Physically Based Modeling

Course Organizer

Andrew Witkin
Pixar Animation Studios

Physically based modeling has become an important new approach to computer animation and computer graphics modeling. Although physically based modeling is inherently a mathematical subject, the math involved needn't be any more difficult nor esoteric than the math that underlies many other areas of computer graphics, such as ray tracing or surface modeling. Many papers on the subject have presupposed a specialized mathematical background that many members of the computer graphics community lack. Consequently, many capable computer graphics practitioners, despite their interest in the subject, have simply been put off by the density of the math.

This course addresses the need to make the principles and methods of physically based modeling accessible to a broader computer graphics audience—those who are familiar with mainstream computer graphics and have the usual basic computer graphics math, such as vector/matrix manipulations, but whose first year calculus course may be only dimly re-mem-bered.

Course topics include modeling the dynamics of particle systems and rigid bodies, basic numerical methods for differential equations, simulation of deformable surfaces, collision detection, modeling energy functions and hard constraints, and the dynamics of collision and contact.

Additional material/updates can be found at:

`http://www.pixar.com/aboutpixar/research/pbm2001`

Course Schedule

8:30 am **Introduction**

8:45 am **Differential Equation Basics** **Witkin**

Vector fields and integral curves; initial value problems; basic numerical methods; modular implementation of differential equation solvers.

9:30 am **Particle Dynamics** **Witkin**

F=ma; phase space; basic forces: gravity, drag, springs, etc. Simple particle collisions; structured implementation of interactive mass-and-spring systems.

10:00 am **Break**

10:15 am **Particle Dynamics (Cont'd)** **Witkin**

10:30 am **Implicit Methods** **Baraff**

Penalty methods and the problem of stiffness: visualizing the problem; how to avoid it; what to do if you can't; Simulating large systems.

11:15 am **Cloth and Fur Energy Functions** **Kass**

Point-volume comparisons, convex and nonconvex polyhedra, coherence based methods, curved surfaces.

12:00 pm **Lunch**

1:30 pm **Rigid Body Dynamics** **Baraff**

Center of mass and inertia tensor; orientation and angular velocity; force, torque, and Newton's laws; rigid body equations of motion: how to simulate rigid bodies.

2:15 pm **Constrained Dynamics** **Witkin**

"Tinkertoy" systems: rigid rods instead of springs. Using constraint forces to avoid stiffness. Lagrange multipliers: solving for constraint forces. basics of collision and contact.

3:00 pm **Break**

3:15 pm **Collision and Contact** **Baraff**

Impulses; one-sided constraints; multiple constraint; discontinuities.

4:00 pm **Dynamics in *Monsters, Inc.*** **Witkin/Baraff/Kass**

Course Speakers

Andrew Witkin joined Pixar Animation Studios in 1998 as a Senior Animation Scientist. He was previously on the faculty of Carnegie Mellon University, from 1988 through 1998. Prior to joining the faculty at Carnegie Mellon, Andrew Witkin was director of the Perception and Graphics groups at Schlumberger's Palo Alto Research Lab. He received a BA in Psychology from Columbia College in 1975 and a PhD in Psychology from MIT in 1980. Dr. Witkin has published extensively in the areas of Computer Vision and Computer Graphics. He serves as an associate editor for ACM Transactions on Graphics, has served on numerous conference program committees, and is a fellow of the American Association for Artifical Intellgience. His awards include Best Paper prizes at the National Conference on Artifical Intellegence and the International Joint conference on Artifical Intellgence, the Grand Prix for Animation at the 1987 Parigraph competition in Paris, France, and the Grand Prix for Computer Graphics, Prix Ars Electronica 1992, Linz, Austria. He is the recipient of this year's ACM SIGGRAPH Computer Graphics Achievement Award.

David Baraff joined Pixar Animation Studios in 1998 as a Senior Animation Scientist. Prior to his arrival at Pixar, he was an Associate Professor of Robotics, and Computer Science at Carnegie Mellon University. David Baraff received his Ph.D. from Cornell University in 1992, where he was a graduate student in Cornell's Department of Computer Science, and Program of Computer Graphics. Before and during his graduate studies, he also worked at Bell Laboratories' Computer Technology Research Laboratory doing computer graphics research, including real-time 3D interactive animation and games. After receiving his Ph.D., he joined the faculty of Carnegie Mellon University. In 1995, he was named an ONR Young Investigator. His research interests include physical simulation and modeling for computer graphics, robotics, and animation.

Michael Kass is a Senior Scientist at Pixar Animation Studios where he developed the tools for physically-based clothing animation that were used on Pixar's Academy Award winning short film "Geri's game." He received his B.A. from Princeton in 1982, his M.S. from M.I.T in 1984, and his Ph. D. from Stanford in 1988. Dr. Kass has received numerous awards for his research on physically-based methods in computer graphics and computer vision including several conference best paper awards, the Prix Ars Electronica for the image "Reaction Diffusion Texture Buttons" and the Imagina Grand Prix for the animation "Splash Dance." Before joining Pixar in 1995, Dr. Kass held research positions at Schlumberger Palo Alto Research, Apple Computer, and was Director of Technology at Live Picture Inc.

Contents

Physically Based Modeling
Differential Equation Basics

Andrew Witkin and David Baraff
Pixar Animation Studios

Differential Equation Basics

Andrew Witkin and David Baraff
Pixar Animation Studios

1 Initial Value Problems

Differential equations describe the relation between an unknown function and its derivatives. To *solve* a differential equation is to find a function that satisfies the relation, typically while satisfying some additional conditions as well. In this course we will be concerned primarily with a particular class of problems, called *initial value problems*. In a canonical initial value problem, the behavior of the system is described by an ordinary differential equation (ODE) of the form

$$\dot{\mathbf{x}} = f(\mathbf{x}, t),$$

where f is a known function (i.e. something we can evaluate given $\mathbf{x}$ and t,) $\mathbf{x}$ is the *state* of the system, and $\dot{\mathbf{x}}$ is $\mathbf{x}$'s time derivative. Typically, $\mathbf{x}$ and $\dot{\mathbf{x}}$ are vectors. As the name suggests, in an initial value problem we are given $\mathbf{x}(t_0) = \mathbf{x}_0$ at some starting time t_0, and wish to follow $\mathbf{x}$ over time thereafter.

The generic initial value problem is easy to visualize. In $2D$, $\mathbf{x}(t)$ sweeps out a curve that describes the motion of a point $\mathbf{p}$ in the plane. At any point $\mathbf{x}$ the function f can be evaluated to provide a 2-vector, so f defines a vector field on the plane (see figure 1.) The vector at $\mathbf{x}$ is the velocity that the moving point $\mathbf{p}$ must have if it ever moves through $\mathbf{x}$ (which it may or may not.) Think of f as *driving* $\mathbf{p}$ from point to point, like an ocean current. Wherever we initially deposit $\mathbf{p}$, the "current" at that point will seize it. Where $\mathbf{p}$ is carried depends on where we initially drop it, but once dropped, all future motion is determined by f. The trajectory swept out by $\mathbf{p}$ through f forms an *integral curve* of the vector field. See figure 2.

We wrote f as a function of both $\mathbf{x}$ and t, but the derivative function may or may not depend directly on time. If it does, then not only the point $\mathbf{p}$ but the the vector field itself moves, so that $\mathbf{p}$'s velocity depends not only on where it is, but on when it arrives there. In that case, the derivative $\dot{\mathbf{x}}$ depends on time in *two ways:* first, the derivative vectors themselves wiggle, and second, the point $\mathbf{p}$, because it moves on a trajectory $\mathbf{x}(t)$, sees different derivative vectors at different times. This dual time dependence shouldn't lead to confusion if you maintain the picture of a particle floating through an undulating vector field.

2 Numerical Solutions

Standard introductory differential equation courses focus on *symbolic* solutions, in which the functional form for the unknown function is to be guessed. For example, the differential equation $\dot{x} = -kx$, where $\dot{x}$ denotes the time derivative of x, is satisfied by $x = e^{-kt}$.

In contrast, we will be concerned exclusively with *numerical* solutions, in which we take discrete *time steps* starting with the initial value $\mathbf{x}(t_0)$. To take a step, we use the derivative function

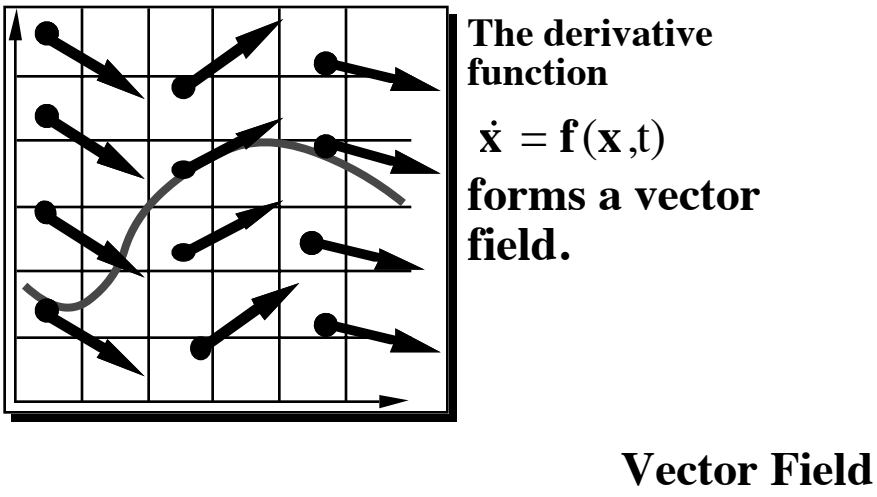


Figure 1: The derivative function $f(\mathbf{x}, t)$. defines a vector field.

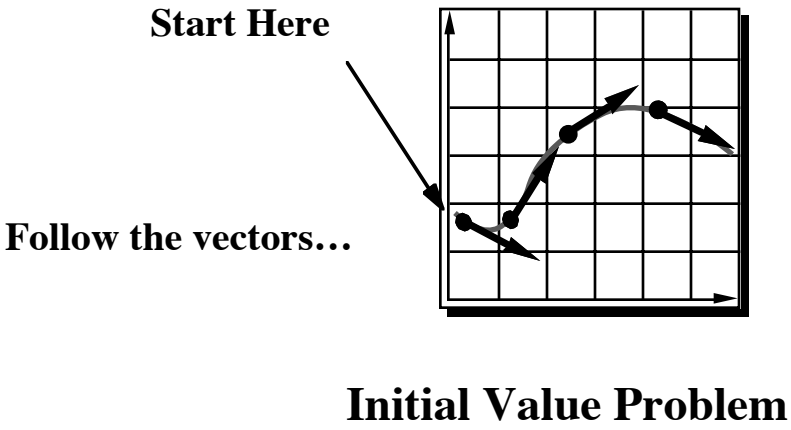
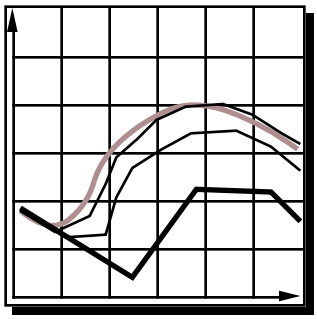


Figure 2: An initial value problem. Starting from a point $\mathbf{x}_0$, move with the velocity specified by the vector field.



- **Simplest numerical solution method**
- **Discrete time steps**
- **Bigger steps, bigger errors.**

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t\, f(\mathbf{x},t)$$

Euler's Method

Figure 3: Euler's method: instead of the true integral curve, the approximate solution follows a polygonal path, obtained by evaluating the derivative at the beginning of each leg. Here we show how the accuracy of the solution degrades as the size of the time step increases.

f to calculate an approximate change in $\mathbf{x}$, $\Delta\mathbf{x}$, over a time interval Δt, then increment $\mathbf{x}$ by $\Delta\mathbf{x}$ to obtain the new value. In calculating a numerical solution, the derivative function f is regarded as a black box: we provide numerical values for $\mathbf{x}$ and t, receiving in return a numerical value for $\dot{\mathbf{x}}$. Numerical methods operate by performing one or more of these *derivative evaluations* at each time step.

2.1 Euler's Method

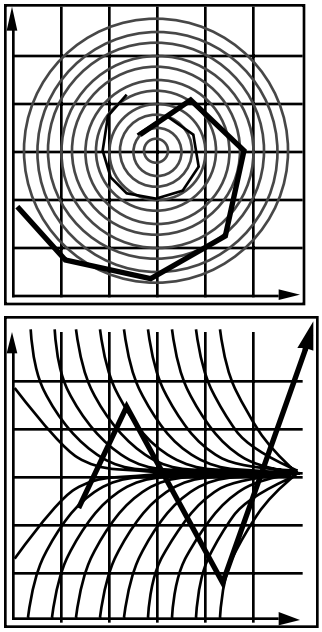
The simplest numerical method is called Euler's method. Let our initial value for $\mathbf{x}$ be denoted by $\mathbf{x}_0 = \mathbf{x}(t_0)$ and our estimate of $\mathbf{x}$ at a later time $t_0 + h$ by $\mathbf{x}(t_0 + h)$, where h is a *stepsize* parameter. Euler's method simply computes $\mathbf{x}(t_0 + h)$ by taking a step in the derivative direction,

$$\mathbf{x}(t_0 + h) = \mathbf{x}_0 + h\dot{\mathbf{x}}(t_0).$$

You can use the mental picture of a 2D vector field to visualize Euler's method. Instead of the real integral curve, $\mathbf{p}$ follows a polygonal path, each leg of which is determined by evaluating the vector f at the beginning, and scaling by h. See figure 3.

Though simple, Euler's method is not accurate. Consider the case of a $2D$ function f whose integral curves are concentric circles. A point $\mathbf{p}$ governed by f is supposed to orbit forever on whichever circle it started on. Instead, with each Euler step, $\mathbf{p}$ will move on a straight line to a circle of larger radius, so that its path will follow an outward spiral. Shrinking the stepsize will slow the rate of this outward drift, but never eliminate it.

Moreover, Euler's method can be unstable. Consider a $1D$ function $f = -kx$, which should make the point $\mathbf{p}$ decay exponentially to zero. For sufficiently small step sizes we get reasonable



Inaccuracy:
Error turns x(t) from a circle into the spiral of your choice.

Instability: off to Neptune!

Two Problems

Figure 4: Above: the real integral curves form concentric circles, but Euler's method always spirals outward, because each step on the current circle's tangent leads to a circle of larger radius. Shrinking the stepsize doesn't cure the problem, but only reduces the rate at which the error accumulates. Below: too large a stepsize can make Euler's method diverge.

behavior, but when $h > 1/k$, we have $|\Delta x| > |x|$, so the solution oscillates about zero. Beyond $h = 2/k$, the oscillation diverges, and the system blows up. See figure 4.

Finally, Euler's method isn't even efficient. Most numerical solution methods spend nearly all their time performing derivative evaluations, so the computational cost *per step* is determined by the number of evaluations per step. Though Euler's method only requires one evaluation per step, the real efficiency of a method depends on the size of the steps it lets you take—while preserving accuracy and stability—as well as on the cost per step. More sophisticated methods, even some requiring as many as four or five evaluations per step, can greatly outperform Euler's method because their higher cost per step is more than offset by the larger stepsizes they allow.

To understand how we go about improving on Euler's method, we need to look more closely at the error that the method produces. The key to understanding what's going on is the *Taylor series*: Assuming $\mathbf{x}(t)$ is smooth, we can express its value at the end of the step as an infinite sum involving the the value and derivatives at the beginning:

$$\mathbf{x}(t_0 + h) = \mathbf{x}(t_0) + h\dot{\mathbf{x}}(t_0) + \frac{h^2}{2!}\ddot{\mathbf{x}}(t_0) + \frac{h^3}{3!}\dddot{\mathbf{x}}(t_0) + \ldots + \frac{h^n}{n!}\frac{\partial^n \mathbf{x}}{\partial t^n} + \ldots$$

As you can see, we get the Euler update formula by *truncating* the series, discarding all but the first two terms on the right hand side. This means that Euler's method would be correct only if all derivatives beyond the first were zero, i.e. if $\mathbf{x}(t)$ were linear. The *error term,* the difference between the Euler step and the full, untruncated Taylor series, is dominated by the leading term, $(h^2/2)\ddot{\mathbf{x}}(t_0)$. Consequently, we can describe the error as $O(h^2)$ (read *"Order h squared".*) Suppose

that we chop our stepsize in half; that is, we take steps of size $\frac{h}{2}$. Although this produces only about one fourth the error we got with a stepsize of h, we have to take twice as many steps over any given interval. That means that the error we accumulate over an interval t_0 to t_1 depends linearly upon h. Theoretically, using Euler's method we can numerically compute $\mathbf{x}$ over an interval t_0 to t_1 with as little error as we want, by choosing a suitably small h. In practice, a great many timesteps might be required, depending on the error and the function f.

2.2 The Midpoint Method

If we were able to evaluate $\ddot{\mathbf{x}}$ as well as $\dot{\mathbf{x}}$, we could acheive $O(h^3)$ accuracy instead of $O(h^2)$ simply by retaining one additional term in the truncated Taylor series:

$$\mathbf{x}(t_0 + h) = \mathbf{x}(t_0) + h\dot{\mathbf{x}}(t_0) + \frac{h^2}{2}\ddot{\mathbf{x}}(t_0) + O(h^3). \tag{1}$$

Recall that the time derivative $\dot{\mathbf{x}}$ is given by a function $f(\mathbf{x}(t), t)$. For simplicity in what follows, we will assume that the derivative function f does depends on time only indirectly through $\mathbf{x}$, so that $\dot{\mathbf{x}} = f(\mathbf{x}(t))$. The chain rule then gives

$$\ddot{\mathbf{x}} = \frac{\partial f}{\partial \mathbf{x}}\dot{\mathbf{x}} = f' f.$$

To avoid having to evaluate f',which would often be complicated and expensive, we can approximate the second-order term just in terms of f, and substitute the approximation into equation 1, leaving us with $O(h^3)$ error. To do this, we perform another Taylor expansion, this time of the function of f,

$$f(\mathbf{x}_0 + \Delta\mathbf{x}) = f(\mathbf{x}_0) + \Delta\mathbf{x} f'(\mathbf{x}_0) + O(\Delta\mathbf{x}^2). \tag{2}$$

We first introduce $\ddot{\mathbf{x}}$ into this expression by choosing

$$\Delta\mathbf{x} = \frac{h}{2} f(\mathbf{x}_0)$$

so that

$$f(\mathbf{x}_0 + \frac{h}{2} f(\mathbf{x}_0)) = f(\mathbf{x}_0) + \frac{h}{2} f(\mathbf{x}_0) f'(\mathbf{x}_0) + O(h^2) = f(\mathbf{x}_0) + \frac{h}{2}\ddot{\mathbf{x}}(t_0) + O(h^2),$$

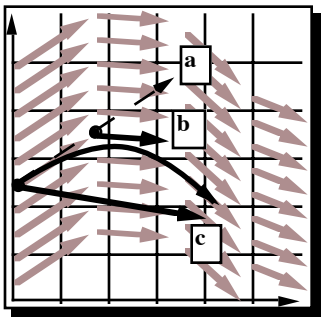
where $\mathbf{x}_0 = \mathbf{x}(t_0)$. We can now multiply both sides by h (turning the $O(h^2)$ term into $O(h^3)$) and rearrange, yielding

$$\frac{h^2}{2}\ddot{\mathbf{x}} + O(h^3) = h(f(\mathbf{x}_0 + \frac{h}{2} f(\mathbf{x}_0)) - f(\mathbf{x}_0).$$

Substituting the right hand side into equation 1 gives the update formula

$$\mathbf{x}(t_0 + h) = \mathbf{x}(t_0) + h(f(\mathbf{x}_0 + \frac{h}{2} f(\mathbf{x}_0)).$$

This formula first evaluates an Euler step, then performs a second derivative evaluation at the midpoint of the step, using the midpoint evaluation to update $\mathbf{x}$. Hence the name *midpoint method*. The midpoint method is correct to within $O(h^3)$, but requires two evaluations of f. See figure 5 for a pictorial view of the method.



a. Compute an Euler step

$$\Delta \mathbf{x} = \Delta t\, \mathbf{f}(\mathbf{x}, t)$$

b. Evaluate f at the midpoint

$$\mathbf{f}_{mid} = \mathbf{f}\left(\frac{x + \Delta x}{2}, \frac{t + \Delta t}{2}\right)$$

c. Take a step using the midpoint value

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t\, \mathbf{f}_{mid}$$

The Midpoint Method

Figure 5: The midpoint method is a 2nd-order solution method. a) an euler step is computed, b) the derivative is evaluated again at the step's midpoint, and the second evaluation is used to calculate the step. The integral curve—the actual solution—is shown as c.

