A Robot Soccer Simulator:
A Case Study for Rigid Body Contact

Eric Larsen
Sony Computer Entertainment America R&D

1 Introduction

This paper focuses on contact problems of rigid body simulation, mainly how to model resting contact with spring-damper penalty methods and semi-implicit integration. My main demo project was a “Robot Soccer Simulator”, which simulates simple wheeled robots, based on the "small-size league" robots of Robocup.

I'm not an expert in rigid-body simulation, but I felt justified in writing down some of my experiences with the subject. I've often wished for more papers that focused on implementation problems, or that spent a little more time giving background information. So writing this paper is an attempt to create a resource that I would have wanted to read a long time ago. Therefore, I try to be honest and not spare embarrassing details when helpful.

Before reading this, you should be familiar with David Baraff’s notes on rigid-body simulation from the SIGGRAPH 97 course notes [Bar97]. Brian Mirtich's Ph.D. thesis [Mir PhD] on impulse-based simulation also introduces some of the concepts and terminology I use.

I was inspired to try penalty methods with implicit integration after a talk by David Wu on the subject, at the Dec. 1999 GDC Hardcore Technical Seminar. I didn’t however get detailed notes on his approach. By the time I started my project, I found the *Implementation Details* section of Brian Mirtich’s *Timewarp Rigid Body Simulation* paper [Mir00], which gave a few concrete details and pointers to information. I ended up following this paper closely. Many thanks to Brian Mirtich for giving me some extra hints when I needed them.

2 Penalty Methods *vs.* Analytical Force Solving

In choosing to implement a penalty method, I skipped a popular alternative, analytical force solving. I’ll give a brief summary of each, ignoring friction for now. But please don’t use only this paper to decide what to try. You should definitely do your own survey of simulation references.

2.1 Analytical Force Solving

Assume that you have a set of rigid bodies, that you have defined points of contact between the ones that touch, and that at each contact, either the relative normal velocity is positive (meaning the bodies are separating at the contact), or zero (meaning the bodies rest at the contact). Contacts with zero relative normal velocity are usually called *resting contacts*.

Forces such as gravity can result in negative relative normal acceleration at resting contacts, so you want to apply forces at the resting contacts to prevent bodies from interpenetrating. However, there are some constraints on the forces you can apply and the accelerations that can result. For instance, the resulting normal acceleration at each contact must be positive and the normal force zero, or else the normal acceleration must be zero and the normal force positive. In other words, there should be no force if the bodies are separating at the contact; otherwise, you should apply enough repulsive force that the bodies stay at rest at the contact point.

For a configuration of resting rigid bodies and contact points, a matrix relates the set of contact forces to the accelerations that result at the contacts. A *force solver* must find forces and accelerations that are related by this matrix, and meet the desired constraints. One method poses the problem as a Linear Complimentary Problem (LCP). Baraff offered a simple LCP solution method in [Bar94], although simple is a relative term – the other LCP papers I've seen are very math intensive. (It does help to read [Bar97], which derives the *A* matrix mentioned in the paper for the frictionless case.) Baraff has also formulated the problem as a Quadratic Programming problem [Bar89]. Regardless of the solution method, you obtain a set of contact forces, then employ an integrator to simulate the resulting motion of the bodies, which results in resting or separating contacts at the end of the timestep.

But there were warnings at the GDC seminar about force solvers: exponential running times or breakdowns in certain cases. So, by the end of the seminar, I was receptive to the talk by David Wu on penalty methods with implicit integration.

2.2 Penalty Methods

With a penalty method, to prevent penetrations at contact points, you define a penalty force function that increases with penetration. A common choice is a spring-damper force that acts only in the normal direction. If the bodies separate at the contact, you break the spring-damper, but if they penetrate at the contact, the spring-damper force resists.

On first glance, this approach seems a lot simpler than analytical force solving. Each contact force is computed separately by a simple formula, so there is no LCP to worry about. But one of the prices you pay is less rigid behavior – bodies will always penetrate at least a little. Also, to make the penetrations small, you have to use stiff spring-dampers, but typical (explicit) integrators can have unstable behavior integrating stiff spring-dampers. This means you are likely to see bodies jump into the air, or even rocket off screen. An implicit or semi-implicit integrator can address this problem, but these do involve much more effort than an explicit integrator.

3 Impulses

I should address one of the loose ends I left above. How do you get only positive or zero relative normal velocities at all contact points? Negative relative normal velocities indicate a collision between bodies at a contact, and to prevent the bodies from interpenetrating, you need to reverse or nullify the velocity at the contact point. Such direct changes to velocity are made with impulses, not forces, so your simulator will also need an impulse system.

A simple impulse system addresses one contact at a time. When you detect a negative relative normal velocity between two bodies, you apply an impulse there. If relative normal velocity at a contact falls below some threshold, you can also use an impulse to cancel it out completely, to make it a true resting contact. A good reference on computing impulses at a contact is [Cha] (check out [Mir PhD] for an explanation of the *collision matrix*).

The problem is, applying an impulse to correct velocity at one contact disrupts the velocity at other contacts. One solution is to propagate collisions through a series of contacting bodies. The even simpler solution that I used was to keep applying impulses at all contact points until all relative normal velocities are greater than some small negative threshold.

If propagation approaches can only give you relative normal velocities above a negative threshold, rather than strictly zero, how do you keep bodies from gradually moving into one another? If you're using an analytical force solving method, you will already need to implement a "stabilization" method to handle drift in contact positions and velocities. My guess is that canceling out the small velocities after impulse propagation falls inside this domain. As for a spring-damper approach, this is a lesser concern – spring-dampers automatically stabilize position and velocity.

However, another impulse approach is similar to force-solving, and solves for collision impulses that produce zero and positive normal velocities. [Bar89] discusses this idea, and [Kaw] gives details on using Baraff's LCP solver to find a set of valid impulses. Apparently though, analytical solutions don't always give the appearance of collision propagation, which may be what you want. For instance, you might want to see a ball hit a line of balls and send one off the opposite side. With an analytical impulse solver, several balls might go one way, and several the opposite [Bar89]. Potentially, some initial propagation could be done before employing an impulse solver.

4 Contact Positions

4.1 Points of Collision

How do you find the point where two bodies collide? A standard way is to use a closest-points algorithm – when two objects are very close, inside a *collision envelope* [Mir PhD], you can declare the closest points to be contact points. For convex shapes, there are fast closest-points algorithms to choose from. Having the feature (point, edge, or face) where the closest point resides on each convex object can useful as well, as I'll discuss later.

There are several closest-points algorithms to consider. One is Lin-Canny, which involves "feature-walking" [LC]. You start with a feature from each of two convex objects, and if necessary move to neighboring features on each object until you are at the closest two. At this point, you get the closest points from these features. This algorithm is notable for its use of frame-to-frame *coherence*, which means closest features often don't change much between calls, so the amount of walking each call may be minimal. Brian Mirtich implemented a variant of Lin-Canny to improve robustness – he called the package V-Clip [V-Clip]. SWIFT, from Stephen Ehmann [SWIFT], is another Lin-Canny variant that improves the feature walking of the algorithm with multi-resolution models.

Another closest-points algorithm for convex objects is GJK, developed by Gilbert, Johnson, and Keerthi [GJK]. This also iterates to find the closest points, but uses extremal points on the convex objects to build simplices (points, line segments, triangles, and tetrahedra). The simplex on the last iteration provides the closest points of the objects. The original algorithm uses a single simplex produced with *Minkowski Sums*, but I implemented a variant that iterates two 3D simplices – the two simplices can be used as features. Notes on this and some source code are available on the Playstation2 developer website. Other people have made alterations to the original GJK. Stephen Cameron discussed how to exploit coherence in GJK's extremal point checks [Cam]. Gino van den Bergen built a collision detection package around GJK, named SOLID [SOLID], and incorporated exact primitive types such as boxes, cones and cylinders [Berg].

