Modelling and Solving Physical Constraints

Not Me

7 Dec 2013

THIS WAS TAKEN FROM AN ARTICLE BY RANDY GAUL ON tutsplus.com, I AM NOT THE AUTHOR. tustplus.com's LATEX FORMATING WAS BROKEN AT THE TIME SO I MADE A COPY HERE TO MAKE IT EASIER FOR MYSELF TO REFER TO. THE ORIGINAL ARTICLE CAN BE FOUND HERE

In general form, a constraint is a scalar equation equal to some value (usually zero).

$$C(l_1, a_1, l_2, a_2) = 0 \tag{1}$$

The l and a terms in (1) are my own notation: l refers to linear while a refers to angular. The subscripts 1 and 2 refer to the two objects within the constraint. As you can see, there exist linear and angular inputs to a constraint equation, and each must be a scalar value.

Let's take a step back to look at the distance constraint. The distance constraint wants to drive the distance between two anchor points on two bodies to be equal to some scalar value:

$$C(l_1, a_1, l_2, a_2) = \frac{1}{2}[(P_2 - P_1)^2 - L^2] = 0 \tag{2}$$

L is the length of the rod connecting both bodies; P_1 and P_2 are the positions of the two bodies.

In its current form, this constraint is an equation of position. This sort of position equation is non-linear, which makes solving it very hard. A method of solving this equation can be to instead derive the position constraint (with respect to time) and use a velocity constraint. Resulting velocity equations are linear, making them solvable. Solutions can then be integrated using some sort of integrator back into positional form.

In general form, a velocity constraint is of the form:

$$\dot{C}(l_1, a_1, l_2, a_2) = 0 \tag{3}$$

The dot above the C in (3) refers to the derivative of C with respect to time. This is common notation when dealing with the study of physics.

During the derivative, a new term J appears via chain rule:

$$\dot{C}(l_1, a_1, l_2, a_2) = JV = 0 \tag{4}$$

The time derivative of C creates a velocity vector and a Jacobian. The Jacobian is a 1x6 matrix containing scalar values corresponding to each degree of freedom. In a pairwise constraint, a Jacobian will typically contain 12 elements (enough to contain the l and a terms for both bodies A and B.

A system of constraints can form a joint. A joint can contain many constraints restricting degrees of freedom in various ways. In this case, the Jacobian will be a matrix where the number of rows is equal to the number of constraints active in the system.

The Jacobian is derived offline, by hand. Once a Jacobian is acquired, code to compute and use the Jacobian can be created. As you can see from (4), the velocity V is transformed from Cartesian space to constraint space. This is important because in constraint space the origin is known. In fact, any target can be known. This means that any constraint can be derived to yield a Jacobian that can transform forces from Cartesian space to constraint space.

In constraint space, given a target scalar, the equation can move either towards or away from the target. Solutions can easily be obtained in constraint space to move the current state of a rigid body towards a target state. These solutions can then be transformed out of constraint space back into Cartesian space like so:

$$F = \lambda J^T \tag{5}$$

F is a force in Cartesian space, where J^T is the inverse (transposed) Jacobian. λ (lambda) is a scalar multiplier.

Think of the Jacobian as a velocity vector, where each row is a vector itself (of two scalar values in 2D, and three scalar values in 3D):

$$J = \begin{bmatrix} l_1 \\ a_1 \\ l_2 \\ a_2 \end{bmatrix} \tag{6}$$

To multiply V by J mathematically would involve matrix multiplication. However, most elements are zero, and this is why we treat the Jacobian as a vector. This allows us to define our own operation for computing JV, as in (4).

$$JV = \begin{bmatrix} l_1 & a_1 & l_2 & a_2 \end{bmatrix} \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} \tag{7}$$

Here, v represents linear velocity, and ω represents angular velocity. (7) can be written down as a few dot products and multiplications in order to provide a more efficient computation compared to full matrix multiplication:

$$JV = l_1 \cdot v_1 + a_1 \cdot \omega_1 + l_2 \cdot v_2 + a_2 \cdot \omega_2 \tag{8}$$

The Jacobian can be thought of as a direction vector in constraint space. This direction always points towards the target in the direction that requires the least work to be done. Since this "direction" Jacobian is derived offline, all that needs to be solved for is the magnitude of the force to be applied in order to uphold the constraint. This magnitude is called λ. λ can be known as the Lagrange Multiplier. I myself have not formally studied Lagrangian Mechanics, however a study of Lagrangian Mechanics is not necessary in order to simply implement constraints. (I am proof of that!) λ can be solved using a constraint solver (more on this later). Solving for Jacobians

In Erin Catto's paper, there exists a simple outline for hand-deriving Jacobians. The steps are:

Start with constraint equation C Compute time derivative $\dot{C}$ Isolate all velocity terms Identify J by inspection

The only hard part is computing the derivative, and this can come with practice. In general, hand-deriving constraints is difficult, but gets easier with time.

Let's derive a valid Jacobian for use in solving a distance constraint. We can start at Step 1 with (2). Here are some details for Step 2:

$$\dot{C} = (P_2 - P_1)(\dot{P}_2 - \dot{P}_1) \tag{9}$$

$$\dot{C} = (P_2 - P_1)((v_2 + \omega_2 \times r_2) - (v_1 + \omega_1 \times r_1)) \tag{10}$$

r_1 and r_2 are vectors from the center of mass to the anchor point, for bodies 1 and 2 respectively.