We don't have to stop with an error of $O(h^3)$. By evaluating f a few more times, we can eliminate higher and higher orders of derivatives. The most popular procedure for doing this is a method called Runge-Kutta of order 4 and has an error per step of $O(h^5)$. (The Midpoint method could be called Runge-Kutta of order 2.) We won't derive the fourth order Runge-Kutta method, but the formula for computing $\mathbf{x}(t_0 + h)$ is listed below:

$$\begin{aligned} k_1 &= hf(\mathbf{x}_0, t_0) \\ k_2 &= hf(\mathbf{x}_0 + \frac{k_1}{2}, t_0 + \frac{h}{2}) \\ k_3 &= hf(\mathbf{x}_0 + \frac{k_2}{2}, t_0 + \frac{h}{2}) \\ k_4 &= hf(\mathbf{x}_0 + k_3, t_0 + h) \\ \mathbf{x}(t_0 + h) &= \mathbf{x}_0 + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4. \end{aligned}$$

3 Adaptive Stepsizes

Whatever the underlying method, a major problem lies in determing a good stepsize. Ideally, we want to choose h as large as possible—but not so large as to give us an unreasonable amount of error, or worse still, to induce instability. If we choose a fixed stepsize, we can only proceed as fast as the "worst" sections of $\mathbf{x}(t)$ will allow. What we would like to do is to vary h as we march forward in time. Whenever we can make h large without incurring too much error, we should do so. When h has to be reduced to avoid excessive error, we want to do that also. This is the idea of

adaptive stepsizing: varying h over the course of solving the ODE.

Here we'll be present adaptive stepsizing for Euler's method. The basic idea is as follows. Lets assume we have a given stepsize h, and we want to know how much we can consider changing it.

Suppose we compute two estimates for $\mathbf{x}(t_0 + h)$. We compute an estimate $\mathbf{x}_a$, by taking an Euler step of size h from t_0 to $t_0 + h$. We also compute an estimate $\mathbf{x}_b$ by taking *two* Euler steps of size $h/2$, from t_0 to $t_0 + h$. Both $\mathbf{x}_a$ and $\mathbf{x}_b$ differ from the true value of $\mathbf{x}(t_0 + h)$ by $O(h^2)$. That means that $\mathbf{x}_a$ and $\mathbf{x}_b$ differ from each other by $O(h^2)$. As a result, we can write that a measure of the current error e is

$$e = |\mathbf{x}_a - \mathbf{x}_b|$$

This gives us a convenient estimate to the error in taking an Euler step of size h.

Suppose that we are willing to have an error of as much as 10^{-4} per step, and that the current error is only 10^{-8}. Since the error goes up as h^2, we can increase the stepsize to

$$\left(\frac{10^{-4}}{10^{-8}}\right)^{\frac{1}{2}} h = 100h.$$

Conversely, if we currently had an error of 10^{-3}, and could only tolerate an error of 10^{-4}, we would have to decrease the stepsize to

$$\left(\frac{10^{-4}}{10^{-3}}\right)^{\frac{1}{2}} h \approx .316h.$$

Adaptive stepsizing is a highly recommended technique.

4 Implementation

The ODEs we will want to solve may represent many things—for instance, a collection of masses and springs, some rigid bodies, or a deformable object. We want to implement ODE solvers and the models on which they operate in a way that isolates each from the internal details of the other. This will make it possible to change solvers easily, and also make the solver code reusable. Fortunately, this kind of modularity is not difficult to acheive, since all solvers can be expressed in terms of a small, stereotyped set of operations. Presumably, the system of ODE-governed objects will be embodied in a structure of some kind. The approach is to write type-specific code that operates on this structure to perform the standard operations, then to implement solvers in terms of these generic operations.

From the solver's viewpoint, the system on which it operates is a black-box function $f(\mathbf{x}, t)$. The solver needs to be able to evaluate f, as required, at any values of $\mathbf{x}$ and t, and then to install the updated $\mathbf{x}$ and t when a time step is taken. To support these operations, the object that represents the ODE being solved must be able to handle these requests from the solver:

- Return dim ($\mathbf{x}$). Since $\mathbf{x}$ and $\dot{\mathbf{x}}$ may be vectors, the solver must know their length, to allocate storage, perform vector arithmetic ops, etc.
- Get/set $\mathbf{x}$ and t. The solver must be able to install new values at the end of a step. In addition, a multi-step method must set $\mathbf{x}$ and t to intermediate values in the course of performing derivative evaulations.
- Evaluate f at the current $\mathbf{x}$ and t.

In an object-oriented language, these operations would naturally be implemented as generic functions that are handled in a type-specific way. In a non-object-oriented language generic functions would be faked by installing pointers to type-specific functions in structure slots, or simply by passing the function pointers as arguments to the solver. Later on we will consider in detail how these operations are to be implemented for specific models such as particle-and-spring systems.

References

[1] W.H. Press, B.P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, England, 1988.

Physically Based Modeling
Particle System Dynamics

Andrew Witkin
Pixar Animation Studios

Particle System Dynamics

Andrew Witkin
Pixar Animation Studios

1 Introduction

Particles are objects that have mass, position, and velocity, and respond to forces, but that have no spatial extent. Because they are simple, particles are by far the easiest objects to simulate. Despite their simplicity, particles can be made to exhibit a wide range of interesting behavior. For example, a wide variety of nonrigid structures can be built by connecting particles with simple damped springs. In this portion of the course we cover the basics of particle dynamics, with an emphasis on the requirements of interactive simulation.

2 Phase Space

The motion of a Newtonian particle is governed by the familiar $\mathbf{f} = m\mathbf{a}$, or, as we will write it here, $\ddot{\mathbf{x}} = \mathbf{f}/m$. This equation differs from the canonical ODE developed in the last chapter because it involves a second time derivative, making it a *second order* equation. To handle a second order ODE, we convert it to a first-order one by introducing extra variables. Here we create a variable v to represent velocity, giving us a pair of coupled first-order ODE's $\dot{\mathbf{v}} = \mathbf{f}/m, \quad \dot{\mathbf{x}} = \mathbf{v}$. The position and velocity $\mathbf{x}$ and $\mathbf{v}$ can be concatenated to form a 6-vector. This position/velocity product space is called *phase space*. In components, the phase space equation of motion is $[\dot{x}_1, \dot{x}_2, \dot{x}_3, \dot{v}_1, \dot{v}_2, \dot{v}_3] = [v_1, v_2, v_3, f_1/m, f_2/m, f_3/m]$, which, assuming force is a function of $\mathbf{x}$ and t, matches our canonical form $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$. A system of n particles is described by n copies of the equation, concatenated to form a $6n$-long vector. Conceptually, the whole system may be regarded as a point moving through $6n$-space.

We can still visualize the phase-space ODE in terms of a planar vector field, though only for a $1D$ particle, by letting one axis represent the particle's position and the other, its velocity. If each point in the phase plane represents a pair $[x, v]$, then the derivative vector is $[v, f/m]$. All the ideas of integral curves, polygonal approximations, and so forth, carry over intact to phase space. Only the interpretation of the trajectory is changed.

3 Basic Particle Systems

In implementing particle systems, we want to maintain two views of our model: from "outside," especially from the point of view of the ODE solver, the model should look like a monolith—a point in a high-dimensional space, whose time derivative may be evaluated at will. From within, the model should be a structured—a collection of distinct interacting objects. This duality will be recurring theme in the course.

A particle simulation involves two main parts—the particles themselves, and the entities that apply forces to particles. In this section we consider just the former, deferring until the next section the specifics of force calculation. Our goal here is to describe structures that could represent a particle and a system of particles, and to show in a concrete way how to implement the generic operations required by ODE solvers.

Particles have mass, position, and velocity, and are subjected to forces, leading to an obvious structure definition, which in C might look like:

```
typedef struct{
  float m;       /* mass */
  float *x;      /* position vector */
  float *v;      /* velocity vector */
  float *f;      /* force accumulator */
} *Particle;
```

In practice, there would probably be extra slots describing appearance and other properties. A system of particles could be represented in an equally obvious way, as

```
typedef struct{
  Particle *p;    /* array of pointers to particles */
  int n;          /* number of particles */
  float t;       /* simulation clock */
} *ParticleSystem;
```

Assume that we have a function `CalculateForces()` that, called on a particle system, adds the appropriate forces into each particle's `f` slot. Don't worry for know about what that function actually does. Then the operations that comprise the ODE solver interface could be written as follows:

```
/* length of state derivative, and force vectors */
int ParticleDims(ParticleSystem p){
  return(6 * p->n);
};

/* gather state from the particles into dst */
int ParticleGetState(ParticleSystem p, float *dst){
  int i;
  for(i=0; i < p->n; i++){
    *(dst++) = p->p[i]->x[0];
    *(dst++) = p->p[i]->x[1];
    *(dst++) = p->p[i]->x[2];
    *(dst++) = p->p[i]->v[0];
    *(dst++) = p->p[i]->v[1];
    *(dst++) = p->p[i]->v[2];
  }
}
```

```
/* scatter state from src into the particles */
int ParticleSetState(ParticleSystem p, float *src){
  int i;
  for(i=0; i < p->n; i++){
  p->p[i]->x[0] = *(src++);
  p->p[i]->x[1] = *(src++);
  p->p[i]->x[2] = *(src++);
  p->p[i]->v[0] = *(src++);
  p->p[i]->v[1] = *(src++);
  p->p[i]->v[2] = *(src++);
  }
}

/* calculate derivative, place in dst */
int ParticleDerivative(ParticleSystem p, float *dst){
 int i;
 Clear_Forces(p);   /* zero the force accumulators */
 Compute_Forces(p); /* magic force function */
  for(i=0; i < p->n; i++){
    *(dst++) = p->p[i]->v[0];    /* xdot = v */
    *(dst++) = p->p[i]->v[1];
    *(dst++) = p->p[i]->v[2];
    *(dst++) = p->p[i]->f[0]/m; /* vdot = f/m */
    *(dst++) = p->p[i]->f[1]/m;
    *(dst++) = p->p[i]->f[2]/m;
  }
}
```

Having defined these operations, and assuming some utility routines and temporary vectors, an Euler solver be written as

```
void EulerStep(ParticleSystem p, float DeltaT){
  ParticleDeriv(p,temp1);   /* get deriv */
  ScaleVector(temp1,DeltaT)       /* scale it */
  ParticleGetState(p,temp2);      /* get state */
  AddVectors(temp1,temp2,temp2);  /* add -> temp2 */
  ParticleSetState(p,temp2);      /* update state */
  p->t += DeltaT;                 /* update time */
}
```

The structures representing a particle and a particle system are shown visually in figures 1 and 2. The interface between a particle system and a differential equation solver is illustrated in figure 3.

4 Forces

All particles are essentially alike. In contrast, the objects that give rise to forces are heterogeneous. As a matter of implementation, we would like to make it easy to extend the set of force-producing

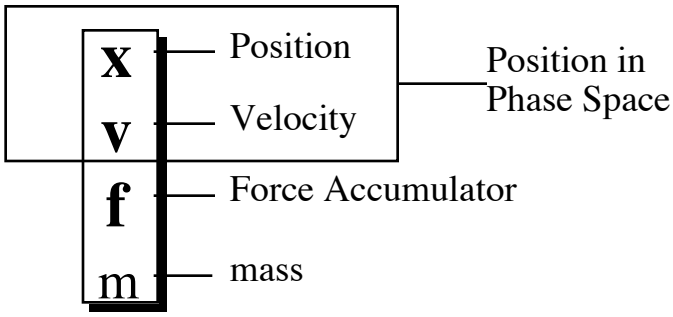


Figure 1: A particle may be represented by a structure containing its position, velocity, force, and mass. The six-vector formed by concatenating the position and velocity comprises the point's position in phase space.

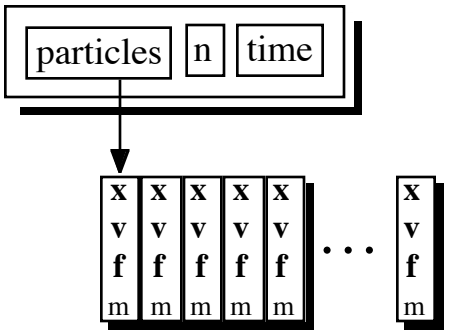


Figure 2: A bare particle system is essentially just a list of particles.

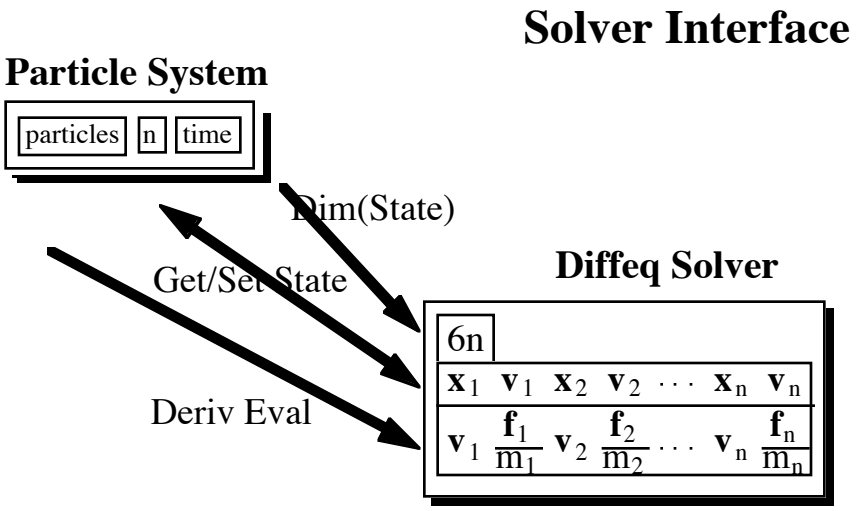


Figure 3: The relation between a particle system and a differential equation solver.

objects without modifying the basic particle system model. We accomplish this by having the particle system maintain a list of force objects, each of which has access to any or all particles, and each of which "knows" how to apply its own forces. The `Calculateforces` function, used above, simply traverses the list of force structures, calling each of their `ApplyForce` functions, with the particle system itself as sole argument. This leaves the real work of force calculation to the individual objects. See figures 4 and 5

Forces can be grouped into three broad categories:

- Unary forces, such as gravity and drag, that act independently on each particle, either exerting a constant force, or one that depends on one or more of particle position, particle velocity, and time.
- n-ary forces, such as springs, that apply forces to a fixed set of particles.
- Forces of spatial interaction, such as attraction and repulsion, that may act on any or all pairs of particles, depending on their positions.

Each of these raises somewhat different implementation issues. We will now consider each in turn.

4.1 Unary forces

Gravity. Global earth gravity (as opposed to particle-particle attraction) is trivial to implement. The gravitational force on each particle is $\mathbf{f} = m\mathbf{g}$, where $\mathbf{g}$ is a constant vector (presumably pointing down) whose magnitude is the gravitational constant. If all particles are to feel the same gravity, which they need not in a simulation, then gravitational force is applied simply by traversing the

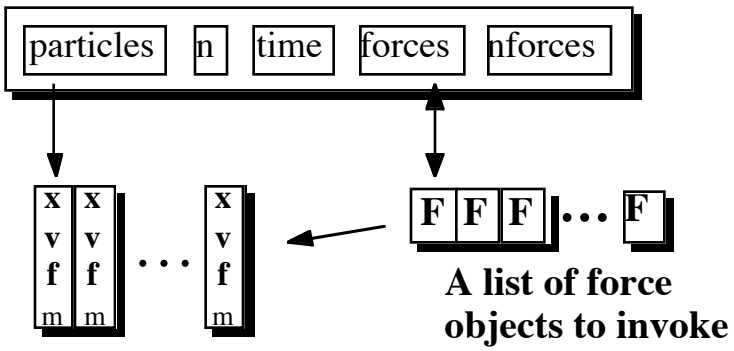


Figure 4: A particle system augmented to contain a list of *force objects*. Each force object points at the particles that it influences, and contains a function that knows how to compute the force on each affected particle.

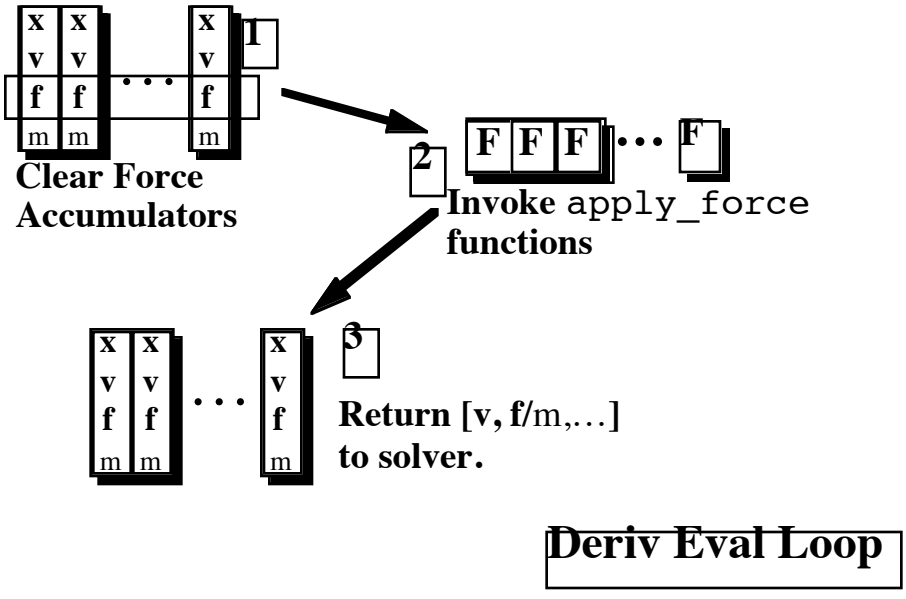


Figure 5: The derivative evaluation loop for a particle system with force objects.

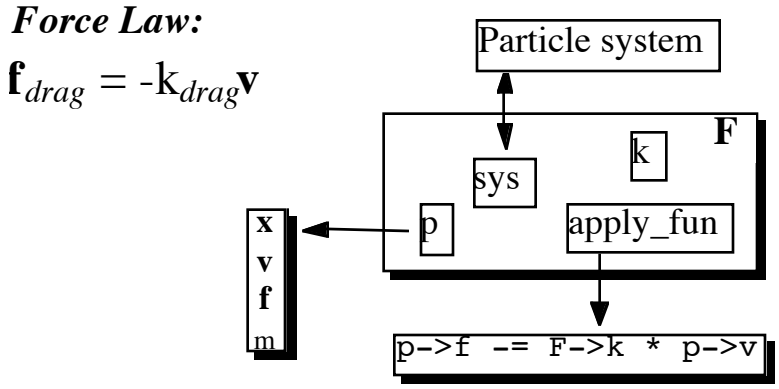


Figure 6: Schematic view of a force object implementing viscous drag. The object points at the particle to which drag is being applied, and also points to a function that implements the force law for drag.

system's particle list, and adding the appropriate force into each particles force accumulator. Gravity is basic enough that it could reasonably be wired it into the particle system, rather than maintaining a distinct "gravity object."