Incidentally, I worked on a closest-points library in graduate school, PQP, for arbitrary collections of triangles, but I don't recommend using this in real time applications. The convex object methods can be orders of magnitude faster, and given the range of choices available now, I think they're the best. You can build objects out of convex pieces and call GJK or Lin-Canny on the pieces, so it is still possible to define interesting geometry.

4.2 Finding Collisions

Suppose you want to simulate the collision of two asteroids. From what I've said above, you could wait until the asteroids are really close, compute their closest points as contact points, and apply the collision impulse. But if you're generating new asteroid positions at regular time intervals, you may find that the asteroids are far away in one frame, but intersect in the next

one. Alternatively, they might be moving so fast that they pass entirely through each other from one time to the next, so you might not even know there was a collision.

I dealt with the first problem by allowing backtracking to the time of collision, with ideas borrowed from [Bar95] and [Mir PhD]. I simulate a fixed step forward, and if the asteroids intersect at the end of this step, I search over the time interval for when their distance is greater than zero, but less than the collision envelope. This involves advancing both bodies to an intermediate time, and checking the distance between them with the closest-points algorithm, stepping forward or backward in time depending on whether you the measured distance is too large or too small. I use simple bisection of the time interval, although a root-finding algorithm would probably locate the time of contact more quickly. However, I think root finding works best if you evaluate negative distances (penetration depths) as well, which I did not.

I don't use integration to advance bodies in time, but simple linear interpolation of the state variables between the two frames. Linear interpolation might be problematic at larger time steps, but in my applications, running at 60 Hz, it seems to work fine. Particularly when you are using an expensive integration method, some kind of cheaper interpolation scheme is essential for these kinds of searches.

Some people might take issue with backing up the simulation clock. You may not have to do integration to backtrack, but you will have to integrate forward from the point you backtrack to, which will increase your integration costs per frame. I think some people try to just move bodies to separate them at the end of the frame, rather than rewind the clock.

The second problem mentioned was missing collisions, and I didn't build anything into my code to handle this. But at the GDC resting-contact seminar, David Wu mentioned the idea of "extruding" convex objects from the start position to the end position to see whether the extrusions of two object overlap. He cited a property of GJK that allows you to use it on these extruded objects efficiently. Generally, you can define an object to be the convex hull of two or more convex shapes, and use GJK on this hull, without having to compute the hull explicitly. All you have to do is modify the extremal-points check of GJK to find the most extremal point among all the shapes.

[Mir PhD] describes a method of bounding time of impact and stepping only that far. This solves both problems of missing collisions and finding collision times. However, conservative estimations of the time of impact can be overly small when bodies are close. Another collision scheduling strategy was also published by Mirtich [Mir00], which applies the *Timewarp* distributed computing algorithm to simulation. *Timewarp Simulation* allows for more time independence between bodies, which can greatly reduce integration costs of large-scale simulations. However, missed collision events are handled by a state rollback, which must be limited in a real-time (or at least interactive) context. In interactive applications, the time of the last rendered frame is as far as you can roll back the state.

4.3 Resting Contacts

In contrast with collision detection or closest-points algorithms, there are not many references for computing contact regions between two objects. However, both Baraff and Mirtich suggest defining contact regions using feature pairs, which may be vertex-vertex, vertex-edge, vertex-face, or edge-edge feature pairs [Bar97] [Mir98].

For instance, the contact region between a block and a tabletop may be defined by four vertex-face feature pairs (Fig. 1(a)). If the block is pushed to an edge of the table, and starts to tilt off the edge, the region may be defined by two edge-edge feature pairs (Fig. 1(b)).

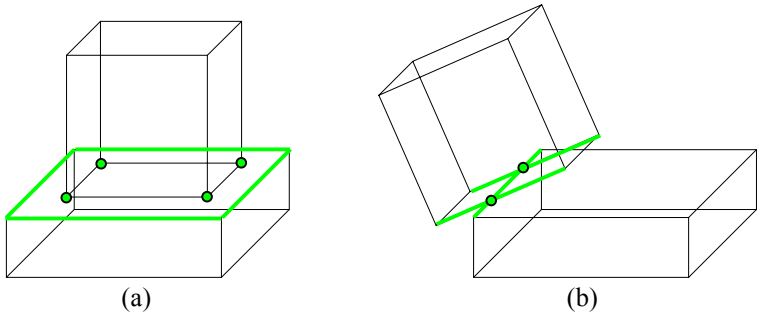

Fig. 1: (a) four vertex-face contacts (b) two edge-edge contacts

In [Mir98], the closest points of these feature pairs provide the contact points used in applying forces between bodies. You initialize the set with the closest features when two objects first share a resting contact. Afterwards, when you call the closest-features algorithm, if the closest features reported are not in the set, you add them. Of course this strategy does depend on maintaining some space between objects with a collision envelope so there's some room for one feature pair to get closer than others. If you find that the bodies intersect when you call the closest-features algorithm, you'll have to back time up to find the moment before collision occurs and after a new feature pair appears. Mirtich suggests that when a new feature pair is discovered, time should be backed up further, to when the feature pair's closest points first enter the collision envelope. This maintains the collision envelope distance between bodies. Feature pairs are removed when their closest points are too far apart, or slide to the exterior of a feature (an endpoint of an edge, or the boundary of a face).

Penalty methods allow sustained overlap between bodies, so how can you use a closest feature algorithm, which depends on objects being separate? This is what I did: as soon as two objects are joined by a resting contact, the normal at this contact is an axis along with penetration depth is computed. If the penetration of the entire objects is larger than the penetration of these features (by a small margin), new contact features are needed (Fig. 2(a)). Time is backed up to when the penetration is small (Fig. 2(b)). The objects are separated along the normal, and the closest features are found and added to the feature set (Fig. 2(c)). This feature pair now defines the new normal along which penetration is measured, and the penetration difference is returned to 0 (Fig. 2(d)).

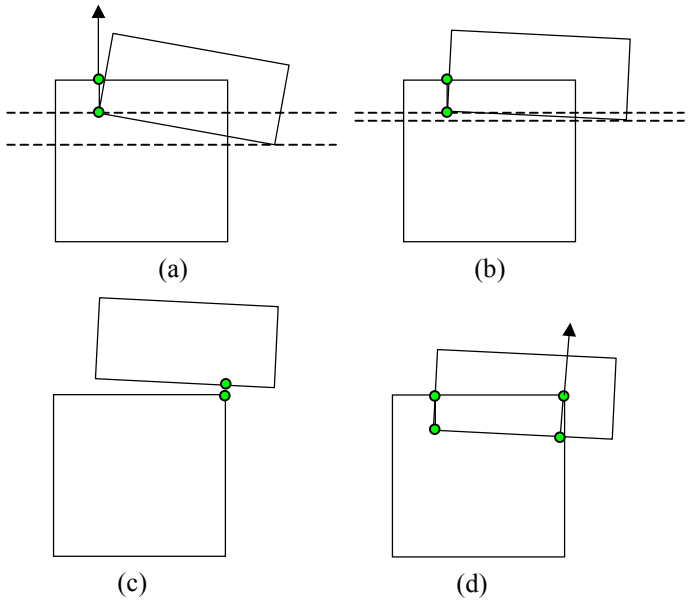

Fig. 2: (a) search for new contact based on *depth difference* (b) search of time interval for small depth difference. (c) separation of bodies to find new contact point. (d) new contact is added to set.

4.4 Resting Contact Flaws

This contact finding strategy was more reliable than I first imagined, but it has a serious shortcoming. Notice I do not back up the simulation to when new contacts enter the collision envelope. I've found that this time can be well before the last rendered frame. But this means that you attach spring-dampers "pre-stretched", and a pre-stretched spring-damper contains potential energy. As the spring-damper compresses, it may add some kinetic energy to the bodies. Contact forces shouldn't increase system kinetic energy, and if used properly, spring-dampers won't. As long as they enter the simulation un-stretched, any stretch they get will be by first absorbing kinetic energy, so they will only release energy that was previously in the system. Actually, since the damper always drains kinetic energy, when a spring stretches and compresses, there should always be some net loss of kinetic energy.