The next step is to isolate the velocity terms. To do this, we'll make use of the scalar triple product identity:

$$(P_2 - P_1) = d \tag{11}$$

$$\dot{C} = (d \cdot v_2 + d \cdot \omega_2 \times r_2) - (d \cdot v_1 + d \cdot \omega_1 \times r_1) \tag{12}$$

$$\dot{C} = (d \cdot v_2 + \omega_2 \cdot r_2 \times d) - (d \cdot v_1 + \omega_1 \cdot r_1 \times d) \tag{13}$$

The last step is to identify the Jacobian by inspection. In order to do this, all the coefficients of all velocity terms (V and ω) will be used as the Jacobian elements. Therefore:

$$J = \begin{bmatrix} -d & -r_1 \times d & d & r_2 \times d \end{bmatrix} \tag{14}$$

Some more Jacobians

Contact constraint (interpenetration constraint), where n is the contact normal:

$$J = \begin{bmatrix} -n & -r_1 \times n & n & r_2 \times n \end{bmatrix} \tag{15}$$

Friction constraint (active during penetration), where t is an axis of friction (2D has one axis, 3D has two):

$$J = \begin{bmatrix} -t & -r_1 \times t & t & r_2 \times t \end{bmatrix} \tag{16}$$

Solving Constraints

Now that we have an understanding of what a constraint is, we can talk about how to solve them. As stated earlier, once a Jacobian is hand-derived, we only need to solve for λ. Solving a single constraint in isolation is easy, but solving many constraints simultaneously is hard, and very inefficient (computationally). This poses a problem, as games and simulations will likely want to have many constraints active all at once.

An alternative method to solving all constraints simultaneously (globally solve) would be to solve the constraints iteratively. By solving for approximations of the solution, and feeding in previous solutions to the equations, we can converge on the solution.

One such iterative solver is known as Sequential Impulses, as dubbed by Erin Catto. Sequential Impulses is very similar to Projected Gauss Seidel. The idea is to solve all constraints, one at a time, multiple times. The solutions will invalidate each other, but over many iterations each individual constraint will converge and a global solution can be achieved. This is good! Iterative solvers are fast.

Once a solution is achieved, an impulse can be applied to both bodies in the constraint in order to enforce the constraint.

To solve a single constraint, we can use the following equation:

$$\lambda = \frac{-(JV + b)}{JM^{-1}J^T} \tag{17}$$

M^{-1} is the mass of the constraint; b is the bias (more on this later).

This is a matrix containing the inverse mass and inverse inertia of both rigid bodies in the constraint. The following is the mass of the constraint; note that m^{-1} is the inverse mass of a body, while I^{-1} is the inverse inertia of a body:

$$M^{-1} = \begin{bmatrix} m_1^{-1} & 0 & 0 & 0 \\ 0 & I_1^{-1} & 0 & 0 \\ 0 & 0 & m_2^{-1} & 0 \\ 0 & 0 & 0 & I_2^{-1} \end{bmatrix} \tag{18}$$

Although M^{-1} is theoretically a matrix, please do not actually model it as such (most of it is zeroes!). Instead, be smart about what sort of calculations you do.

$JM^{-1}J^T$ is known as the constraint mass. This term is calculated one time and used to solve for λ. We calculate it for a system like so:

$$JM^{-1}J^T = (l_1 \cdot l_1) * m_1^{-1} + (l_2 \cdot l_2) * m_2^{-1} + a_1 * (I_1^{-1} a_1) + a_2 * (I_2^{-1} a_2) \tag{19}$$

Please note that you must invert (19) in order to compute (17).

The above information is all that is needed to solve a constraint! A force in Cartesian space can be solved for and used to update the velocity of an object, in order to enforce a constraint. Please recall (5):

$$F = \lambda J^T V_{final} = V_{initial} + m^{-1} * F \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} += \begin{bmatrix} m_1^{-1} & 0 & 0 & 0 \\ 0 & I_1^{-1} & 0 & 0 \\ 0 & 0 & m_2^{-1} & 0 \\ 0 & 0 & 0 & I_2^{-1} \end{bmatrix} \begin{bmatrix} \lambda * l_1 \\ \lambda * a_1 \\ \lambda * l_2 \\ \lambda * a_2 \end{bmatrix} \tag{20}$$

Constraint Drift

Due to the linearization of the non-linear position equations, some information is lost. This results in solutions that don't quite satisfy the original position equation, but do satisfy the velocity

equations. This error is known as constraint drift. One can think of this error as the result of a tangent line approximation.

There are a few different ways to solve such errors, all of which approximate the error and apply some form of correction. The simplest is known as Baumgarte.

Baumgarte is a small addition of energy in constraint space, and accounts for the b term in the previous equations. To account for bias, here is a modified version of (4):

$$\dot{C} = JV + b = 0 \tag{21}$$

To calculate a Baumgarte term and apply it as a bias, we must inspect the original constraint equation and identify a suitable method for calculating error. Baumgarte is in the form:

$$JV = -\beta C \tag{22}$$

β (Baumgarte term) is a tunable, unit-less, simulation-dependent factor. It is usually between 0.1 and 0.3.

In order to calculate the bias term, let's look at the equation for the non-penetration constraint (15) before deriving with respect to time, where n is the contact normal:

$$C = \begin{bmatrix} -x_1 & -r_1 & x_2 & r_2 \end{bmatrix} \cdot \vec{n} \tag{23}$$

The above equation is saying that the scalar error of C is the inter-penetration depth between two rigid bodies.