Viscous Drag. Ideal viscous drag has the form $\mathbf{f} = -k_d\mathbf{v}$, where the constant k_d is called the *coefficient of drag*. The effect of drag is to resist motion, making a particle gradually come to rest in the absence of other influences. It is highly reccommended that at least a small amount of drag be applied to each particle, if only to enhance numerical stability. Excessive drag, however, makes it appear that the particles are floating in molasses. Like gravity, drag can be implemented as a wired-in special case. A force object implementing viscous drag is shown in figure 6.

4.2 *n*-ary forces

Our canonical example of a binary force is a Hook's law spring. In a basic mass-and-spring simulation, the springs are the structural elements that hold everything together. The spring forces between a pair of particles at positions **a** and **b** are

$$\mathbf{f}_a = -\left[k_s(|\mathbf{l}| - r) + k_d\frac{\dot{\mathbf{l}}\cdot\mathbf{l}}{|\mathbf{l}|}\right]\frac{\mathbf{l}}{|\mathbf{l}|}, \quad \mathbf{f}_b = -\mathbf{f}_a, \tag{1}$$

where $\mathbf{f}_a$ and $\mathbf{f}_b$ are the forces on **a** and **b**, respectively, $\mathbf{l} = \mathbf{a} - \mathbf{b}$, r is the rest length, k_s is a spring constant, and k_d is a damping constant. $\dot{\mathbf{l}}$, the time derivative of $\mathbf{l}$, is just $\mathbf{v}_a - \mathbf{v}_b$, the difference between the two particles' velocities.

In equation 1, the spring force magnitude is proportional to the difference between the actual length and the rest length, while the damping force magnitude is proportional to a and b's speed of

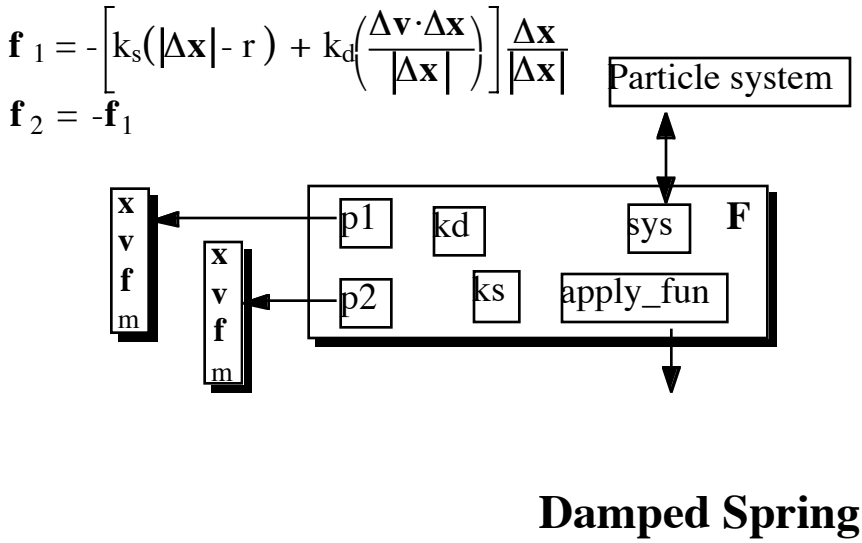


Figure 7: A schematic view of a force object implementing a damped spring that attaches particles p_1 and p_2.

approach. Equal and opposite forces act on each particle, along the line that joins them. The spring damping differs from global drag in that it acts symmetrically on the two particles, having no effect on the motion of their common center of mass. Later, we will learn a general procedure for deriving this kind of force expression.

A damped spring can be implemented straightforwardly as a structure that points to the pair of particles it connects. The code that applies the forces according to equation 1 fetches the positions and velocities from the two particle structures, performs its calculations, and sums the results into the particles' force accumulators. In an object-oriented environment, this operation would be implemented as a generic function. In bare C, the force object would contain a pointer to an ordinary C function. A force object for a damped spring is shown in figure 7

4.3 Spatial Interaction Forces

A spring applies forces to a fixed pair of particles. In contrast, spatial interaction forces may act on *any* pair (or n-tuple) of particles. For local interaction forces, particles begin to interact when they come close, and stop when they move apart. Spatially interacting particles have been used as approximate models for fluid behavior, and large-scale particle simulations are widely used in physics [1]. A complication in large-scale spatial interaction simulations is that the force calculation is $O(n^2)$ in the number of particles. If the interactions are local, efficiency may be improved through the use of spatial buckets.

5 User Interaction

An interactive mass-and-spring simulation is an ideal first implementation project in physically based modeling, because such simulations are relatively easy to implement, and because interactive performance can be acheived even on low-end computers. The main ingredients of a basic mass-and-spring simulation are model construction and model manipulation. Model construction can be a simple matter of mouse-clicking to create particles and connect them with springs. Interactive manipulation requires no more than the ability to grab and drag mass points. Although there is barely any difference mathematically between $2D$ and $3D$ simulations, supporting $3D$ user interaction is more challenging.

Most of the implementation issues are standard, and will not be dealt with here. However, we give a few useful tips:

Controlled particles. Particles whose motion is *not* governed by forces provide a number of interesting possibilities. Fixed particles serve as anchors and pivots. Particles whose motion is procedurally controlled (e.g. moving on a circle) can provide dynamic elements such as motors. All that need be done to implement controlled particles is to prevent the ODE solver from updating their positions. One subtle point, though, is that the velocities as well as positions of controlled particles must be maintained at their correct values. Otherwise, velocity-dependent forces such as damped spring forces will behave incorrectly.

Structures. A variety of interesting non-rigid structures—beams, blocks, etc.—can be built out of masses and springs. By allowing several springs to meet at a single particle, these pieces can be connected with a variety of joints. With some experimentation and ingenuity it is possible to construct entire mechanisms, complete with motors, out of masses and springs. The topic of regular mass-and-spring lattices as an approximation to continuum models will be discussed later.[2]

Mouse springs. The simplest way to manipulate mass-and-spring models is to use the mouse directly to control the positions of grabbed particles. However, this method is not recommended because very rapid mouse motions can lead to stability problems. These problems can be avoided by coupling the grabbed particle to the mouse position using a spring.

6 Energy Functions

Generically, the position-, velocity-, and time-dependent formulae that we use to calculate forces are known as *force laws*. Forces laws are not laws of physics. Rather, they form part of our description of the system we are modeling. Some of the standard ones, like linear springs and dashpots, represent time-honored idealizations of the behavior of real materials and structures. However, if we wanted to accurately model the behavior of a pair of particles connected by, say, a strand of gooey taffy, the resulting equations would probably be even messier than the taffy.

Often, we can regard force laws as things we *design* to hold things in a desired configuration—for instance a spring with nonzero rest length makes the points it connects "want" to be a fixed distance apart. In many cases it is possible to specify the desired configuration by giving a function that reaches zero exactly when things are "happy." We can call this kind of function a *behavior function*. For example, a behavior function that says that two particles a and b should be in the same place is just $C(\mathbf{a}, \mathbf{b}) = \mathbf{a} - \mathbf{b}$ (which is a vector expression each of whose components is supposed to vanish.) If instead we want a and b to be distance r apart, then a suitable behavior function is $C(a, b) = |a - b| - r$ (which is a scalar expression.)

Later on, when we study constrained dynamics, we will use this kind of function as a way to

specify constraints, and we will consider in detail the problem of maintaining such constraints accurately. For now, we will be content to impose forces that pull the system toward the desired state, but that compete with other forces. These energy-based forces can be used to impose approximate, sloppy constraints. However, attempting to make them accurate by increasing the spring constant leads to numerical instability.[3]

Converting a behavior function $\mathbf{C}(\mathbf{x}_1 \ldots, \mathbf{x}_n)$ into a force law is a pure cookbook procedure. We first define a scalar potential energy function

$$E = \frac{k_s}{2}\mathbf{C} \cdot \mathbf{C},$$

where k_s is a generalized stiffness constant. Since the force due to a scalar potential is minus the energy gradient, the force on particle $\mathbf{x}_i$ due to $\mathbf{C}$ is

$$\mathbf{f}_i = \frac{-\partial E}{\partial \mathbf{x}_i} = -k_s \mathbf{C} \frac{\partial \mathbf{C}}{\partial \mathbf{x}_i}.$$

In general $\mathbf{C}$ is a vector, and this expression denotes its product with the transpose of the *Jacobian* matrix $\partial C/\partial x_i$. We will look much more closely at this kind of expression when we study constraint methods, and in particular Lagrange multipliers. For now, it is sufficent to think of the forces f_i as generalized spring forces that attract the system to states that satisfy $\mathbf{C} = 0$. When a behavior function depends on a number of particles' positions, we get a different force expression for each by using C's derivative with respect to that particle.

The force we just defined isn't quite the one we want: in the absence of any damping, this conservative force will cause the system to oscillate about $\mathbf{C} = 0$. To add damping, we modify the force expression to be

$$\mathbf{f}_i = (-k_s \mathbf{C} - k_d \dot{\mathbf{C}}) \frac{\partial \mathbf{C}}{\partial \mathbf{x}_i}, \tag{2}$$

where k_d is a generalized damping constant, and $\dot{\mathbf{C}}$ is the time derivative of C. Note that when you derive expressions for $\dot{\mathbf{C}}$, you will be using the fact that $\dot{\mathbf{x}}_i = \mathbf{v}_i$. So, in a trivial case, if $\mathbf{C} = \mathbf{x}_1 - \mathbf{x}_2$, it follows that $\dot{\mathbf{C}} = \mathbf{v}_1 - \mathbf{v}_2$.

As an extremely simple example, we take $\mathbf{C} = \mathbf{x}_1 - \mathbf{x}_2$, which wants the points to coincide. We have

$$\frac{\partial \mathbf{C}}{\partial \mathbf{x}_1} = \mathbf{I}, \quad \frac{\partial \mathbf{C}}{\partial \mathbf{x}_2} = -\mathbf{I},$$

where $\mathbf{I}$ is the identity matrix. The time derivative is

$$\dot{\mathbf{C}} = \mathbf{v}_1 - \mathbf{v}_2.$$

So, substituting into equation 2, we have

$$\mathbf{f}_1 = -k_s(\mathbf{x}_1 - \mathbf{x}_2) - k_d(\mathbf{v}_1 - \mathbf{v}_2), \quad \mathbf{f}_2 = k_s(\mathbf{x}_1 - \mathbf{x}_2) + k_d(\mathbf{v}_1 - \mathbf{v}_2),$$

which is just the force law for a damped zero-rest-length spring.

As another example, we use the behavior function

$$C = |\mathbf{l}| - r,$$

where $\mathbf{l} = \mathbf{x}_1 - \mathbf{x}_2$, which says the two points should be distance r apart. Its derivative w.r.t. $\mathbf{l}$ is

$$\frac{\partial C}{\partial \mathbf{l}} = \frac{\mathbf{l}}{|\mathbf{l}|},$$

a unit vector in the direction of $\mathbf{l}$. Then, since $\mathbf{l} = \mathbf{x}_1 - \mathbf{x}_2$,

$$\frac{\partial C}{\partial \mathbf{x}_1} = \frac{\partial C}{\partial \mathbf{l}}, \quad \frac{\partial C}{\partial \mathbf{x}_2} = -\frac{\partial C}{\partial \mathbf{l}}.$$

The time derivative of is

$$\dot{C} = \frac{\mathbf{l} \cdot \dot{\mathbf{l}}}{|\mathbf{l}|} = \frac{\mathbf{l} \cdot (\mathbf{v}_1 - \mathbf{v}_2)}{|\mathbf{l}|}.$$

These expressions are then substituted into the general expression of equation 2 to get the forces. You should verify that this produces the damped spring force of equation 1.

7 Particle/Plane Collisions and Contact

The general collision and contact problem is difficult, to say the least. Later in the course we will examine rigid body collision and contact. Here we only consider, in bare bones form, the simplest case of particles colliding with a plane (e.g. the ground or a wall.) Even these simple collision models can add significant interest to an interactive simulation.

7.1 Detection

There are two parts to the collision problem: detecting collisions, and responding to them. Although general collision detection is hard, particle/plane collision detection is trivial. If $\mathbf{P}$ is a point on the plane, and $\mathbf{N}$ is a normal, pointing *inside* (i.e. on the legal side of the barrier,) then we need only test the sign of $(\mathbf{X} - \mathbf{P}) \cdot \mathbf{N}$ to detect a collision of point $\mathbf{X}$ with the plane. A value greater than zero means it's inside, less than zero means it's outside (where it isn't allowed to be) and zero means it's in contact.

If after an ODE step a collision is detected, the *right* thing to do is to solve (perhaps by linear interpolation between the old and new positions) for the instant of contact, and roll back the whole system to that time. A less accurate but easier alternative is just to displace the point that has collided.

7.2 Response

To describe collision response, we need to partition velocity and force vectors into two orthogonal components, one normal to the collision surface, and the other parallel to it. If $\mathbf{N}$ is the normal to the collision plane, then the *normal component* of a vector $\mathbf{x}$ is $\mathbf{x}_n = (\mathbf{N} \cdot \mathbf{x})\mathbf{x}$, and the *tangential component* is $\mathbf{x}_t = \mathbf{x} - \mathbf{x}_n$.

The simplest collision to consider is an elastic collision without friction. Here, the normal component of the particle's velocity is negated, whereafter the particle goes its merry way. In an inelastic collision, the normal velocity component is instead multiplied by $-r$, where r is a constant between zero and one, called the *coefficient of restitution*. At $r = 0$, the particle doesn't bounce at all, and $r = .9$ is a superball.

7.3 Contact

If a particle is on the collision surface, with zero normal velocity, then it is in *contact*. If a particle is pushed *into* the contact plane ($\mathbf{N} \cdot \mathbf{f} < 0$) a *contact force* $\mathbf{f}_c = -(\mathbf{N} \cdot \mathbf{f})\mathbf{f}$ is exerted, exactly canceling

the normal component of f. However, if the applied force points *away* from the contact plane, no contact force is exerted (unless the surface is sticky,) the particle begins to accelerate away from the surface, and contact is broken.

In the very simplest linear friction model, the frictional force is $-k_f(-\mathbf{f} \cdot \mathbf{N})\mathbf{v}_t$, a drag force that acts in the tangential direction, with magnitude proportional to the normal force. To model a perfectly non-slippery surface, $\mathbf{v}_t$ is simply zeroed.

References

[1] R.W Hocknew and J.W. Eastwood. *Computer Simulation Using Particles*. Adam Hilger, New York, 1988.

[2] Gavin S. P. Miller. The motion dynamics of snakes and worms. *Computer Graphics*, 22:169–178, 1988.

[3] Andrew Witkin, Kurt Fleischer, and Alan Barr. Energy constraints on parameterized models. *Computer Graphics*, 21(4):225–232, July 1987.

Physically Based Modeling
Implicit Methods for Differential Equations

David Baraff
Pixar Animation Studios

Implicit Methods for Differential Equations

David Baraff
Pixar Animation Studios

1 Implicit Methods

The methods we have looked at for solving differential equations in the first section of these notes (Euler's method, the midpoint method) are all called "explicit" methods for solving ODE's. However, sometimes an ODE can become "stiff," in which case explicit methods don't do a very good job of solving them. Whenever possible, it is desirable to change your problem formulation so that you don't have to solve a stiff ODE. Sometimes however that's not possible, and you have just have to be able to solve stiff ODE's. If that's the case, you'll usually have to use an ODE solution method which is "implicit."

2 Example Stiff ODE

First, what is the meaning and cause of stiff equations? Lets consider an example that arises frequently in dynamics. Suppose that we have a particle, with position $(x(t), y(t))$, and suppose that we want the y-coordinate to always be zero. One way of doing this is to add a component $-ky(t)$ to $\dot{y}(t)$ where k is a large positive constant. If k is large enough, then the particle will never move too far away from $y(t) = 0$, since the $-ky(t)$ term always brings $y(t)$ back towards zero. However, lets assume that there is no restriction on the x-coordinate, and that we want a user to be able to pull the particle arbitrarily along the x-axis. So lets assume that over some time interval our differential equation is simply

$$\dot{\mathbf{X}}(t) = \frac{d}{dt}\begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} -x(t) \\ -ky(t) \end{pmatrix}. \tag{2–1}$$

(We'll also assume that the particle doesn't start exactly with $y_0 = 0$.) What's happening here is that the particle is strongly attracted to the line $y = 0$, and less strongly towards $x = 0$. If we solve the ODE far enough forward in time, we expect the particle's location to converge towards $(0, 0)$ and then stay there once it arrives.

Now suppose we use Euler's method to solve the equation. If we take a step of size h, we get

$$\mathbf{X}_{new} = \mathbf{X}_0 + h\dot{\mathbf{X}}(t_0) = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + h\begin{pmatrix} -x_0 \\ -ky_0 \end{pmatrix}.$$

This yields

$$\mathbf{X}_{new} = \begin{pmatrix} x_0 - hx_0 \\ y_0 - hky_0 \end{pmatrix} = \begin{pmatrix} (1-h)x_0 \\ (1-hk)y_0 \end{pmatrix}.$$

If we look at the y component of this equation, we see that if $|1 - hk| > 1$ then the y_{new} we compute will have an absolute value which is larger than $|y_0|$. In other words, if $|1 - hk| > 1$, Euler's method will not converge to an answer: each step will result in a value of y_{new} which is larger than the last. Technically, Euler's method is unstable for $|1 - hk| > 1$. Thus, we better have $1 - hk > -1$ or $hk < 2$ if we hope to converge. The largest step we can hope to take is less than $2/k$.

Now, if k is a large number, we'll have to take very small steps. This means that the particle slides towards $(0, 0)$ excruciatingly slowly. Even though the particle may nearly satisfy $y_0 = 0$, we have to take such small steps that the particles' progress along the x-axis is pretty much nonexistent. That's the embodiment of a stiff ODE. In this case, the stiffness arises from making k very large in order to keep the particle close to the line $y = 0$. Later on, when we connect particles with second-order dynamics together with springs, we'll experience exactly the same effect: stiff ODE's. Even if we use a more sophisticated explicit method such as fourth-order Runge-Kutta, we may do a little better in the size of our steps, but we'll still have major problems.

Now as we said above, the name of the game is to pose your dynamics problems so that you don't experience stiff ODE's. However, when you can't, you'll have to turn towards an implicit solution method. The method we'll show below is the simplest of the implicit methods, and its based on taking an Euler step "backwards."

3 Solving Stiff ODE's

Given a differential equation

$$\frac{d}{dt}\mathbf{X}(t) = \boldsymbol{f}(\mathbf{X}(t)),$$

the explicit Euler update would be $\mathbf{X}_{new} = \mathbf{X}_0 + h\boldsymbol{f}(\mathbf{X}(t_0))$, to advance the system forward h in time. For a stiff problem though, we change the update to instead be

$$\mathbf{X}_{new} = \mathbf{X}_0 + h\boldsymbol{f}(\mathbf{X}_{new}) \tag{3–1}$$

That is, we're going to evaluate $\boldsymbol{f}$ at the point we're aiming at, rather than where we came from. (If you think about reversing the world and running everything backwards, the above equation makes perfect sense. Then the equation says "if you were at $\mathbf{X}_{new}$, and took a step $-h\boldsymbol{f}(\mathbf{X}_{new})$, you'd end up at $\mathbf{X}_0$." So if your differential equation represents a system that is reversible in time, this step makes sense. It's just finding a point $\mathbf{X}_{new}$ such that if you ran time backwards, you'd end up at $\mathbf{X}_0$.) So, we're looking for a $\mathbf{X}_{new}$ such that f, evaluated there, times h, points directly back at where we came from. Unfortunately, we can't in general solve for $\mathbf{X}_{new}$, unless f happens to be a linear function.