This isn't just academic – the extra energy can cause real problems. A domino can rock back and forth indefinitely, or a cylinder may pick up speed as it rolls. Trying to adjust the damper to siphon more energy is a hit or miss strategy in my experience. Really, you want a contact finding algorithm that attaches only un-stretched springs, but I didn't have time to explore alternatives. Unfortunately, you have to worry about the opposite problem as well – energy loss can be too great. If your springs are weak enough, the damper will get to act over too large a distance, and a cylinder will look like it's rolling through mud.

Another problem is that the integration seems to perform best when springs are close to equibrium, that is, not stretching or compressing very fast. When you attach pre-stretched springs, you may put extra strain on the integration.

5. Spring-damper Details

5.1 Spring and Damper Constants

Spring and damper constants, k_s and k_d respectively, control the stiffness of the penalty. Brian Mirtich recommended setting these constants to yield a *critically damped* mass-spring-damper system. (All the dynamics textbooks I've seen discuss critically damped systems.) Suppose the gravity constant is *g*. To make a body with mass *m* rest on *n* contacts and penetrate to a depth of *d*, you would set:

$$k_s = \frac{m \cdot g}{n \cdot d}$$

In order to make this a critically damped system, you would set:

$$k_d = 2 \cdot \sqrt{\frac{m \cdot k_s}{n}}$$

This only gives you a critically damped system when all the spring-dampers have the same displacement and velocity, but it's just an approximation.

If constants are chosen per body pair, they will not account for the weight of neighboring bodies. This can cause the bottom object of a stack to penetrate too greatly. Some kind of "global" scheme for setting these constants might seem better, but then, changing k_s or k_d for a spring-damper during its lifetime can cause the spring-dampers to gain or lose potential energy. In *Robot Soccer*, to keep the soccer ball from getting pushed into walls by the more massive robots, the spring constant was kept uniform at all contacts and tuned to the robot masses.

5.2 Attractive Forces

You want the spring-dampers to only repel objects, not pull them together. A spring-damper shouldn't be active when contact points have separated. But contacts may separate during the integration step, imparting some attractive force for a part of the step. I admit that I never addressed this. I simply break spring-dampers at the end of the integration step if the contacts became separated. Attractive forces can also occur if the damper force exceeds the spring force as two objects are separating. I did try to address that – spring-dampers are inactivated at the start of an integration step at contacts with a large enough relative normal velocity.

6. Semi-Implicit Integration

6.1 Explicit, Implicit, and Semi-Implicit Integration

Now I can focus on the really challenging part – semi-implicit integration. Explicit methods like Euler, Midpoint, and Runge-Kutta are typical integrator choices, all simple to implement. Unfortunately, they don't work very well on stiff spring-dampers, because the solution results can diverge. Fig. 3 shows plots of Euler integration results in a 1D spring-damper system, using two different time steps:

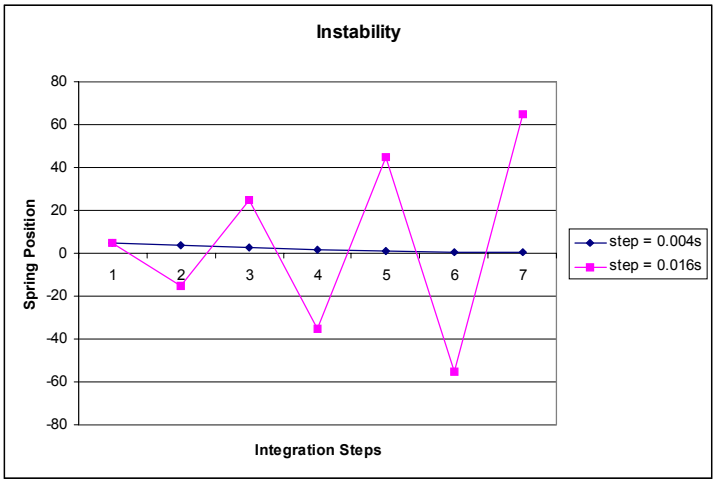

Fig. 3: Instability of Explicit Euler Integration

Runge-Kutta can increase the step size at which the system explodes, but you still can get into a frustrating game of tuning mass, gravity, spring, and damper constants to see if you can keep things from jumping around. Implicit or semi-implicit integration methods largely save you from this headache.

Here's a short primer on the differences of the integration methods. An explicit Euler integrator tries to solve the following differential equation:

$$\frac{d\vec{y}}{dt} = \vec{f}(\vec{y}, t)$$

using the explicit formula:

$$\vec{y}_{n+1} = \vec{y}_n + h \cdot \vec{f}(\vec{y}_n, t_n)$$

It's explicit, because the unknown values are computed directly from known values. But you can see intuitively why this can be unstable. If you stretch out a mass on a stiff spring-damper, the forces used to calculate $\vec{f}$ can be huge at the start of the step. Of course, $\vec{f}$ instantly decays as the spring-damper compresses, but the explicit formula is blind to that. The result can be a repeated overshoot that leads to the divergent behavior in the (Fig. 3).

Naturally, implicit formulas are the basis of implicit integration methods. Here's the Implicit Euler Formula:

$$\vec{y}_{n+1} = \vec{y}_n + h \cdot \vec{f}(\vec{y}_{n+1}, t_{n+1})$$

Obviously you can't evaluate $\vec{f}(\vec{y}_{n+1}, t_{n+1})$ directly because you don't yet know $\vec{y}_{n+1}$, so you have to solve the equation for $\vec{y}_{n+1}$. Depending on the nature of $\vec{f}$, the resulting equation can be nonlinear and difficult to solve, but this formula has a strong stability property (something called *A-stability*).

To circumvent the difficulty of solving nonlinear equations, semi-implicit methods were devised to "linearize" the equations. The semi-implicit Euler formula is:

$$\vec{y}_{n+1} = \vec{y}_n + h \cdot \left(\vec{f}(\vec{y}_n, t_n) + J(\vec{y}_n, t_n) \cdot (\vec{y}_{n+1} - \vec{y}_n) \right)$$

J is a *Jacobian*, which is a matrix of partial derivatives sometimes written as $\partial \vec{f} / \partial \vec{y}$ – element $J_{[i,j]}$ is partial derivative of $f_{[i]}$ with respect to $y_{[j]}$. Basically, J specifies the change in $\vec{f}$ with respect to the change in $\vec{y}$, which means that $J(\vec{y}_n, t_n) \cdot (\vec{y}_{n+1} - \vec{y}_n)$ approximates the change in $\vec{f}$ over the time step. Therefore, $\left(\vec{f}(\vec{y}_n, t_n) + J(\vec{y}_n, t_n) \cdot (\vec{y}_{n+1} - \vec{y}_n) \right)$ approximates $\vec{f}(\vec{y}_{n+1}, t_{n+1})$. The formula can be reorganized as:

$$\left(\frac{1}{h} \cdot (Identity\ Matrix) - J(\vec{y}_n, t_n) \right) \cdot (\vec{y}_{n+1} - \vec{y}_n) = \vec{f}(\vec{y}_n, t_n)$$

to show that you must solve a linear system to find $\vec{y}_{n+1}$.

Semi-implicit methods require that $\vec{f}$ be a function differentiable by $\vec{y}$. Fortunately, forces in a penalty method are a function of rigid body parameters – hence, you can derive J.

I must admit that for some time I didn't know there was any other kind of implicit integration besides semi-implicit integration. In fact, semi-implicit methods do not have the same guaranteed stability – Numerical Recipes says they are only "usually stable". When I tried semi-implicit Euler, the simulations were better behaved than with explicit methods, but still blew up from time to time. David Wu uses the real implicit Euler. Unfortunately, not realizing early enough that there was a true implicit Euler, I never tried the approach myself. I can however report on the method Brian Mirtich used, the Rosenbrock semi-implicit integrator.

