An Introduction to Physically Based Modeling: Rigid Body Simulation II—Nonpenetration Constraints

David Baraff
Robotics Institute
Carnegie Mellon University

Part II. Nonpenetration Constraints

6 Problems of Nonpenetration Constraints

Now that we know how to write and implement the equations of motion for a rigid body, let's consider the problem of preventing bodies from inter-penetrating as they move about an environment. For simplicity, suppose we simulate dropping a point mass (i.e. a single particle) onto a fixed floor. There are several issues involved here.

Because we are dealing with rigid bodies, that are totally non-flexible, we don't want to allow any inter-penetration at all when the particle strikes the floor. (If we considered our floor to be flexible, we might allow the particle to inter-penetrate some small distance, and view that as the floor actually deforming near where the particle impacted. But we don't consider the floor to be flexible, so we don't want any inter-penetration at all.) This means that at the instant that the particle actually comes into contact with the floor, what we would like is to abruptly change the velocity of the particle. This is quite different from the approach taken for flexible bodies. For a flexible body, say a rubber ball, we might consider the collision as occurring gradually. That is, over some fairly small, but non-zero span of time, a force would act between the ball and the floor and change the ball's velocity. During this time span, the ball would deform, due to the force. The more rigid we made the ball, the less the ball would deform, and the faster this collision would occur. In the limiting case, the ball is infinitely rigid, and can't deform at all. Unless the ball's downward velocity is halted instantaneously, the ball will inter-penetrate the floor somewhat. In rigid body dynamics then, we consider collisions as occurring instantaneously.

This means we have two types of contact we need to deal with. When two bodies are in contact at some point p, and they have a velocity towards each other (as in the particle striking the floor), we call this *colliding contact*. Colliding contact requires an instantaneous change in velocity. Whenever a collision occurs, the state of a body, which describes both position, and velocity, undergoes a discontinuity in the velocity. The numerical routines that solve ODE's do so under the assumption that the state $Y(t)$ always varies smoothly. Clearly, requiring $Y(t)$ to change discontinuously when a collision occurs violates that assumption.

We get around this problem as follows. If a collision occurs at time t_c, we tell the ODE solver to stop. We then take the state at this time, $Y(t_c)$, and compute how the velocities of bodies involved in the collision must change. We'll call the state reflecting these new velocities $Y(t_c)^+$. Note that $Y(t_c)$ and $Y(t_c)^+$ agree for all spatial variables (position and orientation), but will be different for the velocity variables of bodies involved in the collision at time t_c. We then restart the numerical solver, with the new state $Y(t_c)$, and instruct it to simulate forward from time t_c.

Whenever bodies are resting on one another at some point p (e.g. imagine the particle in contact with the floor with zero velocity), we say that the bodies are in *resting contact*. In this case, we compute a force that prevents the particle from accelerating downwards; essentially, this force is the weight of the particle due to gravity (or whatever other external forces push on the particle). We call the force between the particle and the floor a *contact force*. Resting contact clearly doesn't require us to stop and restart the ODE solve at every instant; from the ODE solver's point of view, contact forces are just a part of the force returned by `Compute_Force_and_Torque`.

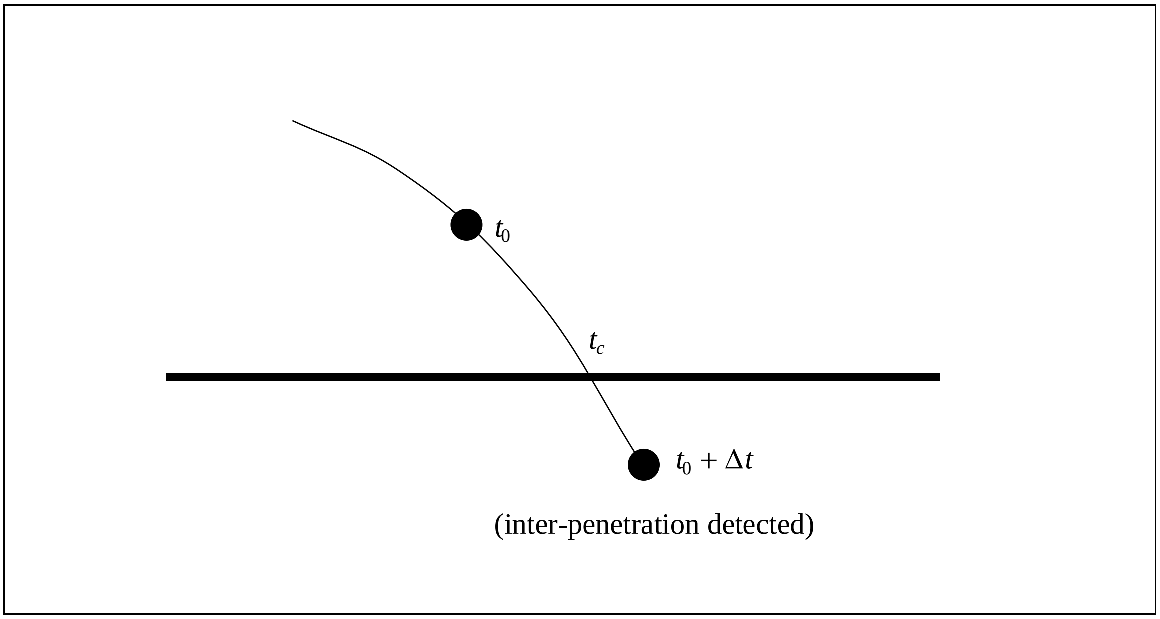

Figure 13: At time $t_0 + \Delta t$, the particle is found to lie below the floor. Thus, the actual time of collision t_c lies between the time of the last known legal position, t_0, and $t_0 + \Delta t$.

So far then, we have two problems we'll need to deal with: computing velocity changes for *colliding contact*, and computing the contact forces that prevent inter-penetration. But before we can tackle these problems we have to deal with the geometric issue of actually detecting contact between bodies. Let's go back to dropping the particle to the floor. As we run our simulation, we compute the position of the particle as it drops towards the floor at specific time values (figure 13). Suppose we consider the particle at times t_0, $t_0 + \Delta t$, $t_0 + 2\Delta t$ etc.[5] and suppose the time of collision, t_c, at which the particle actually strikes the floor, lies between t_0 and $t_0 + \Delta t$. Then at time t_0, we find that the particle lies above the floor, but at the next time step, $t_0 + \Delta t$, we find the particle is beneath the floor, which means that inter-penetration has occurred.

If we're going to stop and restart the simulator at time t_c, we'll need to compute t_c. All we know so far is that t_c lies between t_0 and $t_0 + \Delta t$. In general, solving for t_c exactly is difficult, so we solve for t_c numerically, to within a certain tolerance. A simple way of determining t_c is to use a numerical method called *bisection*[14]. If at time $t_0 + \Delta t$ we detect inter-penetration, we inform the ODE solver that we wish to restart back at time t_0, and simulate forward to time $t_0 + \Delta t/2$. If the simulator reaches $t_0 + \Delta t/2$ without encountering inter-penetration, we know the collision time t_c lies between $t_0 + \Delta t/2$ and $t_0 + \Delta t$. Otherwise, t_c is less than $t_0 + \Delta t/2$, and we try to simulate from t_0 to $t_0 + \Delta t/4$. Eventually, the time of collision t_c is computed to within some suitable numerical tolerance. The accuracy with which t_c is found depends on the collision detection routines. The collision detection routines have some parameter ϵ. We decide that our computation of t_c is "good enough" when the particle inter-penetrates the floor by no more than ϵ, and is less than ϵ above the floor. At this point we declare that the particle is in contact with the floor (figure 14).

The method of bisection is a little slow, but its easy to implement and quite robust. A faster method involves actually predicting the time t_c of the collision, based on examining $\mathbf{Y}(t_0)$ and $\mathbf{Y}(t_0 + \Delta t)$. Baraff[1, 2] describes how to make such predictions. How to actually implement all of

[5] The ODE solver doesn't have to proceed with equal size time steps though.

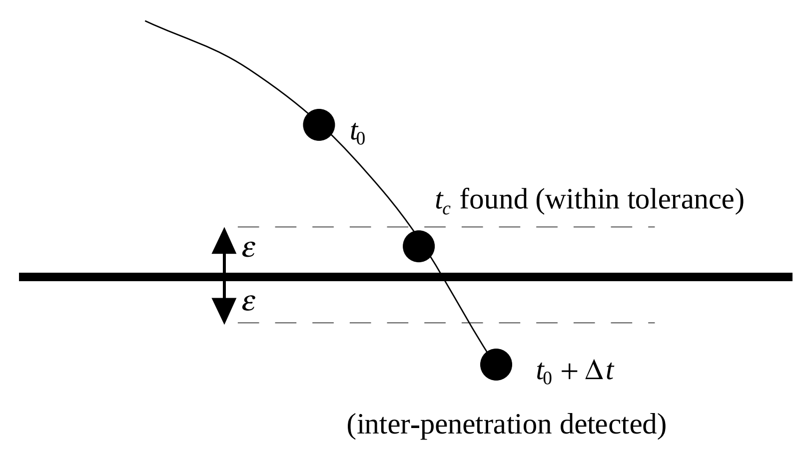

Figure 14: When the particle is found to be within some tolerance ϵ of contacting the floor, then t_c is considered to have been computed to within sufficient accuracy.

this depends on how you interact with your ODE routines. One might use exception handling code to signal the ODE of various events (collisions, inter-penetration), or pass some sort of messages to the ODE solver. We'll just assume that you have some way of getting your ODE solver to progress just up to the point t_c.

Once you actually reach the time of a collision, or whenever you're in a state $Y(t)$ where no inter-penetration has occurred, a geometric determination has to be made to find all the points of contact. (Just because you may be looking for the time of collision between two bodies A and B doesn't mean you get to neglect resting contact forces between other bodies C and D. Whenever you're trying to move the simulation forward, you'll need to compute the point of contact between bodies and the contact forces at those points.) There is a vast amount of literature dealing with the collision detection problem. For instance, some recent SIGGRAPH papers dealing with the subject are Von Herzen, Barr and Zatz[17] and Moore and Wilhelms[12]; in robotics, a number of papers of interest are Canny[4], Gilbert and Hong[6], Meyer[11] and Cundall[5]. Preparata and Shamos[13] describes many approaches in computational geometry to the problem. In the next section, we'll briefly describe a collision detection "philosophy" that leads to very efficient algorithms, for the sorts of simulation these course notes are concerned with. Actual code for the algorithms is fairly easy to write, but a little too lengthy to fit in these notes. Following this, we'll move on to consider colliding and resting contact.

7 Collision Detection

The collision detection algorithm begins with a preprocessing step, in which a bounding box for each rigid body is computed (a box with sides parallel to the coordinate axes). Given n such bounding boxes, we will want to quickly determine all pairs of bounding boxes that overlap. Any pair of rigid bodies whose bounding boxes do not overlap need not be considered any further. Pairs of rigid bodies whose bounding boxes do overlap require further consideration. We'll first describe how to efficiently check for inter-penetration or contact points between rigid bodies defined as convex polyhedra. Then we'll show how to perform the bounding box check efficiently.

As described in section 1, the simulation process consists of the repeated computation of the derivative of the state vector, $\frac{d}{dt}\mathbf{Y}(t)$, at various times t. The numerical ODE solver is responsible for choosing the values of t at which the state derivative is to be computed. For any reasonably complicated simulation, the values of t chosen are such that the state $\mathbf{Y}$ does not change greatly between successive values of t. As a result, there is almost always great geometric coherence between successive time steps. At a time step $t_0 + \Delta t$, the idea is to take advantage of the collision detection results computed at the previous time step t_0.

7.1 Convex Polyhedra

Our primary mechanism for exploiting coherence will be through the use of *witnesses*. In our context, given two convex polyhedra A and B, a witness is some piece of information that can be used to quickly answer the "yes/no" question "are A and B disjoint"? We will utilize coherence by caching witnesses from one time step to the next; hopefully a witness from the previous time step will be a witness during the current time step.

Since we are considering convex polyhedra, two polyhedra do not inter-penetrate if and only if a separating plane between them exists. A separating plane between two polyhedra is a plane such that each polyhedron lies on a different side of the plane. A given plane can be verified to be a separating plane by testing to make sure that all of the vertices of A and B lie on opposite sides of the plane. Thus, a separating plane is a witness to the fact that two convex polyhedra do not inter-penetrate. If a separating plane does not exist, then the polyhedra must be inter-penetrating.

The cost of initially finding a witness (for the very first time step of the simulation, or the first time two bodies become close enough to require more than a bounding box test) is unavoidable. A simple way to find a separating plane initially is as follows. If a pair of convex polyhedra are disjoint or contacting (but not inter-penetrating), then a separating plane exists with the following property: either the plane contains a face of one of the polyhedra, or the plane contains an edge from one of the polyhedra and is parallel to an edge of the other polyhedra. (That is, the separating plane's normal is the cross product of the two edge directions, and the plane itself contains one of the edges.) We will call the face or edges in question the *defining* face or edges. Initially, we simply check all possible combinations of faces and edges to see if one such combination forms a separating plane (figure 15). Although this is inefficient, it's done so infrequently that the inefficiency is unimportant. For subsequent time steps, all we need to do is form a separating plane from the defining face or edges found during the previous time step, and then verify the plane to see that it is still valid (figure 16).

On those (rare) occasions when the cached face or two edges fails to form a valid separating plane (figure 17), faces or edges adjacent to the previously cached face or edges can be examined to see if they form a separating plane; however, this happens infrequently enough that it may be simpler to start from scratch and compute a new separating plane without using any prior knowledge.

Once the separating place has been found, the contact region between the two polyhedra is determined, assuming the polyhedra are not disjoint. Contact points between the two polyhedra can only occur on the separating plane. Given the separating plane, the contact points can be quickly and efficiently determined by comparing only those faces, edges, and vertices of the polyhedra that are coincident with the separating plane.