To cope with this, we'll replace $\boldsymbol{f}(\mathbf{X}_{new})$ with a linear approximation, again based on f's Taylor series. Lets define $\Delta\mathbf{X}$ by $\Delta\mathbf{X} = \mathbf{X}_{new} - \mathbf{X}_0$. Using this, we rewrite equation (3–1) as

$$\mathbf{X}_0 + \Delta\mathbf{X} = \mathbf{X}_0 + h\boldsymbol{f}(\mathbf{X}_0 + \Delta\mathbf{X}).$$

or just

$$\Delta\mathbf{X} = h\boldsymbol{f}(\mathbf{X}_0 + \Delta\mathbf{X}).$$

Next, lets approximate $\boldsymbol{f}(\mathbf{X}_0 + \Delta\mathbf{X})$ by

$$\boldsymbol{f}(\mathbf{X}_0) + f'(\mathbf{X}_0)\Delta\mathbf{X}.$$

(Note that since $\boldsymbol{f}(\mathbf{X}_0)$ is a vector, the derivative $f'(\mathbf{X}_0)$ is a matrix.) Using this approximation, we can approximate $\Delta\mathbf{X}$ with

$$\Delta\mathbf{X} = h\left(\boldsymbol{f}(\mathbf{X}_0) + f'(\mathbf{X}_0)\Delta\mathbf{X}\right).$$

or

$$\Delta\mathbf{X} - hf'(\mathbf{X}_0)\Delta\mathbf{X} = h\boldsymbol{f}(\mathbf{X}_0)$$

Rewriting this as

$$\left(\frac{1}{h}\mathbf{I} - f'(\mathbf{X}_0)\right)\Delta\mathbf{X} = \boldsymbol{f}(\mathbf{X}_0),$$

where $\mathbf{I}$ is the identity matrix, we can solve for $\Delta\mathbf{X}$ as

$$\Delta\mathbf{X} = \left(\frac{1}{h}\mathbf{I} - f'(\mathbf{X}_0)\right)^{-1}\boldsymbol{f}(\mathbf{X}_0) \tag{3–2}$$

Computing $\mathbf{X}_{new} = \mathbf{X}_0 + \Delta\mathbf{X}$ is clearly more work than using an explicit method, since we have to solve a linear system at each step. While this would seem a serious weakness, computationally speaking, don't despair (yet). For many types of problems, the matrix f' will be sparse—for example, if we are simulating a spring-lattice, f' will have a structure which matches the connectivity of the particles. (For a discussion of sparsity and solution techniques, see Baraff and Witkin [1]. Basic material in Press *et al*. [2] will also prove useful.) As a result, it is usually possible to solve equation (3–2) in linear time (i.e. time proportional to the dimension of $\mathbf{X}$). In such cases, the payoff is dramatic: we can usually take considerably large timesteps without losing stability (i.e. without divergence, as happens with the explicit case if the stepsize is too large). The time taken in solving each linear system is thus more than offset by the fact that our timesteps are often orders of magnitude bigger than we could manage using an explicit method. (Of course, the code needed to do all this is much more complicated than in the explicit case; like we said, make your problems un-stiff if you can, and if not, pay the price.)

Lets apply the implicit method to equation (2–1). We have that $\boldsymbol{f}(\mathbf{X}(t))$ is

$$\boldsymbol{f}(\mathbf{X}(t)) = \begin{pmatrix} -x(t) \\ -ky(t) \end{pmatrix}.$$

Differentiating with respect to $\mathbf{X}$ yields

$$f'(\mathbf{X}(t)) = \frac{\partial}{\partial\mathbf{X}}\boldsymbol{f}(\mathbf{X}(t)) = \begin{pmatrix} -1 & 0 \\ 0 & -k \end{pmatrix}$$

Then the matrix $\frac{1}{h}\mathbf{I} - f'(\mathbf{X}_0)$ is

$$\begin{pmatrix} \frac{1}{h}+1 & 0 \\ 0 & \frac{1}{h}+k \end{pmatrix} = \begin{pmatrix} \frac{1+h}{h} & 0 \\ 0 & \frac{1+kh}{h} \end{pmatrix}$$

Inverting this matrix, and multiplying by $\boldsymbol{f}(\mathbf{X}_0)$ yields

$$\begin{aligned}\Delta\mathbf{X} &= \begin{pmatrix} \frac{1+h}{h} & 0 \\ 0 & \frac{1+kh}{h} \end{pmatrix}^{-1} \begin{pmatrix} -x_0 \\ -ky_0 \end{pmatrix} \\ &= \begin{pmatrix} \frac{h}{h+1} & 0 \\ 0 & \frac{h}{1+kh} \end{pmatrix} \begin{pmatrix} -x_0 \\ -ky_0 \end{pmatrix} \\ &= -\begin{pmatrix} \dfrac{h}{h+1}x_0 \\ \dfrac{h}{1+kh}ky_0 \end{pmatrix}\end{aligned}$$

What is the limit on the stepsize in this case? The answer is: there is no limit! In this case, if we let h grow to infinity, we get

$$\lim_{h\to\infty} \Delta\mathbf{X} = \lim_{h\to\infty} -\begin{pmatrix} \dfrac{h}{h+1}x_0 \\ \dfrac{h}{1+kh}ky_0 \end{pmatrix} = -\begin{pmatrix} x_0 \\ \dfrac{1}{k}ky_0 \end{pmatrix} = -\begin{pmatrix} x_0 \\ y_0 \end{pmatrix}.$$

This means that we achieve $\mathbf{X}_{new} = \mathbf{X}_0 + (-\mathbf{X}_0) = \mathbf{0}$ in a single step! For a general stiff ODE, we won't be able to take steps of arbitrary size, but we will be able to take much larger steps using an implicit method than using an explicit method. The extra cost of solving a linear equation is more than made up by the time saved by taking large timesteps.

4 Solving Second-Order Equations

Most dynamics problems are written in terms of a second-order differential equation:

$$\ddot{\mathbf{x}}(t) = \boldsymbol{f}(\mathbf{x}(t), \dot{\mathbf{x}}(t)). \tag{4–1}$$

This equation is easily converted to a first-order differential equation by adding new variables. If we define $\mathbf{v} = \dot{\mathbf{x}}$, then we can rewrite equation (4–1) as

$$\frac{d}{dt}\begin{pmatrix} \mathbf{x}(t) \\ \mathbf{v}(t) \end{pmatrix} = \begin{pmatrix} \mathbf{v}(t) \\ \boldsymbol{f}(\mathbf{x}(t), \mathbf{v}(t)) \end{pmatrix} \tag{4–2}$$

which is a first-order system. However, applying the backward Euler method to equation (4–2) results in a linear system of size $2n \times 2n$ where n is the dimension of $\mathbf{x}$. A fairly simple transformation allows us to reduce the size of the problem to solving an $n \times n$ linear system instead. It is important to note that both the $2n \times 2n$ and $n \times n$ systems will have the same degree of sparsity, so solving the smaller system will be faster.

The $n \times n$ system that needs to be solved is derived as follows. Let us simplify notation by writing $\mathbf{x_0} = \mathbf{x}(t_0)$ and $\mathbf{v_0} = \mathbf{v}(t_0)$. We also define $\Delta\mathbf{x} = \mathbf{x}(t_0 + h) - \mathbf{x}(t_0)$ and $\Delta\mathbf{v} = \mathbf{v}(t_0 + h) - \mathbf{v}(t_0)$. The backward Euler update, applied to equation (4–2), yields

$$\begin{pmatrix} \Delta\mathbf{x} \\ \Delta\mathbf{v} \end{pmatrix} = h\begin{pmatrix} \mathbf{v_0} + \Delta\mathbf{v} \\ \boldsymbol{f}(\mathbf{x_0} + \Delta\mathbf{x}, \mathbf{v_0} + \Delta\mathbf{v}) \end{pmatrix}. \tag{4–3}$$

Applying a Taylor series expansion to $\boldsymbol{f}$—which in this context is a function of both $\mathbf{x}$ and $\mathbf{v}$—yields the first-order approximation

$$\boldsymbol{f}(\mathbf{x_0} + \Delta\mathbf{x}, \mathbf{v_0} + \Delta\mathbf{v}) = \mathbf{f_0} + \frac{\partial \boldsymbol{f}}{\partial \mathbf{x}}\Delta\mathbf{x} + \frac{\partial \boldsymbol{f}}{\partial \mathbf{v}}\Delta\mathbf{v}.$$

In this equation, the derivative $\partial\boldsymbol{f}/\partial\mathbf{x}$ is evaluated for the state $(\mathbf{x_0}, \mathbf{v_0})$ and similarly for $\partial\boldsymbol{f}/\partial\mathbf{v}$. Substituting this approximation into equation (4–3) yields the linear system

$$\begin{pmatrix} \Delta\mathbf{x} \\ \Delta\mathbf{v} \end{pmatrix} = h \begin{pmatrix} \mathbf{v_0} + \Delta\mathbf{v} \\ (\mathbf{f_0} + \frac{\partial \boldsymbol{f}}{\partial \mathbf{x}}\Delta\mathbf{x} + \frac{\partial \boldsymbol{f}}{\partial \mathbf{v}}\Delta\mathbf{v}) \end{pmatrix}. \tag{4–4}$$

Taking the bottom row of equation (4–4) and substituting $\Delta\mathbf{x} = h(\mathbf{v_0} + \Delta\mathbf{v})$ yields

$$\Delta\mathbf{v} = h\left(\mathbf{f_0} + \frac{\partial \boldsymbol{f}}{\partial \mathbf{x}}h(\mathbf{v_0} + \Delta\mathbf{v}) + \frac{\partial \boldsymbol{f}}{\partial \mathbf{v}}\Delta\mathbf{v}\right).$$

Letting $\mathbf{I}$ denote the identity matrix, and regrouping, we obtain

$$\left(\mathbf{I} - h\frac{\partial \boldsymbol{f}}{\partial \mathbf{v}} - h^2\frac{\partial \boldsymbol{f}}{\partial \mathbf{x}}\right)\Delta\mathbf{v} = h\left(\mathbf{f_0} + h\frac{\partial \boldsymbol{f}}{\partial \mathbf{x}}\mathbf{v_0}\right) \tag{4–5}$$

which we then solve for $\Delta\mathbf{v}$. Given $\Delta\mathbf{v}$, we trivially compute $\Delta\mathbf{x} = h(\mathbf{v_0} + \Delta\mathbf{v})$.

The above assumes that the function $\boldsymbol{f}$ has no direct dependence on time; in the case that $\boldsymbol{f}$ varies directly with time (for example, if $\boldsymbol{f}$ describes time-varying external forces, or references moving points or coordinate frames that are not variables of $\mathbf{x}$) then equation (4–5) needs an additional term to account for this dependence:

$$\left(\mathbf{I} - h\frac{\partial \boldsymbol{f}}{\partial \mathbf{v}} - h^2\frac{\partial \boldsymbol{f}}{\partial \mathbf{x}}\right)\Delta\mathbf{v} = h\left(\mathbf{f_0} + h\frac{\partial \boldsymbol{f}}{\partial \mathbf{x}}\mathbf{v_0} + \frac{\partial \boldsymbol{f}}{\partial t}\right) \tag{4–6}$$

References

[1] D. Baraff and A. Witkin. Large steps in cloth simulation. *Computer Graphics* (*Proc. SIGGRAPH*), 1998.

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

Physically Based Modeling
Constrained Dynamics

Andrew Witkin
Pixar Animation Studios

Constrained Dynamics

Andrew Witkin
Pixar Animation Studios

1 Beyond penalty methods

The idea of constrained particle dynamics is that our description of the system includes not only particles and forces, but restrictions on the way the particles are permitted to move. For example, we might constrain a particle to move along a specified curve, or require two particles to remain a specified distance apart. The problem of constrained dynamics is to make the particles obey Newton's laws, and at the same time obey the geometric constraints.

As we learned earlier, energy functions provide a sloppy, approximate constraint mechanism. A spring with rest length r makes the particles it connects "want" to be distance r apart. However, the spring force competes with all other forces acting on the particles—gravity, other springs, forces applied by the user, etc. The constraint can only win this tug-of-war if its spring constant is large enough to overpower all competing influences, so that very small displacements induce large restoring forces. As we saw in the last section, this is really no solution because it gives rise to stiff differential equations which are all but numerically intractible. The use of extra energy terms to impose constraints is known as the *penalty method*. If we want both accurate constraints and numerical tractibility, then penalty methods will not fill the bill.

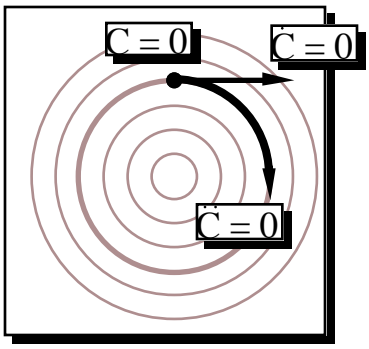
Penalty constraints work, to the extent they do, because the restoring forces cancel applied forces that would otherwise break the constraints. The fundamental difficulty with penalty constraints is that the applied forces and restoring forces communicate only indirectly, through displacements. In effect, the displacements produced by applied forces act as signals that tell the constraint what restoring force is required. This is not a good communication mechanism because it is impossible to acheive accuracy without stiffness.

The basic approach to avoiding this problem is to directly calculate the forces required to maintain the constraints, rather than relying on displacements and restoring forces to do the job. The job of these *constraint forces* is to cancel just those parts of the applied forces that act against the constraints. Since forces influence acceleration, this means specifically that the constraint forces must convert the particles' accelerations into "legal" accelerations that are consistent with the constraints.

2 A bead on a wire

We introduce the approach using the simple example of a $2D$ particle constrained to move on the unit circle. We can express the constraint by writing a scalar behavior function, as we did in Chapter C to create energy functions,

$$C(\mathbf{x}) = \frac{1}{2}(\mathbf{x} \cdot \mathbf{x} - 1), \tag{1}$$



Point-on-circle constraint:

$$C = \tfrac{1}{2}\left(\mathbf{x}\cdot\mathbf{x} - 1\right)$$

$C = 0$ *legal position*
$\dot{C} = 0$ *legal velocity*
$\ddot{C} = 0$ *legal acceleration*

Add in a constraint force that ensures legal acceleration.

Maintaining Constraints Differentially

Figure 1: If the initial position and velocity are consistent with the constraints, then the constraint can be maintained by ensuring that the acceleration is always legal thereafter.

so that the legal positions of $\mathbf{x}$ are all those that satisfy $C(\mathbf{x}) = 0$. The function C is an *implicit function* for the constraint. If $\mathbf{x}$ is a legal position, then the legal *velocities* are all those that satisfy

$$\dot{C} = \mathbf{x} \cdot \dot{\mathbf{x}} = 0. \tag{2}$$

In turn, the legal *accelerations* are all those that satisfy

$$\ddot{C} = \ddot{\mathbf{x}} \cdot \mathbf{x} + \dot{\mathbf{x}} \cdot \dot{\mathbf{x}} = 0. \tag{3}$$

If we start out with a legal position and velocity, then to maintain the constraint, in principal, we need only ensure that equation 3 is satisfied at every instant thereafter. See figure 1 The particle's acceleration is

$$\ddot{\mathbf{x}} = \frac{\mathbf{f} + \hat{\mathbf{f}}}{m}, \tag{4}$$

where $\mathbf{f}$ is the given applied force, and $\hat{\mathbf{f}}$ is the as yet unknown constraint force. Substituting for $\ddot{\mathbf{x}}$ in equation 3 gives

$$\ddot{C} = \frac{\mathbf{f} + \hat{\mathbf{f}}}{m} \cdot \mathbf{x} + \dot{\mathbf{x}} \cdot \dot{\mathbf{x}} = 0, \tag{5}$$

or

$$\hat{\mathbf{f}} \cdot \mathbf{x} = -\mathbf{f} \cdot \mathbf{x} - m\dot{\mathbf{x}} \cdot \dot{\mathbf{x}}. \tag{6}$$

2.1 The principal of virtual work

We have only one equation and two unknowns—the two components of $\hat{\mathbf{f}}$—so we cannot solve for the constraint force without an additional condition. We get that condition by requiring that

the constraint force never add energy to nor remove energy from the system, i.e. the constraint is passive and lossless. The kinetic energy is

$$T = \frac{m}{2}\dot{\mathbf{x}} \cdot \dot{\mathbf{x}},$$

and its time derivative is

$$\dot{T} = m\ddot{\mathbf{x}} \cdot \dot{\mathbf{x}} = m\mathbf{f} \cdot \dot{\mathbf{x}} + m\hat{\mathbf{f}} \cdot \dot{\mathbf{x}},$$

which is the *work* done by $\mathbf{f}$ and $\hat{\mathbf{f}}$. Requiring that the constraint not change the energy means that the last term must be zero, i.e. that the constraint force does no work. A subtle point is that we are enjoining the constraint force from *ever* doing work, rather than saying that it happens not to be at the moment. We therefore require thet $\hat{\mathbf{f}} \cdot \dot{\mathbf{x}}$ vanish for *every legal* $\dot{\mathbf{x}}$, i.e.

$$\hat{\mathbf{f}} \cdot \dot{\mathbf{x}} = 0, \forall \dot{\mathbf{x}} \mid \mathbf{x} \cdot \dot{\mathbf{x}} = 0.$$

This condition simply states that $\hat{\mathbf{f}}$ must point in the direction of $\mathbf{x}$, so we can rewrite the constraint force as

$$\hat{\mathbf{f}} = \lambda \mathbf{x},$$

where λ is an unknown scalar. Substituting for $\hat{\mathbf{f}}$ in equation 6, and solving for λ gives

$$\lambda = \frac{-\mathbf{f} \cdot \mathbf{x} - m\dot{\mathbf{x}} \cdot \dot{\mathbf{x}}}{\mathbf{x} \cdot \mathbf{x}}. \tag{7}$$

Having solved for λ, we calculate $\hat{\mathbf{f}} = \lambda \mathbf{x}$, then $\ddot{\mathbf{x}} = (\hat{\mathbf{f}} + \mathbf{f})/m$, and proceed with the simulation in the usual way.

In its general form, the condition we imposed on $\hat{\mathbf{f}}$ is known as *the principal of virtual work*.[2] See figure 2 for an illustration.

2.2 Feedback

If we were solving the differential equation exactly, this procedure would keep the particle exactly on the unit circle, provided we began with valid initial conditions. In practice, we know that numerical solutions to ODE's drift. We must add an extra feedback term to prevent this numerical drift from accumulating, turning the circle into an outward spiral. The feedback term can be just a damped spring force, pulling the particle back onto a unit circle. The feedback force needs to be added in *after* the constraint force calculation, or else the constraint force will dutifully cancel it! We will discuss feedback in more detail in the next section.