The Rosenbrock integrator is a generalized form of Runge-Kutta. It offers extra stability over semi-implicit Euler – it does three $\vec{f}$ evaluations and solves four linear equations, and you can specify the error bounds of the result, which allows the integrator to detect any "divergent" results and take a smaller step.

This may seem like a very bad path to start down. If each integration step can fail, the integrator is a more unpredictable component of your simulator. I agree that this is a major weakness. Spring-dampers can cause integration retries when they're allowed to stretch too far or if they're hit with a high velocity. Most often though, failures due to spring-dampers are rare, not being overly strict with error bounds. With stiff spring and damper constants, the implicit integration was nearly always faster than taking the many explicit integration steps that were needed for stability, and I never saw things blow up.

6.2 Computing Jacobians

Definitely the hardest part of the semi-implicit integration process is deriving the Jacobian for the integrator. My code used the following differential equation per body, obtained from David Baraff's course notes:

$$d/dt(\vec{y}) = \vec{f}(\vec{y})$$

where

$$\vec{y} = \begin{pmatrix} \vec{x} \\ \vec{q} \\ \vec{P} \\ \vec{L} \end{pmatrix}, \vec{f} = \begin{pmatrix} \dot{\vec{x}} \\ \dot{\vec{q}} \\ \vec{F} \\ \vec{\tau} \end{pmatrix}$$

$\vec{x}$ is position, $\vec{q}$ is an orientation (quaternions are preferred for ease of re-normalization, and compact representation), $\vec{P}$ is linear momentum, $m\dot{\vec{x}}$, and $\vec{L}$ is angular momentum which is the *inertial tensor*, I, times the *angular velocity*, $\vec{\omega}$. $\vec{F}$ is the total force, and $\vec{\tau}$ is the total torque.

$\vec{f}$ can always be computed from the current state:

$$\dot{\vec{x}} = \vec{P}/mass$$

$$\dot{\vec{q}} = \frac{1}{2} \begin{pmatrix} \omega_{[0]} \\ \omega_{[1]} \\ \omega_{[2]} \\ 0 \end{pmatrix} \cdot \vec{q}$$

[Bar97], $\vec{F}$ and $\vec{\tau}$ are computed from your penalty force law or your force solver.

The differential equation for a system of rigid bodies is the union of all the per body equations. You end up with long arrays of state and a long array of derivatives.

$$d/dt\begin{pmatrix}\begin{pmatrix}\vec{x}_A\\ \vec{q}_A\\ \vec{P}_A\\ \vec{L}_A\\ \vec{x}_B\\ \vec{q}_B\\ \vec{P}_B\\ \vec{L}_B\\ \vec{x}_C\\ \vec{q}_C\\ \vec{P}_C\\ \vec{L}_C\\ \vdots\end{pmatrix}\end{pmatrix}=\begin{pmatrix}\dot{\vec{x}}_A\\ \dot{\vec{q}}_A\\ \vec{F}_A\\ \vec{\tau}_A\\ \dot{\vec{x}}_B\\ \dot{\vec{q}}_B\\ \vec{F}_B\\ \vec{\tau}_B\\ \dot{\vec{x}}_C\\ \dot{\vec{q}}_C\\ \vec{F}_C\\ \vec{\tau}_C\\ \vdots\end{pmatrix}$$

Since $\vec{x}$, $\vec{P}$, and $\vec{L}$ each have 3 dimensions, and $\vec{q}$ has 4, there are 13 variables per body. The differential equation for a system of *n* bodies thus has dimensions 13*n*. Since the Jacobian is the matrix of $\vec{f}$ variables differentiated with respect to $\vec{y}$ variables, the Jacobian's dimensions are $13n \times 13n$. It's better though to think of it as an $n \times n$ matrix of 13x13 blocks. Here's what a Jacobian looks like for a system of two bodies:

$$\begin{pmatrix} \begin{pmatrix} \frac{\partial \dot{\vec{x}}_A}{\partial \vec{x}_A} & \frac{\partial \dot{\vec{x}}_A}{\partial \vec{q}_A} & \frac{\partial \dot{\vec{x}}_A}{\partial \vec{P}_A} & \frac{\partial \dot{\vec{x}}_A}{\partial \vec{L}_A} \\ \frac{\partial \dot{\vec{q}}_A}{\partial \vec{x}_A} & \frac{\partial \dot{\vec{q}}_A}{\partial \vec{q}_A} & \frac{\partial \dot{\vec{q}}_A}{\partial \vec{P}_A} & \frac{\partial \dot{\vec{q}}_A}{\partial \vec{L}_A} \\ \frac{\partial \vec{F}_A}{\partial \vec{x}_A} & \frac{\partial \vec{F}_A}{\partial \vec{q}_A} & \frac{\partial \vec{F}_A}{\partial \vec{P}_A} & \frac{\partial \vec{F}_A}{\partial \vec{L}_A} \\ \frac{\partial \vec{\tau}_A}{\partial \vec{x}_A} & \frac{\partial \vec{\tau}_A}{\partial \vec{q}_A} & \frac{\partial \vec{\tau}_A}{\partial \vec{P}_A} & \frac{\partial \vec{\tau}_A}{\partial \vec{L}_A} \end{pmatrix} & \begin{pmatrix} \frac{\partial \dot{\vec{x}}_A}{\partial \vec{x}_B} & \frac{\partial \dot{\vec{x}}_A}{\partial \vec{q}_B} & \frac{\partial \dot{\vec{x}}_A}{\partial \vec{P}_B} & \frac{\partial \dot{\vec{x}}_A}{\partial \vec{L}_B} \\ \frac{\partial \dot{\vec{q}}_A}{\partial \vec{x}_B} & \frac{\partial \dot{\vec{q}}_A}{\partial \vec{q}_B} & \frac{\partial \dot{\vec{q}}_A}{\partial \vec{P}_B} & \frac{\partial \dot{\vec{q}}_A}{\partial \vec{L}_B} \\ \frac{\partial \vec{F}_A}{\partial \vec{x}_B} & \frac{\partial \vec{F}_A}{\partial \vec{q}_B} & \frac{\partial \vec{F}_A}{\partial \vec{P}_B} & \frac{\partial \vec{F}_A}{\partial \vec{L}_B} \\ \frac{\partial \vec{\tau}_A}{\partial \vec{x}_B} & \frac{\partial \vec{\tau}_A}{\partial \vec{q}_B} & \frac{\partial \vec{\tau}_A}{\partial \vec{P}_B} & \frac{\partial \vec{\tau}_A}{\partial \vec{L}_B} \end{pmatrix} \\ \begin{pmatrix} \frac{\partial \dot{\vec{x}}_B}{\partial \vec{x}_A} & \frac{\partial \dot{\vec{x}}_B}{\partial \vec{q}_A} & \frac{\partial \dot{\vec{x}}_B}{\partial \vec{P}_A} & \frac{\partial \dot{\vec{x}}_B}{\partial \vec{L}_A} \\ \frac{\partial \dot{\vec{q}}_B}{\partial \vec{x}_A} & \frac{\partial \dot{\vec{q}}_B}{\partial \vec{q}_A} & \frac{\partial \dot{\vec{q}}_B}{\partial \vec{P}_A} & \frac{\partial \dot{\vec{q}}_B}{\partial \vec{L}_A} \\ \frac{\partial \vec{F}_B}{\partial \vec{x}_A} & \frac{\partial \vec{F}_B}{\partial \vec{q}_A} & \frac{\partial \vec{F}_B}{\partial \vec{P}_A} & \frac{\partial \vec{F}_B}{\partial \vec{L}_A} \\ \frac{\partial \vec{\tau}_B}{\partial \vec{x}_A} & \frac{\partial \vec{\tau}_B}{\partial \vec{q}_A} & \frac{\partial \vec{\tau}_B}{\partial \vec{P}_A} & \frac{\partial \vec{\tau}_B}{\partial \vec{L}_A} \end{pmatrix} & \begin{pmatrix} \frac{\partial \dot{\vec{x}}_B}{\partial \vec{x}_B} & \frac{\partial \dot{\vec{x}}_B}{\partial \vec{q}_B} & \frac{\partial \dot{\vec{x}}_B}{\partial \vec{P}_B} & \frac{\partial \dot{\vec{x}}_B}{\partial \vec{L}_B} \\ \frac{\partial \dot{\vec{q}}_B}{\partial \vec{x}_B} & \frac{\partial \dot{\vec{q}}_B}{\partial \vec{q}_B} & \frac{\partial \dot{\vec{q}}_B}{\partial \vec{P}_B} & \frac{\partial \dot{\vec{q}}_B}{\partial \vec{L}_B} \\ \frac{\partial \vec{F}_B}{\partial \vec{x}_B} & \frac{\partial \vec{F}_B}{\partial \vec{q}_B} & \frac{\partial \vec{F}_B}{\partial \vec{P}_B} & \frac{\partial \vec{F}_B}{\partial \vec{L}_B} \\ \frac{\partial \vec{\tau}_B}{\partial \vec{x}_B} & \frac{\partial \vec{\tau}_B}{\partial \vec{q}_B} & \frac{\partial \vec{\tau}_B}{\partial \vec{P}_B} & \frac{\partial \vec{\tau}_B}{\partial \vec{L}_B} \end{pmatrix} \end{pmatrix}$$