However, if no separating plane can be found, then the two polyhedra must be inter-penetrating. When two polyhedra inter-penetrate, it is almost always the case that either a vertex of one poly-

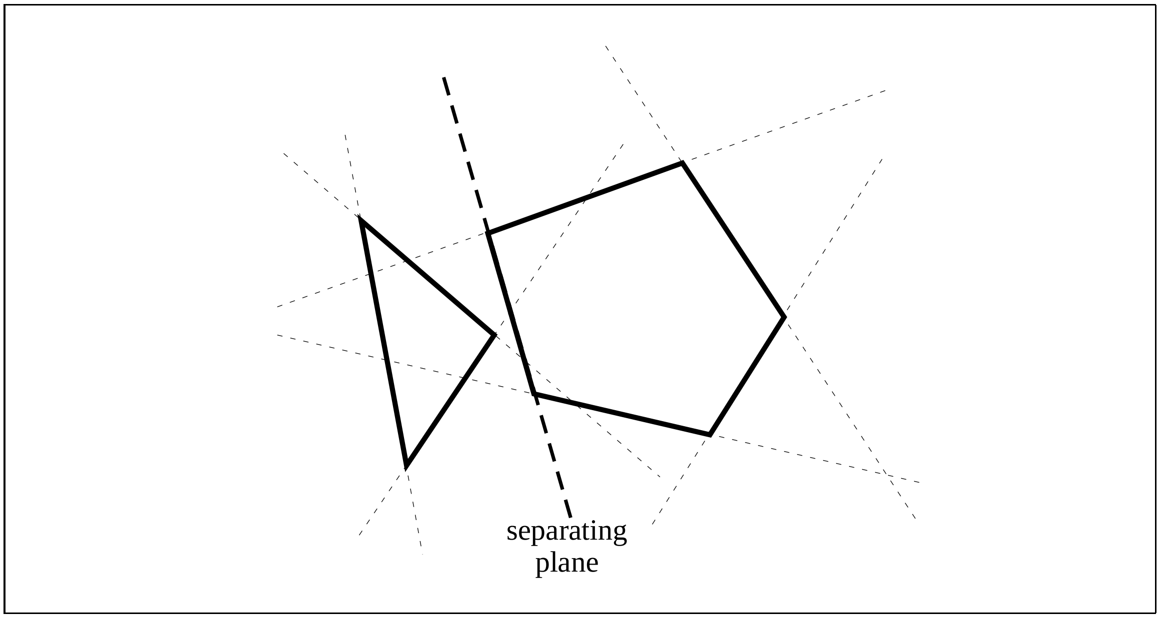

Figure 15: Exhaustive search for a separating plane. Only one face of the two polygons forms a separating plane.

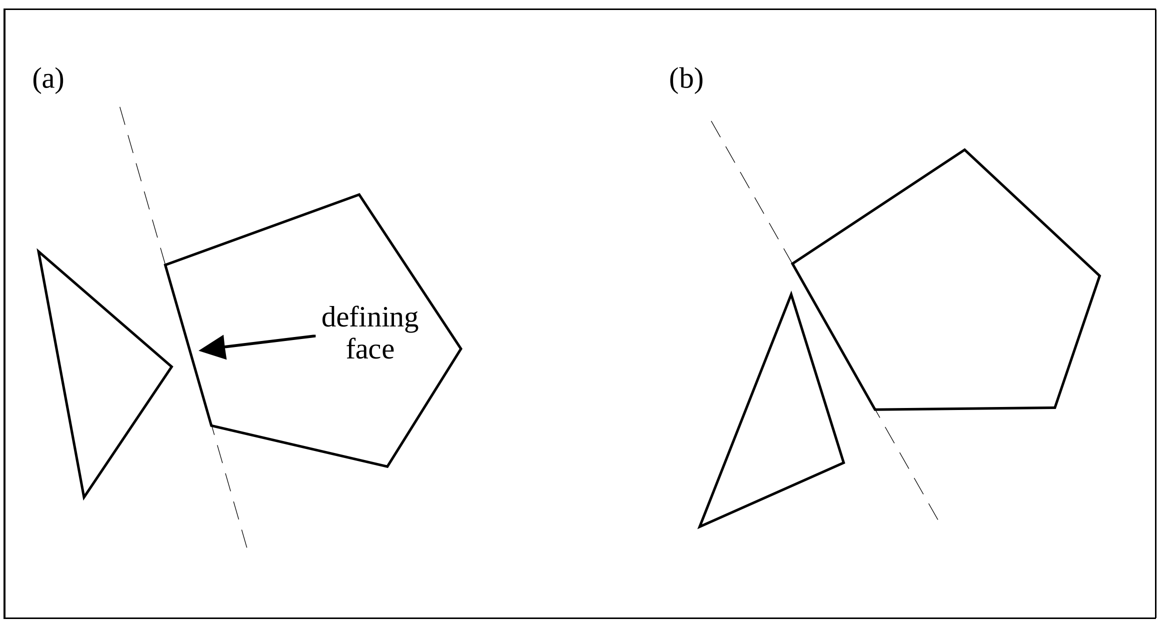

Figure 16: (a) At this time step, the separating plane is defined by a face of one of the polygons. (b) At the next time step, the polygons have moved, but the same face still defines a separating plane.

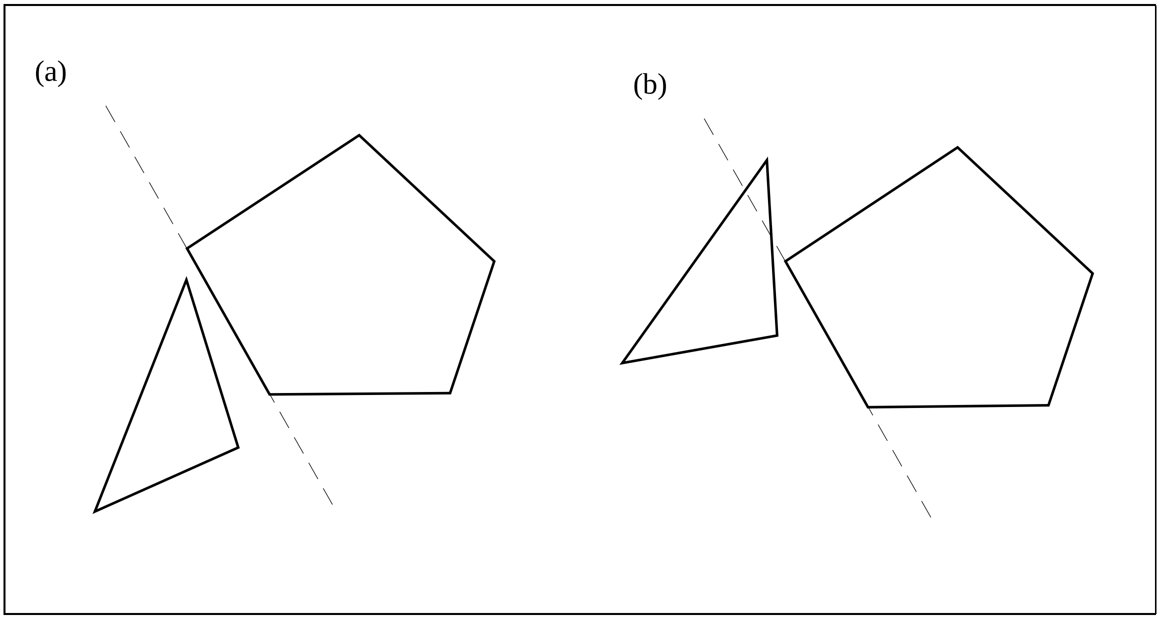

Figure 17: The face that has been defining a separating plane no longer does so, and a new separating plane must be found.

hedron is inside the other, or an edge of one polyhedron has intersected a face of the other.[6] In this case, the inter-penetrating vertex, or intersecting edge and face are cached as a witness to the inter-penetration. Since this indicates a collision at some earlier time, the simulator will back up and attempt to compute $\frac{d}{dt}\mathbf{Y}(t)$ at some earlier time. Until the collision time is determined, the first action taken by the collision/contact determination step will be to check the cached vertex or edge and face to see if they indicate inter-penetration. Thus, until the collision time is found, states in which the inter-penetration still exists are identified as such with a minimum of computational overhead.

7.2 Bounding Boxes

To reduce the number of pairwise collision/contact determinations necessary, a bounding box hierarchy is imposed on the bodies in the simulation environment. If two bounding boxes are found not to overlap, no further comparisons involving the contents of the boxes are needed. Given a collection of n rectangular bounding boxes, aligned with the coordinate axes, we would like to efficiently determine all pairs of boxes that overlap. A naive pairwise comparison of all pairs requires $O(n^2)$ work and is too inefficient, unless the number of bodies is small. Computational geometry algorithms exist that can solve this problem in time $O(nlogn+k)$ where k is the number of pairwise overlaps; a general result is that the problem can be solved in time $O(nlog^{d-2}n+k)$ for d-dimensional bounding boxes[13]. Using coherence, we can achieve substantially better performance.

[6] An exception is the following. Stack two cubes of equal size atop one another so that their contacting faces exactly coincide. Lower the top one. This produces an inter-penetration such that no vertex is inside either cube, and no edge penetrates through any face.

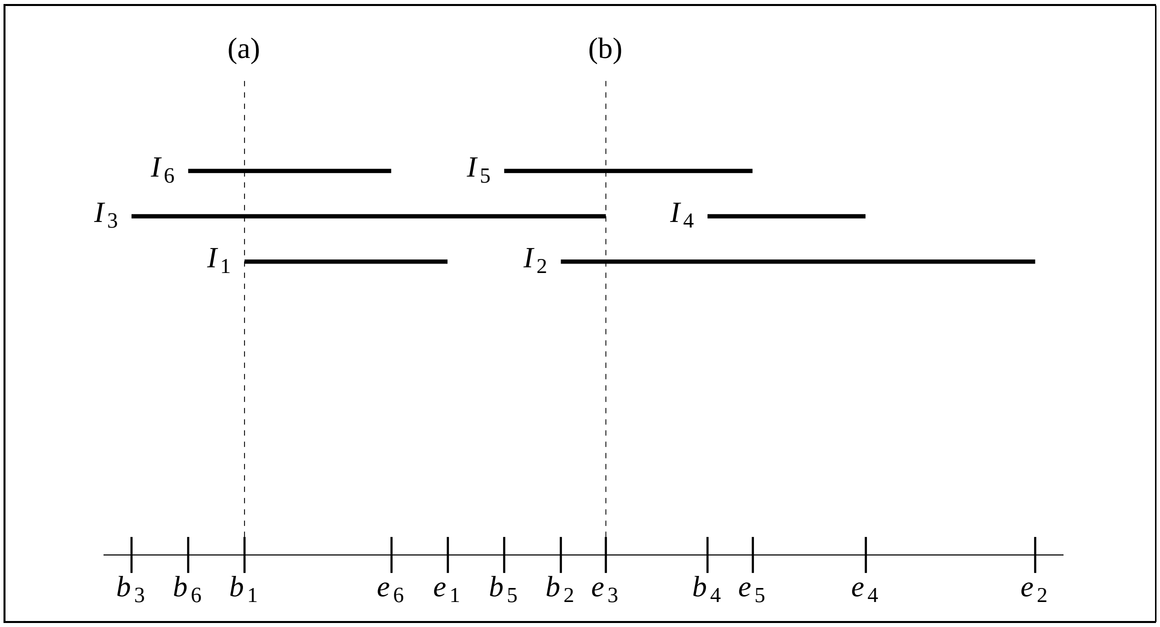

Figure 18: The sweep/sort algorithm. (a) When b_1 is encountered, the active list contains intervals 3 and 6; interval 1 is reported to overlap with these two intervals. Interval 1 is added to the active list and the algorithm continues. (b) When e_3 is encountered, the active list contains intervals 2, 3 and 5. Interval 3 is removed from the active list.

7.2.1 The one-dimensional case

Consider the problem of detecting overlap between *one*-dimensional bounding boxes, aligned with the coordinate system. Such a bounding box can be described simply as an interval $[b, e]$ where b and e are real numbers. Let us consider a list of n such intervals, with the ith interval being $[b_i, e_i]$. The problem is then defined to be the determination of all pairs i and j such that the intervals $[b_i, e_i]$ and $[b_j, e_j]$ intersect.

The problem can be solved initially by a *sort and sweep* algorithm. A sorted list of all the b_i and e_i values is created, from lowest to highest. The list is then swept, and a list of *active* intervals, initially empty, is maintained. Whenever some value b_i is encountered, all intervals on the active list are output as overlapping with interval i, and interval i is then added to the list (figure 18a). Whenever some value e_i is encountered , interval i is removed from the active list (figure 18b). The cost of this process is $O(n \log n)$ to create the sorted list, $O(n)$ to sweep through the list, and $O(k)$ to output each overlap. This gives a total cost of $O(n \log n + k)$, and is an optimal algorithm for initially solving the problem.

Subsequent comparisons can be improved as follows. First, there is no need to use an $O(n \log n)$ algorithm to form the sorted list of b_i and e_i values. It is considerably more efficient to start with the order found for b_i and e_i values from the previous time step; if coherence is high, this ordering will be nearly correct for the current time step. A sorting method called an *insertion sort*[15] is used to permute the "nearly sorted" list into a sorted list. The insertion sort algorithm works by moving items towards the beginning of the list, until a smaller item is encountered. Thus, the second item is interchanged with the first if necessary, then the third item is moved towards the beginning of the list until its proper place is found, and so on; each movement of an item indicates a change in the ordering of two values. After the last item on the list has been processed, the list is in order. Such

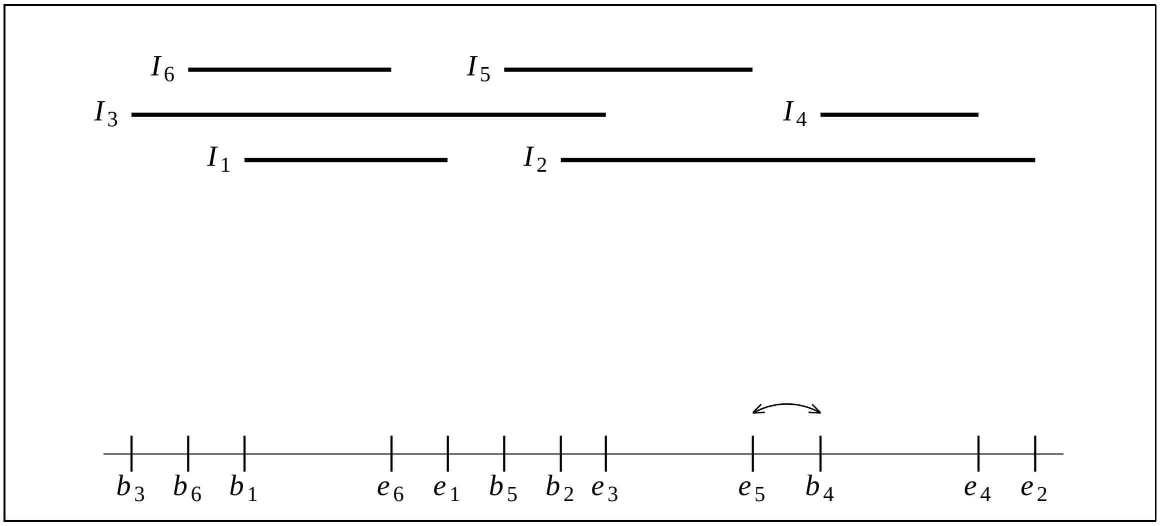

Figure 19: A coherence-based method of detecting overlaps. The order produced in figure 18 is nearly correct for this arrangement of intervals. Only b_4 and e_5 need to be exchanged. When the exchange occurs, the change in overlap status between interval 4 and 5 is detected.

a sort takes time $O(n+c)$ where c is the number of exchanges necessary. For example, the only difference between figures 19 and 18 is that interval 4 has moved to the right. Starting from the ordered list of b_i and e_i values of figure 18, only a single exchange is necessary to sort the list for figure 19. The insertion sort is not recommendeded as a sorting procedure in general, since it may require $O(n^2)$ exchanges; however, it is a good algorithm for sorting a nearly sorted list, which is what occurs in our highly coherent environment. To complete the algorithm, note that if two intervals i and j overlap at the previous time step, but not at the current time step, one or more exchanges involving either a b_i or e_i value and a b_j or e_j value must occur. The converse is true as well when intervals i and j change from not overlapping at the previous time step to overlapping at the current time step.

