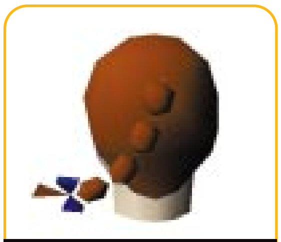
How to Simulate a Ponytail, Part 2

by Chris Hecker

When I was a kid, I used to leave my room — or wherever I went, really — a total mess. I'd like to be able to say that I've changed and am a really tidy and responsible person now, but that would be a stretch. I am, however, going to clean up the awful mess I left us in after the last issue, where we had a bunch of equations, a whole lot of terms, and not much of a clue about what to do to get a ponytail simulator out the other end.



Swinging ponytail screenshot from the sample application used in last month's article.

When We Last Left Our Heroes...

There's no escaping the fact that you need to read last month's part one article ("How to Simulate a Ponytail," March 2000) in order to read this part two. I can't review it in any meaningful way, so I'm just going to set up our initial conditions from the end of last month's article and move on. Figure 1 shows the bodies and notation we're using, and Table 1 contains the equations we ended up with.

In part one, we decided to do the derivation for two constrained bodies first, to keep things manageable, and then later generalize it to the longer chain of bodies that make up the ponytail. At the end of part one, we had written out equations for the linear and angular accelerations of our simple two-body system when they were affected by the constraint force, f_c, as you can see in Equations 1 and 2. We also determined that Equation 3 was going to be the constraint equation we would attempt to satisfy at all times during the simulation using the constraint force. If we could satisfy Equation

Chris Hecker (checker@d6.com) is the Editor-at-Large of Game Developer.

3, and our simulation started with the position and velocity constraints satisfied, we'd have a constrained rigid body simulator. Finally, I said the end product of all the plugging and chugging with equations would be a linear system of equations looking like Equation 4: $Af_c = b$. We'd solve this equation for the force of constraint, and then apply it back to the objects to stick them together.

Plug 'n' Chug

We originally derived Equation 3 because we needed the constraint equation to be in terms of accelerations rather than positions or velocities. Now that we've got it in acceleration space, we can enforce it with f_c, since we know forces can directly affect accelerations. Still, it's not immediately obvious how to get our f_c into Equation 3, where it can do some good.

Equation 3 is too abstract for our needs. It simply says the acceleration of the two constraint endpoints must be equal. It makes sense that the force of constraint, f_c, can affect the accelerations of the endpoints by pushing and pulling on the bodies, but how do we show this mathematically? First, we need to express the endpoint accelerations in terms of the body's linear and angular accelerations, which we know are directly affected by f_c via Equations 1 and 2.

Remember from part one (or from my original physics articles from *Game Developer*, referenced at the end of this article) that the equation for the acceleration of a point fixed on a rigid body — say, Body A — looks like this:

$$\ddot{p}_A = \ddot{R}_A + \alpha_A \times r_A + \omega_A \times (\omega_A \times r_A) \qquad \text{Eq. 5}$$

Equation 5 contains the second derivative of R_A (the vector to the center of mass of the body) and α_A (the angular acceleration of the body). These quantities are definitely affected by f_c as shown in Equations 1 and 2.

If we substitute Equation 5 and its counterpart for Body B into Equation 3, we get a very long equation. Then, if we substitute Equations 1 and 2 and their counterparts for Body B into the very long equation, we get an *extremely* long equation. At that point, our extremely long equation is in terms of our only unknown, f_c, and we can munge it around until we get something that looks like Equation 4. We could do this, but we'd probably go insane trying to keep all the terms straight with all their subscripts and whatnot, and I know I'd go insane trying to type all the intermediate stages into the evil Equation Editor.

We'll take a step back, and just work with Equation 5 for a little while. We can move ahead under the assumption that anything we do to Equation 5, we can do to its Body B partner. If we can simplify Equation 5 before substituting it into Equation 3, then we can do the same for the B version and we'll stay sane.