Fortunately, many of these partial derivatives will be zero. I'll use *[X]* to denote all elements corresponding to body *X*. When two bodies, *A* and *B*, don't have any contacts between them. $f_{[A]}$ has no direct dependence on $y_{[B]}$ and $f_{[B]}$ has no direct dependence on $y_{[A]}$. In this case, blocks $J_{[A,B]}$ and $J_{[B,A]}$ are entirely 0. Even when two bodies share contacts, not all of one's $\vec{f}$ terms will depend on all the other's $\vec{y}$ terms, leaving groups of 0 elements within each block.

6.3 Contact Groups

The zero blocks allow you to split the integration of the entire system into smaller integration problems. Specifically, each *contact group* can be integrated independently. A contact group is a cluster of contacting bodies. If you create a graph in which bodies are nodes, and edges link directly contacting bodies, each contact group is a connected component of the graph. Each contact group contributes equations that are independent from those of other contact groups. This allows you to integrate each contact group independently – you lump their Jacobian

blocks together into a “group Jacobian”. Incidentally, fixed or kinematic bodies should not link different contact groups together – contact groups can share the same fixed or kinematic bodies and their linear equations will still remain independent.

6.4 Jacobian Mathematics

The type of math involved in computing Jacobians is differentiating vector expressions by vectors. Last year’s GDC proceedings had a cloth simulation paper by Dean Macri that covered this topic. But I should mention that he defined row *i* column *j* of $\partial\vec{a}/\partial\vec{b}$ as $\partial a_{[j]}/\partial b_{[i]}$ whereas I define it as $\partial a_{[i]}/\partial b_{[j]}$. I’m sure you can use either with appropriate transpositions, but I wanted to stay consistent with Numerical Recipes.

Here’s a contrived example involving two 3 element vectors $\vec{a}$ and $\vec{b}$. If $\vec{a}$ is defined as:

$$\vec{a} = \begin{pmatrix} 10b_{[0]}b_{[1]} + b_{[2]} \\ b_{[0]} + 5b_{[1]} \\ b_{[1]} + b_{[0]}b_{[2]} \end{pmatrix}$$

then:

$$\frac{\partial\vec{a}}{\partial\vec{b}} = \begin{pmatrix} 10b_{[1]} & 10b_{[0]} & 1 \\ 1 & 5 & 0 \\ b_{[2]} & 1 & b_{[0]} \end{pmatrix}$$

But suppose you have a more complicated expression for $\vec{a}$, such as:

$$\vec{a} = M\vec{v} + \vec{v} \times \vec{w}$$

where $\vec{v}$ and $\vec{w}$ and M may contain elements of $\vec{b}$. You could expand each matrix and vector operation to get a separate equation for each element of $\vec{a}$, but you would most likely end up with very lengthy expressions that are hard to simplify. Fortunately, there are differentiation rules that can result in more concise derivations and efficient code. I’ll give some of these differentiation rules, but first let me define:

$$\tilde{\vec{a}} = \begin{pmatrix} 0 & -a_{[2]} & a_{[1]} \\ a_{[2]} & 0 & -a_{[0]} \\ -a_{[1]} & a_{[0]} & 0 \end{pmatrix}$$

This is the “*” matrix in David Baraff’s tutorial, but I preferred the ~ notation used by Chris Hecker.

Here are some differentiation rules involving matrices, vectors, and scalars:

$$\frac{\partial \vec{u} \cdot \vec{v}}{\partial \vec{w}} = \vec{u}^T \frac{\partial \vec{v}}{\partial \vec{w}} + \vec{v}^T \frac{\partial \vec{u}}{\partial \vec{w}}$$

$$\frac{\partial \vec{u} \times \vec{v}}{\partial \vec{w}} = \tilde{\vec{u}} \frac{\partial \vec{v}}{\partial \vec{w}} - \tilde{\vec{v}} \frac{\partial \vec{u}}{\partial \vec{w}}$$

$$\frac{\partial s\vec{v}}{\partial \vec{w}} = s\frac{\partial \vec{v}}{\partial \vec{w}} + \begin{pmatrix} \frac{\partial s}{\partial w_{[0]}}\vec{v} & \frac{\partial s}{\partial w_{[1]}}\vec{v} & \frac{\partial s}{\partial w_{[2]}}\vec{v} & \cdots \end{pmatrix}$$

$$\frac{\partial M\vec{v}}{\partial \vec{w}} = M\frac{\partial \vec{v}}{\partial \vec{w}} + \begin{pmatrix} \frac{\partial M}{\partial w_{[0]}}\vec{v} & \frac{\partial M}{\partial w_{[1]}}\vec{v} & \frac{\partial M}{\partial w_{[2]}}\vec{v} & \cdots \end{pmatrix}$$

$$\frac{\partial MN}{\partial s} = M\frac{\partial N}{\partial s} + \frac{\partial M}{\partial s}N$$

I also needed a rule for quaternion multiplication, which I obtained by splitting the quaternion into its vector and scalar parts:

$$\vec{q} = \begin{pmatrix} \vec{v} \\ s \end{pmatrix}$$

$$\vec{q}_A\vec{q}_B = \begin{pmatrix} \vec{v}_A \times \vec{v}_B + s_A\vec{v}_B + s_B\vec{v}_A \\ s_As_B - \vec{v}_A \cdot \vec{v}_B \end{pmatrix}$$

The differentiation rule is:

$$\frac{\partial(\vec{q}_A\vec{q}_B)}{\partial \vec{w}} = \begin{pmatrix} \tilde{\vec{v}}_A\frac{\partial \vec{v}_B}{\partial \vec{w}} - \tilde{\vec{v}}_B\frac{\partial \vec{v}_A}{\partial \vec{w}} + s_A\frac{\partial \vec{v}_B}{\partial \vec{w}} + \begin{pmatrix} \frac{\partial s_A}{\partial w_{[0]}}\vec{v}_B & \frac{\partial s_A}{\partial w_{[1]}}\vec{v}_B & \frac{\partial s_A}{\partial w_{[2]}}\vec{v}_B & \cdots \end{pmatrix} + s_B\frac{\partial \vec{v}_A}{\partial \vec{w}} + \begin{pmatrix} \frac{\partial s_B}{\partial w_{[0]}}\vec{v}_A & \frac{\partial s_B}{\partial w_{[1]}}\vec{v}_A & \frac{\partial s_B}{\partial w_{[2]}}\vec{v}_A & \cdots \end{pmatrix} \\ s_A\frac{\partial s_B}{\partial \vec{w}} + s_B\frac{\partial s_A}{\partial \vec{w}} - \vec{v}_A{}^T\frac{\partial \vec{v}_B}{\partial \vec{w}} - \vec{v}_B{}^T\frac{\partial \vec{v}_A}{\partial \vec{w}} \end{pmatrix}$$