Thus, if we maintain a table of overlapping intervals at each time step, the table can be updated at each time step with a total cost of $O(n+c)$. Assuming coherence, the number of exchanges c necessary will be close to the actual number k of changes in overlap status, and the extra $O(c-k)$ work will be negligible. Thus, for the one-dimensional bounding box problem, the coherence view yields an efficient algorithm of extreme (if not maximal) simplicity that approaches optimality as coherence increases.

7.2.2 The three-dimensional case

Efficient computational geometry algorithms for solving the bounding box intersection problem in $\mathbb{R}^3$ are much more complicated than the sort and sweep method for the one-dimensional case. However, these algorithms all have in common a step that is essentially a sort along a coordinate axis, as in the one-dimensional case. Each bounding box is described as three independent intervals $[b_i^{(x)}, e_i^{(x)}]$, $[b_i^{(y)}, e_i^{(y)}]$, and $[b_i^{(z)}, e_i^{(z)}]$ which represent the intervals spanned on the three coordinate axes by the ith bounding box. Thus, our first thought towards improving the efficiency of a computational geometry algorithm for coherent situations would be to sort a list containing the $b_i^{(x)}$ and $e_i^{(x)}$ values, and similarly for the y and z axes. Again, such a step will involve $O(n+c)$ work, where c is now the total number of exchanges involved in sorting all three lists. However, if we observe that checking two bounding boxes for overlap is a constant time operation, it follows that if we simply check bounding boxes i and j for overlap whenever an exchange is made between values indexed by i and j (on any coordinate axis), we will detect all changes in overlap status in $O(n+c)$ time.

Again, we can maintain a table of overlapping bounding boxes, and update it at each time step in $O(n+c)$ time. The extra work involved is again $O(c-k)$. For the three-dimensional case, extra work can occur if the extents of two bounding boxes change on one coordinate axis without an actual change of their overlap status. In practice, the extra work done has been found to be completely negligible, and the algorithm runs essentially in time $O(n+k)$.

8 Colliding Contact

For the remainder of these notes, we're going to be concerned with examining the bodies in our simulator at a particular instant of time t_0. At this time t_0, we assume that no bodies are inter-penetrating, and that the simulator has already determined which bodies contact, and at which points. To simplify matters, we'll imagine that all bodies are polyhedra, and that every contact point between bodies has been detected. We'll consider contacts between polyhedra as either *vertex/face* contacts or *edge/edge* contacts. A vertex/face contact occurs when a vertex on one polyhedra is in contact with a face on another polyhedra. An edge/edge contact occurs when a pair of edges contact; it is assumed in this case that the two edges are not collinear. (Vertex/vertex and vertex/edge contacts are degenerate, and are not considered in these notes.) As examples, a cube resting on a plane would be described as four vertex/face contacts, one contact at each corner of the cube. A cube resting on a table, but with its bottom face hanging over the edge of the table would still be described as four contacts; two vertex/face contacts for the vertices on the table, and two edge/edge contacts, one on each edge of the cube that crosses over an edge of the table.

Each contact is represented by a structure

```
struct Contact {
        RigidBody       *a,     /* body containing vertex */
                        *b;     /* body containing face */
        triple          p,      /* world-space vertex location */
                        n,      /* outwards pointing normal of face */
                        ea,     /* edge direction for A */
                        eb;     /* edge direction for B */

        boolean         vf;     /* TRUE if vertex/face contact */
};

int     Ncontacts;
Contact *Contacts;
```

If the contact is a vertex/face contact, then the variable a points to the rigid body that the contact vertex is attached to, while b points to the body the face is attached to. We'll call these two bodies A and B respectively. For vertex/face contacts, the variable n is set to the outwards pointing unit normal of the contact face of body B, and the variables ea and eb are unused.

For edge/edge contacts, ea is a triple of unit length, that points in the direction of the contacting

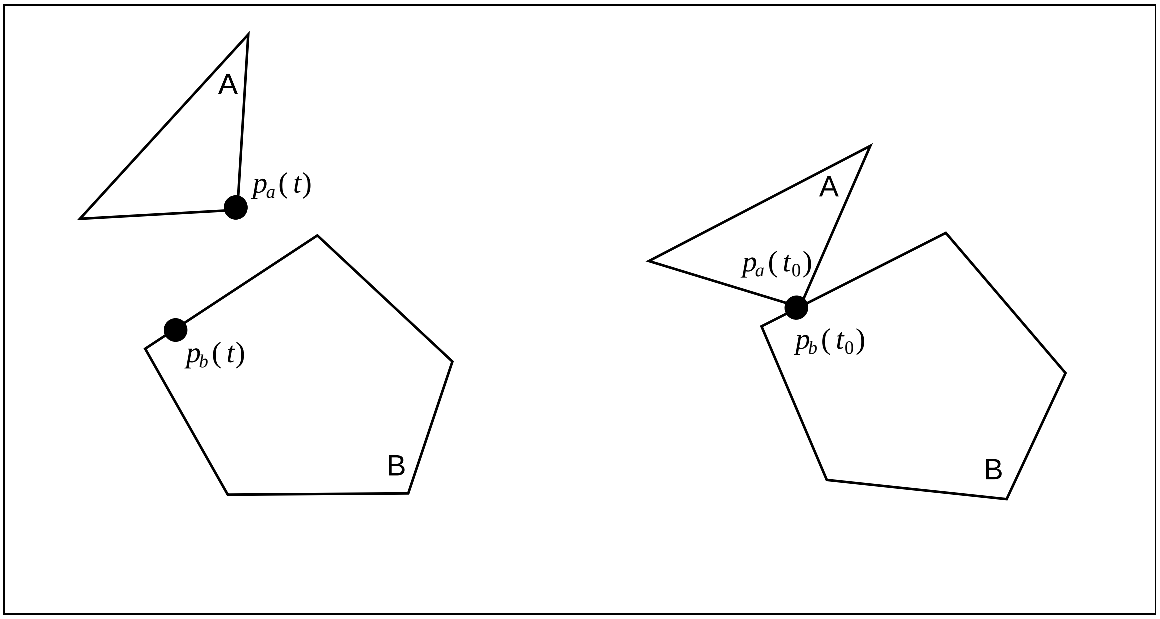

Figure 20: (a) The points $p_a(t)$ and $p_b(t)$ for a vertex/face contact. (b) At time t_0, the bodies come into contact at $p_a(t_0) = p_b(t_0)$.

edge of body A (pointed to by `a`). Similarly, `eb` is a unit vector giving the direction that the contact edge on body B points. For edge/edge contacts, `n` denotes a unit vector in the `ea` $\times$ `eb` direction. We'll adopt the convention that the two contacting bodies are labeled A and B such that the normal direction `ea` $\times$ `eb` points outwards from B, towards A, as it does for vertex/face contacts.

For both types of contact, the position of the contact in world space (which is either the contact vertex, or the point where the two edges intersect) is given by `p`. The collision detection routines are responsible for discovering all the contact points, setting `Ncontacts` to the number of contact points, and allocating space for and initializing an array of `Contact` structures.

The first thing we'll need to do is examine the data in each `Contact` structure to see if colliding contact is taking place. For a given contact point, the two bodies A and B contact at the point p. Let $p_a(t)$ denote the particular the point on body A that satisfies $p_a(t_0) = p$. (For vertex/face contacts, this point will be the vertex itself. For edge/edge contacts, it is some particular point on the contact edge of A.) Similarly, let $p_b(t)$ denote the particular point on body B that coincides with $p_a(t_0) = p$ at time t_0 (figure 20). Although $p_a(t)$ and $p_b(t)$ are coincident at time t_0, the *velocity* of the two points at time t_0 may be quite different. We will examine this velocity to see if the bodies are colliding or not.

From section 2.5, we can calculate the velocity of the vertex point, $\dot{p}_a(t_0)$ by the formula

$$\dot{p}_a(t_0) = v_a(t_0) + \omega_a(t_0) \times (p_a(t_0) - x_a(t_0)) \tag{8–1}$$

where $v_a(t)$ and $\omega_a(t)$ are the velocities for body A. Similarly, the velocity of the contact point on the face of B is

$$\dot{p}_b(t_0) = v_b(t_0) + \omega_b(t_0) \times (p_b(t_0) - x_b(t_0)). \tag{8–2}$$

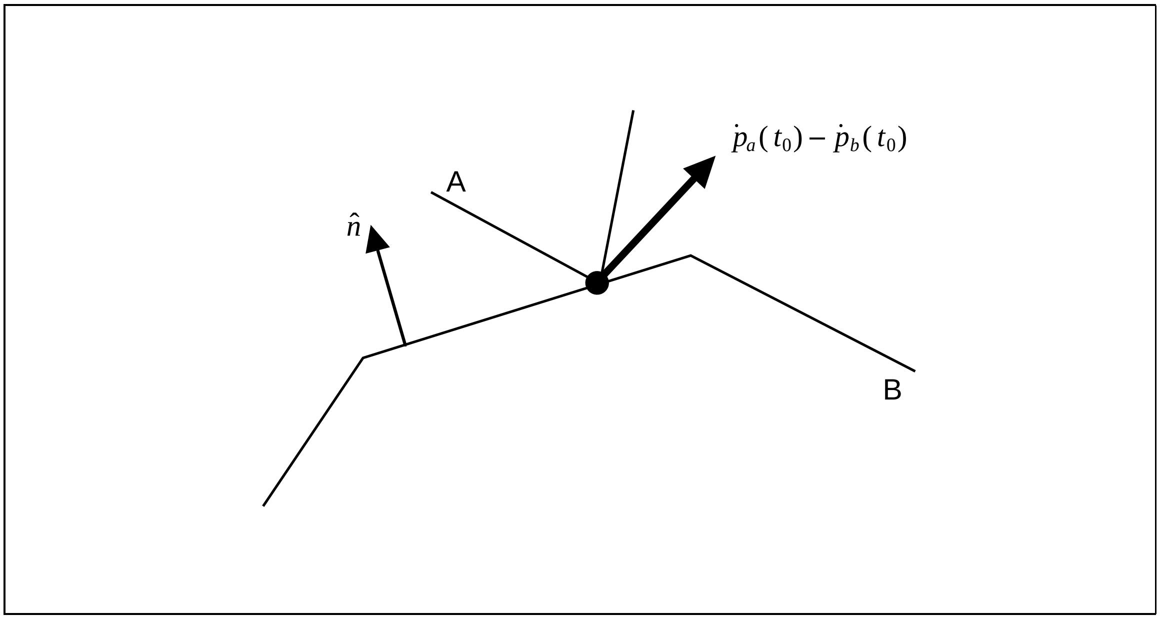

Figure 21: The vector $\dot{p}_a(t_0) - \dot{p}_b(t_0)$ points in the same direction as $\hat{n}(t_0)$; the bodies are separating.

Let's examine the quantity

$$v_{rel} = \hat{n}(t_0) \cdot (\dot{p}_a(t_0) - \dot{p}_b(t_0)), \tag{8–3}$$

which is a scalar. In this equation, $\hat{n}(t_0)$ is the unit surface normal, described by the variable `n`, for each contact point. The quantity v_{rel} gives the component of the relative velocity $\dot{p}_a(t_0) - \dot{p}_b(t_0)$ in the $\hat{n}(t_0)$ direction. Clearly, if v_{rel} is positive, then the relative velocity $\dot{p}_a(t_0) - \dot{p}_b(t_0)$ at the contact point is in the positive $\hat{n}(t_0)$ direction. This means that the bodies are moving apart, and that this contact point will disappear immediately after time t_0 (figure 21). We don't need to worry about this case. If v_{rel} is zero, then the bodies are neither approaching nor receding at p (figure 22). This is exactly what we mean by resting contact, and we'll deal with it in the next section.

In this section, we're interested in the last possibility, which is $v_{rel} < 0$. This means that the relative velocity at p is opposite $\hat{n}(t_0)$, and we have colliding contact. If the velocities of the bodies don't immediately undergo a change, inter-penetration will result (figure 23).

How do we compute the change in velocity? Any force we might imagine acting at p, no matter how strong, would require at least a small amount of time to completely halt the relative motion between the bodies. (No matter how strong your car brakes are, you still need to apply them *before* you hit the brick wall. If you wait until you've contacted the wall, it's too late...) Since we want bodies to change their velocity instantly though, we postulate a new quantity J called an *impulse*. An impulse is a vector quantity, just like a force, but it has the units of momentum. Applying an impulse produces an instantaneous change in the velocity of a body. To determine the effects of a given impulse J, we imagine a large force F that acts for a small time interval Δt. If we let F go to infinity and Δt go to zero in such a way that

$$F \Delta t = J \tag{8–4}$$

then we can derive the effect of J on a body's velocity by considering how the velocity would change if we let the force F act on it for Δt time.

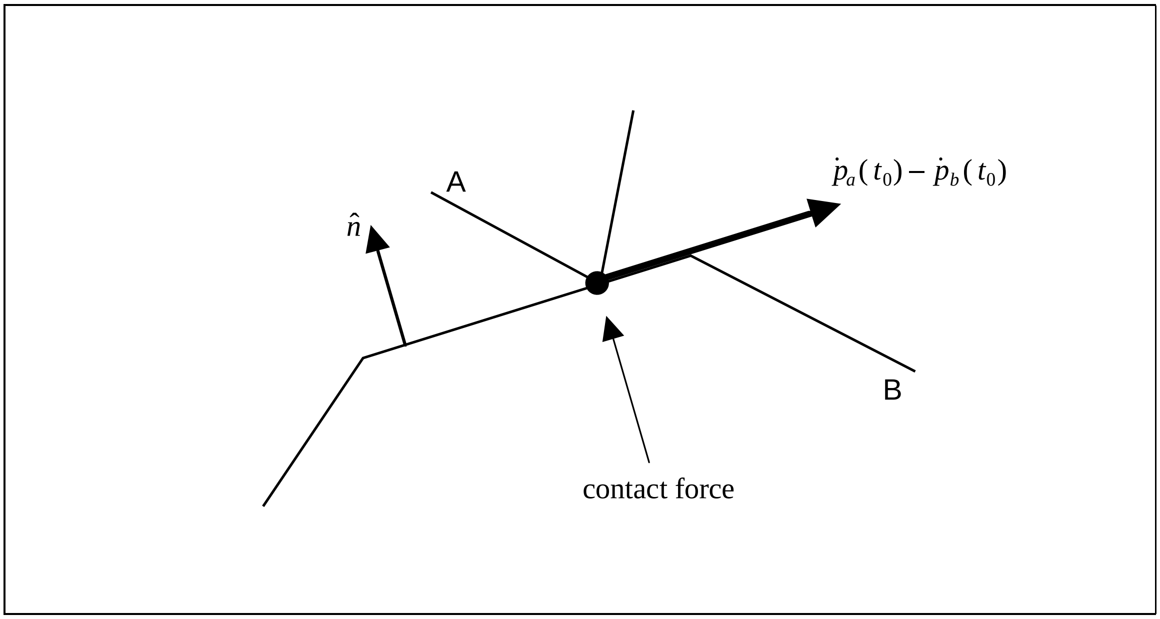

Figure 22: The vector $\dot{p}_a(t_0) - \dot{p}_b(t_0)$ is perpendicular to $\hat{n}(t_0)$; the bodies are in resting contact. A contact force may be necessary to prevent bodies from accelerating towards each other.