One Term at a Time

Look at the first term on the right hand side of Equation 5, the acceleration of R_A. Equation 1 just drops into Equation 5 in place of this term, and we get Equation 6:

$$\ddot{p}_A = M_A^{-1} f_c + M_A^{-1} F_{EA} + \alpha_A \times r_A + \omega_A \times (\omega_A \times r_A) \qquad \text{Eq. 6}$$

This is already starting to get messy. We can simplify a bit by introducing a new term, b_A. We'll use b_A to hold all of the "known" terms in the equation. The known terms are those that contain quantities whose values we know how to calculate at any given time. So, as we discussed in part one, the external forces are all known at a given timestep, meaning we can stuff the F_{EA} term into b_A. Also, the last term in Equation 6 is known because it only contains angular velocities and the position vector, r_A, both of which are known at any timestep since they were integrated forward from a previous

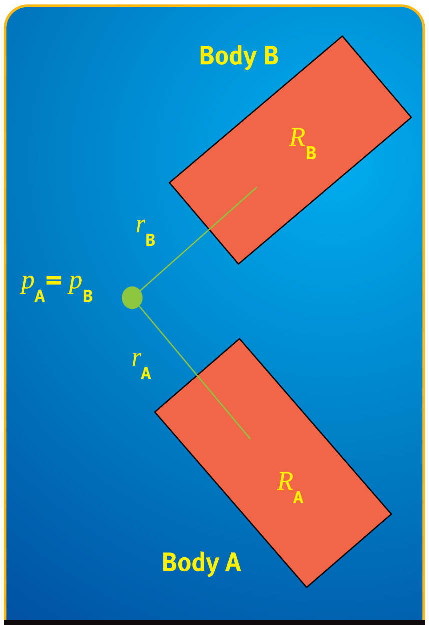


FIGURE 1. *Bodies and notation from last month's article.*

TABLE 1. *Review equations from last month's article.*

ACCELERATION EQUATIONS. Equations 1 and 2 are the linear and angular acceleration equations for the rigid body A in terms of the external forces and torques (denoted with a *E* subscript) and the force of constraint, f_C. Body B's equations would be the same with *B* subscripts and a $-f_C$ in place of the f_C terms because the constraint force is applied negatively to Body B.

$$\ddot{R}_A = M_A^{-1} f_c + M_A^{-1} F_{EA} \qquad \text{Eq. 1}$$

$$\alpha_A = I_A^{-1}(r_A \times f_c) + I_A^{-1}\tau_{EA} - I_A^{-1}(\omega_A \times L_A) \qquad \text{Eq. 2}$$

CONSTRAINT EQUATION. Equation 3 is the second derivative of the position constraint equation, $p_A - p_B = 0$. This equation specifies that the positions (and velocities and accelerations) of the endpoints of the constraint vectors must be equal at all times.

$$\ddot{p}_A - \ddot{p}_B = 0 \qquad \text{Eq. 3}$$

OUR GOAL. At the end of this article, we'd better have a system of linear equations that looks like Equation 4.

$$Af_c = b \qquad \text{Eq. 4}$$

timestep. So, our b_A looks like this so far:

$$b_A = M_A^{-1}F_{EA} + \omega_A \times (\omega_A \times r_A) \qquad \text{Eq. 7}$$

And, our simplified Equation 6 looks like this:

$$\ddot{p}_A = M_A^{-1}f_c + \alpha_A \times r_A + b_A \qquad \text{Eq. 8}$$

The substitution of Equation 2 for α_A is more complicated. First, the α_A is inside a cross product, which means the entire right-hand side of Equation 2 is going to have to go into the first term of that cross product. This makes perfect sense mathematically: α_A is a vector, and so the right-hand side of Equation 2 is a vector as well — it's simply composed of a bunch of other vectors. It's a big mess symbolically, though, because replacing the single symbol on the left-hand side of Equation 2 with the multi-term expression on the right-hand side makes the cross product pretty much unreadable, and we have to use parentheses to keep everything straight.