I'm not going to write down the full derivation of the Jacobian, and not just because it's easier for me that way. I think you're better off not just typing in my math. You might want to do things differently, and I could make a mistake somewhere. But I will give you a lengthy example. Consider just the diagonal Jacobian block for a body *A*, $\partial f_{[A]}/\partial y_{[A]}$. This block is the following matrix of partial derivatives:

$$\begin{pmatrix} \frac{\partial \dot{\vec{x}}_A}{\partial \vec{x}_A} & \frac{\partial \dot{\vec{x}}_A}{\partial \vec{q}_A} & \frac{\partial \dot{\vec{x}}_A}{\partial \vec{P}_A} & \frac{\partial \dot{\vec{x}}_A}{\partial \vec{L}_A} \\ \frac{\partial \dot{\vec{q}}_A}{\partial \vec{x}_A} & \frac{\partial \dot{\vec{q}}_A}{\partial \vec{q}_A} & \frac{\partial \dot{\vec{q}}_A}{\partial \vec{P}_A} & \frac{\partial \dot{\vec{q}}_A}{\partial \vec{L}_A} \\ \frac{\partial \vec{F}_A}{\partial \vec{x}_A} & \frac{\partial \vec{F}_A}{\partial \vec{q}_A} & \frac{\partial \vec{F}_A}{\partial \vec{P}_A} & \frac{\partial \vec{F}_A}{\partial \vec{L}_A} \\ \frac{\partial \vec{\tau}_A}{\partial \vec{x}_A} & \frac{\partial \vec{\tau}_A}{\partial \vec{q}_A} & \frac{\partial \vec{\tau}_A}{\partial \vec{P}_A} & \frac{\partial \vec{\tau}_A}{\partial \vec{L}_A} \end{pmatrix}$$

Some of the components are zero though, and $\partial \dot{x}_A / \partial P_A$ is trivial:

$$\begin{pmatrix} 0 & 0 & \begin{pmatrix} 1/mass_A & 0 & 0 \\ 0 & 1/mass_A & 0 \\ 0 & 0 & 1/mass_A \end{pmatrix} & 0 \\ 0 & \frac{\partial \dot{\vec{q}}_A}{\partial \vec{q}_A} & 0 & \frac{\partial \dot{\vec{q}}_A}{\partial \vec{L}_A} \\ \frac{\partial \vec{F}_A}{\partial \vec{x}_A} & \frac{\partial \vec{F}_A}{\partial \vec{q}_A} & \frac{\partial \vec{F}_A}{\partial \vec{P}_A} & \frac{\partial \vec{F}_A}{\partial \vec{L}_A} \\ \frac{\partial \vec{\tau}_A}{\partial \vec{x}_A} & \frac{\partial \vec{\tau}_A}{\partial \vec{q}_A} & \frac{\partial \vec{\tau}_A}{\partial \vec{P}_A} & \frac{\partial \vec{\tau}_A}{\partial \vec{L}_A} \end{pmatrix}$$

I'll show my derivation of the component $\partial \dot{\vec{q}}_A / \partial \vec{q}_A$, dropping the *A* subscript for convenience. Again, I'll use $\vec{q} = \begin{pmatrix} \vec{v} \\ s \end{pmatrix}$:

$$\frac{\partial \dot{\vec{q}}}{\partial \vec{q}} = \frac{\partial\left(\frac{1}{2}\begin{pmatrix}\omega_{[0]}\\ \omega_{[1]}\\ \omega_{[2]}\\ 0\end{pmatrix}\cdot \vec{q}\right)}{\partial \vec{q}} = \frac{1}{2}\frac{\partial\begin{pmatrix}\vec{\omega}\times\vec{v} + s\vec{\omega}\\ -\vec{\omega}\cdot\vec{v}\end{pmatrix}}{\partial \vec{q}}$$

$$\frac{\partial\begin{pmatrix}\vec{\omega}\times\vec{v}+s\vec{\omega}\\ -\vec{\omega}\cdot\vec{v}\end{pmatrix}}{\partial \vec{q}} = \begin{pmatrix}\tilde{\vec{\omega}}\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\end{pmatrix} - \tilde{\vec{v}}\frac{\partial\vec{\omega}}{\partial\vec{q}} + s\frac{\partial\vec{\omega}}{\partial\vec{q}} + \begin{pmatrix}\vec{0}&\vec{0}&\vec{0}&\vec{\omega}\end{pmatrix}\\ -\vec{\omega}^T\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\end{pmatrix} - \vec{v}^T\frac{\partial\vec{\omega}}{\partial\vec{q}}\end{pmatrix}$$

$$= \begin{pmatrix}\begin{pmatrix}\tilde{\vec{\omega}} & \vec{\omega}\end{pmatrix} - \tilde{\vec{v}}\frac{\partial\vec{\omega}}{\partial\vec{q}} + s\frac{\partial\vec{\omega}}{\partial\vec{q}}\\ -\begin{pmatrix}\omega_{[0]}&\omega_{[1]}&\omega_{[2]}&0\end{pmatrix} - \vec{v}^T\frac{\partial\vec{\omega}}{\partial\vec{q}}\end{pmatrix}$$

$\partial\vec{\omega}/\partial\vec{q}$ is still needed. But first, let me denote the rotation matrix of the body as *R*, and the constant body-relative inertial tensor as I_{body}. Again, I refer you to [Bar97] for context.

$$\frac{\partial\vec{\omega}}{\partial\vec{q}} = \frac{\partial(RI_{body}{}^{-1}R^T\vec{L})}{\partial\vec{q}}$$

$$= RI_{body}{}^{-1}R^T\frac{\partial\vec{L}}{\partial\vec{q}} + \begin{pmatrix}\frac{\partial(RI_{body}{}^{-1}R^T)}{\partial q_{[0]}}\vec{L} & \frac{\partial(RI_{body}{}^{-1}R^T)}{\partial q_{[1]}}\vec{L} & \frac{\partial(RI_{body}{}^{-1}R^T)}{\partial q_{[2]}}\vec{L} & \frac{\partial(RI_{body}{}^{-1}R^T)}{\partial q_{[3]}}\vec{L}\end{pmatrix}$$

$\partial\vec{L}/\partial\vec{q}$ is 0 so the first term vanishes. I'll derive $\frac{\partial(RI_{body}{}^{-1}R^T)}{\partial q_{[0]}}$:

$$\frac{\partial(RI_{body}{}^{-1}R^T)}{\partial q_{[0]}} = R\frac{\partial(I_{body}{}^{-1}R^T)}{\partial q_{[0]}} + \frac{\partial R}{\partial q_{[0]}}I_{body}{}^{-1}R^T$$

$$= RI_{body}{}^{-1}\frac{\partial R^T}{\partial q_{[0]}} + \frac{\partial R}{\partial q_{[0]}}I_{body}{}^{-1}R^T$$

But since,

$$\left(RI_{body}{}^{-1}\frac{\partial R^T}{\partial q_{[0]}} \right) = \left(\frac{\partial R}{\partial q_{[0]}} I_{body}{}^{-1} R^T \right)^T$$

you get the slighter simpler result:

$$\frac{\partial (RI_{body}{}^{-1}R^T)}{\partial q_{[0]}} = \left(\frac{\partial R}{\partial q_{[0]}} I_{body}{}^{-1} R^T \right) + \left(\frac{\partial R}{\partial q_{[0]}} I_{body}{}^{-1} R^T \right)^T$$

The last component that you need is $\partial R / \partial q_{[0]}$. You first need an expression for R in terms of $\vec{q}$. I used

$$R = \frac{1}{q_{[0]}{}^2 + q_{[1]}{}^2 + q_{[2]}{}^2 + q_{[3]}{}^2} \begin{pmatrix} q_{[0]}{}^2 - q_{[1]}{}^2 - q_{[2]}{}^2 + q_{[3]}{}^2 & 2(q_{[0]}q_{[1]} - q_{[3]}q_{[2]}) & 2(q_{[0]}q_{[2]} + q_{[1]}q_{[3]}) \\ 2(q_{[0]}q_{[1]} + q_{[3]}q_{[2]}) & -(q_{[0]}{}^2 - q_{[1]}{}^2 + q_{[2]}{}^2 - q_{[3]}{}^2) & -2(q_{[1]}q_{[2]} + q_{[0]}q_{[3]}) \\ 2(q_{[0]}q_{[2]} - q_{[1]}q_{[3]}) & 2(q_{[1]}q_{[2]} + q_{[0]}q_{[3]}) & -(q_{[0]}{}^2 + q_{[1]}{}^2 - q_{[2]}{}^2 - q_{[3]}{}^2) \end{pmatrix}$$

which looks different from the usual quaternion-to-matrix formula because it contains a normalization of $\vec{q}$. I'm hoping you can handle $\partial R / \partial q_{[0]}$, $\partial R / \partial q_{[1]}$, $\partial R / \partial q_{[2]}$, and $\partial R / \partial q_{[3]}$ on your own.