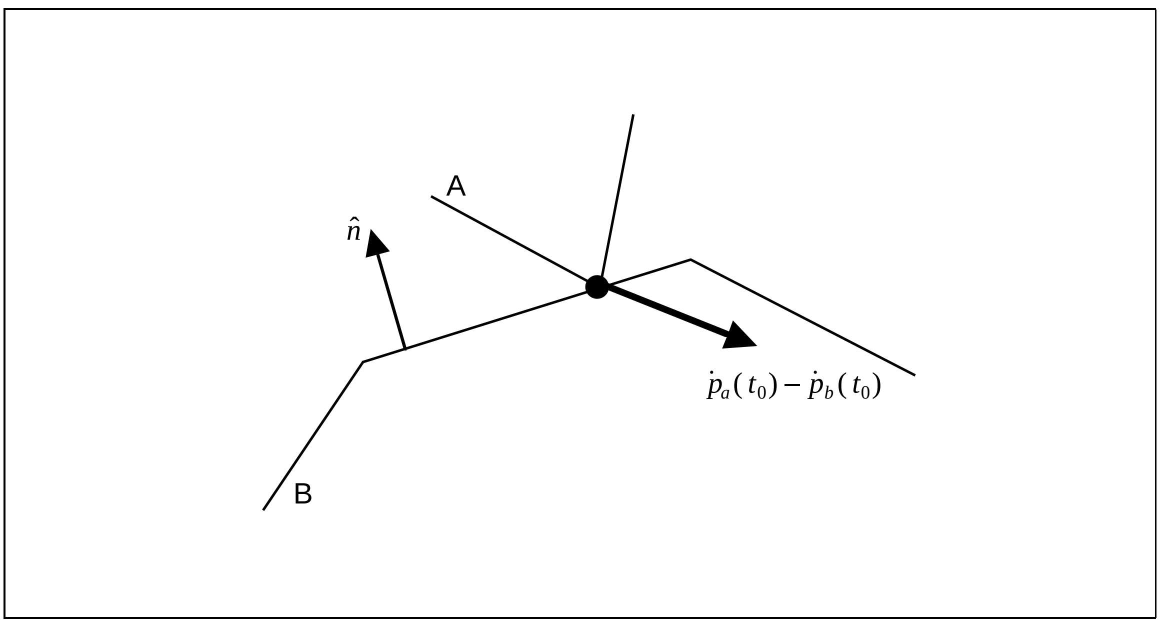

Figure 23: Colliding contact. The relative velocity $\dot{p}_a(t_0) - \dot{p}_b(t_0)$ is directed inwards, opposite $\hat{n}(t_0)$. Unless the relative velocity is abruptly changed, inter-penetration will occur immediately after time t_0.

For example, if we apply an impulse J to a rigid body with mass M, then the change in linear velocity Δv is simply

$$\Delta v = \frac{J}{M}. \tag{8–5}$$

Equivalently, the change in linear momentum ΔP is simply $\Delta P = J$. If the impulse acts at the point p, then just as a force produces a torque, J produces an impulsive torque of

$$\tau_{impulse} = (p - x(t)) \times J. \tag{8–6}$$

As one would imagine, the impulsive torque $\tau_{impulse}$ also gives rise to a change in angular momentum ΔL of $\Delta L = \tau_{impulse}$. The change in angular velocity is simply $I^{-1}(t_0)\tau_{impulse}$, assuming the impulse was applied at time t_0.

When two bodies collide, we will apply an impulse between them to change their velocity. For frictionless bodies, the direction of the impulse will be in the normal direction, $\hat{n}(t_0)$. Thus, we can write the impulse J as

$$J = j\hat{n}(t_0) \tag{8–7}$$

where j is an (as yet) undetermined scalar that gives the magnitude of the impulse. We'll adopt the convention that the impulse J acts positively on body A, that is, A is subject to an impulse of $+j\hat{n}(t_0)$, while body B is subject to an equal but opposite impulse $-j\hat{n}(t_0)$ (figure 24). We compute j by using an empirical law for collisions. Let's let $\dot{p}_a^-(t_0)$ denote the velocity of the contact vertex of A prior to the impulse being applied, and let $\dot{p}_a^+(t_0)$ denote the velocity after we apply the impulse J. Let $\dot{p}_b^-(t_0)$ and $\dot{p}_b^+(t_0)$ be defined similarly. Using this notation, the initial relative velocity in the normal direction is

$$v_{rel}^- = \hat{n}(t_0) \cdot (\dot{p}_a^-(t_0) - \dot{p}_b^-(t_0)); \tag{8–8}$$

after the application of the impulse,

$$v_{rel}^+ = \hat{n}(t_0) \cdot (\dot{p}_a^+(t_0) - \dot{p}_b^+(t_0)). \tag{8–9}$$

The empirical law for frictionless collisions says simply that

$$v_{rel}^+ = -\epsilon v_{rel}^-. \tag{8–10}$$

The quantity ϵ is called the *coefficient of restitution* and must satisfy $0 \le \epsilon \le 1$. If $\epsilon = 1$, then $v_{rel}^+ = -v_{rel}^-$, and the collision is perfectly "bouncy"; in particular, no kinetic energy is lost. At the other end of the spectrum, $\epsilon = 0$ results in $v_{rel}^+ = 0$, and a maximum of kinetic energy is lost. After this sort of collision, the two bodies will be in resting contact at the contact point p (figure 25).

Calculating the magnitude j of the impulse $J = j\hat{n}(t_0)$ is fairly simple, although the equations are a bit tedious to work through. Let's define the displacements r_a and r_b as $p - x_a(t_0)$, and $p - x_b(t_0)$. If we let $v_a^-(t_0)$ and $\omega_a^-(t_0)$ be the pre-impulse velocities of body A, and $v_a^+(t_0)$ and $\omega_a^+(t_0)$ be the post-impulse velocities, we can write

$$\dot{p}_a^+(t_0) = v_a^+(t_0) + \omega_a^+(t_0) \times r_a \tag{8–11}$$

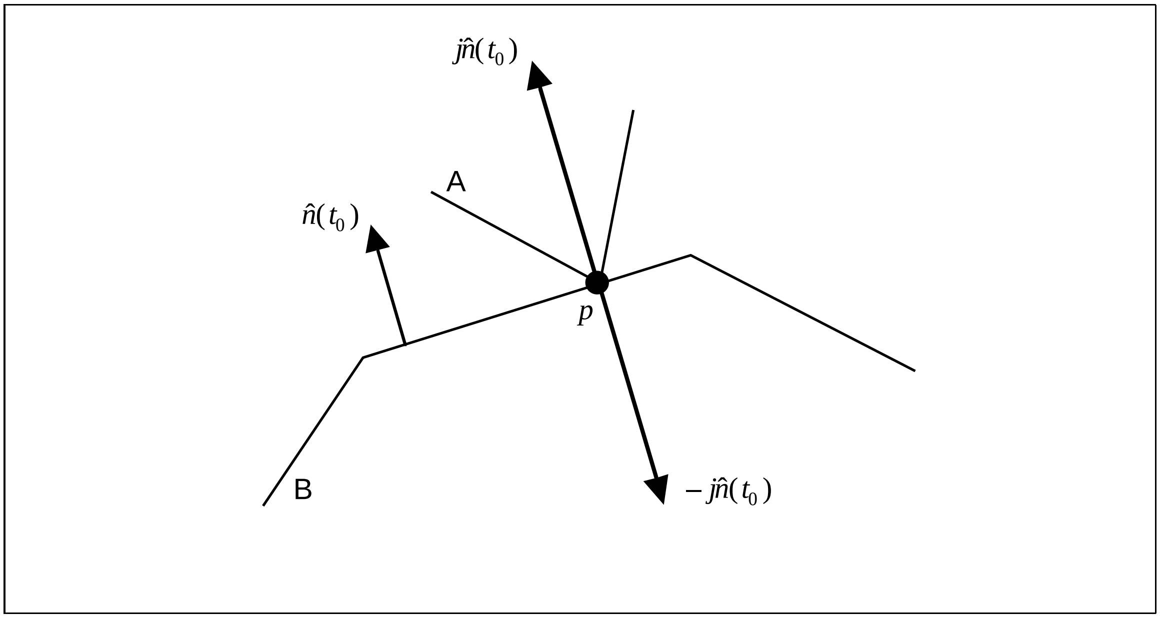

Figure 24: The impulse between two bodies at a contact point. An impulse of $j\hat{n}(t_0)$ acts on A, while an impulse of $-j\hat{n}(t_0)$ acts on B.

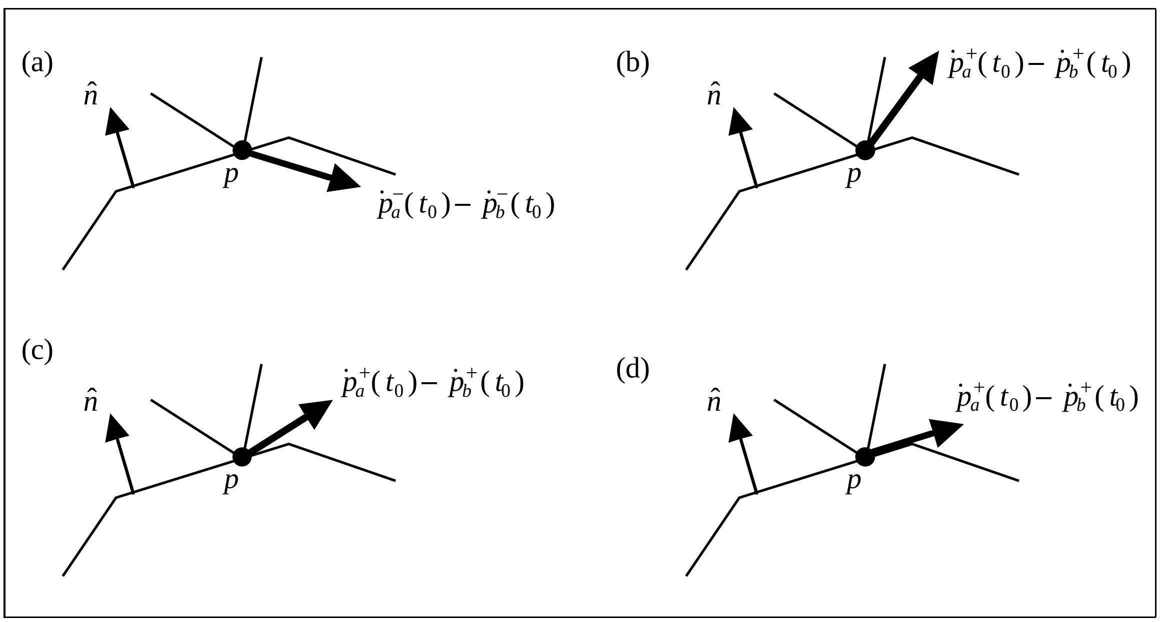

Figure 25: (a) The relative velocity before application of the impulse. (b) The component of the relative velocity in the $\hat{n}(t_0)$ direction is reversed for an $\epsilon = 1$ collision. The relative velocity perpendicular to $\hat{n}(t_0)$ remains the same. (c) A collision with $0 < \epsilon < 1$. The bodies bounce away in the $\hat{n}(t_0)$ direction with less speed than they approached. (d) A collision with $\epsilon = 0$. The bodies do not bounce away from each other, but the relative velocity perpendicular to $\hat{n}(t_0)$ is unaffected by the collision.

along with

$$v_a^+(t_0) = v_a^-(t_0) + \frac{j\hat{n}(t_0)}{M_a} \qquad \text{and} \qquad \omega_a^+(t_0) = \omega_a^-(t_0) + I_a^{-1}(t_0)\left(r_a \times j\hat{n}(t_0)\right) \tag{8–12}$$

where M_a is the mass of body A, and $I_a(t_0)$ is its inertia tensor. Combining the two previous equations yields

$$\begin{aligned}\dot{p}_a^+(t_0) &= \left(v_a^-(t_0) + \frac{j\hat{n}(t_0)}{M_a}\right) + \left(\omega_a^-(t_0) + I_a^{-1}(t_0)\left(r_a \times j\hat{n}(t_0)\right)\right) \times r_a \\ &= v_a^-(t_0) + \omega_a^-(t_0) \times r_a + \left(\frac{j\hat{n}(t_0)}{M_a}\right) + \left(I_a^{-1}(t_0)\left(r_a \times j\hat{n}(t_0)\right)\right) \times r_a \\ &= \dot{p}_a^- + j\left(\frac{\hat{n}(t_0)}{M_a} + I_a^{-1}(t_0)\left(r_a \times \hat{n}(t_0)\right)\right) \times r_a.\end{aligned} \tag{8–13}$$

It is important to note the form of $\dot{p}_a^+(t_0)$: it is a simple linear function of j. For body B, an opposite impulse $-j\hat{n}(t_0)$ acts, yielding

$$\dot{p}_b^+(t_0) = \dot{p}_b^- - j\left(\frac{\hat{n}(t_0)}{M_b} + I_b^{-1}(t_0)\left(r_b \times \hat{n}(t_0)\right)\right) \times r_b. \tag{8–14}$$

This yields

$$\begin{aligned}\dot{p}_a^+(t_0) - \dot{p}_b^+ = \left(\dot{p}_a^+(t_0) - \dot{p}_b^+\right) + j\Bigg(&\frac{\hat{n}(t_0)}{M_a} + \frac{\hat{n}(t_0)}{M_b} + \\ &\left(I_a^{-1}(t_0)\left(r_a \times \hat{n}(t_0)\right)\right) \times r_a + \left(I_b^{-1}(t_0)\left(r_b \times \hat{n}(t_0)\right)\right) \times r_b\Bigg).\end{aligned} \tag{8–15}$$

To calculate v_{rel}^+, we dot this expression with $\hat{n}(t_0)$. Since $\hat{n}(t_0)$ is of unit length, $\hat{n}(t_0) \cdot \hat{n}(t_0) = 1$, and we obtain

$$\begin{aligned}v_{rel}^+ &= \hat{n}(t_0) \cdot (\dot{p}_a^+(t_0) - \dot{p}_b^+) \\ &= \hat{n}(t_0) \cdot (\dot{p}_a^-(t_0) - \dot{p}_b^-) + j\Bigg(\frac{1}{M_a} + \frac{1}{M_b} + \\ &\quad \hat{n}(t_0) \cdot \left(I_a^{-1}(t_0)\left(r_a \times \hat{n}(t_0)\right)\right) \times r_a + \hat{n}(t_0) \cdot \left(I_b^{-1}(t_0)\left(r_b \times \hat{n}(t_0)\right)\right) \times r_b\Bigg) \\ &= v_{rel}^- + j\Bigg(\frac{1}{M_a} + \frac{1}{M_b} + \\ &\quad \hat{n}(t_0) \cdot \left(I_a^{-1}(t_0)\left(r_a \times \hat{n}(t_0)\right)\right) \times r_a + \hat{n}(t_0) \cdot \left(I_b^{-1}(t_0)\left(r_b \times \hat{n}(t_0)\right)\right) \times r_b\Bigg).\end{aligned} \tag{8–16}$$

