expression graphs generated by complicated spacetime problems. Were I to write this program again, I would place a much greater emphasis on an expression representation which was memory-efficient. This would mean not using the STL at all in the symbolic math package.

I also assumed that a modern computer would be sufficiently fast to evaluate enormous expression graphs on the fly. It turned out that often traversal of the expression graphs was extremely slow. Therefore, I now think that the expression graphs should only be used during a precomputation phase to generate compilable code, as Witkin and Kass suggested in their original paper [19]. This would greatly speed up the evaluation of the various dynamics quantities needed by the optimizer.

Other things that I would like to try in order to improve performance are implementing second-order automatic differentiation, and using a faster formulation of the dynamics equations (e.g., Balafoutis' [1]).

Overall, though, I am decently satisfied with my results. While it is disappointing that I could not get Luxo to jump, I am very happy that at least the particle problem works correctly. Obtaining that simple result involved a lot of work, and the particle's success alone makes this project worthwhile. Academically, I am pleased with the amount of new material that this project introduced me to. I was forced to learn a lot about subjects of which I had had no previous knowledge (e.g., mathematical programming, rigid body dynamics, numerical optimization), and feel as though I am now partially conversant in many of them. Implementing spacetime constraints was an extremely beneficial experience, regardless of the results. I am fortunate, and thankful, to have had this opportunity.

References

[1] C.A. Balafoutis, R.V. Patel: *Dynamic Analysis of Robot Manipulators: A Cartesian Tensor Approach*, Kluwer Academic Publishers, 1991.

[2] L.S. Brotman, A.N. Netravali: Motion interpolation by optimal control. In *Proceedings of SIGGRAPH '88*, Vol. 22, ACM, pp. 309-315, 1988.

[3] S. Chenney, D.A. Forsyth: Sampling constrained animations *???*

[4] M. Cohen: Interactive spacetime control for animation, *Proceedings of SIGGRAPH '92*, ACM, 1992.

[5] J. Denavit, R.S. Hartenburg: A kinematic notation for lower-pair mechanisms based on matrices, *ASME Journal of Applied Mathematics*, vol. 23, 1955.

[6] R. Featherstone: *Robot Dynamic Algorithms*, Kluwer Academic Publishers, 1987.

[7] C.C. Paige, M.A. Saunders: LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software*, vol. 8, ACM, 1982.

[8] P.E. Gill, W. Murray, M.H. Wright: *Practical Optimization*, Academic Press, 1981.

[9] M. Gleicher: Motion editing with spacetime constraints, *1997 Symposium on Interactive 3D Graphics*, ACM, 1997.

[10] M. Gleicher: Retargeting motion to new characters, *Proceedings of SIGGRAPH '98*, ACM, 1998.

[11] A. Griewank: On automatic differentiation, *Mathematical Programming: Recent Developments and Applications*, Kluwer Academic Publishers, 1989.

[12] J.T. Ngo, J. Marks: Spacetime constraints revisited *Proceedings of SIGGRAPH '93*, ACM, 1993.

[13] Z. Popovic, A. Witkin: Physically based motion transformation, *Proceedings of SIGGRAPH '99*, ACM, 1999.

[14] W.H. Press, B. Flannery: *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, 1986.

[15] G. Rodriguez: Kalman filtering, smoothing and recursive robot arm forward and inverse dynamics, *IEEE J. of Robotics and Automation*, Vol. RA-3, No. 6, pp. 624-639, 1987.

[16] C. Rose, B. Guenter, B. Bodenheimer, M. Cohen: Efficient generation of motion transitions using spacetime constraints, *Proceedings of SIGGRAPH '96*, ACM, 1996.

[17] Y. Saad, M.H. Schulz: GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, *SIAM Journal on Scientific and Statistical Computing*, vol. 7, 1986.

[18] D. Tang, J.T. Ngo, J. Marks: N-body spacetime constraints *Journal of Visualization and Computer Animation*, 1995.

[19] A. Witkin, M. Kass: Spacetime constraints, *Proceedings of SIGGRAPH '88*, ACM, 1988.