CS 229 Project:
Motion from Spacetime Constraints

Matt Ahrens (mahrens@cs.brown.edu)
Brett Levin (blevin@cs.brown.edu)

April 19, 2001

1 Abstract

We have implemented a simplified form of the spacetime constraint animation method described by Witkin & Kass, in which constrained optimization techniques are applied to kinematic equations to derive physically plausible motion.[1] To make this a feasible final project, we have foregone the "function box" GUI for specifying the system in favor of a simple text format.

2 Details

Functionally, we've broken the tasks down as follows:

- **Problem specification**. We list the scalar independent variables (S_i), constraint functions ($C_i(S_j)$), and the objective function ($R(S_j)$) to be minimized in a text file, which is parsed at runtime into a symbolic representation.
- **Symbolic differentiation and evaluation** of the above functions, used to aid the optimization process. Witkin & Kass employ a *Sequential Quadratic Programming* technique, which relies upon Jacobian and Hessian derivative matrices of the system.
- **Iterative refinement** of the S_i via SQP or other numeric methods.
- **Visualization** via an Open GL front-end, possibly allowing runtime tweaking of system parameters (e.g. keyframes).

Matt worked mainly on the first two parts, functional representation and functional differentiation and evaluation, which tie together closely. Brett worked mainly on the refinement algorithm and visualization.

Now we'll take a look at the particulars of implementing each aspect.

[1] Andrew Witkin and Michael Kass, *Spacetime Constraints*, Siggraph 1988

2.1 Problem Specification

In the spacetime constraint method, we pose the animation problem in equation form for constrained optimization. We have the following elements, with examples to show our syntax:

- constants—used for things like gravity and keyframes. Example:

```
constants:
xi=0, yi=0, xf=10.0, yf=0, l=5.2, m0, m1, I0, I1, g=9.8;
```

- scalar independent variables—the state variables, such as the positions of and forces on objects at each frame of the animation, that we are trying to solve for. Scalar variables implicitly get indexed by frame, but this process is invisible to the user when writing the problem specification. The user only has to specify the components of the "state vector" of the system, and any per-frame indexing happens automatically. Example:

```
variables:
x, y, a0, a1;
```

- constraint functions—functions on the scalar variables which we drive to zero during the course of optimization in order to enforce as constraints. Constraint functions have a specified *duration*, describing the time range over which they apply. This lets us do things like constrain Luxo's base to the ground over the first and last 10 frames. The text file format supports taking time-index derivatives of scalar variables, in order to get things like velocity (dx/dt) in the form of finite differences. Example:

```
constraints:
[0, n/10]: # following constraints apply to frames [0, n/10)
x - xi;
y - yi;

[0, n]: # following constraints apply to all frames
m0*x0'' + m1*(-(l1*Cos(a0 + a1)* (a0' + a1')^2) -
l1*Sin(a0 + a1)*(a0'' + a1'') + x0'');
```

- objective function—the overarching function which we are trying to minimize. This may be energy usage, fuel consumption, or something of the kind, and may include penalties for things like extreme joint angles. Example:

```
objective:
[0, n]:
# sum of the torques over frames 0 to n
(I0 * a0'')^2 + (I1 * a1'')^2;
```

We parse in the problem specification file using a parser generated by Lex and Yacc. We represent the equations internally a expression trees, which makes it easy to apply algebraic transformations (such as differentiation or simplification) to the expressions.

2.2 Symbolic Differentiation

Given the above functions, we need to be able to find their derivatives with respect to the state variables for use in the linear optimization process. This amounts to differentiating an expression tree with respect to a given variable. We produce the partial derivatives once at startup time, by recursively applying the chain rule for addition, subtraction, multiplication, division, exponentiation, sine, and cosine. Since we are simply recursively applying the chain rule to each node of the expression tree, very large expression trees are generated.

2.3 Algebraic Simplification

Luckily, the large expressions generated by the differentiation process can be easily simplified, as can be seen in this example:

$$\begin{array}{rl} & \frac{d}{dt} l \cdot m + t^2/y - cos(t+m) \\ \Rightarrow & 0 \cdot m + l \cdot 0 + (y \cdot (1 \cdot t^1) - 0 \cdot t^2)/y^2 - 0 - sin(t+m) \cdot (1+0) \\ \Rightarrow & t/y - sin(t+m) \end{array}$$

It was typical to have partial derivatives with hundreds of arithmetic operations. By applying algebraic transformations to these expression trees, we dramatically reduce the number of arithmetic operations required to evaluate these expressions. The simplification process could typically eliminate more than 99% of the arithmetic operations. Due to the nature of the equations, most of the partial derivatives simplified to zero. In the following examples, e represents an expression, and l represents a literal value. The transformations include straightforward simplifications such as:

$$\begin{array}{rcl} 0+e & \Rightarrow & e \\ 0*e & \Rightarrow & 0 \\ 1*e & \Rightarrow & e \end{array}$$

as well as more complex transformations such as:

$$\begin{array}{rcl} l*e_1 + l*e_2 & \Rightarrow & l*(e_1+e_2) \\ 0 - l*e & \Rightarrow & (-l)*e \\ (l_1*e_1)*(l_2*e_2) & \Rightarrow & (l_1*l_2)*(e_1*e_2) \\ (e_1/e_2)*e_3 & \Rightarrow & (e_1*e_3)/e_2 \end{array}$$

and also strength reduction transformations, in which we substitute a (possibly more complicated) expression which is easier for the computer to calculate. For example:

$$\begin{array}{rcl} e^l & \Rightarrow & \prod_0^l e \\ e/l & \Rightarrow & e*(1/l) \end{array}$$

Once we have simplified the derivatives, it is straightforward to determine which derivatives are always zero, so that we can take into account this sparsity information when evaluating the expressions and also when solving the linear systems.