By expressing v_{rel}^+ in terms of j and v_{rel}^-, we can compute j according to equation (8–10). If we substitute equation (8–16) into equation (8–10), we get

$$\begin{aligned}v_{rel}^- + j\Bigg(\frac{1}{M_a} + \frac{1}{M_b} + \hat{n}(t_0) \cdot \left(I_a^{-1}(t_0)\left(r_a \times \hat{n}(t_0)\right)\right) \times r_a + \\ \hat{n}(t_0) \cdot \left(I_b^{-1}(t_0)\left(r_b \times \hat{n}(t_0)\right)\right) \times r_b\Bigg) = -\epsilon v_{rel}^-.\end{aligned} \tag{8–17}$$

Finally, solving for j,

$$j = \frac{-(1+\epsilon)v_{rel}^-}{\frac{1}{M_a} + \frac{1}{M_b} + \hat{n}(t_0) \cdot \left(I_a^{-1}(t_0)\left(r_a \times \hat{n}(t_0)\right)\right) \times r_a + \hat{n}(t_0) \cdot \left(I_b^{-1}(t_0)\left(r_b \times \hat{n}(t_0)\right)\right) \times r_b}. \quad (8\text{–}18)$$

Let's consider some actual code (written for clarity, not speed). First, we determine if two bodies are in colliding contact.

```
/*
        Operators: if 'x' and 'y' are triples,
        assume that 'x ^ y' is their cross product,
        and 'x * y' is their dot product.
*/

/* Return the velocity of a point on a rigid body */
triple pt_velocity(Body *body, triple p)
{
        return body->v + (body->omega ^ (p - body->x));
}

/*
   Return TRUE if bodies are in colliding contact.  The
   parameter 'THRESHOLD' is a small numerical tolerance
   used for deciding if bodies are colliding.
*/
boolean colliding(Contact *c)
{
        triple  padot = pt_velocity(c->a, p),   /* ṗa⁻(t0) */
                pbdot = pt_velocity(c->b, p);   /* ṗb⁻(t0) */
        double  vrel = c->n * (padot - pbdot);  /* vrel⁻ */

        if(vrel > THRESHOLD)    /* moving away */
                return FALSE;
        if(vrel > -THRESHOLD)   /* resting contact */
                return FALSE;
        else                    /* vrel < -THRESHOLD */
                return TRUE;
}
```

Next, we'll loop through all the contact points until all the collisions are resolved, and actually compute and apply an impulse.

```
void collision(Contact *c, double epsilon)
{
        triple  padot = pt_velocity(c->a, p),   /* ṗa⁻(t0) */
                pbdot = pt_velocity(c->b, p),   /* ṗb⁻(t0) */
                n = c->n,                       /* n̂(t0) */
```

```
                ra = p - c->a->x,               /* r_a */
                rb = p - c->b->x;               /* r_b */
        double  vrel = n * (padot - pbdot),     /* v_rel^- */
                numerator = -(1 + epsilon) * vrel;

        /* We'll calculate the denominator in four parts */
        double  term1 = 1 / c->a->mass,
                term2 = 1 / c->b->mass,
                term3 = n * ((c->a->Iinv * (ra ^ n)) ^ ra),
                term4 = n * ((c->b->Iinv * (rb ^ n)) ^ rb);

        /* Compute the impulse magnitude */

        double  j = numerator / (term1 + term2 + term3 + term4);
        triple force = j * n;

        /* Apply the impulse to the bodies */
        c->a->P += force;
        c->b->P -= force;
        c->a->L += ra ^ force;
        c->b->L -= rb ^ force;

        /* recompute auxiliary variables */
        c->a->v = c->a->P / c->a->mass;
        c->b->v = c->b->P / c->b->mass;

        c->a->omega = c->a->Iinv * c->a->L;
        c->b->omega = c->b->Iinv * c->b->L;
}

void find_all_collisions(Contact contacts[], int ncontacts)
{
        boolean had_collision;
        double  epsilon = .5;

        do {
                had_collision = FALSE;

                for(int i = 0; i < ncontacts; i++)
                        if(colliding(&contacts[i]))
                        {
                                collision(&contacts[i], epsilon);
                                had_collision = TRUE;

                                /* Tell the solver we had a collision */
                                ode_discontinuous();
```

```
                        }

            } while(had_collision == TRUE);
}
```

Note several things. First, $\epsilon = .5$ was chosen arbitrarily. In a real implementation, we'd allow the user to use different values of ϵ depending on which two bodies were colliding. Also, every time we find a collision, we have to rescan the list of contacts, since bodies that were at rest may no longer be so, and new collisions may develop. If there are initially several collisions to be resolved (such as a cube dropped flat onto a plane, with all four vertices colliding at once), the order of the contact list may have an effect on the simulation. There is a way to compute impulses at more than one contact point at a time, but it more complicated, and is based on the concepts used for resting contact in the next section. For further information, see Baraff[1].

Incidentally, if you want to have certain bodies that are "fixed", and cannot be moved (such as floors, or walls), you can use the following trick: for such bodies, let $\frac{1}{\texttt{mass}}$ be zero; also let the inverse inertia tensor also be the 3×3 zero matrix. You can either special-case the code to check if a body is supposed to be fixed, or you can recode the definition of `RigidBody` to have the variable `invmass` instead of `mass`. For ordinary bodies, `invmass` is the inverse of the mass, while for fixed bodies, `invmass` is zero. The same goes for the inertia tensor. (Note that nowhere in any of the dynamics computations (including the next section) is the mass or inertia tensor ever used; only their inverses are used, so you won't have to worry about dividing by zero.) The same trick can be used in the next section on resting contact to simulate bodies that can support any amount of weight without moving.

9 Resting Contact

The case of resting contact, when bodies are neither colliding nor separating at a contact point, is the last (and hardest) dynamics problem we'll tackle in these notes. To implement what's in this section, you'll have to obtain a fairly sophisticated piece of numerical software, which we'll describe below.

At this point, let's assume we have a configuration with n contact points. At each contact point, bodies are in resting contact, that is, the relative velocity v_{rel}, from section 8, is zero (to within the numerical tolerance `THRESHOLD`). We can say that this is so, because colliding contact is eliminated by the routine `find_all_collisions`, and any contact points with v_{rel} larger than `THRESHOLD` can be safely ignored, since the bodies are separating there.

As was the case for colliding contact, at each contact point, we have a contact force that acts normal to the contact surface. For the case of colliding contact, we had an impulse $j\hat{n}(t_0)$ where j was an unknown scalar. For resting contact, at each contact point there is some force $f_i\hat{n}_i(t_0)$, where f_i is an unknown scalar, and $\hat{n}_i(t_0)$ is the normal at the ith contact point (figure 26). Our goal is to determine what each f_i is. In computing the f_i's, they must all be determined at the same time, since the force at the ith contact point may influence one or both of the bodies of the j contact point. In section 8, we wrote how the velocity of the contact points $p_a(t_0)$ and $p_b(t_0)$ changed with respect to j. We'll do the same thing here, but now we'll have to describe how the *acceleration* of $p_a(t_0)$ and $p_b(t_0)$ depends on each f_i.

For colliding contact, we had an empirical law which related the impulse strength j to the relative velocity and a coefficient of restitution. For resting contact, we compute the f_i's subject to not one, but three conditions. First, the contact forces must prevent inter-penetration; that is, the contact

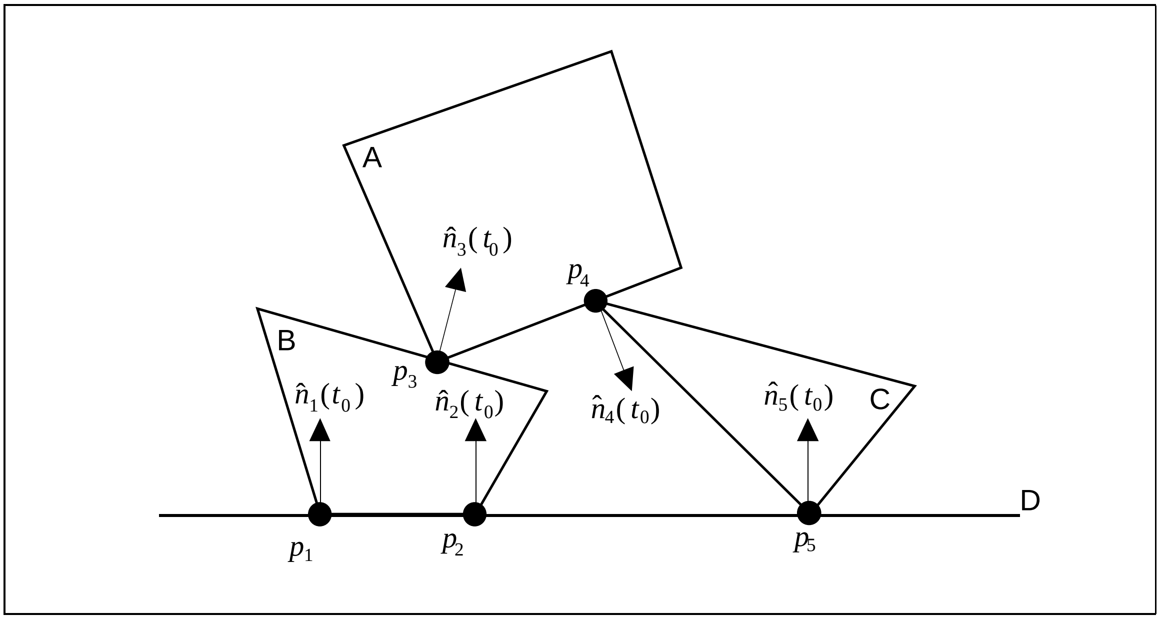

Figure 26: Resting contact. This configuration has five contact points; a contact force acts between pairs of bodies at each contact point.

forces must be strong enough to prevent two bodies in contact from being pushed "towards" one another. Second, we want our contact forces to be repulsive; contact forces can push bodies apart, but can never act like "glue" and hold bodies together. Last, we require that the force at a contact point become zero if the bodies begin to separate. For example, if a block is resting on a table, some force may act at each of the contact points to prevent the block from accelerating downwards in response to the pull of gravity. However, if a very strong wind were to blow the brick upwards, the contact forces on the brick would have to become zero at the instant that the wind accelerated the brick off the table.

Let's deal with the first condition: preventing inter-penetration. For each contact point i, we construct an expression $d_i(t)$ which describes the separation distance between the two bodies near the contact point at time t. Positive distance indicates the bodies have broken contact, and have separated at the ith contact point, while negative distance indicates inter-penetration. Since the bodies are in contact at the present time t_0, we will have $d_i(t_0) = 0$ (within numerical tolerances) for each contact point. Our goal is to make sure that the contact forces maintain $d_i(t) \geq 0$ for each contact point at future times $t > t_0$.

For vertex/face contacts, we can immediately construct a very simple function for $d_i(t)$. If $p_a(t)$ and $p_b(t)$ are the contact points of the ith contact, between bodies A and B, than the distance between the vertex and the face at future times $t \geq t_0$ is given by

$$d_i(t) = \hat{n}_i(t) \cdot (p_a(t) - p_b(t)). \tag{9–1}$$

At time t, the function $d(t)$ measures the separation between A and B near $p_a(t)$. If $d_i(t)$ is zero, then the bodies are in contact at the ith contact point. If $d_i(t) > 0$, then the bodies have lost contact at the ith contact point. However, if $d_i(t) < 0$, then the bodies have inter-penetrated, which is what we need to avoid (figure 27). The same function can also be used for edge/edge contacts; since $\hat{n}_i(t)$

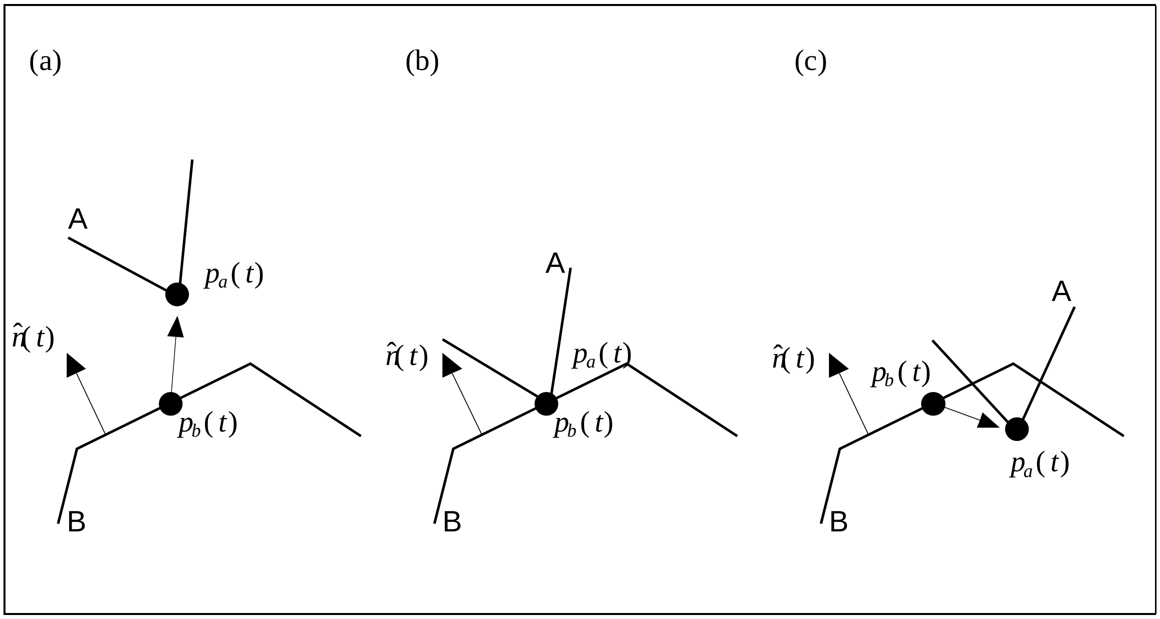

Figure 27: (a) The displacement $p_a(t) - p_b(t)$, indicated by an arrow, points in the same direction as $\hat{n}(t)$. Thus, the distance function $d(t)$ would be positive. (b) The distance function $d(t)$ is zero. (c) The displacement $p_a(t) - p_b(t)$ points in the opposite direction as $\hat{n}(t)$. The distance function $d(t)$ is negative, indicating inter-penetration.

points outwards from B towards A (by convention), $\hat{n}_i(t) \cdot (p_a(t) - p_b(t))$ will be positive if the two contacting edges move so as to separate the bodies.