To compute $\partial \vec{F}_A / \partial \vec{g}_A$, where $\vec{g}_A$ is any one of $\vec{x}_A$, $\vec{q}_A$, $\vec{P}_A$, and $\vec{L}_A$, you can sum the derivatives of individual contact forces, $\partial \vec{F}i_A / \partial \vec{g}_A$, for each contact *i*. You can also exploit the fact that for a contact *i* between bodies *A* and *B*, $\partial \vec{F}i_B / \partial \vec{g}_A = -\partial \vec{F}i_A / \partial \vec{g}_A$. In general, the off diagonal blocks of the Jacobian are almost free when you compute the diagonal blocks.

6.5 Debugging Jacobians

Even if you're comfortable with this type of math, there are a lot of chances to mess it up. A symbolic mathematics package may be able to handle some of this derivation for you, although I didn't look into it. But if so, this could make your results more reliable, and save you from some of the tedium of the process.

And in fact, doing the derivation by hand, I didn't get it all right the first time. When I first tried to use the Rosenbrock integrator, it took extremely small steps. Apparently some methods for stiff problems just use the Jacobian as a kind of hint, but Rosenbrock methods really require that the Jacobian be right. I suspected the Jacobian had some bugs, but I had no idea where to look for them.

Finally, I remembered a way to debug derivative values. Your integrator requires a function to evaluate $\vec{f}$, and you can use this function to evaluate the Jacobian experimentally. To evaluate column *j* of the Jacobian experimentally, you can evaluate $\vec{f}$ at the start of the step, $\vec{f}_{start}$, add a small *delta* to y_j and compute $\vec{f}$ again, $\vec{f}_{delta}$, then just compute $\left(\vec{f}_{delta} - \vec{f}_{start}\right) / delta$.

Just printing out the experimental Jacobian and the one I computed analytically, I found a bunch of numbers that didn't match. It was a lot easier to track down the bugs in the math or the code that were responsible knowing exactly which parts of the Jacobian matrix were wrong. It's hard to say exactly how closely these numbers should match. Generally, I was content with a few leading digits being the same. You might find that some small differences are closed by decreasing *delta*, although I think the numerical error can increase if *delta* is too small. In spite of that, the technique works pretty well at showing you where you've gone wrong. Sometimes I could see the problem was as simple as a wrong sign.

Another piece of general wisdom is, don't try to integrate rigid bodies until you've gotten integration to work on a smaller problem. Start with a single stiff mass-spring-damper system. There won't be as many ways to mess that up, so you can expose any misconceptions more quickly.

6.6 Normals

You must also derive expressions for contact normals in terms of body state, if you choose to think of normals changing during the integration step. I kept my normals fixed in the Robot Soccer demo, but you get less penetration at the end of the integration step if normals can change continuously during the integration step.

However, I had a concern about normals defined by edge pairs. If you define this normal as a normalized cross product of two edges, the normal becomes degenerate when edges become parallel. You must figure out a way to deal with this discontinuity.

6.7 Sparse Matrix Inversion

Even for a contact group, a Jacobian can be sparse. Consider a row of dominos that has fallen over. Even though you have to integrate all the dominos as a group, there are only *O(n)* non-zero blocks in its Jacobian. You can exploit the sparseness of the Jacobian to make solving each linear system efficient.

That's why Brian Mirtich used a sparse solver with the Rosenbrock Integrator, and I used the same sparse matrix solving library [IML] that he cited in his paper. The particular sparse solver I used was the Biconjugate Gradient solver, with diagonal preconditioning (explained in the IML literature). The Biconjugate Gradient only accesses your matrix either to multiply it by a vector or to multiply the transpose of it by a vector. These multiplies can take advantage of the sparseness of your matrix – only the nonzero elements need to be multiplied.

I found that sometimes Biconjugate Gradient fails, i.e., iterates the maximum number of times without meeting the error bounds. This didn't seem to affect the simulations in an obvious way, although failing means the algorithm does more work. I found a reference on the web, [Netlib], that mentioned conditions under which Biconjugate Gradient fails. When I added some checks for these conditions, I found that they did occur occasionally. I tried just exiting the matrix solver with the current solution when the conditions occur, and again didn't notice any obvious side effects. But I wasn't sure this was really a safe fix. I want to understand what's happening better – perhaps a better preconditioner is needed, or I should switch to a more robust solver. There are other choices in the IML library that I didn't get time to play with.

Also, since the entire Jacobian is a collection of 3x3, 3x4, 4x3, or 4x4 matrices, you can use SIMD operations to accelerate the matrix-vector multiplies.

7 Friction

7.1 A Friction Model

In the *Timewarp* paper, Mirtich cited a friction law that approximates friction response with a hyperbolic tangent function (*tanh*). It's simplest to discuss this method in the context of a 1D simulation. Imagine a point mass block sliding down a ramp. You can simulate the block motion along just the "down ramp" axis to turn it into a 1D simulation. The tangent direction is fixed down the ramp in this example, so tangent velocity (v_T) is positive when the block moves down the ramp, and negative when it moves up.

The block's motion in the normal direction is assumed fixed, but you still compute normal force, F_N, as a constant to scale the friction force. Forces acting on the block are the down-ramp component of the gravity force, F_G, and the friction force, F_F. The friction force is calculated as:

$$F_F = -\mu F_N \tanh(v_T / \varepsilon)$$

where μ is the friction constant, and ε is a constant that squeezes the tanh horizontally. First, here's a plot of this function:

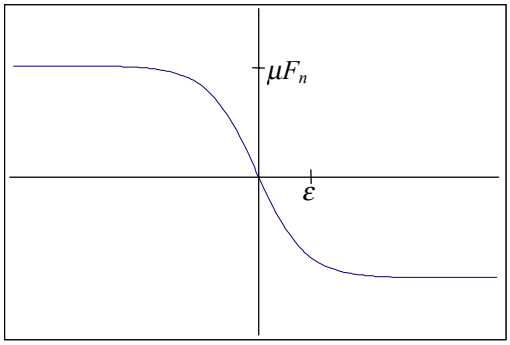

Fig. 4: Friction Response

Although a continuous function, it has several distinct looking components. It has a roughly linear component between $-\varepsilon$ and ε, and roughly constant components below $-\varepsilon$ and above ε. It is basically a smooth approximation of the piecewise friction law given in [Bar91].

If the block is moving faster than ε, the friction force will oppose the direction of motion and have magnitude (roughly) μF_N, thus acting as dynamic friction. But if the block begins to move from rest, the friction force opposing the motion will rapidly increase on the "linear" part of the function. Provided the friction constant is high enough, the friction will reach a strength that cancels F_G, and the block will stop accelerating, i.e., reach a terminal velocity. Since this terminal velocity is smaller than ε, ε is a bound on how fast the block can creep down the ramp. This steep linear component thus approximates static friction, as long as you're willing to redefine *static* as *slowly creeping*. Mirtich, however, stated "static friction is not modeled" [Mir00].