2.3 Geometric Interpretation

When the system is at rest, the constraint force given in equation 7 reduces to

$$\hat{\mathbf{f}} = -\frac{\mathbf{f} \cdot \mathbf{x}}{\mathbf{x} \cdot \mathbf{x}}\mathbf{x}, \tag{8}$$

which has clear geometric interpretation as the vector that orthogonally projects $\mathbf{f}$ onto the circle's tangent. This interpretation makes intuitive sense because the force component that is removed by this projection is the component that points out of the legal motion direction. The orthogonality of the projection also makes sense, because it ensures that the particle will not experience gratuitous accelerations in the allowed direction of motion.

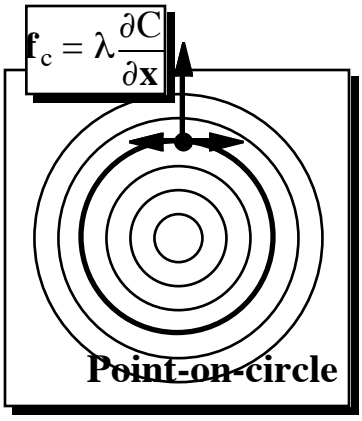


Figure 2: In the case of a point-on-circle constraint, the principle of virtual work simply requires the constraint force to lie in a direction normal to the circle.

When $\dot{\mathbf{x}}$ is nonzero, we unfortunately lose this simple geometric picture, but we can still interpret the addition of the constraint force as a projection. Rather than a force projection, it is the projection of the *acceleration* onto the set of legal accelerations. The velocity-dependent term in equation 7 ensures that the curvature of the particle's trajectory matches that of the circle. some effort, we could try to regain the geometric picture by visualizing a projection in *phase space,* but this is hardly worth the trouble.

3 The general case

In the last section we derived the constraint force expression for a single particle subject to a single scalar constraint. Our goal in this section is to extend this special case to the general one of a whole system of particles, collectively subjected to a number of constraints. The derivation follows the more detailed one presented in [5].

The key to making this a managable task is to adopt a uniform, monolithic view, much as we do in solving ODEs. Rather than considering each particle separately, we lump their positions into a single *state vector,* which we will call **q**. Unlike the phase-space vector that we hand to the solver, this one contains positions only, not velocities,so, in $3D$, it has length $3n$.

To collapse all the particles' equations of motion into one global equation, we next define a *mass matrix,* **M**, whose diagonal elements are the particles' masses, and whose off-diagonal elements are zero. The diagonal mass matrix for n $3D$ points is a $3n \times 3n$ matrix whose diagonal elements are $[m_1, m_1, m_1, m_2, m_2, m_2, \ldots, m_n, m_n, m_n]$. implementation, a diagonal matrix may be represented as a vector. Multiplication of a vector by the matrix is then just element-by-element multiplication. The *inverse* of a diagonal matrix is just the element-by-element reciprocal.

Finally, we concatenate the forces on all the particles, just as we do the positions, to create a global force vector, which we denote by $\mathbf{Q}$. Now we can write the global equation governing the particle system as

$$\ddot{\mathbf{q}} = \mathbf{W}\mathbf{Q},$$

where $\mathbf{W}$ is the inverse of $\mathbf{M}$.

We will also use global notation for the constraints, concatenating all the scalar constraint functions to form a single vector function $\mathbf{C}(\mathbf{q})$. If we have n $3D$ particles, subject to m scalar constraints, then the output of this global constraint function is an m-vector, and its input is a $3n$-vector.

At this point, you may well be wondering how all this global notation will ever actually apply to a real network of particles and constraints. This is an example of just the same kind of duality that we encountered in applying ODE solvers to mass-and-spring models. On the one hand, we want to build particle-and-constraint models as networks of distinct interacting objects. On the other, we want to allow the code that calculates constraint forces to act as if the system on which it operates really were a structureless monolith, just as an ODE solver does. Soon, we will show very concretely how this dual view can be maintained.

As in the point-on-circle example, we assume that the configuration $\mathbf{q}$ and the velocity $\dot{\mathbf{q}}$ are both initially legal, i.e. that $\mathbf{C} = \dot{\mathbf{C}} = 0$. Then our problem is to solve for a constraint force $\hat{\mathbf{Q}}$ that, added to the applied force $\mathbf{Q}$, guarantees that $\ddot{\mathbf{C}} = 0$.

To do this, we will need to take derivatives of $\mathbf{C}$. In the previous section, we had a specific algebraic expression for the constraint function, so we were able to derive expressions for their derivatives as well. Now, since we are regarding $\mathbf{C}$ as an anonymous function of state, we will be writing derivatives generically, using expressions such as $\frac{\partial \mathbf{C}}{\partial \mathbf{q}}$. To keep things down to earth, you should think of expressions such as these, as well as $\mathbf{C}$ itself, as things we are able to evaluate numerically by invoking functions. [1]

By the chain rule,

$$\dot{\mathbf{C}} = \frac{\partial \mathbf{C}}{\partial \mathbf{q}}\dot{\mathbf{q}}.$$

The matrix $\partial\mathbf{C}/\partial\mathbf{q}$ is called the *Jacobian* of $\mathbf{C}$. We will denote it henceforward by $\mathbf{J}$. Differentiating again w.r.t. time gives

$$\ddot{\mathbf{C}} = \dot{\mathbf{J}}\dot{\mathbf{q}} + \mathbf{J}\ddot{\mathbf{q}}.$$

The quantity $\dot{\mathbf{J}}$, the time derivative of the Jacobian, might be a little puzzling. By the chain rule we could write it as $\dot{\mathbf{J}} = \partial\mathbf{J}/\partial\mathbf{q}\dot{\mathbf{q}}$. However, taking the derivative of a matrix w.r.t. a vector yields a rank 3 tensor (essentially, a $3D$ array). We can avoid introducing this new kind of object by writing, equivalently $\dot{\mathbf{J}} = \partial\dot{\mathbf{C}}/\partial\mathbf{q}$. Assuming we have an expression for $\dot{\mathbf{C}}$, this entails only differentiating a vector expression w.r.t. a vector, which is less menacing.

Next, we use the system's equations of motion to replace $\ddot{\mathbf{q}}$ by a force expression, giving

$$\ddot{\mathbf{C}} = \dot{\mathbf{J}}\dot{\mathbf{q}} + \mathbf{J}\mathbf{W}(\mathbf{Q} + \hat{\mathbf{Q}}).$$

Setting $\ddot{\mathbf{C}}$ to zero and re-arranging gives

$$\mathbf{J}\mathbf{W}\hat{\mathbf{Q}} = -\dot{\mathbf{J}}\dot{\mathbf{q}} - \mathbf{J}\mathbf{W}\mathbf{Q}, \tag{9}$$

[1] Speaking of derivatives, it is important to understand what a quantity such as $\partial\mathbf{C}/\partial\mathbf{q}$ is. Since both $\mathbf{C}$ and $\mathbf{q}$ are vectors, the derivative of one with respect to the other is a matrix, obtained by taking the scalar derivative of each component of $\mathbf{C}$ w.r.t. each component of $\mathbf{q}$.

which is the counterpart, in general form, to equation 7. As in the point-on-circle example, we have more unknowns than equations, and once again we introduce the principle of virtual work. The legal velocities (i.e. the ones that don't change $\mathbf{C}$) are all the ones that satisfy $\mathbf{J}\dot{\mathbf{x}} = 0$. To ensure that the constraint force does no work, we therefore require that

$$\hat{\mathbf{Q}} \cdot \dot{\mathbf{x}} = 0, \quad \forall \dot{\mathbf{x}} \mid \mathbf{J}\dot{\mathbf{x}} = 0.$$

All and only vectors $\hat{\mathbf{Q}}$ that satisfy this requirement can be expressed in the form

$$\hat{\mathbf{Q}} = \mathbf{J}^T \lambda,$$

where λ is a vector with the dimension of $\mathbf{C}$.

To understand what this expression means, it helps to regard the matrix $\mathbf{J}$ as a collection of vectors, each of which is the gradient of one of the scalar constraint functions comprising $\mathbf{C}$. Since our fundamental requirement is that $\mathbf{C} = 0$, these gradients are normals to the constraint hypersurfaces, representing the state-space directions in which the system is *not* permitted to move. The vectors that have the form $\mathbf{J}^T \lambda$ are the linear combinations of these gradient vectors, and hence span exactly the set of *prohibited* directions. Restricting the constraint force to this set ensures that its dot product with any *legal* displacement of the system will be zero, which is exactly what the principle of virtual work demands.

In matrix parlance, the set of vectors $\mathbf{J}^T \lambda$ is known as the *null space complement* of $\mathbf{J}$. The *null space* of $\mathbf{J}$ is the set of vectors $\mathbf{v}$ that satisfy $\mathbf{J}\mathbf{v} = 0$. The null space vectors are the *legal* displacements, while the null space complement vectors are the *prohibited* ones.

The components of λ are known as *Lagrange multipliers*. These quantities, which determine how much of each constraint gradient is mixed into the constraint force, are the unknowns for which we must solve. To do so, we replace $\hat{\mathbf{Q}}$ by $\mathbf{J}^T \lambda$ in equation 9, which gives

$$\mathbf{J}\mathbf{W}\mathbf{J}^T \lambda = -\dot{\mathbf{J}}\dot{\mathbf{q}} - \mathbf{J}\mathbf{W}\mathbf{Q}, \tag{10}$$

which is a matrix equation in which all but λ are known. The matrix $\mathbf{J}\mathbf{W}\mathbf{J}^T$ is a square matrix with the dimensions of $\mathbf{C}$. Once λ is obtained, it is multiplied by $\mathbf{J}^T$ to obtain $\hat{\mathbf{Q}}$, which is added to the applied force before calculating acceleration.

We already noted the need for a feedback term to prevent the accumulation of numerical drift. This term can be incorporated directly into the constraint force calculation. Instead of solving for $\ddot{\mathbf{C}} = 0$, as we did above, we solve for

$$\ddot{\mathbf{C}} = -k_s \mathbf{C} - k_d \dot{\mathbf{C}},$$

where k_s and k_d are spring and damping constants. By adding this term, we make the constraint force perform the extra function of returning, with damping, to a valid state should drift occur. The final constraint force equation, with feedback, is

$$\mathbf{J}\mathbf{W}\mathbf{J}^T \lambda = -\dot{\mathbf{J}}\dot{\mathbf{q}} - \mathbf{J}\mathbf{W}\mathbf{Q} - k_s \mathbf{C} - k_d \dot{\mathbf{C}}. \tag{11}$$

The values assigned to k_s and k_d are not critical, since this term only plays the relatively undemanding role of absorbing drift. See [1, 3, 5] for further discussion.

4 Tinkertoys: Implementing Constrained Particle Dynamics

The general formula of equation 11 is just a skeleton. To actually simulate anything, a specific constraint function $\mathbf{C}(\mathbf{q})$ must be provided, in a form that lets us evaluate the function itself and its various derivatives. One way to flesh out the skeleton would be to write down an expression for such a function, symbolically take the required derivatives, substitute these expressions into equation 11 and, after simplifying and massaging the resulting mess, write code that performs the numerical evaluations required for a simulation. This is essentially what we did in section 2, but only for a trivially simple example. As an exercise, you might try working through a somewhat more complicated example, say a double pendulum. If you actually do try it, you will probably be able to carry it through to a working implementation, but it will be readily apparent that this derive-and-implement methodology does not scale up!

Instead of hand-deriving and hand-coding models, we want to build models interactively by snapping the pieces together, drawing freely from a set of useful pre-defined constraints, such as distance and point-on-curve constraints. The main problem we must solve to acheive this goal is to evaluate the global matrices and vectors that comprise equation 11. The evaluations must be quick, and also dynamic in the sense that we can freely change the structure of the model on the fly. Naturally, we want constraints to be modular just as forces are.

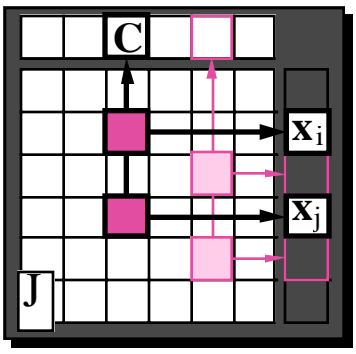
In this section we describe an architecture for constrained particle dynamics simulation that meets these objectives. The approach is to represent individual constraints using objects similar to those that represent forces. Each constraint object is responsible for evaluating the constraint function it represents, and also that functions derivatives. These evaluations produce fragments of the vectors and matrices comprising equation 11. The fragments are then combined dynamically by the constrained particle system.

4.1 The constrained simulation loop

The machinery required to support constraints fits neatly into our basic particle system architecture. From the standpoint of the ODE solver, the main job of the particle system, in both the unconstrained and constrained case, is to perform derivative evaluations. The sequence of steps that the particle system must perform to evaluate the derivative is nearly the same, with one important addition: calculating the constraint force. This is how the extra step fits in:

1. Clear forces: zero each particle's force accumulator.
2. Calculate forces: loop over all force objects, allowing each to add forces to the particles it influences.
3. Calculate constraint forces: On completion of the previous step, each particle's force accumulator contains the total force on that particle. In this step, the global equation 11 is set up and solved, yielding a constraint force on each particle, which is added into the applied force.
4. Calculate the derivative: Divide force by mass to get acceleration, and gather the derivatives into a global vector for the solver.

In this section, we are concerned exclusively with the third step in this sequence.



- **Each constraint contributes one or more *blocks* to the matrix.**
- **Sparsity: many empty blocks.**
- **Modularity: let each constraint compute its own blocks.**

Matrix Block Structure

Figure 3: The block-sparse Jacobian matrix. The constraints are shown above, and the particles along the right side. Each constraint/particle pair corresponds to a block in the matrix. A block is non-zero only if the constraint depends on the particles. Here, we show a binary constraint, such as a distance constraint, that connects two particles. The two shaded blocks are that constraint's only contributions to the sparse Jacobian.

4.2 Block-structured matrices

Each individual constraint contributes a slice to the global constraint vector $\mathbf{C}$, just as each particle contributes a slice to the global state vector $\mathbf{q}$. In addition to a list of particles, and a list of forces, a constrained particle system must also maintain a list of constraints. Evaluating the global vectors such as $\mathbf{C}$ and $\dot{\mathbf{C}}$ is straightforward, assuming that each constraint points to a function that performs its own portion of these evaluations. We simply loop over the constraints, invoking the functions, and placing the results in the global vectors with appropriate offsets. This is essentially the same gather operation that we use for communication with the ODE solver.

The main new ingredient is that we must evaluate global matrices as well as vectors. Whereas constraints and particles each occupy a slice of their respective global vectors, each *constraint-particle pair* occupies a *block* of the global derivative matrix. The vector slice that is "owned" by a constraint can be described by an offset and a length, say i and $ilength$. Similarly, a particle's global station can be described by j and $jlength$. While $jlength$ is always the dimension of the space that the particles live in, $ilength$ may vary from constraint to constraint. The derivative of a constraint with respect to a particle occupies an $ilength \times jlength$ block of the Jacobian matrix $\mathbf{J}$, with originn at position (i, j). See figure 3.

A typical constraint influences at most just a few particles. The value of the constraint function depends on only these particles; its derivative with respect to all other particles is zero. This means that the matrices $\mathbf{J}$ and $\dot{\mathbf{J}}$ are typically very sparse. Of the n blocks per constraint in an n-particle system, a unary constraint contributes only one non-zero block, a binary one two non-zero blocks,

etc. Given this structure, a natural way to represent the sparse matrices is by lists of the non-zero blocks. In this scheme, each matrix block is represented by a structure that specifies the block's origin, (i, j), and dimensions, $(ilength, jlength)$, and that contains an $ilength \times jlength$ float array holding the block's data, e.g.

```
struct{
   int i;
   int j;
   int ilength;
   int jlength;
   float *data;
    };
```

To support constrained particle dynamics using block-sparse matrices, as we will soon see, we must implement only two operations: matrix times vector, and matrix-transpose times vector. Both are simple operations, looping over the matrix blocks, and performing an ordinary matrix multiplication for each block, using the block's i and j as offsets into the destination and source vectors.

4.3 Building the matrices

In addition to holding lists of particles and forces, the constrained particle system will hold a list of constraints, block-sparse matrices to represent $\mathbf{J}$ and $\dot{\mathbf{J}}$, and vectors to hold $\mathbf{C}$, $\dot{\mathbf{C}}$, etc. The structures that represent the constraints may be similar in many respects to the structures we use to represent simple forces, i.e. they point to the particles on which they depend and they point to functions that perform their type-specific operations.

When a constraint is instantiated, matrix blocks must be created for each particle on which the constraint depends, and the blocks must be added to the global matrices. Since the number and shape of blocks involved varies with the constraint type, this initialization may be handled by the constraint in a type-specific way. Thereafter, the constraint must be able to evaluate its portions of the vectors $\mathbf{C}$ and $\dot{\mathbf{C}}$, and of the matrices $\mathbf{J}$ and $\dot{\mathbf{J}}$. The results of the matrix evaluations are placed in the matrix blocks that were created by the constraint on initialization.

All the required global quantities can then be computed simply by looping over the constraints, and invoking the functions that perform these evaluations.

4.4 Solving the linear system

The solution of sparse linear systems is a field unto itself. Of the many available options, we give one that is simple and readily available. A matrix equation of the form $\mathbf{Mx} = b$ may be solved iteratively by finding a vector $\mathbf{x}$ that minimizes $(\mathbf{Mx} - b) \cdot (\mathbf{Mx} - b)$. A conjugate gradient algorithm that solves this problem is given in Numerical Recipes [4], Chapter 2. The conjugate gradient algorithm offers the advantage that it gives a least-squares solution for over-determined systems, and tolerates redundant constraints. The solver takes as arguments two routines which contitute its only access the matrix: vector-times-matrix, and vector-transpose-times matrix. Sparsity is exploited by implementing these routines efficiently. The routine requires $O(n)$ iterations to solve an $\mathbf{n} \times \mathbf{n}$ matrix, and the cost of each iteration is $O(m)$, where m is the number of non-zero entries in the matrix.

The matrix of equation 11 is $\mathbf{JWJ}^T$, where $\mathbf{J}$ is block-sparse and $\mathbf{W}$ is diagonal. We need never actually calculate the matrix. Instead we need only calculate $\mathbf{JWJ}^T\mathbf{x}$, given a vector $\mathbf{x}$. We do this by calculating $\mathbf{J}^T\mathbf{x}$, using the block-sparse matrix-transpose multiply routine described above, then performing an element-by-element multiplication of the result by the vector representing the diagonal $\mathbf{W}$. Finally, the resulting vector is multiplied by $\mathbf{J}$. Since the compound matrix is symmetric, we do not need a separate function for multiplication by the transpose.

Evaluating the right hand side vector of equation 11 is a straightforward application of the block-sparse matrix routines, and standard vector operations.

Finally, once the linear system has been solved, the vector λ is multiplied by $\mathbf{J}^T$ to produce the global constraint force vector $\hat{\mathbf{Q}}$, which is then scattered into the particles' force accumulator.

4.5 Summary

To introduce constraints into a particle system simulation, we add the additional step of constraint force calculation to the derivative evaluation operation. After the ordinary applied forces have been calculated, but before computing accelerations, we perform the following steps:

- Loop over the constraints, letting each evaluate its own portion of $\mathbf{C}$, $\dot{\mathbf{C}}$, $\mathbf{J}$ and $\dot{\mathbf{J}}$. Each constraint points to one or more matrix blocks that receive its contributions to the global matrices.
- Form the right-hand-side vector of equation 11.
- Invoke the conjugate gradient solver to obtain the Lagrange multiplier vector, λ.
- Multiply λ by $\mathbf{J}^T$ to obtain the global constraint force vector, and scatter this force to the particles.