Since $d_i(t_0) = 0$, we have to keep $d_i(t_0)$ from decreasing at time t_0; that is, we have to have $\dot{d}_i(t_0) \geq 0$. What is $\dot{d}_i(t_0)$? Differentiating,

$$\dot{d}_i(t) = \dot{\hat{n}}_i(t) \cdot (p_a(t) - p_b(t)) + \hat{n}_i(t) \cdot (\dot{p}_a(t) - \dot{p}_b(t)). \tag{9–2}$$

Since $d_i(t)$ describes the separation distance, $\dot{d}_i(t)$ will describe the separation *velocity* at time t. However, at time t_0, $p_a(t_0) = p_b(t_0)$, which means that $\dot{d}_i(t_0) = \hat{n}_i(t_0) \cdot (\dot{p}_a(t_0) - \dot{p}_b(t_0))$. This should look familiar: its v_{rel} from the previous section! The function $\dot{d}_i(t_0)$ is a measure of how the bodies are separating, and for resting contact, we know that $\dot{d}(t_0)$ is zero, because the bodies are neither moving towards nor away from each other at a contact point.

At this point then, we have $d_i(t_0) = \dot{d}_i(t_0) = 0$. Now we'll look at $\ddot{d}_i(t_0)$. If we differentiate equation (9–2), we get

$$\begin{aligned} \ddot{d}_i(t) = & \Big(\ddot{\hat{n}}_i(t) \cdot (p_a(t) - p_b(t)) + \dot{\hat{n}}_i(t) \cdot (\dot{p}_a(t) - \dot{p}_b(t))\Big) + \\ & \Big(\dot{\hat{n}}_i(t) \cdot (\dot{p}_a(t) - \dot{p}_b(t)) + \hat{n}_i(t) \cdot (\ddot{p}_a(t) - \ddot{p}_b(t))\Big) \\ = & \ \ddot{\hat{n}}_i(t) \cdot (p_a(t) - p_b(t)) + 2\dot{\hat{n}}_i(t) \cdot (\dot{p}_a(t) - \dot{p}_b(t)) + \hat{n}_i(t) \cdot (\ddot{p}_a(t) - \ddot{p}_b(t)). \end{aligned} \tag{9–3}$$

Since $p_a(t_0) = p_b(t_0)$, we can write $\ddot{d}_i(t_0)$ as

$$\ddot{d}(t_0) = \hat{n}_i(t_0) \cdot (\ddot{p}_a(t_0) - \ddot{p}_b(t_0)) + 2\dot{\hat{n}}_i(t_0) \cdot (\dot{p}_a(t_0) - \dot{p}_b(t_0)). \tag{9–4}$$

The quantity $\ddot{d}_i(t_0)$ measures how the two bodies are accelerating towards each other at the contact point p. If $\ddot{d}_i(t_0) > 0$, the the bodies have an acceleration away from each other, and contact will break immediately after t_0. If $\ddot{d}_i(t_0) = 0$, then contact remains. The case $\ddot{d}_i(t_0) < 0$ must be avoided, for this indicates the bodies are accelerating towards each other. Note that if $\hat{n}_i(t_0)$ is a constant (if body B is fixed), then $\dot{\hat{n}}_i(t_0)$ is zero, leading to further simplifications.

Thus, we satisfy our first condition for contact forces by writing the constraint

$$\ddot{d}_i(t_0) \geq 0 \tag{9–5}$$

for each contact point. Since the acceleration $\ddot{d}_i(t_0)$ depends on the contact forces, this is really a constraint on the contact forces.

Let's turn our attention to the second and third constraints. Since contact forces must always be repulsive, each contact force must act outward. This means that each f_i must be positive, since a force of $f_i \hat{n}_i(t_0)$ acts on body A, and $\hat{n}_i(t_0)$ is the outwards pointing normal of B. Thus, we need

$$f_i \geq 0 \tag{9–6}$$

for each contact point. The third constraint is expressed simply in terms of f_i and $\ddot{d}_i(t_0)$. Since the contact force $f_i \hat{n}_i(t_0)$ must become zero if contact is breaking at the ith contact, this says that f_i must be zero if contact is breaking. We can express this constraint by writing

$$f_i \ddot{d}_i(t_0) = 0; \tag{9–7}$$

if contact is breaking, $\ddot{d}_i(t_0) > 0$ and equation (9–7) is satisfied by requiring $f_i = 0$. If contact is not breaking, then $\ddot{d}_i(t_0) = 0$, and equation (9–7) is satisfied regardless of f_i.

In order to actually find f_i's which satisfy equations (9–5), (9–6), and (9–7), we need to express each $\ddot{d}_i(t_0)$ as a function of the unknown f_i's. It will turn out that we will be able to write each $\ddot{d}_i(t_0)$ in the form

$$\ddot{d}_i(t_0) = a_{i1} f_1 + a_{i2} f_2 + \cdots + a_{in} f_n + b_i. \tag{9–8}$$

In matrix parlance, this means we will be able to write

$$\begin{pmatrix} \ddot{d}_1(t_0) \\ \vdots \\ \ddot{d}_n(t_0) \end{pmatrix} = \mathbf{A} \begin{pmatrix} f_1 \\ \vdots \\ f_n \end{pmatrix} + \begin{pmatrix} b_i \\ \vdots \\ b_n \end{pmatrix} \tag{9–9}$$

where $\mathbf{A}$ is the $n \times n$ matrix of the a_{ij} coefficients of equation (9–8). Although the code needed to calculate the a_{ij}'s and the b_i's is not too complicated, working out the derivations on which the code is based is somewhat tedious. The derivations are worked out in appendix D, along with code to compute the matrix of a_{ij}'s and b_i's.

Appendix D gives an implementation of the routines

```
void    compute_a_matrix(Contact contacts[], int ncontacts,
                         bigmatrix &a);
```

```
void    compute_b_vector(Contact contacts[], int ncontacts,
                         vector &b);
```

where the types `bigmatrix` and `vector` represent matrices and vectors of arbitrary size. The first routine computes the a_{ij}'s, while the second routine computes the b_i's.

Once we've computed all this, we can think about solving equations (9–5), (9–6), and (9–7). This system of equations forms what is called a *quadratic program* (QP); that is, f_i's that satisfy these three equations are found by an algorithm called quadratic programming. Not all quadratic programs can be solved efficiently, but because our contact forces are all normal to the contact surfaces (that is, they do not involve friction), it turns out that our QP can always be solved efficiently. One interesting thing to note is that QP codes can easily handle the case $\ddot{d}_i(t_0) = 0$ instead of $\ddot{d}_i(t_0) \geq 0$. We use $\ddot{d}_i(t_0) = 0$ (and also drop the constraint $f_i \geq 0$) if we wish to constrain two bodies to never separate at a contact point. This enables us to implement hinges, and pin-joints, as well as non-penetration constraints during simulation.

Quadratic programming codes aren't terribly common though; certainly, they are not nearly as common as linear equation codes, and are much harder to implement. The quadratic programming routines used by the author were obtained from the Department of Operations Research at Stanford University. See Gill *et al.*[7, 8, 9] for further details. More recently, we have been using code described by Baraff[3] to solve the quadratic programs. If you are determined to really implement this, we suggest a thorough study of this paper (excepting for the section on contact with friction).

At any rate, let's assume that you've got a working QP solver at your disposal. We'll assume that you pass the matrix **A**, and the vector of b_i's to the QP solver, and you get back the vector of f_i's. Let's pretend the interface is

```
void qp_solve(bigmatrix &a, vector &b, vector &f);
```

Let's see how to compute all the resting contact forces. The following routine is presumably called from `Compute_Force_and_Torque`, after `find_collisions` has been called.

```
void compute_contact_forces(Contact contacts[], int ncontacts, double t)
{
        /* We assume that every element of contacts[]
            represents a contact in resting contact.

           Also, we'll assume that for each element of Bodies[],
           the ‘force’ and ‘torque’ fields have been set to the
           net external force and torque acting on the body, due
           to gravity, wind, etc., perhaps by a call to

                Compute_External_Force_and_Torque_for_all_Bodies(t);
        */

        /* Allocate n × n matrix ‘amat’ and n-vectors ‘fvec’,
           and ‘bvec’. */
```

```
    bigmatrix amat = new bigmatrix(ncontacts, ncontacts);
    vector    bvec = new vector(ncontacts),
              fvec = new vector(ncontacts);

    /* Compute a_ij and b_i coefficients */

    compute_a_matrix(contacts, ncontacts, amat);
    compute_b_vector(contacts, ncontacts, bvec);

    /* Solve for f_j's */
    qp_solve(amat, bmat, fvec);

    /* Now add the resting contact forces we just computed into
       the 'force' and 'torque' field of each rigid body. */

    for(int i = 0; i < ncontacts; i++)
    {
            double     f = fvec[i];                 /* f_i */
            triple     n = contacts[i]->n;          /* n̂_i(t_0) */
            RigidBody  *A = contacts[i]->a,         /* body A */
                       *B = contacts[i]->b;         /* body B */

            /* apply the force 'f n' positively to A... */

            A->force += f * n;
            A->torque += (contacts[i].p - A->x) * (f*n);

            /* and negatively to B */

            B->force -= f * n;
            B->torque -= (contacts[i].p - B->x) * (f*n);
    }
}
```

That's pretty much it! Now that the resting forces have been computed and combined with the external forces, we return control to the ODE solver, and each body goes merrily along its way, in a physically correct manner, without inter-penetration.

Appendix A Motion Equation Derivations

In this appendix, we'll fill in some of the missing details from section 2, with regards to the equations $\dot{P}(t) = F(t)$, $\dot{L}(t) = \tau(t)$, and $L(t) = I(t)\omega(t)$. The derivation method used here is somewhat nonstandard, and was proposed by Andy Witkin. The derivation in this appendix is (we feel) much shorter and considerably more elegant than the one found in traditional sources such as Goldstein[10].

We've described the external force acting on a rigid body in terms of forces $F_i(t)$, where $F_i(t)$ is the external force acting on the ith particle. However, for a rigid body to maintain its shape, there must be some "internal" constraint forces that act between particles in the same body. We will make the assumption that these constraint forces act passively on the system and do not perform any net work. Let $F_{ci}(t)$ denote the net internal constraint force acting on the ith particle. The work performed by F_{ci} on the ith particle from time t_0 to t_1 is

$$\int_{t_0}^{t_1} F_{ci}(t) \cdot \dot{r}_i(t)\, dt$$

where $\dot{r}_i(t)$ is the velocity of the ith particle. The net work over all the particles is the sum

$$\sum_i \int_{t_0}^{t_1} F_{ci}(t) \cdot \dot{r}_i(t)\, dt = \int_{t_0}^{t_1} \sum_i F_{ci}(t) \cdot \dot{r}_i(t)\, dt,$$

which must be zero for any interval t_0 to t_1. This means that the integrand

$$\sum_i F_{ci}(t) \cdot \dot{r}_i(t) \tag{A–1}$$

is itself always zero for any time t. (Henceforth we'll just write these expressions as $\sum F_{ci} \cdot \dot{r}_i = 0$.)

We can use this fact to eliminate any mention of the constraint forces F_{ci} from our derivations. First, some quick notes about the "$*$" operator defined in section 2.3: since $a^*b = a \times b$, and $a \times b = -b \times a$, we get

$$-a^*b = b \times a = b^*a. \tag{A–2}$$

Since a^* is an antisymmetric matrix,

$$(a^*)^T = -a^*. \tag{A–3}$$

Finally, since the "$*$" operator is a linear operator,

$$(\dot{a})^* = \dot{(a^*)} = \frac{d}{dt}(a^*) \tag{A–4}$$

and for a set of vectors a_i

$$\sum a_i^* = \left(\sum a_i\right)^*. \tag{A–5}$$

Recall that we can write the velocity $\dot{r}_i$ as $\dot{r}_i = v + \omega \times (r_i - x)$ where r_i is the particle's location, x is the position of the center of mass, and v and ω are linear and angular velocity. Letting $r'_i = r_i - x$ and using the "$*$" notation,

$$\dot{r}_i = v + \omega^* r'_i = v - r'^*_i \omega. \tag{A–6}$$

Substituting this into $\sum F_{ci} \cdot \dot{r}_i$, which is always zero, yields

$$\sum F_{ci} \cdot (v - r'^*_i \omega) = 0. \tag{A–7}$$

Note that this equation must hold for arbitrary values of v and ω. Since v and ω are completely independent, if we choose ω to be zero, then $\sum F_{ci} \cdot v = 0$ for any choice of v, from which we conclude that in fact $\sum F_{ci} = \mathbf{0}$ is always true. This means that the constraint forces produce no net force. Similarly, choosing v to be zero we see that $\sum -F_{ci} \cdot (r'^*_i \omega) = 0$ for any ω. Rewriting $F_{ci} \cdot (r'^*_i \omega)$ as $F_{ci}{}^T (r'^*_i \omega)$ we get that

$$\sum -F_{ci}{}^T r'^*_i \omega = \left(\sum -F_{ci}{}^T r'^*_i \right) \omega = 0 \tag{A–8}$$

for any ω, so $\sum -F_{ci}{}^T r'^*_i = \mathbf{0}^T$. Transposing, we have

$$\sum -(r'^*_i)^T F_{ci} = \sum (r'_i)^* F_{ci} = \sum r'_i \times F_{ci} = \mathbf{0} \tag{A–9}$$

which means that the internal forces produce no net torque.

We can use the above to derive the rigid body equations of motion. The net force on each particle is the sum of the internal constraint force F_{ci} and the external force F_i. The acceleration $\ddot{r}_i$ of the ith particle is

$$\ddot{r}_i = \frac{d}{dt}\dot{r}_i = \frac{d}{dt}(v - r'^*_i \omega) = \dot{v} - \dot{r}'^*_i \omega - r'^*_i \dot{\omega}. \tag{A–10}$$

Since each individual particle must obey Newton's law $f = ma$, or equivalently $ma - f = \mathbf{0}$, we have

$$m_i \ddot{r}_i - F_i - F_{ci} = m_i(\dot{v} - \dot{r}'^*_i \omega - r'^*_i \dot{\omega}) - F_i - F_{ci} = \mathbf{0} \tag{A–11}$$

for each particle.

To derive $\dot{P} = F = \sum F_i$, we sum equation (A–11) over all the particles. We obtain

$$\sum m_i(\dot{v} - \dot{r}'^*_i \omega - r'^*_i \dot{\omega}) - F_i - F_{ci} = \mathbf{0}. \tag{A–12}$$

Breaking the large sum into smaller ones,

$$\begin{aligned}
\sum m_i(\dot{v} - \dot{r}'^*_i \omega - r'^*_i \dot{\omega}) - F_i - F_{ci} &= \\
\sum m_i \dot{v} - \sum m_i \dot{r}'^*_i \omega - \sum m_i r'^*_i \dot{\omega} - \sum F_i - \sum F_{ci} &= \\
\sum m_i \dot{v} - \left(\sum m_i \dot{r}'_i \right)^* \omega - \left(\sum m_i r'_i \right)^* \dot{\omega} - \sum F_i - \sum F_{ci} &= \\
\sum m_i \dot{v} - \left(\frac{d}{dt} \sum m_i r'_i \right)^* \omega - \left(\sum m_i r'_i \right)^* \dot{\omega} - \sum F_i - \sum F_{ci} &= \mathbf{0}.
\end{aligned} \tag{A–13}$$

Since we are in a center-of-mass coordinate system, equation (2–20) from section 2.6 tells us that $\sum m_i r'_i = \mathbf{0}$, which also means that $\frac{d}{dt} \sum m_i r'_i = \mathbf{0}$. Removing terms with $\sum m_i r'_i$, and the term $\sum F_{ci}$ from the above equation yields

$$\sum m_i \dot{v} - \sum F_i = \mathbf{0} \tag{A–14}$$

or simply $M\dot{v} = \dot{P} = \sum F_i$ as advertised.