We'll concentrate solely on the α_A term in Equation 8 and ignore the other terms for a moment. We substitute in Equation 2 and use the fact that the cross product distributes across addition and subtraction:

$$\alpha_A \times r_A = \left[I_A^{-1}(r_a \times f_c)\right]\times r_A + \left[I_A^{-1}\tau_{EA}\right]\times r_A - \left[I_A^{-1}(\omega_A \times L_A)\right]\times r_A \qquad \text{Eq. 9}$$

Now, before dropping Equation 9 back into Equation 8, let's try to stick a bunch of it into b_A to get it out of the way. The first term on the right-hand side of Equation 9 contains f_c, so we need to keep it around, but the other two terms are both known, since they contain only external torques, velocities, momenta, and positions. Away into b_A they go, leaving us with:

$$b_A = M_A^{-1}F_{EA} + \omega_A \times (\omega_A \times r_A) + \left[I_A^{-1}\tau_{EA} - I_A^{-1}(\omega_A \times L_A)\right]\times r_A \qquad \text{Eq. 10}$$

I "un-distributed" the cross product of the two known terms in Equation 9 when writing Equation 10 to make it a bit shorter.

Our equation for the constraint endpoint acceleration now looks like this:

$$\ddot{p}_A = M_A^{-1}f_c + \left[I_A^{-1}(r_A \times f_c)\right]\times r_A + b_A \qquad \text{Eq. 11}$$

As you can see, it's much simpler than it could have been, but we've still got some work to do.

A Breather

Let's take a break and assess our situation. We have Equation 11, which is an equation for the acceleration of Body A's constraint endpoint in terms of the known quantities (most of which are tucked away in b_A), and the unknown constraint force, f_c. That is, at any given time we can calculate the value of the b_A vector, and plug it into Equation 11. We can also calculate all of the other known terms on the right hand side of Equation 11, such as the mass, inertia tensor, and constraint vector, r_A. The exception is f_c; we don't know it in advance. In fact, our whole goal is to solve for f_c so we can plug it back into Equations 1 and 2 to find the acceleration of the bodies under constraint.

Since f_c is our unknown, we need to get it in a better position to be manipulated. The first term on the right-hand side of Equation 11 is pretty reasonable, since it's just a matrix times f_c. This term looks a bit like Equation 4, so we know we're getting close. However, f_c is stuck inside two cross products in the second term, which is a far cry from Equation 4.

Cross Products

How do we get f_c out of the cross products? Cross products are notoriously hard to manipulate, unless you have the following definitions in your bag of tricks:

$$a \times b = -b \times a \qquad \text{Eq. 12}$$

$$a \times b = \tilde{a}b = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix}\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \qquad \text{Eq. 13}$$

Equation 12 is the familiar rule stating that when you reverse the cross product terms, the resulting vector is negated. This is what you see when you accidentally compute a triangle's normal by crossing the edges in the wrong direction — you get the inverted normal.

Equation 13 defines the "tilde operator," which, when applied to a vector, creates the matrix shown in the equation. It just so happens that this tilde-matrix of vector *a* — also called the "skew symmetric matrix of *a*" — times vector *b* gives the same resulting vector as taking the cross product of *a* and *b* (multiply the matrix-vector product on a piece of paper to double-check it for yourself). This is a great trick, because it turns a cross product into a matrix-vector product, which we know how to manipulate.