5 Lagrangian Dynamics: modeling objects other than particles

In the previous sections we have seen how to constrain the behavior of a particle system through the use of constraint forces. Our starting point for the derivation was the use of *implicit functions*—functions of state that are supposed to be zero—to represent the constraints. Each scalar implicit function defines a hypersurface in state space, and the legal states of the system are those that lie on the intersection of all the hypersurfaces.

Suppose instead that we represented the constraints using a parametric function—a function $\mathbf{q}(\mathbf{u})$, with $\dim \mathbf{u} < \dim \mathbf{q}$, so that $\mathbf{q}(\mathbf{u})$ specifies all and only the legal states. In the case of a unit circle, the parametric function would of course be $\mathbf{x} = [\cos\theta, \sin\theta]$, leaving θ as the single degree of freedom.

In order to use parametric functions to represent constraints, we need to express the constrained system's equations of motion in terms of the new, constrained degrees of freedom, $\mathbf{u}$, rather than the unconstrained $\mathbf{q}$. These new equations, which we will derive in this section, are known as *Lagrange's equations of motion*.[2].

A clear advantage of the parametric constraint representation is that the extra degrees of freedom are actually removed from the system, rather than being neutralized through the use of constraint forces. This, as one would expect, can lead to better performance. However, Lagrangian dynamics has a very serious drawback: it is often difficult or impossible to find a parametric function that

captures the desired constraints. Moreover, in contrast to the implicit form, there is no automatic way to combine multiple constraints. Lagrangian dynamics is therefore unsuitable as a vehicle for interactive model building. Its important role is as an off-line tool for defining new primitive objects that are more complex than particles.

As before, we begin with a collection of particles whose positions are described by a global state vector $\mathbf{q}$, a diagonal mass matrix $\mathbf{M}$, and a global applied force vector $\mathbf{Q}$. We also retain the idea of a constraint force vector $\hat{\mathbf{Q}}$ that satisfies the principle of virtual work. Now, however, the $\mathbf{q}$'s are not independent variables, but are given by a function $\mathbf{q}(\mathbf{u})$. Our goal is to solve for $\ddot{\mathbf{u}}$, accounting for the constraint forces.

In developing the constraint force formulation we made extensive use of the Jacobian of the implicit constraint function. The Jacobian of the parametric function,

$$\mathbf{J} = \frac{\partial \mathbf{q}}{\partial \mathbf{u}},$$

has a different meaning but is equally important. By the chain rule, the *legal* particle velocities are given by

$$\dot{\mathbf{q}} = \mathbf{J}\dot{\mathbf{u}}.$$

The principle of virtual work therefore requires that

$$\hat{\mathbf{Q}}^T \mathbf{J}\dot{\mathbf{u}} = 0, \quad \forall \dot{\mathbf{u}},$$

which simply means that $\mathbf{J}^T\hat{\mathbf{Q}} = 0$. As before, we can write the unconstrained equations of motion as

$$\mathbf{M}\ddot{\mathbf{q}} = \mathbf{Q} + \hat{\mathbf{Q}}.$$

Now, however, instead of solving for the constraint force, we can simply make it go away, by multiplying both sides of the equation by $\mathbf{J}^T$, giving

$$\mathbf{J}^T\mathbf{M}\ddot{\mathbf{q}} - \mathbf{J}^T\mathbf{Q} = 0. \tag{12}$$

Since $\mathbf{q}$ is a function of $\mathbf{u}$, we can now remove $\ddot{\mathbf{q}}$ from the expression, leaving $\ddot{\mathbf{u}}$ as the unknown. Once again invoking the trusty chain rule,

$$\ddot{\mathbf{q}} = \mathbf{J}\ddot{\mathbf{u}} + \dot{\mathbf{J}}\dot{\mathbf{u}}.$$

Substituting this expression into equation 12 gives

$$\mathbf{J}^T\mathbf{M}\mathbf{J}\ddot{\mathbf{u}} + \mathbf{J}^T\mathbf{M}\dot{\mathbf{J}}\dot{\mathbf{u}} - \mathbf{J}^T\mathbf{Q} = 0. \tag{13}$$

which is a matrix equation to be solved for $\ddot{\mathbf{u}}$. Although you will usually see it expressed in a superficially quite different form, equation 13 is equivalent to the classical Lagrangian equation of motion. As we've expressed it here, its close relation to the constraint force formulation should be strikingly clear.

5.1 Hybrid models

The Lagrangian dynamics formulation is well-suited to creating compound objects off-line, while constraint force methods are well-suited to creating constrained models on the fly. In [5] we describe an architecture that combines both methods, allowing constraints to be applied dynamically to complex objects that had been pre-defined using Lagrangian dynamics. In [6], Lagrangian dynamics is used to create simplified non-rigid bodies.

References

[1] Ronen Barzel and Alan H. Barr. A modeling system based on dynamic constaints. *Computer Graphics*, 22:179–188, 1988.

[2] Herbert Goldstein. *Classical Mechanics*. Addision Wesley, Reading, MA, 1950.

[3] John Platt and Alan Barr. Constraint methods for flexible models. *Computer Graphics*, 22:279–288, 1988.

[4] W.H. Press, B.P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, England, 1988.

[5] Andrew Witkin, Michael Gleicher, and William Welch. Interactive dynamics. *Computer Graphics*, 24, 1990. Proc. 1990 Symposium on 3-D Interactive Graphics.

[6] Andrew Witkin and William Welch. Fast animation and control of non-rigid structures. *Computer Graphics*, 24(4):243–252, July 1990. Proc. Siggraph '90.

Physically Based Modeling
Rigid Body Simulation

David Baraff
Pixar Animation Studios

Rigid Body Simulation

David Baraff
Pixar Animation Studios

Introduction

This portion of the course notes deals with the problem of rigid body dynamics. To help get you started simulating rigid body motion, we've provided code fragments that implement most of the concepts discussed in these notes. This segment of the course notes is divided into two parts. The first part covers the motion of rigid bodies that are completely *unconstrained* in their allowable motion; that is, simulations that aren't concerned about collisions between rigid bodies. Given any external forces acting on a rigid body, we'll show how to simulate the motion of the body in response to these forces. The mathematical derivations in these notes are meant to be fairly informal and intuitive.

The second part of the notes tackles the problem of *constrained* motion that arises when we regard bodies as solid, and need to disallow inter-penetration. We enforce these non-penetration constraints by computing appropriate contact forces between contacting bodies. Given values for these contact forces, simulation proceeds exactly as in the unconstrained case: we simply apply all the forces to the bodies and let the simulation unfold as though the motions of bodies are completely unconstrained. If we have computed the contact forces correctly, the resulting motion of the bodies will be free from inter-penetration. The computation of these contact forces is the most demanding component of the entire simulation process.[1]

[1]Collision detection (i.e. determining the points of contact between bodies) runs a close second though!

Part I. Unconstrained Rigid Body Dynamics

1 Simulation Basics

This portion of the course notes is geared towards a full implementation of rigid body motion. In this section, we'll show the basic structure for simulating the motion of a rigid body. In section 2, we'll define the terms, concepts, and equations we need to implement a rigid body simulator. Following this, we'll give some code to actually implement the equations we need. Derivations for some of the concepts and equations we will be using will be left to appendix A.

The only thing you need to be familiar with at this point are the basic concepts (but not the numerical details) of solving ordinary differential equations. If you're not familiar with this topic, you're in luck: just turn back to the beginning of these course notes, and read the section on "Differential Equation Basics." You also might want to read the next section on "Particle Dynamics" as well, although we're about to repeat some of that material here anyway.

Simulating the motion of a rigid body is almost the same as simulating the motion of a particle, so let's start with particle simulation. The way we simulate a particle is as follows. We let a function $x(t)$ denote the particle's location in world space (the space all particles or bodies occupy during simulation) at time t. The function $v(t) = \dot{x}(t) = \frac{d}{dt}x(t)$ gives the velocity of the particle at time t. The *state* of a particle at time t is the particle's position and velocity. We generalize this concept by defining a *state vector* $\mathbf{X}(t)$ for a system: for a single particle,

$$\mathbf{X}(t) = \begin{pmatrix} x(t) \\ v(t) \end{pmatrix}. \tag{1–1}$$

When we're talking about an actual implementation, we have to "flatten" out $\mathbf{X}(t)$ into an array. For a single particle, $\mathbf{X}(t)$ can be described as an array of six numbers: typically, we'd let the first three elements of the array represent $x(t)$, and the last three elements represent $v(t)$. Later, when we talk about state vectors $\mathbf{X}(t)$ that contain matrices as well as vectors, the same sort of operation is done to flatten $\mathbf{X}(t)$ into an array. Of course, we'll also have to reverse this process and turn an array of numbers back into a state vector $\mathbf{X}(t)$. This all comes down to pretty simple bookkeeping though, so henceforth, we'll assume that we know how to convert any sort of state vector $\mathbf{X}(t)$ to an array (of the appropriate length) and vice versa. (For a simple example involving particles, look through the "Particle System Dynamics" section of these notes.)

For a system with n particles, we enlarge $\mathbf{X}(t)$ to be

$$\mathbf{X}(t) = \begin{pmatrix} x_1(t) \\ v_1(t) \\ \vdots \\ x_n(t) \\ v_n(t) \end{pmatrix} \tag{1–2}$$

where $x_i(t)$ and $v_i(t)$ are the position and velocity of the ith particle. Working with n particles is no harder than working with one particle, so we'll let $\mathbf{X}(t)$ be the state vector for a single particle for now (and when we get to it later, a single rigid body).

To actually simulate the motion of our particle, we need to know one more thing—the force acting on the particle at time t. We'll define $F(t)$ as the force acting on our particle at time t. The function $F(t)$ is the sum of all the forces acting on the particle: gravity, wind, spring forces, etc. If the particle has mass m, then the change of $\mathbf{X}$ over time is given by

$$\frac{d}{dt}\mathbf{X}(t) = \frac{d}{dt}\begin{pmatrix} x(t) \\ v(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ F(t)/m \end{pmatrix}. \tag{1–3}$$

Given any value of $\mathbf{X}(t)$, equation (1–3) describes how $\mathbf{X}(t)$ is instantaneously changing at time t. A simulation starts with some initial conditions for $\mathbf{X}(0)$, (i.e. values for $x(0)$ and $v(0)$) and then uses a numerical equation solver to track the change or "flow" of $\mathbf{X}$ over time, for as long as we're interested in. If all we want to know is the particle's location one second from now, we ask the solver to compute $\mathbf{X}(1)$, assuming that time units are in seconds. If we're going to animate the motion of the particle, we'd want to compute $\mathbf{X}(\frac{1}{30})$, $\mathbf{X}(\frac{2}{30})$ and so on.

The numerical method used by the solver is relatively unimportant with respect to our actual implementation. Let's look at how we'd actually interact with a numerical solver, in a C++-like language. Assume we have access to a numerical solver, which we'll generically write as a function named `ode`. Typically, `ode` has the following specification:

```
typedef void (*DerivFunc)(double t, double x[], double xdot[]);

void    ode(double x0[], double xEnd[], int len, double t0,
            double t1, DerivFunc dxdt);
```

We pass an initial state vector to `ode` as an array `x0`. The solver `ode` knows nothing about the inherent structure of `x0`. Since solvers can handle problems of arbitrary dimension, we also have to pass the length `len` of `x0`. (For a system of n particles, we'd obviously have `len` $= 6n$.) We also pass the solver the starting and ending times of the simulation, `t0` and `t1`. The solver's goal is to compute the state vector at time `t1` and return it in the array `xEnd`.

We also pass a function `Dxdt()` to `ode`. Given an array `y` that encodes a state vector $\mathbf{X}(t)$ and a time t, `Dxdt()` must compute and return $\frac{d}{dt}\mathbf{X}(t)$ in the array `xdot`. (The reason we must pass `t` to `Dxdt()` is that we may have time-varying forces acting in our system. In that case, `Dxdt()` would have to know "what time it is" to determine the value of those forces.) In tracing the flow of $\mathbf{X}(t)$ from `t0` to `t1`, the solver `ode` is allowed to call `Dxdt()` as often as it likes. Given that we have such a routine `ode`, the only work we need to do is to code up the routine `Dxdt()` which we'll give as a parameter to `ode`.

Simulating rigid bodies follows exactly the same mold as simulating particles. The only difference is that the state vector $\mathbf{X}(t)$ for a rigid body holds more information, and the derivative $\frac{d}{dt}\mathbf{X}(t)$ is a little more complicated. However, we'll use exactly the same paradigm of tracking the movement of a rigid body using a solver `ode`, which we'll supply with a function `Dxdt()`.

2 Rigid Body Concepts

The goal of this section is to develop an analogue to equation (1–3), for rigid bodies. The final differential equation we develop is given in section 2.11. In order to do this though, we need to define a lot of concepts first and relations first. Some of the longer derivations are found in appendix A. In the next section, we'll show how to write the function `Dxdt()` needed by the numerical solver `ode` to compute the derivative $\frac{d}{dt}\mathbf{X}(t)$ developed in this section.

2.1 Position and Orientation

The location of a particle in space at time t can be described as a vector $x(t)$, which describes the translation of the particle from the origin. Rigid bodies are more complicated, in that in addition to translating them, we can also rotate them. To locate a rigid body in world space, we'll use a vector $x(t)$, which describes the translation of the body. We must also describe the rotation of the body, which we'll do (for now) in terms of a 3×3 rotation matrix $R(t)$. We will call $x(t)$ and $R(t)$ the *spatial variables* of a rigid body.

A rigid body, unlike a particle, occupies a volume of space and has a particular shape. Because a rigid body can undergo only rotation and translation, we define the shape of a rigid body in terms of a fixed and unchanging space called *body space*. Given a geometric description of the body in body space, we use $x(t)$ and $R(t)$ to transform the body-space description into world space (figure 1). In order to simplify some equations we'll be using, we'll require that our description of the rigid body in body space be such that the *center of mass* of the body lies at the origin, $(0, 0, 0)$. We'll define the center of mass more precisely later, but for now, the center of mass can be thought of as a point in the rigid body that lies at the geometric center of the body. In describing the body's shape, we require that this geometric center lie at $(0, 0, 0)$ in body space. If we agree that $R(t)$ specifies a rotation of the body about the center of mass, then a fixed vector r in body space will be rotated to the world-space vector $R(t)r$ at time t. Likewise, if p_0 is an arbitrary point on the rigid body, in body space, then the world-space location $p(t)$ of p_0 is the result of first rotating p_0 about the origin and then translating it:

$$p(t) = R(t)\,p_0 + x(t). \tag{2–1}$$

Since the center of mass of the body lies at the origin, the world-space location of the center of mass is always given directly by $x(t)$. This lets us attach a very physical meaning to $x(t)$ by saying that $x(t)$ is the location of the center of mass in world space at time t. We can also attach a physical meaning to $R(t)$. Consider the x axis in body space i.e. the vector $(1, 0, 0)$. At time t, this vector has direction

$$R(t) \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

in world space. If we write out the components of $R(t)$ as

$$R(t) = \begin{pmatrix} r_{xx} & r_{yx} & r_{zx} \\ r_{xy} & r_{yy} & r_{zy} \\ r_{xz} & r_{yz} & r_{zz} \end{pmatrix}, \tag{2–2}$$

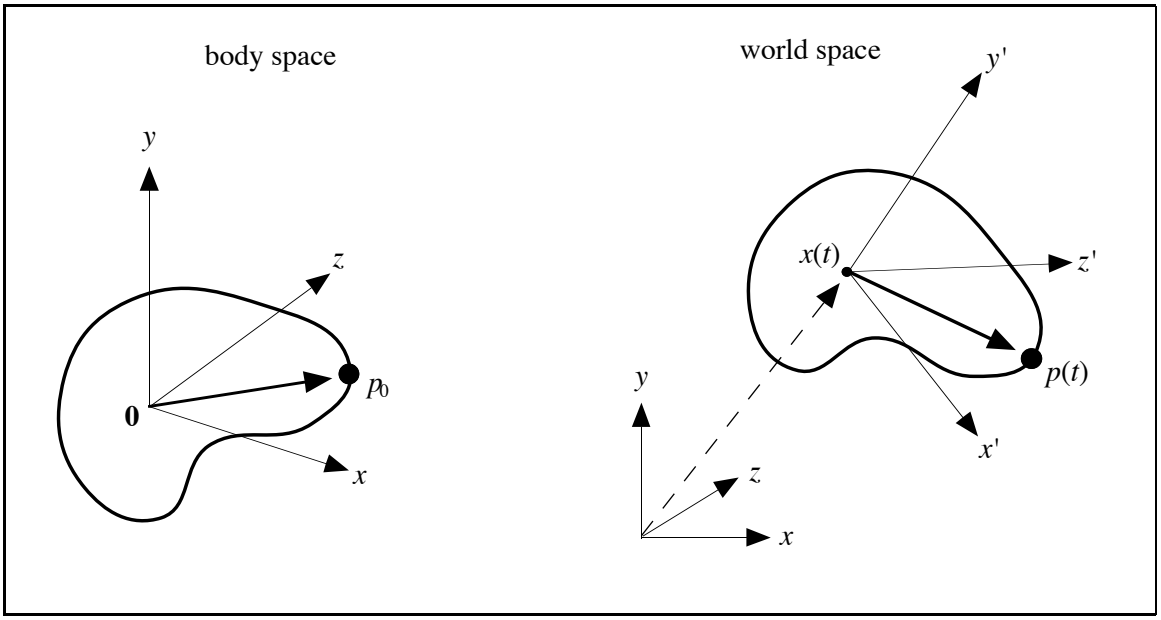


Figure 1: The center of mass is transformed to the point $x(t)$ in world space, at time t. The fixed x, y, and z axes of the body in body space transform to the vectors $x' = R(t)x$, $y' = R(t)y$ and $z' = R(t)z$. The fixed point p_0 in body space is transformed to the point $p(t) = R(t)p_0 + x(t)$.

then

$$R(t)\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix} \tag{2–3}$$

which is the first column of $R(t)$. The physical meaning of $R(t)$ is that $R(t)$'s first column gives the direction that the rigid body's x axis points in, when transformed to world space at time t. Similarly, the second and third columns of $R(t)$,

$$\begin{pmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{pmatrix} \qquad \text{and} \qquad \begin{pmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{pmatrix}$$

are the directions of the y and z axes of the rigid body in world space at time t (figure 2).

2.2 Linear Velocity

For simplicity, we'll call $x(t)$ and $R(t)$ the *position* and *orientation* of the body at time t. The next thing we need to do is define how the position and orientation change over time. This means we need expressions for $\dot{x}(t)$ and $\dot{R}(t)$. Since $x(t)$ is the position of the center of mass in world space, $\dot{x}(t)$ is the velocity of the center of mass in world space. We'll define the *linear velocity* $v(t)$ as this velocity:

$$v(t) = \dot{x}(t). \tag{2–4}$$

If we imagine that the orientation of the body is fixed, then the only movement the body can undergo is a pure translation. The quantity $v(t)$ gives the velocity of this translation.