To obtain $\dot{L} = \tau = \sum r'_i \times F_i$, we again start with equation (A–11). Multiplying both sides by r'^*_i yields

$$r'^*_i m_i (\dot{v} - \dot{r}'^*_i \omega - r'^*_i \dot{\omega}) - r'^*_i F_i - r'^*_i F_{ci} = r'^*_i \mathbf{0} = \mathbf{0}. \tag{A–15}$$

Summing over all the particles, we obtain

$$\sum r'^*_i m_i \dot{v} - \sum r'^*_i m_i \dot{r}'^*_i \omega - \sum r'^*_i m_i r'^*_i \dot{\omega} - \sum r'^*_i F_i - \sum r'^*_i F_{ci} = \mathbf{0}. \tag{A–16}$$

Since $\sum r'^*_i F_{ci} = \mathbf{0}$, we can rearrange this to obtain

$$\left(\sum m_i r'_i\right)^* \dot{v} - \left(\sum m_i r'^*_i \dot{r}'^*_i\right) \omega - \left(\sum m_i r'^*_i r'^*_i\right) \dot{\omega} - \sum r'^*_i F_i = \mathbf{0}. \tag{A–17}$$

Using $\sum m_i r'_i = \mathbf{0}$, we are left with

$$-\left(\sum m_i r'^*_i \dot{r}'^*_i\right) \omega - \left(\sum m_i r'^*_i r'^*_i\right) \dot{\omega} - \sum r'^*_i F_i - = \mathbf{0} \tag{A–18}$$

or, recognizing that $\sum r'^*_i F_i = \sum r'_i \times F_i = \tau$,

$$-\left(\sum m_i r'^*_i \dot{r}'^*_i\right) \omega - \left(\sum m_i r'^*_i r'^*_i\right) \dot{\omega} = \tau. \tag{A–19}$$

We're almost done now: if we refer back to the matrix defined by the "$*$" notation, one can easily verify the relation that the matrix $-a^* a^*$ is equivalent to the matrix $(a^T a)\mathbf{1} - aa^T$ where $\mathbf{1}$ is the 3×3 identity matrix. (This relation is equivalent to the vector rule $a \times (b \times c) = ba^T c - ca^T b$.) Thus

$$\sum -m_i r'^*_i r'^*_i = \sum m_i ((r'^T_i r'_i)\mathbf{1} - r'_i r'^T_i) = I(t). \tag{A–20}$$

Substituting into equation (A–19), this yields

$$\left(\sum -m_i r'^*_i \dot{r}'^*_i\right) \omega + I(t)\dot{\omega} = \tau. \tag{A–21}$$

The above expression is almost acceptable, as it gives an expression for $\dot{\omega}$ in terms of τ, except that it requires us to evaluate the matrix $\sum m_i r'^*_i \dot{r}^*_i$, which is as expensive as computing the inertia tensor from scratch. We'll use one last trick here to clean things up. Since $\dot{r}'_i = \omega \times r'_i$ and $r'^*_i \omega = -\omega \times r'_i$, we can write

$$\sum m_i \dot{r}'^*_i r'^*_i \omega = \sum m_i (\omega \times r'_i)^* (-\omega \times r'_i) = \sum -m_i (\omega \times r'_i) \times (\omega \times r'_i) = \mathbf{0}. \tag{A–22}$$

Thus, we can add $-\sum m_i \dot{r}_i'^* r_i'^* \omega = \mathbf{0}$ to equation (A–21) to obtain

$$\left(\sum -m_i r_i'^* \dot{r}_i'^* - m_i \dot{r}_i'^* r_i'^*\right) \omega + I(t)\dot{\omega} = \tau. \tag{A–23}$$

Finally, since

$$\dot{I}(t) = \frac{d}{dt} \sum -m_i r_i'^* r_i'^* = \sum -m_i r_i'^* \dot{r}_i'^* - m_i \dot{r}_i'^* r_i'^* \tag{A–24}$$

we have

$$\dot{I}(t)\omega + I(t)\dot{\omega} = \frac{d}{dt}(I(t)\omega) = \tau. \tag{A–25}$$

Since $L(t) = I(t)\omega(t)$, this leaves us with the final result that

$$\dot{L}(t) = \tau. \tag{A–26}$$

Appendix B Quaternion Derivations

A formula for $\dot{q}(t)$ is derived as follows. Recall that the angular velocity $\omega(t)$ indicates that the body is instantaneously rotating about the $\omega(t)$ axis with magnitude $|\omega(t)|$. Suppose that a body were to rotate with a constant angular velocity $\omega(t)$. Then the rotation of the body after a period of time Δt is represented by the quaternion

$$[\cos \frac{|\omega(t)|\Delta t}{2}, \sin \frac{|\omega(t)|\Delta t}{2} \frac{\omega(t)}{|\omega(t)|}].$$

Let us compute $\dot{q}(t)$ at some particular instant of time t_0. At times $t_0 + \Delta t$ (for small Δt), the orientation of the body is (to within first order) the result of first rotating by $q(t_0)$ and then further rotating with velocity $\omega(t_0)$ for Δt time. Combining the two rotations, we get

$$q(t_0 + \Delta t) = [\cos \frac{|\omega(t_0)|\Delta t}{2}, \sin \frac{|\omega(t_0)|\Delta t}{2} \frac{\omega(t_0)}{|\omega(t_0)|}]\, q(t_0). \tag{B–1}$$

Making the substitution $t = t_0 + \Delta t$, we can express this as

$$q(t) = [\cos \frac{|\omega(t_0)|(t - t_0)}{2}, \sin \frac{|\omega(t_0)|(t - t_0)}{2} \frac{\omega(t_0)}{|\omega(t_0)|}] q(t_0). \tag{B–2}$$

Let us differentiate $q(t)$ at time t_0. First, since $q(t_0)$ is a constant, let us differentiate

$$[\cos \frac{|\omega(t_0)|(t - t_0)}{2}, \sin \frac{|\omega(t_0)|(t - t_0)}{2} \frac{\omega(t_0)}{|\omega(t_0)|}].$$

At time $t = t_0$,

$$\begin{aligned} \frac{d}{dt} \cos \frac{|\omega(t_0)|(t - t_0)}{2} &= -\frac{|\omega(t_0)|}{2} \sin \frac{|\omega(t_0)|(t - t_0)}{2} \\ &= -\frac{|\omega(t_0)|}{2} \sin 0 = 0. \end{aligned} \tag{B–3}$$

Similarly,

$$\frac{d}{dt}\sin\frac{|\omega(t_0)|(t-t_0)}{2} = \frac{|\omega(t_0)|}{2}\cos\frac{|\omega(t_0)|(t-t_0)}{2} \\ = \frac{|\omega(t_0)|}{2}\cos 0 = \frac{|\omega(t_0)|}{2}. \tag{B–4}$$

Thus, at time $t = t_0$,

$$\begin{aligned}\dot{q}(t) &= \frac{d}{dt}\left([\cos\frac{|\omega(t_0)|(t-t_0)}{2}, \sin\frac{|\omega(t_0)|(t-t_0)}{2}\frac{\omega(t_0)}{|\omega(t_0)|}]\, q(t_0)\right) \\ &= \frac{d}{dt}\left([\cos\frac{|\omega(t_0)|(t-t_0)}{2}, \sin\frac{|\omega(t_0)|(t-t_0)}{2}\frac{\omega(t_0)}{|\omega(t_0)|}]\right) q(t_0) \\ &= [0, \frac{|\omega(t_0)|}{2}\frac{\omega(t_0)}{|\omega(t_0)|}]\, q(t_0) \\ &= [0, \tfrac{1}{2}\omega(t_0)]\, q(t_0) = \tfrac{1}{2}[0, \omega(t_0)]\, q(t_0).\end{aligned} \tag{B–5}$$

The product $[0, \omega(t_0)]\, q(t_0)$ is abbreviated to the form $\omega(t_0)q(t_0)$; thus, the general expression for $\dot{q}(t)$ is

$$\dot{q}(t) = \tfrac{1}{2}\omega(t)q(t). \tag{B–6}$$

Appendix C Some Miscellaneous Formulas

C.1 Kinetic Energy

The kinetic energy T of a rigid body is defined as

$$T = \sum \tfrac{1}{2} m_i \dot{r}_i^T \dot{r}_i. \tag{C–1}$$

Letting $r_i' = r_i - x$, we have $\dot{r}_i = v(t) + r_i'^* \omega$. Thus

$$\begin{aligned}T &= \sum \tfrac{1}{2} m_i \dot{r}_i^T \dot{r}_i \\ &= \sum \tfrac{1}{2} m_i (v + r_i'^*\omega)^T (v + r_i'^*\omega) \\ &= \tfrac{1}{2}\sum m_i v^T v + \sum v^T m_i r_i'^* \omega + \tfrac{1}{2}\sum m_i (r_i'^*\omega)^T (r_i'^*\omega) \\ &= \tfrac{1}{2} v^T \left(\sum m_i\right) v + v^T \left(\sum m_i r_i'\right)^* \omega + \tfrac{1}{2}\omega^T \left(\sum m_i (r_i'^*)^T r_i'^*\right)\omega.\end{aligned} \tag{C–2}$$

Using $\sum m_i r_i' = \mathbf{0}$ and $(r_i'^*)^T = -r_i'^*$, we have

$$T = \tfrac{1}{2} v^T M v + \tfrac{1}{2}\omega^T \left(\sum -m_i r_i'^* r_i'^*\right)\omega = \tfrac{1}{2}(v^T M v + \omega^T I \omega) \tag{C–3}$$

since $I = \sum -m_i r_i'^* r_i'^*$ from appendix A. Thus, the kinetic energy can be decomposed into two terms: a linear term $\frac{1}{2} v^T M v$, and an angular term $\frac{1}{2}\omega^T I \omega$.

C.2 Angular Acceleration

It is often necessary to compute $\dot{\omega}(t)$. Since $L(t) = I(t)\omega(t)$, we know $\omega(t) = I^{-1}(t)L(t)$. Thus,

$$\dot{\omega}(t) = \dot{I}^{-1}(t)L(t) + I^{-1}(t)\dot{L}(t). \tag{C–4}$$

Since we know that $\dot{L}(t) = \tau(t)$, let us consider $\dot{I}^{-1}(t)$. From equation (2–40),

$$I^{-1}(t) = R(t)I_{body}^{-1}R(t)^T,$$

so

$$\dot{I}^{-1}(t) = \dot{R}(t)I_{body}^{-1}R(t)^T + R(t)I_{body}^{-1}\dot{R}(t)^T. \tag{C–5}$$

Since $\dot{R}(t) = \omega(t)^* R(t)$,

$$\dot{R}(t)^T = (\omega(t)^* R(t))^T = R(t)^T(\omega(t)^*)^T. \tag{C–6}$$

Since $\omega(t)^*$ is antisymmetric, (i.e. $(\omega(t)^*)^T = -\omega(t)^*$),

$$\dot{R}(t)^T = -R(t)^T\omega(t)^*. \tag{C–7}$$

This yields

$$\begin{aligned}
\dot{I}^{-1}(t) &= \dot{R}(t)I_{body}^{-1}R(t)^T + R(t)I_{body}^{-1}(-R(t)^T\omega(t)^*) \\
&= \omega(t)^* R(t)I_{body}^{-1}R(t)^T - I^{-1}(t)\omega(t)^* \\
&= \omega(t)^* I^{-1}(t) - I^{-1}(t)\omega(t)^*.
\end{aligned} \tag{C–8}$$

Then

$$\begin{aligned}
\dot{\omega}(t) &= \dot{I}^{-1}(t)L(t) + I^{-1}(t)\dot{L}(t) \\
&= \left(\omega(t)^* I^{-1}(t) - I^{-1}(t)\omega(t)^*\right)L(t) + I^{-1}(t)\dot{L}(t) \\
&= \omega(t)^* I^{-1}(t)L(t) - I^{-1}(t)\omega(t)^* L(t) + I^{-1}(t)\dot{L}(t).
\end{aligned} \tag{C–9}$$

But since $I^{-1}(t)L(t) = \omega(t)$, the first term, $\omega(t)^* I^{-1}(t)L(t)$ is equivalent to $\omega(t)^*\omega(t)$, or $\omega(t) \times \omega(t)$, which is zero. This leaves the final result of

$$\begin{aligned}
\dot{\omega}(t) &= -I^{-1}(t)\omega(t)^* L(t) + I^{-1}(t)\dot{L}(t) \\
&= -I^{-1}(t)\omega(t) \times L(t) + I^{-1}(t)\dot{L}(t) \\
&= I^{-1}(t)(L(t) \times \omega(t)) + I^{-1}(t)\dot{L}(t) \\
&= I^{-1}(t)(L(t) \times \omega(t) + \dot{L}(t)).
\end{aligned} \tag{C–10}$$

We can see from this that even if no forces act, so that $\dot{L}(t)$ is zero, $\dot{\omega}(t)$ can still be non-zero. (In fact, this will happen whenever the angular momentum and angular velocities point in different directions, which in turn occurs when the body has a rotational velocity axis that is not an axis of symmetry for the body.)

C.3 Acceleration of a Point

Given a point of a rigid body with world space coordinate $p(t)$, it is often necessary to compute $\ddot{p}(t)$. Let the body space coordinate that transforms at time t to $p(t)$ be p_0; then

$$p(t) = R(t)p_0 + x(t)$$

If we let $r(t) = p(t) - x(t)$, then

$$\begin{aligned} \dot{p}(t) &= \dot{R}(t)p_0 + \dot{x}(t) = \omega(t)^* R(t)p_0 + v(t) \\ &= \omega(t) \times (R(t)p_0 + x(t) - x(t)) + v(t) \\ &= \omega(t) \times (p(t) - x(t)) + v(t) \\ &= \omega(t) \times r(t) + v(t). \end{aligned} \tag{C–11}$$

Then

$$\begin{aligned} \ddot{p}(t) &= \dot{\omega}(t) \times r(t) + \omega(t) \times \dot{r}(t) + \dot{v}(t) \\ &= \dot{\omega}(t) \times r(t) + \omega(t) \times (\omega(t) \times r(t)) + \dot{v}(t). \end{aligned} \tag{C–12}$$

We can interpret this as follows. The first term, $\dot{\omega}(t) \times r(t)$ is the *tangential* acceleration of the point; that is, $\dot{\omega}(t) \times r(t)$ is the acceleration perpendicular to the displacement $r(t)$ as a result of the body being angularly accelerated. The second term, $\omega(t) \times (\omega(t) \times r(t))$ is the centripetal acceleration of the point; this centripetal acceleration arises because the body is rigid, and points on the body must rotate in a circular orbit about the center of mass. The last term, $\dot{v}(t)$ is the linear acceleration of the point due to the linear acceleration of the center of mass of the body.