Now we can pound on our cross product term. First, let's get f_c on the right side of the expression by applying Equation 12:

$$\left[I_A^{-1}(r_A \times f_c)\right]\times r_A = -r_A \times \left[I_A^{-1}(r_A \times f_c)\right]$$

Next, we "tilde-ize" the outer cross product:

$$-r_A \times \left[I_A^{-1}(r_A \times f_c)\right] = \tilde{r}_A I_A^{-1}(r_A \times f_C)$$

Notice how the outer brackets aren't needed anymore, because now we just have a matrix multiply of the tilde'd r_A and the inverse inertia tensor. Finally, let's tilde-ize the inner cross product:

$$-\tilde{r}_A I_A^{-1}(r_A \times f_c) = -\tilde{r}_A I_A^{-1}\tilde{r}_A f_c$$

Our result is simply three matrices times our unknown vector. Let's put this result back into Equation 11 and group the terms to isolate f_c:

$$\ddot{p}_A = M_A^{-1}f_c - \tilde{r}_A I_A^{-1}\tilde{r}_A f_c + b_A$$

$$\ddot{p}_A = \left[M_A^{-1} - \tilde{r}_A I_A^{-1}\tilde{r}_A\right]f_c + b_A$$

$$\ddot{p}_A = \mathrm{A}_A f_c + b_A \qquad \text{Eq. 14}$$

Now we're in business. I've renamed the grouped matrices "A_A" to highlight the structure of the equations. We have a known matrix, A_A, and a known vector, b_A, and our unknown f_c. We're not quite at Equation 4, but we're extremely close. We only need to get rid of the p_A acceleration term, and that's our cue to substitute back into Equation 3.

Body B

If you look at all the work we did to get from Equation 5 to Equation 14, you'll see that very little of it depended on whether we were talking about Body A or Body B. There's really just one difference between the bodies, and we've already mentioned it: f_c is positive for Body A, but negative for Body B. This means we can simply rewrite all the equations with B subscripts, and if we're careful to substitute in $-f_c$ wherever f_c shows up, we'll have valid equations for Body B. We don't have to rederive everything.

We can actually just write the Equation 14 for Body B by inspection:

$$\ddot{p}_B = A_B(-f_c) + b_B = -A_B f_c + b_B \qquad \text{Eq. 15}$$

The A_B matrix and the b_B vector are calculated exactly as shown above for Body A, and we simply negate the f_c term. Equation 3 tells us we can subtract Equation 15 from Equation 14 to get 0, assuming we're enforcing our constraint properly. Let's write the subtraction, taking care to get our signs right:

$$\ddot{p}_A - \ddot{p}_B = A_A f_c + b_A + A_B f_c - b_B = 0$$

Finally, let's group our terms to match Equation 3:

$$[A_A + A_B] f_c = -b_A + b_B \qquad \text{Eq. 16}$$

That's it. We have a linear system matching Equation 4, where $A = A_A + A_B$ and $b = -b_A + b_B$. A and b are known, and f_c is unknown, meaning that at any given time, we can construct Equation 16 for the two rigid bodies, and then solve it for f_c to find the constraint force that will hold the bodies together.

The Simulation Algorithm

Now we're ready to outline the overall simulation algorithm for two rigid bodies with one constraint:

1. Compute the external forces: F_{EA}, τ_{EA}, F_{EB}, τ_{EB}.
2. Compute the A matrix and the b vector.
3. Solve the linear system for f_c.
4. Apply f_c and the external forces to the objects.
5. Integrate forward.

After step three, f_c is a known force, just like the external forces. The forces can now all be applied to the bodies in the usual way. Remember, f_c is applied at the tip of the constraint vector, so it will induce torque on the objects as well as apply a force to the center of mass. Also remember that f_c is applied negatively to Body B.

The Linear System

So, how do we solve the $Af_c = b$ linear system? This is actually the easiest part of the algorithm in some sense, because there are so many different ways to do it. Solving linear systems on computers is the most studied area of numerical analysis, and there are hundreds of books about doing it right and lots of free source code. You could probably write your own Gaussian Elimination routine in a few lines of C, or you could download some fancy numerical linear algebra package if you're so inclined. If you're just interested in solving the 3×3 system in Equation 16, you could even use Cramer's Rule or just solve the system by hand symbolically, but those techniques won't scale to the larger systems that occur when we add bodies. When I wrote the sample application for this article, I downloaded a simple linear system solver from the web (http://www.netlib.org) and hacked it into my program. See the references for details on the sample application.