There's another way that this is unlike static friction. In most friction models, static friction has the potential to be stronger than dynamic friction, since there is a different friction constant for each. Here, there is only one friction constant.

7.2 Problems with Friction

Note that when a sliding block comes to a stop, the friction response rapidly falls off. Under dynamic friction force (if μ is large enough), $|v_T|$ will fall linearly towards 0. When it becomes less than ε, the friction force falls off to allow it to converge to its small terminal velocity. However, if the friction force were not to fall off when v_T reaches 0, v_T would maintain its linear trend, pass 0 and change sign – the block would start sliding the other direction, that is, back up the ramp.

Although this falloff is built into the above friction function, the integration process can miss it. If $|v_T| > \varepsilon$ at the start of the integration step, you evaluate a dynamic friction force. If this force is kept constant during the step, at the end of the step, v_T may change sign but still have magnitude > ε. Thus v_T can oscillate, making your object jitter and not come to rest.

Several force evaluations per step can mix positive and negative friction forces, helping to reduce the slope of $|v_T|$, but this doesn't seem to always guarantee that v_T will converge to its terminal velocity. You can potentially reduce your time step, or increase ε, but either lose performance or realism.

It occurred to me that the analytical force solving methods must have this problem, or an even worse one, since you have to try to obtain $v_T = 0$ exactly. I only recently looked through these references for an answer, but couldn't find explicit mention of this problem.

Alternatively, impulses computed in [Kaw] can result in $v_T = 0$, so perhaps impulse systems can be used to create static contacts. (Someone at GDC told me that they in fact use an impulse solver for this).

7.3 Rosenbrock and Friction

Alternatively, Rosenbrock Integration error bounds can be used to prevent this kind of jitter. After all, oscillation in v_T is an error – the true v_T curve should level off near zero. I tried setting relative error bounds on v_T , (but no smaller than $\varepsilon/2$) to have the integrator detect when v_T improperly crosses zero.

Detect is a generous word though. What the algorithm does is *fail*, and retry a smaller step. When a successful step is finally taken, the integrator tries another step forward, and can repeat the process. The number of retries can be pretty large just to make a single contact "stick". I didn't realize until recently that failures occur even more often due to underestimating the change in v_T than overestimating. This is a plot of integration behavior on a 1D block-on-ramp simulation:

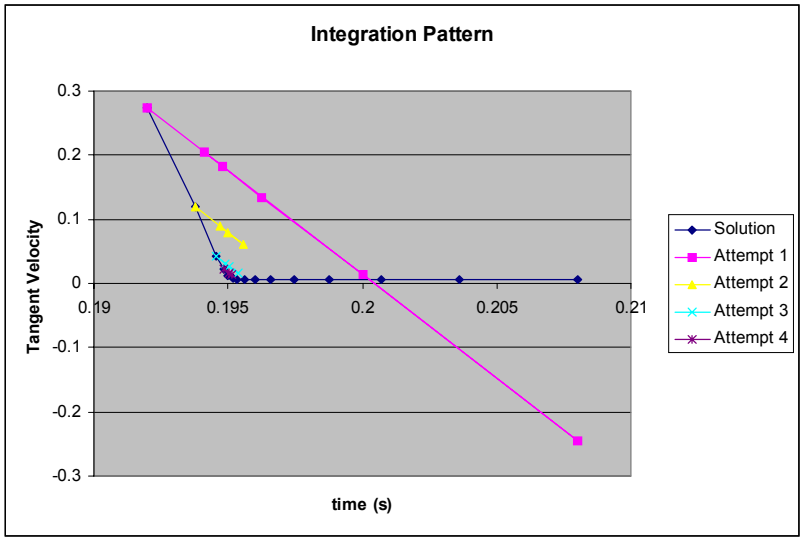

Fig 5: Pattern of failed integration steps.

The failed v_T are along linear paths with a shallower slope than the actual solution, due to the positive forces that get evaluated when v_t changes sign during integration. In order to follow the correct solution within the given error bounds, the integrator must reduce the time step until v_T doesn't change sign within the step.

On the other hand, I would be happy with many of these "failed" results. I'm not concerned that v_T falls on the slope of the actual solution, just that it converges to a small number. I think this could lead to a more efficient solution. However, in *Robot Soccer*, I had to reduce friction-related failures by making ε large, at the expense of lowering the appearance of static friction.

I also attempted a scheme of holding dynamic friction constant over the timestep, and on a change of v_t's sign, searching over the integration interval for a time when v_t is small, then retrying. This did help reduce integration failures, but wasn't as effective at stopping contacts. Plus, this can incur a large number of integration retries anyway.

But while I could clearly make improvements to the *tanh* solution, I have reservations about the general approach. I don't like that it never completely stops rigid bodies from sliding. Attaching tangent spring-dampers when $|v_T| < \varepsilon$ may be a way to really stop slip at a contact, but I didn't get a chance to try it.

8 Conclusion

Should you use these methods in real-time simulation? The answer is, not without some improvements. Integration is the major bottleneck in my simulations. An individual integration step with Rosenbrock won't take very much of the frame for a few objects, since you can take advantage of sparse matrices. But it isn't so fast that you can do it many times per frame.

The integrator currently gets called most frequently in my implementation when trying to bring a contact to a (near) stop. But even supposing this problem is solved, the integrator can only guarantee stability if allowed to retry smaller steps when it must – it will always be somewhat unpredictable. I have to optimize the approach more before I can say whether it has an acceptable degree of unpredictability.

Regardless, I hope that this paper exposed you to some of the basic concepts and common problems of resting contact. Good luck with your simulation projects.

9 References

[Bar89] Baraff, David, "Analytical Methods for Dynamics Simulation of Non-penetrating Rigid Bodies",
Computer Graphics Proceedings, ACM SIGGRAPH, 1989

[Bar91] Baraff, David, "Coping with Friction for Non-penetration Rigid Body Simulation", 1991

[Bar94] Baraff, David, "Fast Contact Force Computation for Nonpenetrating Rigid Bodies", *Computer Graphics Proceedings*, ACM SIGGRAPH, 1994

[Bar95] Baraff, David, "Interactive Simulation of Solid Rigid Bodies", *IEEE Computer Graphics and Applications*, 1995

[Bar97] Baraff, David, "An Introduction to Physically Based Modeling: Rigid Body Simulation I & Rigid Body Simulation II", SIGGRAPH 1997

[Berg] van den Bergen, Gino "A Fast and Robust GJK Implementation for Collision Detection of Convex Objects", 1998

[Cam] Cameron, Stephen “Enhancing GJK: Computing Minimum and Penetration Distances between Convex Polyhedra, 1997

[Cha] Chatterjee, Anindya and Andy Ruina “A New Algebraic Rigid Body Collision Law Based On Impulse Space Considerations”, *Journal of Applied Mechanics*

[GJK] Gilbert, Johnson, Keerthi “A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space”, 1988

[IML] http://math.nist.gov/iml++

[Kaw] Kawachi, Katsuaki *et al.*, “Technical Issues on Simulating Impulse and Friction in Three Dimensional Rigid Body Dynamics”

[LC] Lin, Ming and John Canny, “A fast algorithm for incremental distance calculation”

[Mir PhD] Mirtich, Brian, Ph.D. Thesis, University of California at Berkeley, 1996

[Mir98] Mirtich, Brian, “Rigid Body Contact: Collision Detection to Force Computation”, MERL Technical Report, 1998

[Mir00] Mirtich, Brian, “Timewarp Rigid Body Simulation”, *Computer Graphics Proceedings*, ACM SIGGRAPH, 2000

[Netlib] http://www.netlib.org/linalg/html_templates/node19.html

[NR] Press, William H. et al., “Numerical Recipes in C: The Art of Scientific Computing”

[SOLID] http://www.win.tue.nl/cs/tt/gino/solid

[SWIFT] http://www.cs.unc.edu/~geom/SWIFT/

[V-Clip] http://www.merl.com/projects/vclip