2.4 Expression evaluation

The most straightforward way to evaluate our expression trees is to recursively evaluate each node using an algorithm such as

```
double evaluate(node n) {
        if (n represents a symbol)
            return the current value of n;
        else /* n is a simple operation such as add, multiply, etc */
            return evaluate(n's left child) op evaluate(n's right child)
}
```

However, this type of algorithm is extremely slow, since we must make a function call for each arithmetic operation. To make this faster, we compile the expressions into machine code at run time, which reduces the time required to evaluate the Jacobian by a factor of 10, since each arithmetic operation now simply requires one floating-point instruction. Since no function calls are required (we compute the sine and cosine by a polynomial approximation), we can almost always keep the current state and temporary values in registers, so very few loads and stores are needed.

We use conditional move instructions instead of branching instructions when computing the sine and cosine, so it would be farily straightforward to take into account instruction timing information (such as latency and grouping rules) when scheduling instructions. On the UltraSparc IIi, our simplistic instruction scheduling algorithm achieves approximately 3 cycles per instruction. We estimate that an instruction scheduling algorithm which took into account timing information would be able to achieve at least 1 cycle per instruction.

2.5 Iterative Refinment

Witkin & Kass employ a Sequential Quadratic Programming method which they detail in their paper. The method amounts to finding 1st- and 2nd-order Newton-Raphson steps on the state variables which minimize the constraint and objective functions, respectively. This process is iteratived until a satisfactory solution is found. At the heart of the process is solving two sparse linear systems in sequence. Since the matrices involved can grow very large (the Hessian for example grows as the square of the number of frames), quick solutions to these systems (employing the sparsity patterns of the matrices) are crucial. The original paper uses a conjugate gradient method to solve these systems without having to find matrix inverses; we're currently trying to get this approach working.

2.6 Visualization

While ideally this whole process would be instantaneous, we recognize that this probably won't be the case until we are farther along on various exponential technological development curves. That said, we provide feedback as the refinement algorithm runs, showing intermediate solutions as they are reached and allowing the watchful user to bail out if the refinment process runs amok, numerically speaking.

Since a crucial aspect of the spacetime technique is that time is discretized into frames, and since more frames require more processing, we linearly interpolate the resulting state variables to smooth out motion inbetween frames and allow for reasonable-looking results without a huge number of frames (time samples).

Additionally, our visualizer can display force and torque vectors, which is useful for debugging.

3 Comments

3.1 Appeal

The draw of this technique is its promise to yield physically true animation with little effort on the animator's part by discovering "good" motion implicit in the physical formulation of the system. In the application to Luxo jumping, this method is particularly apt in that the objective function (muscle power spent) reflects the fitness function of energy efficiency that real life organisms have spent millions of years optimizing against.

3.2 Limitations

Despite the magical and serendipitous allure of the promised results, this method suffers some very real drawbacks:

- It's unnatural, from the animator's point of view. Using this approach requires knowledge of how to model the specific system mathematically, as well as knowing when and where to bring ad-hoc techniques into play (e.g. gluing Luxo's base to the ground for the first 3 frames; giving 'style' points in the objective function for a pretty jump; etc.) This reflects the fact that this approach tries to recreate the cause of a given animation — the physical roots of motion — rather than mimicking a desired effect. As a result, this approach inverts the approach taken by most tools for animators, whereby actions (tweaking a keyframe, for example) have direct and predictable results.

- Inflexibile. Changing any system parameters means re-running the optimization process. (This could be partially avoided if all that changes are constants, since the existing solution could be used as the basis for further

SQP iteration. But if the number of frames changes, the entire animation must be scrapped.) Furthermore, there is no notion of branching possibilities: there is no provision for arbitrary collision or interpenetration constraints, for example. Any bodily contact, like Luxo's contact with the floor, needs to be addressed by the arguably ad-hoc approach of furnishing additional constraints, which unavoidably make assumptions about the time duration of a given contact. This is very affirmatively *not* a general physical animation algorithm; the fact that it revolves highly around supplying constraints until the desired behavior is observed is testament to the fact that it is only well suited for animation problems that are highly constrained to begin with.

- Limited realm of application. The idea of formulating the Lagrangian of a system and minimizing its' derivatives with respect to the generalized coordinates does not seem feasible beyond simple articulated rigid bodies. More complex bodies, with more degrees of freedom, seem like they would render this approach computationally intractable.

4 Results

Our system as it stands is capable of parsing in a general syntax (described above) for posing problems of constrained optimization, of calculating arbitrary derivatives of the system, of applying SQP iteration to the system, and of providing visual output for a couple different geometric layouts.

We succesfully used the technique for a few simpler problems, including

- A 2D case where a particle needs to pass through a number of waypoints with a minimal amount of acceleration, given varying degrees of gravity
- Giving various standing poses for Luxo to move between

Despite a wide range of amusing (or disheartening, depending on your perspective) results, we were not able to succeed with our original goal of animating Luxo jumping. We suspect the problem lies either in our implementation of SQP or in our execution of the Lagrangian dynamics of Luxo. We tried to use the Lagrangian technique with a simpler system consisting of a single segment with 3 degrees of freedom (position and orientation), but weren't able to get reasonble results with it, either.