Kinematic Control

Before we generalize our derivation to multiple bodies, let's talk about how kinematic control fits into our formulation. From last month, you'll remember the head of the character is kinematically controlled, meaning its movements are already known from an animation. It's easy to integrate animated bodies into our algorithm.

First, we need to be able to generate values for the known position, velocity, and acceleration of the kinematically controlled body at a given point in time. We can find the equations for these values from our animation system in most cases. The position equation is simplest — we have to have that around if we're animating the body in the first place. The velocity and acceleration equations are attained by differentiating the equation for position. If we're interpolating keyframes, then the interpolation function will give us the velocity at a given time when we differentiate it. Another differentiation gives us the acceleration. For example, if we're linearly interpolating positions between keyframes, the velocity will be constant and the acceleration will be zero. Linear interpolation is not continuous at the keyframes themselves because the direction changes sharply, so be careful about differentiating in those areas. If we're linearly interpolating joint angle keyframes, our velocities and accelerations will be nonlinear, but still derivable from the equations. If we're doing a more sophisticated continuous spline interpolation, our derivatives will be even more complicated, but we should still be able to attain equations for the velocity and acceleration.

Once we've gotten the linear and angular positions, velocities, and accelerations from our animation system, we use these known values everywhere they appear in our equations. Equations 1 and 2 for the kinematically controlled body become known values, rather than equations depending on the force of constraint. All of the animating body's kinematic quantities are now known and end up in the b vector. When we compute f_c, the computed force will make the dynamically controlled body obey the movement of the constraint due to the kinematically controlled body's animation. We don't apply the constraint force to the animating body because, well, it's animated, not simulated.

Multiple Constraints

Two bodies does not a ponytail make. Do we have to rederive everything when we want to do three bodies with two constraints in a chain, not to mention N bodies with $N - 1$ constraints? Thankfully, the answer is no. The equations for multiple bodies and constraints are very similar to those we've already derived, but we need to talk a bit more about the structure of the multi-constraint problem before we can extend them.

The first thing to notice is that we have to solve for all of the constraint forces simultaneously. If I have a chain of three bodies, and I pull up on the top body, not only does the middle body have to feel the yank, but the bottom body does as well. If we didn't solve simultaneously, the force of the pull would ripple down the chain in the order we solved the constraints, and the chain would separate. This is not the behavior we want.

Because we need to solve simultaneously, all of the constraints need to be represented in the equations we write. This forces us to develop some new notation that will scale to multiple constraints. Bodies can now have two constraints attached to them, rather than just having one as in our two-body derivation. It turns out that it makes the most sense to be "constraint-centric" in our notation, numbering the constraints and having them refer to the bodies rather than having the bodies refer to the constraints.

Figure 2 shows this notation. The constraints are numbered 1, 2, and 3; if we were being completely general we'd call them $i - 1$, i, and $i + 1$. We're going to talk about the middle joint, number 2. We'll call the forces at each constraint f_{c1}, f_{c2}, and f_{c3}. The constraints attach the two bodies on either side of the joint, and I've chosen the subscripts u and d to stand for the "upstairs" body and the "downstairs" body relative to the joint in the figure. So, the body between joints 1 and 2 is the upstairs body of joint 2, and the other body is the downstairs body of joint 2. The constraint endpoint of the top body for joint 2 is denoted p_{2u}, and the endpoint from the bottom body is p_{2d}, and so on. The constraint endpoints and the r vectors are still attached to their respective bodies, but they're numbered relative to the joints. Notice that the u-body of joint 2 is the d-body of joint 1.