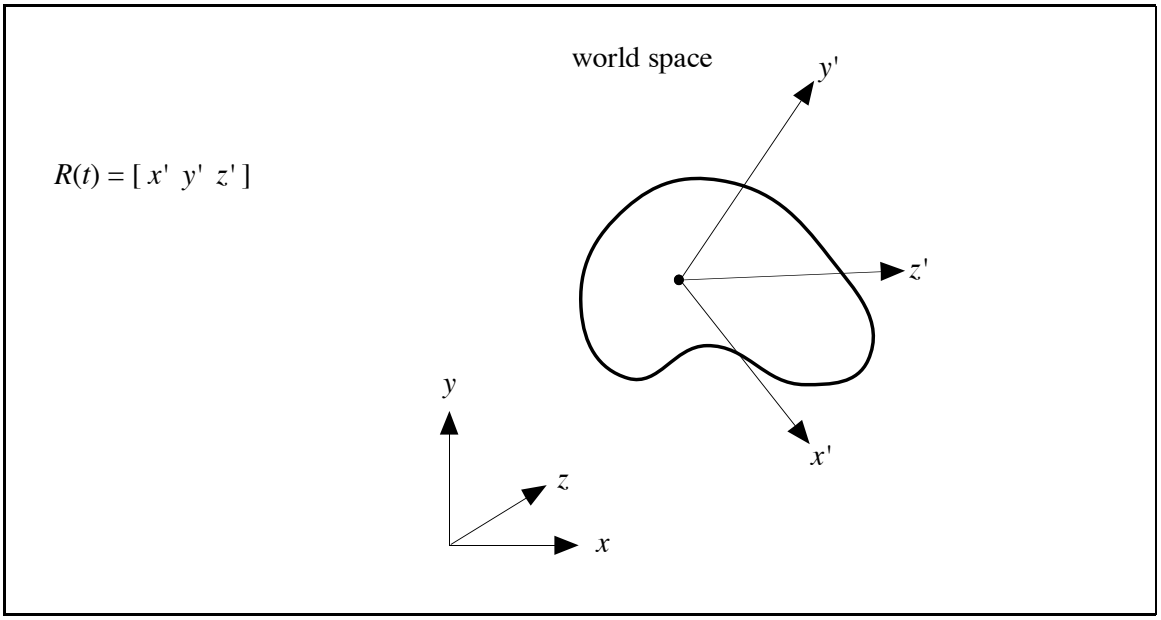


Figure 2: Physical interpretation of the orientation matrix $R(t)$. At time t, the columns of $R(t)$ are the world-space directions that the body-space x, y, and z axes transform to.

2.3 Angular Velocity

In addition to translating, a rigid body can also spin. Imagine however that we freeze the position of the center of mass in space. Any movement of the points of the body must therefore be due to the body spinning about some axis that passes through the center of mass. (Otherwise the center of mass would itself be moving). We can describe that spin as a vector $\omega(t)$. The *direction* of $\omega(t)$ gives the direction of the axis about which the body is spinning (figure 3). The magnitude of $\omega(t)$, $|\omega(t)|$, tells how fast the body is spinning. $|\omega(t)|$ has the dimensions of revolutions/time; thus, $|\omega(t)|$ relates the angle through which the body will rotate over a given period of time, if the angular velocity remains constant. The quantity $\omega(t)$ is called the *angular velocity*.

For linear velocity, $x(t)$ and $v(t)$ are related by $v(t) = \frac{d}{dt}x(t)$. How are $R(t)$ and $\omega(t)$ related? (Clearly, $\dot{R}(t)$ cannot be $\omega(t)$, since $R(t)$ is a matrix, and $\omega(t)$ is a vector.) To answer this question, let's remind ourselves of the physical meaning of $R(t)$. We know that the columns of $R(t)$ tell us the directions of the transformed x, y and z body axes at time t. That means that the columns of $\dot{R}(t)$ must describe the *velocity* with which the x, y, and z axes are being transformed. To discover the relationship between $\omega(t)$ and $R(t)$, let's examine how the change of an arbitrary vector in a rigid body is related to the angular velocity $\omega(t)$.

Figure 4 shows a rigid body with angular velocity $\omega(t)$. Consider a vector $r(t)$ at time t specified in world space. Suppose that we consider this vector fixed to the body; that is, $r(t)$ moves along with the rigid body through world space. Since $r(t)$ is a direction, it is independent of any translational effects; in particular, $\dot{r}(t)$ is independent of $v(t)$. To study $\dot{r}(t)$, we decompose $r(t)$ into vectors a and b, where a is parallel to $\omega(t)$ and b is perpendicular to $\omega(t)$. Suppose the rigid body were to maintain a constant angular velocity, so that the tip of $r(t)$ traces out a circle centered on the $\omega(t)$ axis (figure 4). The radius of this circle is $|b|$. Since the tip of the vector $r(t)$ is instantaneously moving along this circle, the instantaneous change of $r(t)$ is perpendicular to both b and $\omega(t)$. Since the tip of $r(t)$ is moving in a circle of radius b, the instantaneous velocity of $r(t)$ has magnitude $|b||\omega(t)|$.

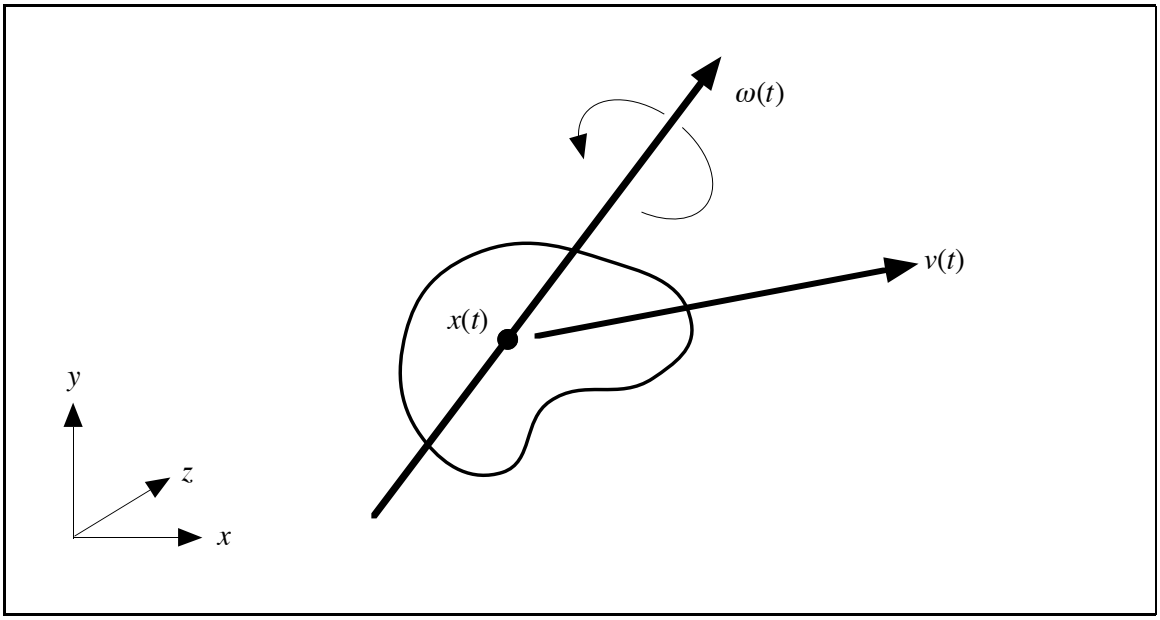


Figure 3: Linear velocity $v(t)$ and angular velocity $\omega(t)$ of a rigid body.

Since b and $\omega(t)$ are perpendicular, their cross product has magnitude

$$|\omega(t) \times b| = |\omega(t)|\,|b|. \tag{2–5}$$

Putting this together, we can write $\dot{r}(t) = \omega(t) \times (b)$. However, since $r(t) = a + b$ and a is parallel to $\omega(t)$, we have $\omega(t) \times a = 0$ and thus

$$\dot{r}(t) = \omega(t) \times b = \omega(t) \times b + \omega(t) \times a = \omega(t) \times (b + a). \tag{2–6}$$

Thus, we can simply express the rate of change of a vector as

$$\dot{r}(t) = \omega(t) \times r(t). \tag{2–7}$$

Let's put all this together now. At time t, we know that the direction of the x axis of the rigid body in world space is the first column of $R(t)$, which is

$$\begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix}.$$

At time t, the derivative of the first column of $R(t)$ is just the rate of change of this vector: using the cross product rule we just discovered, this change is

$$\omega(t) \times \begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix}.$$

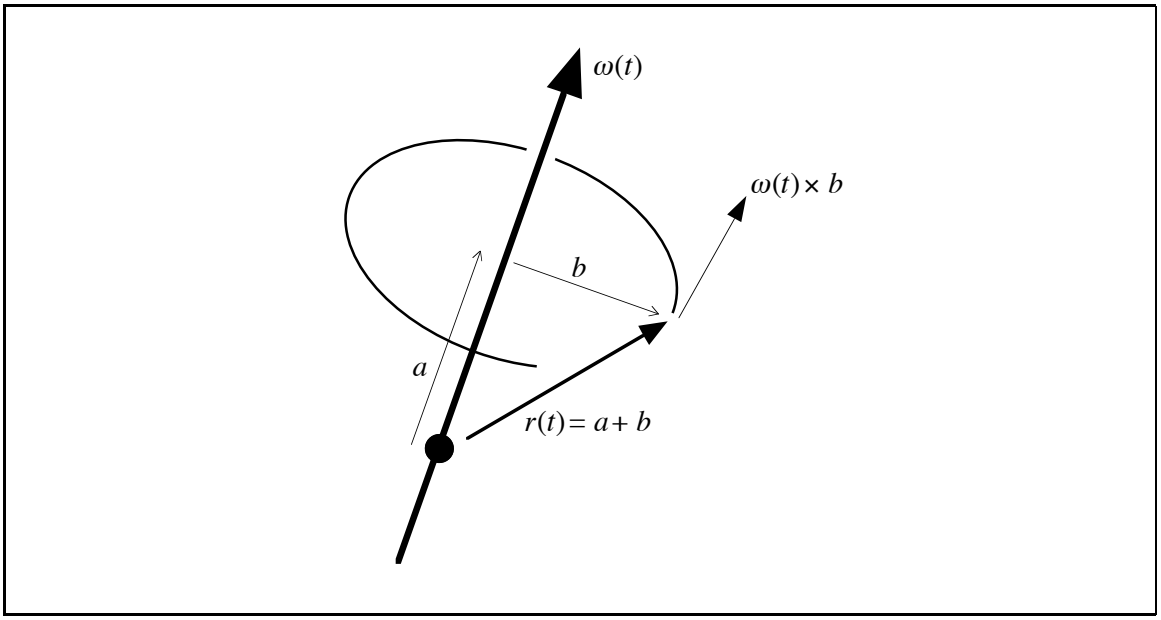


Figure 4: The rate of change of a rotating vector. As the tip of $r(t)$ spins about the $\omega(t)$ axis, it traces out a circle of diameter $|b|$. The speed of the tip of $r(t)$ is $|\omega(t)||b|$.

The same obviously holds for the other two columns of $R(t)$. This means that we can write

$$\dot{R} = \left(\omega(t) \times \begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix} \quad \omega(t) \times \begin{pmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{pmatrix} \quad \omega(t) \times \begin{pmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{pmatrix} \right). \tag{2–8}$$

This is too cumbersome an expression to tote around though. To simplify things, we'll use the following trick. If a and b are 3-vectors, then $a \times b$ is the vector

$$\begin{pmatrix} a_y b_z - b_y a_z \\ -a_x b_z + b_x a_z \\ a_x b_y - b_x a_y \end{pmatrix}.$$

Given the vector a, let us define a^* to be the matrix

$$\begin{pmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{pmatrix}.$$

Then[2]

$$a^* b = \begin{pmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{pmatrix} \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \begin{pmatrix} a_y b_z - b_y a_z \\ -a_x b_z + b_x a_z \\ a_x b_y - b_x a_y \end{pmatrix} = a \times b. \tag{2–9}$$

[2]This looks a little too "magical" at first. Did someone discover this identity accidentally? Is it a relation that just happens to work? This construct can be derived by considering what's known as *infinitesimal* rotations. The interested reader might wish to read chapter 4.8 of Goldstein[10] for a more complete derivation of the a^* matrix.

Using the "$*$" notation, we can rewrite $\dot{R}(t)$ more simply as

$$\dot{R}(t) = \left(\omega(t)^* \begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix} \quad \omega(t)^* \begin{pmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{pmatrix} \quad \omega(t)^* \begin{pmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{pmatrix} \right). \tag{2–10}$$

By the rules of matrix multiplication, we can factor this into

$$\dot{R}(t) = \omega(t)^* \left(\begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix} \quad \begin{pmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{pmatrix} \quad \begin{pmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{pmatrix} \right) \tag{2–11}$$

which is a matrix-matrix multiplication. But since the matrix on the right is $R(t)$ itself, we get simply that

$$\dot{R}(t) = \omega(t)^* R(t). \tag{2–12}$$

This, at last, gives us the relation we wanted between $\dot{R}(t)$ and $\omega(t)$. Note the correspondence between $\dot{r}(t) = \omega(t) \times r(t)$ for a vector, and $\dot{R}(t) = \omega(t)^* R(t)$ for the rotation matrix.

2.4 Mass of a Body

In order to work out some derivations, we'll need to (conceptually) perform some integrations over the volume of our rigid body. To make these derivations simpler, we're going to temporarily imagine that a rigid body is made up of a large number of small particles. The particles are indexed from 1 to N. The mass of the ith particle is m_i, and each particle has a (constant) location r_{0i} in body space. The location of the ith particle in world space at time t, denoted $r_i(t)$, is therefore given by the formula

$$r_i(t) = R(t) r_{0i} + x(t). \tag{2–13}$$

The total mass of the body, M, is the sum

$$M = \sum_{i=1}^{N} m_i. \tag{2–14}$$

(Henceforth, summations are assumed to be summed from 1 to N with index variable i.)

2.5 Velocity of a Particle

The *velocity* $\dot{r}_i(t)$ of the ith particle is obtained by differentiating equation (2–13): using the relation $\dot{R}(t) = \omega^* R(t)$, we obtain

$$\dot{r}_i(t) = \omega^* R(t) r_{0i} + v(t). \tag{2–15}$$

We can rewrite this as

$$\begin{aligned} \dot{r}_i(t) &= \omega(t)^* R(t) r_{0i} + v(t) \\ &= \omega(t)^* \left(R(t) r_{0i} + x(t) - x(t) \right) + v(t) \\ &= \omega(t)^* \left(r_i(t) - x(t) \right) + v(t) \end{aligned} \tag{2–16}$$

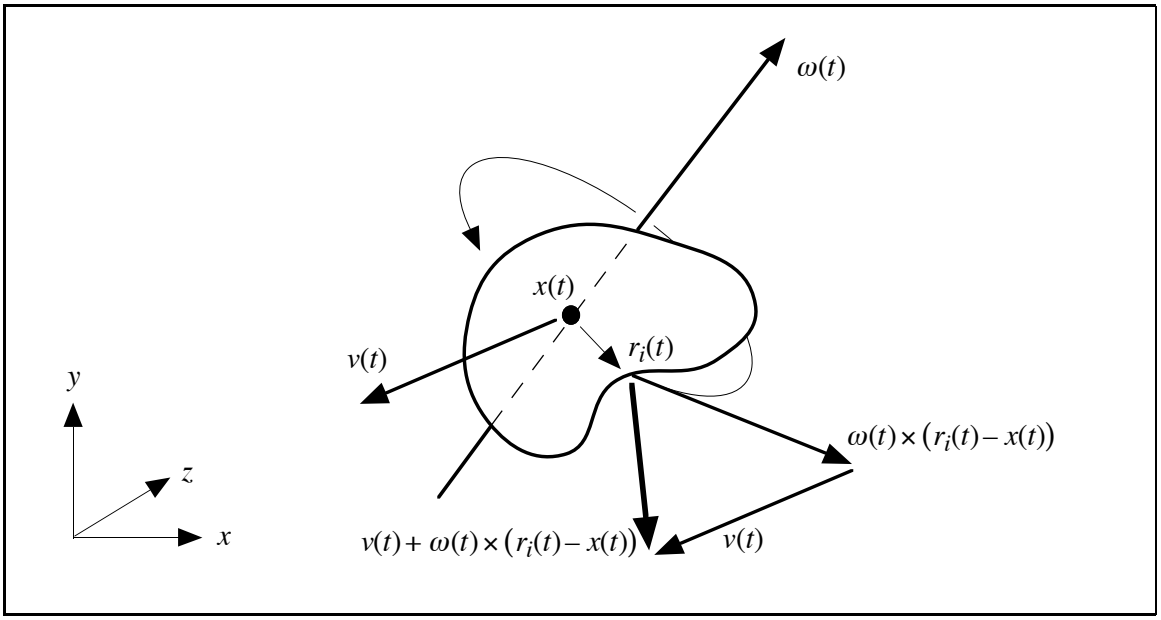


Figure 5: The velocity of the ith point of a rigid body in world space. The velocity of $r_i(t)$ can be decomposed into a linear term $v(t)$ and an angular term $\omega(t) \times (r_i(t) - x(t))$.

using the definition of $r_i(t)$ from equation (2–13). Recall from the definition of the "$*$" operator that $\omega(t)^* a = \omega(t) \times a$ for any vector a. Using this, we can simply write

$$\dot{r}_i(t) = \omega(t) \times (r_i(t) - x(t)) + v(t). \tag{2–17}$$

Note that this separates the velocity of a point on a rigid body into two components (figure 5): a linear component $v(t)$, and an angular component $\omega \times (r_i(t) - x(t))$.

2.6 Center of Mass

Our definition of the center of mass is going to enable us to likewise separate the dynamics of bodies into linear and angular components. The center of mass of a body in world space is defined to be

$$\frac{\sum m_i r_i(t)}{M} \tag{2–18}$$

where M is the mass of the body (i.e. the sum of the individual masses m_i). When we say that we are using a center of mass coordinate system, we mean that in body space,

$$\frac{\sum m_i r_{0i}}{M} = \mathbf{0} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}. \tag{2–19}$$

Note that this implies that $\sum m_i r_{0i} = \mathbf{0}$ as well.

We have spoken of $x(t)$ as being the location of the center of mass at time t. Is this true? Yes: since the ith particle has position $r_i(t) = R(t)r_{0i} + x(t)$ at time t, the center of mass at time t is

$$\frac{\sum m_i r_i(t)}{M} = \frac{\sum m_i (R(t)r_{0i} + x(t))}{M} = \frac{R(t)\sum m_i r_{0i} + \sum m_i x(t)}{M} = x(t)\frac{\sum m_i}{M} = x(t).$$

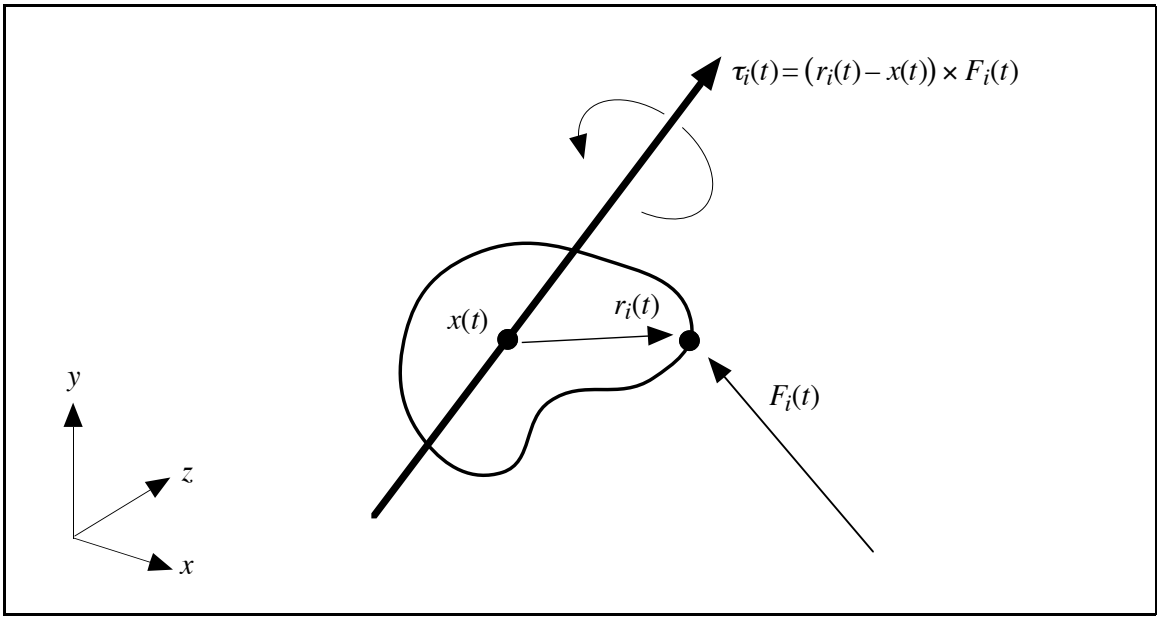


Figure 6: The torque $\tau_i(t)$ due to a force $F_i(t)$ acting at $r_i(t)$ on a rigid body.