Appendix D Resting Contact Derivations

If you're determined to implement resting contact in your simulator, you'll need the derivations and the code in this appendix. This is probably not a fun appendix to work through; then again, this wasn't a fun appendix to write! The derivations in here are somewhat terse, but the code at the end of the appendix will hopefully make things clearer.

D.1 Derivations

We need to express $\ddot{d}_i(t_0)$ in terms of all the unknown f_i's. It will turn out that we'll be able to write each $\ddot{d}_i(t_0)$ in the form

$$\ddot{d}_i(t_0) = a_{i1} f_1 + a_{i2} f_2 + \cdots + a_{in} f_n + b_i. \tag{D–1}$$

Given i and j, we need to know how $\ddot{d}_i(t_0)$ depends on f_j, that is, we need to know a_{ij}. Also, we need to compute the constant term b_i.

Let's start by determining a_{ij} and ignoring the constant part b_i. We'll assume the ith contact involves two bodies A and B. From equation (9–4), we can write $\ddot{d}_i(t_0)$ as

$$\ddot{d}(t_0) = \hat{n}_i(t_0) \cdot (\ddot{p}_a(t_0) - \ddot{p}_b(t_0)) + 2\dot{\hat{n}}_i(t_0) \cdot (\dot{p}_a(t_0) - \dot{p}_b(t_0)) \tag{D–2}$$

where $p_a(t_0) = p_i = p_b(t_0)$ is the contact point for the ith contact at time t_0. The term $2\dot{\hat{n}}_i(t_0) \cdot (\dot{p}_a(t_0) - \dot{p}_b(t_0))$ is a velocity dependent term (i.e. you can immediately calculate it without knowing the forces involved), and is part of b_i, so we'll ignore this for now.

So we only need to know how $\ddot{p}_a(t_0)$ and $\ddot{p}_b(t_0)$ depend on f_j, the magnitude of the jth contact force. Consider the jth contact. If body A is not one of the bodies involved in the jth contact, then $\ddot{p}_a(t_0)$ is independent of f_j, because the jth contact force does not act on body A. Similarly, if B is also not one of the two bodies involved in the jth contact, then $\ddot{p}_b(t_0)$ is also independent of f_j. (For example, in figure 26, the acceleration of the contact points at the first contact is completely unaffected by the contact force acting at the fifth contact. Thus, $\ddot{d}_1(t_0)$ would be completely independent of f_5. Conversely, $\ddot{d}_5(t_0)$ is completely independent of f_1.)

Suppose though that in the jth contact, body A is involved. For definiteness, suppose that in the jth contact, a force of $j\hat{n}_j(t_0)$ acts on body A, as opposed to $-j\hat{n}_j(t_0)$. Let's derive how $\ddot{p}_a(t_0)$ is affected by the force $j\hat{n}_j(t_0)$ acting on A.

From equation (C–12), we can write

$$\ddot{p}_a(t) = \dot{v}_a(t) + \dot{\omega}_a(t) \times r_a(t) + \omega_a(t) \times (\omega_a(t) \times r_a(t)) \tag{D–3}$$

where $r_a(t) = p_a(t) - x_a(t)$, and $x_a(t)$, $v_a(t)$, and $\omega_a(t)$ are all the variables associated with body A. We know that $\dot{v}_a(t)$ is the linear acceleration of body A, and is equal to the total force acting on A divided by the mass. Thus, a force of $j\hat{n}_j(t_0)$ contributes

$$\frac{f_j\hat{n}_j(t_0)}{m_a} = f_j\frac{\hat{n}_j(t_0)}{m_a} \tag{D–4}$$

to $\dot{v}_a(t)$ and thus $\ddot{p}_a(t)$. Similarly, consider $\dot{\omega}_a(t)$, from equation (C–10):

$$\dot{\omega}_a(t) = I_a^{-1}(t)\tau_a(t) + I_a^{-1}(t)(L_a(t) \times \omega_a(t))$$

where $\tau_a(t)$ is the total torque acting on body A. If the jth contact occurs at the point p_j, then the force $j\hat{n}_j(t_0)$ exerts a torque of

$$(p_j - x_a(t_0)) \times f_j\hat{n}_j(t_0).$$

Thus, the angular contribution to $\ddot{p}_a(t_0)$ is

$$f_j\left(I_a^{-1}(t_0)\left((p_j - x_a(t_0)) \times \hat{n}_j(t_0)\right)\right) \times r_a. \tag{D–5}$$

The total dependence of $\ddot{p}_a(t_0)$ on f_j is therefore

$$f_j\left(\frac{\hat{n}_j(t_0)}{m_a} + \left(I_a^{-1}(t_0)\left((p_j - x_a(t_0)) \times \hat{n}_j(t_0)\right)\right) \times r_a\right).$$

Now, if a force of $-f_j\hat{n}(t_0)$ had acted on A instead, we'd get the same dependence, but with a minus sign in front of f_j. Clearly, $\ddot{p}_b(t_0)$ depends on f_j in the same sort of manner. Once we compute how $\ddot{p}_a(t_0)$ and $\ddot{p}_b(t_0)$ depend on f_j, we combine the results together and take the dot product with $\hat{n}_i(t_0)$, to see how $\ddot{d}_i(t_0)$ depends on f_j. This gives us a_{ij}. Confused? See the code below.

We still need to compute b_i. We know that $\ddot{d}_i(t_0)$ contains the constant term

$$2\dot{\hat{n}}_i(t_0) \cdot (\dot{p}_a(t_0) - \dot{p}_b(t_0)).$$

But we also have to take into account the contributions to $\ddot{p}_a(t_0)$ and $\ddot{p}_b(t_0)$ due to known external forces such as gravity, as well as the force-independent terms $\omega_a(t_0) \times (\omega_a(t_0) \times r_a)$ and $(I_a^{-1}(t_0)(Ł_a(t_0) \times \omega_a(t_0))) \times r_a$. If we let the net external force acting on A be $F_a(t_0)$, and the net external torque be $\tau_a(t_0)$, then from equations (D–4) and (D–5), we get that $F_a(t_0)$ contributes

$$\frac{F_a(t_0)}{m_a}$$

and that $\tau_a(t_0)$ contributes

$$\left(I_a^{-1}(t_0)\tau_a(t_0)\right) \times r_a.$$

Thus, the part of $\ddot{p}_a(t_0)$ that is independent from all the f_j's is

$$\frac{F_a(t_0)}{m_a} + \left(I_a^{-1}(t_0)\tau_a(t_0)\right) \times r_a + \omega_a(t_0) \times (\omega_a(t_0) \times r_a) + \left(I_a^{-1}(t_0)(Ł_a(t_0) \times \omega_a(t_0))\right) \times r_a$$

and similarly for $\ddot{p}_b(t_0)$. To compute b_i, we combine the constant parts of $\ddot{p}_a(t_0)$, $\ddot{p}_b(t_0)$, dot with $\hat{n}_i(t_0)$, and add the term $2\dot{\hat{n}}_i(t_0) \cdot (\dot{p}_a(t_0) - \dot{p}_b(t_0))$.

D.2 Code

Here's the code to implement the above derivations. Let's start by computing the constant b_i terms.

```
/* return the derivative of the normal vector */
triple compute_ndot(Contact *c)
{
        if(c->vf)       /* vertex/face contact */
        {
                /* The vector ‘n’ is attached to B, so... */
                return c->b->omega Θ c->n;
        }
        else
        {
                /* This is a little trickier. The unit normal ‘n’ is
                        n̂ = (ea × eb) / ‖ea × eb‖.
                   Differentiating n̂ with respect to time is left
                   as an exercise... but here’s some code */

                triple  eadot = c->a->omega Θ ea,       /* ė_a */
                        ebdot = c->b->omega Θ eb;       /* ė_b */
                        n1 = ea * eb,
                        z = eadot * eb + ea * ebdot;
                double  l = length(n1);

                n1 = n1 / length;                       /* normalize */

                return (z - ((z * n) * n)) / l;
        }
}

void compute_b_vector(Contact contacts[], int ncontacts, vector &b)
{
        for(int i = 0; i < ncontacts; i++)
        {
                Contact *c = &contacts[i];
                Body    *A = c->a,
                        *B = c->b;
                triple  n = c->n,               /* n̂_i(t_0) */
                        ra = c->p - A->x,       /* p − x_a(t_0) */
                        rb = c->p - B->x;       /* p − x_b(t_0) */


                /* Get the external forces and torques */
                triple  f_ext_a = A->force,
                        f_ext_b = B->force,
                        t_ext_a = A->torque,
```

```
                t_ext_b = B->torque;

        triple  a_ext_part, a_vel_part,
                b_ext_part, b_vel_part;

        /* Operators: ‘Θ’ is for cross product, ‘*’, is for
           dot products (between two triples), or matrix-vector
           multiplication (between a matrix and a triple). */

        /* Compute the part of p̈a(t0) due to the external
           force and torque, and similarly for p̈b(t0). */

        a_ext_part = f_ext_a / A->mass +
                        ((A->Iinv * t_ext_a) Θ ra),
        b_ext_part = f_ext_b / B->mass +
                        ((B->Iinv * t_ext_b) Θ rb);

        /* Compute the part of p̈a(t0) due to velocity,
           and similarly for p̈b(t0). */

        a_vel_part = (A->omega Θ (A->omega Θ ra)) +
                        ((A->Iinv * (A->L * A->omega)) Θ ra);

        b_vel_part = (B->omega Θ (B->omega Θ rb)) +
                        ((B->Iinv * (B->L * B->omega)) Θ rb);


        /* Combine the above results, and dot with n̂i(t0) */
        double  k1 = n * ((a_ext_part + a_vel_part) -
                          (b_ext_part + b_vel_part));
        triple  ndot = compute_ndot(c);

        /* See section 8 for ‘pt_velocity’ definition */
        double  k2 = 2 * ndot * (pt_velocity(A, c->p) -
                                 pt_velocity(B, c->p));

        b[i] = k1 + k2;
    }
}
```

Computing the a_{ij} terms is a little more tricky, because we have to keep track of how the jth contact force affects the ith contact point. The following routine is not the most efficient way to do things, because with a good data structure, you can tell in advance which of the a_{ij}'s are going to be zero. Still unless you're working with really huge numbers of contacts, not too much extra work will be done.

```
void compute_a_matrix(Contact contacts[], int ncontacts, bigmatrix &a)
{
        for(int i = 0; i < ncontacts; i++)
                for(int j = 0; j < ncontacts; j++)
                        a[i,j] = compute_aij(contacts[i], contacts[j]);
}

double compute_aij(Contact ci, Contact cj)
{
        /* If the bodies involved in the ith and jth contact are
           distinct, then a_ij is zero. */

        if((ci.a != cj.a) && (ci.b != cj.b) &&
           (ci.a != cj.b) && (ci.b != cj.a))
                return 0.0;

        Body    *A = ci.a,
                *B = ci.b;
        triple  ni = ci.n,              /* n̂_i(t_0) */
                nj = cj.n,              /* n̂_j(t_0) */
                pi = ci.p,              /* ith contact point location */
                pj = cj.p,              /* jth contact point location */
                ra = pi - A->x,
                rb = pi - B->x;

        /* What force and torque does contact j exert on body A? */
        triple  force_on_a = 0,
                torque_on_a = 0;

        if(cj.a == ci.a)
        {
                /* force direction of jth contact force on A */
                force_on_a = nj;

                /* torque direction */
                torque_on_a = (pj - A->x) ⊖ nj;
        }
        else if(cj.b == ci.a)
        {
                force_on_a = - nj;
                torque_on_a = (pj - A->x) ⊖ nj;
        }

        /* What force and torque does contact j exert on body B? */
        triple  force_on_b = 0,
                torque_on_b = 0;
```

```
        if(cj.a == ci.b)
        {
                /* force direction of jth contact force on B */
                force_on_b = nj;

                /* torque direction */
                torque_on_b = (pj - B->x) ^ nj;
        }
        else if(cj.b == ci.b)
        {
                force_on_b = - nj;
                torque_on_b = (pj - B->x) ^ nj;
        }

        /* Now compute how the jth contact force affects the linear
           and angular acceleration of the contact point on body A */

        triple  a_linear = force_on_a / A->mass,
                a_angular = (A->Iinv * torque_on_a) * ra;

        /* Same for B */

        triple  b_linear = force_on_b / B->mass,
                b_angular = (B->Iinv * torque_on_b) * rb;

        return ni * ((a_linear + a_angular) - (b_linear + b_angular));
}
```

References

[1] D. Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *Computer Graphics* (*Proc. SIGGRAPH*), volume 23, pages 223–232. ACM, July 1989.

[2] D. Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. In *Computer Graphics* (*Proc. SIGGRAPH*), volume 24, pages 19–28. ACM, August 1990.

[3] D. Baraff. Fast contact force computation for nonpenetrating rigid bodies. *Computer Graphics* (*Proc. SIGGRAPH*), 28:23–34, 1994.

[4] J. Canny. Collision detection for moving polyhedra. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(2), 1986.

[5] P.A. Cundall. Formulation of a three-dimensional distinct element model—Part I. A scheme to represent contacts in a system composed of many polyhedral blocks. *International Journal of Rock Mechanics, Mineral Science and Geomechanics*, 25, 1988.

[6] E.G. Gilbert and S.M. Hong. A new algorithm for detecting the collision of moving objects. In *International Conference on Robotics and Automation*, pages 8–13. IEEE, 1989.

[7] P. Gill, S. Hammarling, W. Murray, M. Saunders, and M. Wright. User's guide for LSSOL: A Fortran package for constrained linear least-squares and convex quadratic programming. Technical Report Sol 86-1, Systems Optimization Laboratory, Department of Operations Research, Stanford University, 1986.

[8] P. Gill, W. Murray, M. Saunders, and M. Wright. User's guide for QPSOL: A Fortran package for quadratic programming. Technical Report Sol 84-6, Systems Optimization Laboratory, Department of Operations Research, Stanford University, 1984.

[9] P. Gill, W. Murray, M. Saunders, and M. Wright. User's guide for NPSOL: A Fortran package for nonlinear programming. Technical Report Sol 86-2, Systems Optimization Laboratory, Department of Operations Research, Stanford University, 1986.

[10] H. Goldstein. *Classical Mechanics*. Addison-Wesley, Reading, 1983.

[11] W. Meyer. Distance between boxes: Applications to collision detection and clipping. In *International Conference on Robotics and Automation*, pages 597–602. IEEE, 1986.

[12] P.M. Moore and J. Wilhelms. Collision detection and reponse for computer animation. In *Computer Graphics* (*Proc. SIGGRAPH*), volume 22, pages 289–298. ACM, August 1988.

[13] F.P. Preparata and M.I. Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.

[14] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes*. Cambridge University Press, 1986.

[15] R. Sedgewick. *Algorithms*. Addison-Wesley, 1983.

[16] K. Shoemake. Animating rotation with quaternion curves. In *Computer Graphics* (*Proc. SIGGRAPH*), volume 19, pages 245–254. ACM, July 1985.

[17] B. Von Herzen, A. Barr, and H. Zatz. Geometric collisions for time-dependent parametric surfaces. In *Computer Graphics* (*Proc. SIGGRAPH*), volume 24, pages 39–48. ACM, August 1990.