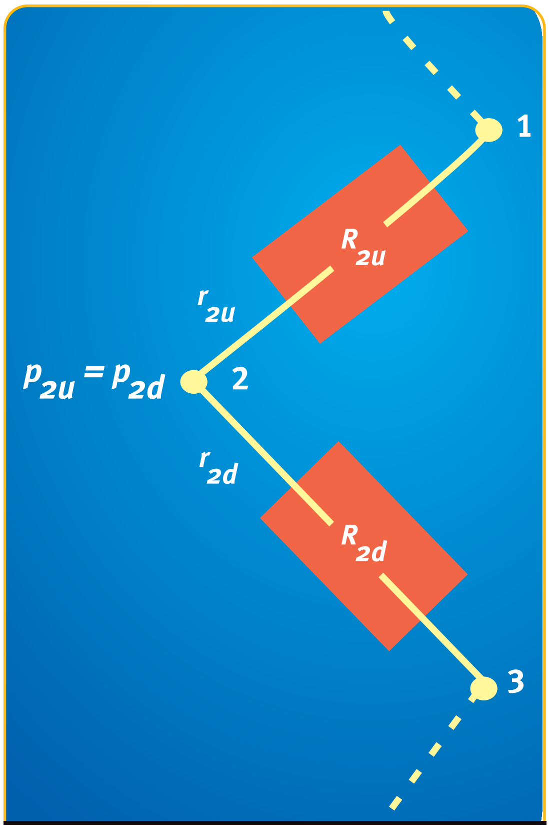


FIGURE 2. *A multi-constraint system.*

While I make no claims to the elegance of this notation, it will let us get the job done.

General Acceleration Equations

Given the new notation, we could rederive all of our equations for the new general constraint. We don't have the space for that (and it's almost identical to our previous derivation), so we're going to skip ahead and show the structure of the equations we end up with for joint 2. The accelerations of the two endpoints associated with joint 2 look like this:

$$\ddot{p}_{2u} = A_{2u2} f_{c2} - A_{2u1} f_{c1} + b_{2u} \qquad \text{Eq. 17}$$

$$\ddot{p}_{2d} = -A_{2d2} f_{c2} + A_{2d3} f_{c3} + b_{2d} \qquad \text{Eq. 18}$$

A single joint is affected not only by its own constraint force, but also by the constraint forces on either side of it. This is because the body accelerations are modified by all the constraint forces acting on them, and those body accelerations appear in the equation for the constraint. Put another way, the top body's motion at joint 2 is dependent on what joint 1's force is doing, in addition to what joint 2's force is doing.

I haven't described what the A matrices look like exactly, but they'll be very similar in composition to the A matrices we derived above, so we can just deal with them symbolically here. They're subscripted to describe their function: A_{2u1} is the matrix for joint 2's upstairs body that multiplies f_{c1}. In English, A_{2u1} describes the acceleration effect f_{c1} has on joint 2's upstairs endpoint. Put yet another way, A_{2u1} maps the force from joint 1 to an acceleration at joint 2, through the body. To belabor the point a bit more, if you look at the expression that makes up A_{2u1} (once you've derived it, of course!), you'll see that it converts f_{c1} to accelerations on the center of mass, and then maps those accelerations out to p_{2u}.

Although I didn't mention this way of thinking above, the original A_A and A_B matrices from the two-body derivation work the same way. The linear acceleration is transferred through the M^{-1} term, and the angular acceleration is transferred through the cross product (or tilde matrix) and inertia tensor term. In A_A and A_B we're mapping the constraint force from the joint, down through the body, and back up to the same joint, but the principle still applies. In Equations 17 and 18, A_{2u2} and A_{2d2} are similar to A_A and A_B, since they map f_{c2} to acceleration back at joint 2.

We've also adopted the convention of applying the constraint force positively to the *u*-body and negatively to the *d*-body, which accounts for the negative signs in Equations 17 and 18.

The Multi-Constraint System

If we subtract Equation 18 from Equation 17 and group terms, we get the constraint equation we must satisfy for joint 2:

$$-A_{2u1}f_{c1} + \left[A_{2u2} + A_{2d2}\right]f_{c2} - A_{2d3}f_{c3} = -b_{2u} + b_{2d}$$

Eq. 19

This equation has the three unknown vectors in it, but it's only one vector equation. To solve a linear system, we need as many equations as we have unknowns. Where will we find the other equations? From the other constraints, naturally.

Let's assume for the moment that the system in Figure 2 has four bodies and the three constraints shown. In other words, although they're only hinted at in the figure, there is a body above joint 1 and a body below joint 3. These outlying bodies each has only one constraint (joint 1 for the upper body and joint 3 for the lower body, obviously), and they are the endpoints of the chain in this example. Given this system, the equation for joint 1 is:

$$\left[A_{1u1} + A_{1d1}\right]f_{c1} - A_{1d2}f_{c2} = -b_{1u} + b_{1d}$$

Eq. 20

And the equation for joint 3 is:

$$-A_{3u2}f_{c2} + \left[A_{3u3} + A_{3d3}\right]f_{c3} = -b_{3u} + b_{3d}$$

Eq. 21

Notice that Equations 20 and 21 have only two constraint forces each in them, as opposed to the three forces in Equation 19. The equation for the end joints in a chain will have only two constraint forces because the extremal bodies don't have a joint on their "far" side (or they wouldn't be very extremal, now would they?).

Now for a bit of matrix magic. Equation 20 contains f_{c1} and f_{c2}, Equation 19 contains f_{c1}, f_{c2}, and f_{c3}, and Equation 21 contains f_{c2} and f_{c3}. Each of these equations depends on one or more of the other ones. This is the mathematical expression of our statement above that the constraints must be solved simultaneously. We can construct a single large matrix equation that contains all of these equations by stacking them up, like so:

$$\begin{vmatrix} A_{1u1} + A_{1d1} & -A_{1d2} & 0 \\ -A_{2u1} & A_{2u2} + A_{2d2} & -A_{2d3} \\ 0 & -A_{3u2} & A_{3u3} + A_{3d3} \end{vmatrix} \begin{vmatrix} f_{c1} \\ f_{c2} \\ f_{c3} \end{vmatrix} = \begin{vmatrix} -b_{1u} + b_{1d} \\ -b_{2u} + b_{2d} \\ -b_{3u} + b_{3d} \end{vmatrix}$$

Eq. 22

If you perform the matrix multiply in Equation 22, you can see you get the exact equations listed above. Furthermore, Equation 22 is just another $Af_c = b$ linear system, where A now stands for the compound matrix in Equation 22, and f_c and *b* (with no numbered subscripts) stand for the stacked vectors. Instead of a 3×3 system, we now have a 9×9 system, but it's still a linear system and the same rules apply to solving it. Throw Equation 22 into a linear solver, apply the individual f_c vectors back to their appropriate objects, and you've got a constrained system.

From here, it should be pretty clear how to extend this math to an arbitrary number of bodies and constraints. The A matrix and the associated vectors keep growing, but the structure is exactly the same.

The Linear System Revisited

I said you can solve the $Af_c = b$ system using a generic linear solver, which is true. However, there are more efficient ways of solving the particular matrix generated by our algorithm that take advantage of its special properties. Efficiency is incredibly important when doing constrained dynamics because linear systems such as $Af_c = b$ have $O(n^3)$ complexity in the general case, where *n* is the number of rows in the matrix. This means that every constraint equation you add as an additional row makes the system much slower to solve. $O(n^3)$ complexity is not the kind of slowness that waiting for next year's CPU can fix.

The most important special characteristic of our A matrix is its sparsity structure. You can see this structure developing in Equation 22, and as you add more bodies and constraints you can see it even better: the constraint submatrices stay on the diagonal of the matrix and its neighboring columns, and the rest of the matrix is zero. This makes intuitive sense given that a constraint depends only on itself (which corresponds to the diagonal element) and its two constraint neighbors (the off-diagonal elements). The official name for the sparsity structure of the A matrix is "block tridiagonal," for somewhat obvious reasons. What's more, the A matrix is symmetric, although this fact is not completely clear from our derivation. And finally, it's "positive definite," assuming the constraint equations are well formed. A positive definite matrix is roughly analogous to restricting a real number to be greater than zero, rather than allowing it to be zero or negative. You can learn more about these characteristics in a good numerical linear algebra book.

Taken together, these properties mean we can write (or download) a custom linear solver that will solve our systems in $O(n)$ time. $O(n)$ is definitely the kind of problem that AMD and Intel will make faster every year.

Numerical Accuracy

It's really a shame that all of this math I've presented to you doesn't actually work when you type it into the computer. Well, that's a bit extreme, but the world of floating-point numerics is far removed from that of symbolic equations, and we have to do a bit more work to get them to match up.

The major problem is that we're using numerically computed forces to affect accelerations, which are then numerically integrated to find new positions. This algorithm has two big numerical holes in it. For starters, the forces we compute are not going to be exact because of floating-point errors accumulated during the formulation of the $Af_c = b$ system and its solution. This means the f_c terms aren't going to exactly enforce the constraint equations when they're applied in floating point. To compound matters, the integrator is going to introduce even more numerical errors, since we're using forces to keep a position constraint together. These two sources of error mean the objects will slowly drift apart. At first the errors will be small. If you subtract the positions of the two endpoints of a constraint, you'll see the result is not exactly the zero vector after a few steps. Eventually, the objects will have drifted far enough apart so you can see the gap. This is bad.

There are many ways of dealing with this drift, and we don't have the space to talk about any of them in depth. I favor a method called Baumgarte Stabilization, which basically places tiny springs on the joints that are adjusted to suck up the numerical error as it develops. The springs don't actually provide any physical support (the f_c terms still do that), but they do a great job of keeping the joints together in the face of floating-point errors. The sample application implements Baumgarte Stabilization to fight the drift problem. It's easy to implement and it works well.

Other methods include directly correcting for the drift in position space, and other techniques. Now that you know the math behind constrained dynamics, you'll have no trouble following the numerical accuracy discussions in the books referenced on my web site.

Miscellanea

Phew! That's a lot of equations, but we've accomplished a lot. We've actually accomplished even more than we set out to, because all of this math is valid for completely general constraints. When I say general, I mean it in two ways: topologically and in terms of the joint types.

Topologically, our constraint-centric viewpoint means we can have as many constraints coming off a rigid body as we like. We'll need to modify our notation slightly to support this, and the location of the elements in the Equation 22 matrix will change a bit depending on the interconnection of the bodies, but making an octopus would be no problem, even though the root body has eight constraints on it.

As far as joint types go, Equation 3 is just one of an infinite number of acceleration constraints this math can enforce. Again, pieces of the derivation change, but the overall structure stays the same, regardless of whether you're simulating spherical joints such as Equation 3, or hinges, or prismatics, or whatever. Look at the references on my web site for books about writing different constraint equations, or give it a try yourself. The important thing to remember is to get it clear in your head what you'd like the joint to be, and then write down an equation that describes that joint. Differentiate it then plug and chug.

Hopefully, with the general interest in physics for games that's been growing for the past few years, we'll start to see a lot of special effects using real, consistent physics. Then, when everybody's comfortable with the math and implementation issues, we can start to work on integrating physics with game play and game physics can finally live up to its potential. ■

FOR FURTHER INFO

My dynamics web site, including the ponytail sample app, articles, references, and more:

http://www.d6.com/users/checker/dynamics.htm

Acknowledgements

I forgot last time to thank Lisa Washburn of Vector Graphics (http://www.vectorg.com) for the model of the head and the ponytail pieces.