Additionally, the relation

$$\sum m_i(r_i(t) - x(t)) = \sum m_i(R(t)r_{0i} + x(t) - x(t)) = R(t)\sum m_i r_{0i} = \mathbf{0} \tag{2–20}$$

is also very useful.

2.7 Force and Torque

When we imagine a force acting on a rigid body due to some external influence (e.g. gravity, wind, contact forces), we imagine that the force acts on a particular particle of the body. (Remember that our particle model is conceptual only. We can have a force act at any geometrical location on or inside the body, because we can always imagine that there happens to be a particle at that exact location.) The location of the particle the force acts on defines the location at which the force acts. We will let $F_i(t)$ denote the total force from external forces acting on the ith particle at time t. Also, we define the external *torque* $\tau_i(t)$ acting on the ith particle as

$$\tau_i(t) = (r_i(t) - x(t)) \times F_i(t). \tag{2–21}$$

Torque differs from force in that the torque on a particle depends on the location $r_i(t)$ of the particle, relative to the center of mass $x(t)$. We can intuitively think of the direction of $\tau_i(t)$ as being the axis the body would spin about due to $F_i(t)$, if the center of mass were held firmly in place (figure 6).

The total external force $F(t)$ acting on the body is the sum of the $F_i(t)$:

$$F(t) = \sum F_i(t) \tag{2–22}$$

while the total external torque is defined similarly as

$$\tau(t) = \sum \tau_i(t) = \sum (r_i(t) - x(t)) \times F_i(t). \tag{2–23}$$

Note that $F(t)$ conveys no information about where the various forces acted on the body; however, $\tau(t)$ does tell us something about the distribution of the forces $F_i(t)$ over the body.

2.8 Linear Momentum

The linear momentum p of a particle with mass m and velocity v is defined as

$$p = mv. \tag{2–24}$$

The total linear momentum $P(t)$ of a rigid body is the sum of the products of the mass and velocity of each particle:

$$P(t) = \sum m_i \dot{r}_i(t). \tag{2–25}$$

From equation (2–17), the velocity $\dot{r}_i(t)$ of the ith particle is $\dot{r}_i(t) = v(t) + \omega(t) \times (r_i(t) - x(t))$. Thus, the total linear momentum of the body is

$$\begin{aligned} P(t) &= \sum m_i \dot{r}_i(t) \\ &= \sum \Big(m_i v(t) + m_i \omega(t) \times (r_i(t) - x(t)) \Big) \\ &= \sum m_i v(t) + \omega(t) \times \sum m_i \left(r_i(t) - x(t) \right). \end{aligned} \tag{2–26}$$

Because we are using a center of mass coordinate system, we can apply equation (2–20) and obtain

$$P(t) = \sum m_i v(t) = \left(\sum m_i \right) v(t) = M v(t). \tag{2–27}$$

This gives us the nice result that the total linear momentum of our rigid body is the same as if the body was simply a particle with mass M and velocity $v(t)$. Because of this, we have a simple transformation between $P(t)$ and $v(t)$: $P(t) = Mv(t)$ and $v(t) = P(t)/M$. Since M is a constant,

$$\dot{v}(t) = \frac{\dot{P}(t)}{M}. \tag{2–28}$$

The concept of linear momentum lets us express the effect of the total force $F(t)$ on a rigid body quite simply. Appendix A derives the relation

$$\dot{P}(t) = F(t) \tag{2–29}$$

which says that the change in linear momentum is equivalent to the total force acting on a body. Note that $P(t)$ tells us nothing about the rotational velocity of a body, which is good, because $F(t)$ also conveys nothing about the change of rotational velocity of a body!

Since the relationship between $P(t)$ and $v(t)$ is simple, we will be using $P(t)$ as a state variable for our rigid body, instead of $v(t)$. We could of course let $v(t)$ be a state variable, and use the relation

$$\dot{v}(t) = \frac{F(t)}{M}. \tag{2–30}$$

However, using $P(t)$ instead of $v(t)$ as a state variable will be more consistent with the way we will be dealing with angular velocity and acceleration.

2.9 Angular Momentum

While the concept of linear momentum is pretty intuitive ($P(t) = Mv(t)$), the concept of angular momentum (for a rigid body) is not. The only reason that one even bothers with the angular momentum of a rigid body is that it lets you write simpler equations than you would get if you stuck with angular velocity. With that in mind, it's probably best not to worry about attaching an intuitive physical explanation to angular momentum—all in all, it's a most unintuitive concept. Angular momentum ends up simplifying equations because it is conserved in nature, while angular *velocity* is not: if you have a body floating through space with no torque acting on it, the body's angular momentum is constant. This is not true for a body's angular velocity though: even if the angular momentum of a body is constant, the body's angular *velocity* may not be! Consequently, a body's angular velocity can vary even when no force acts on the body. Because of this, it ends up being simpler to choose angular momentum as a state variable over angular velocity.

For linear momentum, we have the relation $P(t) = Mv(t)$. Similarly, we define the total angular momentum $L(t)$ of a rigid body by the equation $L(t) = I(t)\omega(t)$, where $I(t)$ is a 3×3 matrix (technically a rank-two tensor) called the *inertia tensor*, which we will describe momentarily. The inertia tensor $I(t)$ describes how the mass in a body is distributed relative to the body's center of mass. The tensor $I(t)$ depends on the orientation of a body, but does not depend on the body's translation. Note that for both the angular and the linear case, momentum is a linear function of velocity—it's just that in the angular case the scaling factor is a matrix, while it's simply a scalar in the linear case. Note also that $L(t)$ is independent of any translational effects, while $P(t)$ is independent of any rotational effects.

The relationship between $L(t)$ and the total torque $\tau(t)$ is very simple: appendix A derives

$$\dot{L}(t) = \tau(t), \tag{2–31}$$

analogous to the relation $\dot{P}(t) = F(t)$.

2.10 The Inertia Tensor

The inertia tensor $I(t)$ is the scaling factor between angular momentum $L(t)$ and angular velocity $\omega(t)$. At a given time t, let r'_i be the displacement of the ith particle from $x(t)$ by defining $r'_i = r_i(t) - x(t)$. The tensor $I(t)$ is expressed in terms of r'_i as the symmetric matrix

$$I(t) = \sum \begin{pmatrix} m_i(r'^2_{iy} + r'^2_{iz}) & -m_i r'_{ix} r'_{iy} & -m_i r'_{ix} r'_{iz} \\ -m_i r'_{iy} r'_{ix} & m_i(r'^2_{ix} + r'^2_{iz}) & -m_i r'_{iy} r'_{iz} \\ -m_i r'_{iz} r'_{ix} & -m_i r'_{iz} r'_{iy} & m_i(r'^2_{ix} + r'^2_{iy}) \end{pmatrix} \tag{2–32}$$

For an actual implementation, we replace the finite sums with integrals over a body's volume in world space. The mass terms m_i are replaced by a density function. At first glance, it seems that we would need to evaluate these integrals to find $I(t)$ whenever the orientation $R(t)$ changes. This would be prohibitively expensive to do during a simulation unless the body's shape was so simple (for example, a sphere or cube) that that the integrals could be evaluated symbolically.

Fortunately, by using body-space coordinates we can cheaply compute the inertia tensor for any orientation $R(t)$ in terms of a precomputed integral in body-space coordinates. (This integral is typically computed before the simulation begins and should be regarded as one of the input parameters

describing a physical property of the body.) Using the fact that ${r_i'}^T r_i' = r_{ix}'^2 + r_{iy}'^2 + r_{iz}'^2$, we can rewrite $I(t)$ as the difference

$$I(t) = \sum m_i {r_i'}^T r_i' \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \begin{pmatrix} m_i r_{ix}'^2 & m_i r_{ix}' r_{iy}' & m_i r_{ix}' r_{iz}' \\ m_i r_{iy}' r_{ix}' & m_i r_{iy}'^2 & m_i r_{iy}' r_{iz}' \\ m_i r_{iz}' r_{ix}' & m_i r_{iz}' r_{iy}' & m_i r_{iz}'^2 \end{pmatrix} \tag{2–33}$$

Taking the outer product multiplication of r_i' with itself, that is

$$r_i' {r_i'}^T = \begin{pmatrix} r_{ix}' \\ r_{iy}' \\ r_{iz}' \end{pmatrix} \begin{pmatrix} r_{ix}' & r_{iy}' & r_{iz}' \end{pmatrix} = \begin{pmatrix} r_{ix}'^2 & r_{ix}' r_{iy}' & r_{ix}' r_{iz}' \\ r_{iy}' r_{ix}' & r_{iy}'^2 & r_{ix}' r_{iz}' \\ r_{iz}' r_{ix}' & r_{iz}' r_{iy}' & r_{iz}'^2 \end{pmatrix} \tag{2–34}$$

and letting $\mathbf{1}$ denote the 3×3 identity matrix, we can express $I(t)$ simply as

$$I(t) = \sum m_i (({r_i'}^T r_i')\mathbf{1} - r_i' {r_i'}^T) \tag{2–35}$$

How does this help?

Since $r_i(t) = R(t) r_{0i} + x(t)$ where r_{0i} is a constant, $r_i' = R(t) r_{0i}$. Then, since $R(t)R(t)^T = \mathbf{1}$,

$$\begin{aligned} I(t) &= \sum m_i (({r_i'}^T r_i')\mathbf{1} - r_i' {r_i'}^T) \\ &= \sum m_i ((R(t)r_{0i})^T (R(t)r_{0i})\mathbf{1} - (R(t)r_{0i})(R(t)r_{0i})^T) \\ &= \sum m_i (r_{0i}^T R(t)^T R(t) r_{0i}\mathbf{1} - R(t) r_{0i} r_{0i}^T R(t)^T) \\ &= \sum m_i ((r_{0i}^T r_{0i})\mathbf{1} - R(t) r_{0i} r_{0i}^T R(t)^T). \end{aligned} \tag{2–36}$$

Since $r_{0i}^T r_{0i}$ is a scalar, we can rearrange things by writing

$$\begin{aligned} I(t) &= \sum m_i ((r_{0i}^T r_{0i})\mathbf{1} - R(t) r_{0i} r_{0i}^T R(t)^T) \\ &= \sum m_i (R(t)(r_{0i}^T r_{0i}) R(t)^T \mathbf{1} - R(t) r_{0i} r_{0i}^T R(t)^T) \\ &= R(t) \left(\sum m_i ((r_{0i}^T r_{0i})\mathbf{1} - r_{0i} r_{0i}^T) \right) R(t)^T. \end{aligned} \tag{2–37}$$

If we define I_{body} as the matrix

$$I_{body} = \sum m_i ((r_{0i}^T r_{0i})\mathbf{1} - r_{0i} r_{0i}^T) \tag{2–38}$$

then from the previous equation we have

$$I(t) = R(t) I_{body} R(t)^T. \tag{2–39}$$

Since I_{body} is specified in body-space, it is constant over the simulation. Thus, by precomputing I_{body} for a body before the simulation begins, we can easily compute $I(t)$ from I_{body} and the orientation matrix $R(t)$. Section 5.1 derives the body-space inertia tensor for a rectangular object in terms of an integral over the body's volume in body space.

Also, the inverse of $I(t)$ is given by the formula

$$\begin{aligned} I^{-1}(t) &= \left(R(t)I_{body}R(t)^T\right)^{-1} \\ &= \left(R(t)^T\right)^{-1} I_{body}^{-1}R(t)^{-1} \\ &= R(t)I_{body}^{-1}R(t)^T \end{aligned} \tag{2–40}$$

since, for rotation matrices, $R(t)^T = R(t)^{-1}$ and $\left(R(t)^T\right)^T = R(t)$. Clearly, I_{body}^{-1} is also a constant during the simulation.

2.11 Rigid Body Equations of Motion

Finally, we have covered all the concepts we need to define the state vector $\mathbf{X}(t)$! For a rigid body, we will define $\mathbf{X}(t)$ as

$$\mathbf{X}(t) = \begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix}. \tag{2–41}$$

Thus, the state of a rigid body is its position and orientation (describing spatial information), and its linear and angular momentum (describing velocity information). The mass M of the body and body-space inertia tensor I_{body} are constants, which we assume we know when the simulation begins. At any given time, the auxiliary quantities $I(t)$, $\omega(t)$ and $v(t)$ are computed by

$$v(t) = \frac{P(t)}{M}, \quad I(t) = R(t)I_{body}R(t)^T \qquad \text{and} \qquad \omega(t) = I(t)^{-1}L(t). \tag{2–42}$$

The derivative $\frac{d}{dt}\mathbf{X}(t)$ is

$$\frac{d}{dt}\mathbf{X}(t) = \frac{d}{dt}\begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ \omega(t)^*R(t) \\ F(t) \\ \tau(t) \end{pmatrix}. \tag{2–43}$$

The next section gives an implementation for the function `Dxdt()` that computes $\frac{d}{dt}\mathbf{X}(t)$.

One final note: rather than represent the orientation of the body as a matrix $R(t)$ in $\mathbf{X}(t)$, it is better to use *quaternions*. Section 4 discusses using quaternions in place of rotation matrices. Briefly, a quaternion is a type of four element vector that can be used to represent a rotation. If we replace $R(t)$ in $\mathbf{X}(t)$ with a quaternion $q(t)$, we can treat $R(t)$ as an auxiliary variable that is computed directly from $q(t)$, just as $\omega(t)$ is computed from $L(t)$. Section 4 derives a formula analogous to $\dot{R}(t) = \omega(t)^*R(t)$, that expresses $\dot{q}(t)$ in terms of $q(t)$ and $\omega(t)$.

3 Computing $\frac{d}{dt}\mathbf{X}(t)$

Lets consider an implementation of the function `Dxdt()` for rigid bodies. The code is written in C++, and we'll assume that we have datatypes (classes) called `matrix` and `triple` which implement, respectively, 3×3 matrices and points in 3-space. Using these datatypes, we'll represent

a rigid body by the structure

```
struct RigidBody {
    /* Constant quantities */
    double  mass;           /* mass M */
    matrix  Ibody,          /* I_body */
            Ibodyinv;       /* I_body^-1 (inverse of I_body) */

    /* State variables */
    triple  x;              /* x(t) */
    matrix  R;              /* R(t) */
    triple  P,              /* P(t) */
            L;              /* L(t) */

    /* Derived quantities (auxiliary variables) */
    matrix  Iinv;           /* I^-1(t) */
    triple  v,              /* v(t) */
            omega;          /* ω(t) */

    /* Computed quantities */
    triple  force,          /* F(t) */
            torque;         /* τ(t) */
};
```

and assume a global array of bodies

```
RigidBody Bodies[NBODIES];
```

The constant quantities `mass`, `Ibody` and `Ibodyinv` are assumed to have been calculated for each member of the array `Bodies`, before simulation begins. Also, the initial conditions for each rigid body are specified by assigning values to the state variables `x`, `R`, `P` and `L` of each member of `Bodies`. The implementation in this section represents orientation with a rotation matrix; section 4 describes the changes necessary to represent orientation by a quaternion.

We communicate with the differential equation solver `ode` by passing arrays of real numbers. Several bookkeeping routines are required:

```
/* Copy the state information into an array */
void StateToArray(RigidBody *rb, double *y)
{
    *y++ = rb->x[0];            /* x component of position */
    *y++ = rb->x[1];            /* etc. */
    *y++ = rb->x[2];

    for(int i = 0; i < 3; i++)  /* copy rotation matrix */
        for(int j = 0; j < 3; j++)
            *y++ = rb->R[i,j];
```

```
    *y++ = rb->P[0];
    *y++ = rb->P[1];
    *y++ = rb->P[2];

    *y++ = rb->L[0];
    *y++ = rb->L[1];
    *y++ = rb->L[2];
}
```

and

```
/* Copy information from an array into the state variables */
void ArrayToState(RigidBody *rb, double *y)
{
    rb->x[0] = *y++;
    rb->x[1] = *y++;
    rb->x[2] = *y++;

    for(int i = 0; i < 3; i++)
        for(int j = 0; j < 3; j++)
            rb->R[i,j] = *y++;

    rb->P[0] = *y++;
    rb->P[1] = *y++;
    rb->P[2] = *y++;

    rb->L[0] = *y++;
    rb->L[1] = *y++;
    rb->L[2] = *y++;

    /* Compute auxiliary variables... */

    /* v(t) = P(t)/M */
    rb->v = rb->P / mass;

    /* I^-1(t) = R(t) I_body^-1 R(t)^T */
    rb->Iinv = R * Ibodyinv * Transpose(R);

    /* ω(t) = I^-1(t) L(t) */
    rb->omega = rb->Iinv * rb->L;
}
```

Note that `ArrayToState` is responsible for computing values for the auxiliary variables `Iinv`, `v` and `omega`. We'll assume that the appropriate arithmetic operations have been defined between real numbers, `triple`'s and `matrix`'s, and that `Transpose` returns the transpose of a matrix.

Examining these routines, we see that each rigid body's state is represented by $3+9+3+3=18$ numbers. Transfers between all the members of `Bodies` and an array `y` of size $18 \cdot$ `NBODIES` are implemented as

```
#define STATE_SIZE      18

void ArrayToBodies(double x[])
{
    for(int i = 0; i < NBODIES; i++)
        ArrayToState(&Bodies[i], &x[i * STATE_SIZE]);
}
```

and

```
void BodiesToArray(double x[])
{
    for(int i = 0; i < NBODIES; i++)
        StateToArray(&Bodies[i], &x[i * STATE_SIZE]);
}
```

Now we can implement `Dxdt()`. Let's assume that the routine

```
void    ComputeForceAndTorque(double t, RigidBody *rb);
```

computes the force $F(t)$ and torque $\tau(t)$ acting on the rigid body `*rb` at time `t`, and stores $F(t)$ and $\tau(t)$ in `rb->force` and `rb->torque` respectively. `ComputeForceAndTorque` takes into account all forces and torques: gravity, wind, interaction with other bodies etc. Using this routine, we'll define `Dxdt()` as

```
void Dxdt(double t, double x[], double xdot[])
{
    /* put data in x[] into Bodies[] */
    ArrayToBodies(x);

    for(int i = 0; i < NBODIES; i++)
    {
        ComputeForceAndTorque(t, &Bodies[i]);
        DdtStateToArray(&Bodies[i],
                        &xdot[i * STATE_SIZE]);
    }
}
```

The numerical solver `ode` calls calls `Dxdt()` and is responsible for allocating enough space for the arrays `y`, and `xdot` (`STATE_SIZE` $\cdot$ `NBODIES` worth for each). The function which does the real work of computing $\frac{d}{dt}\mathbf{X}(t)$ and storing it in the array `xdot` is `ddtStateToArray`:

```
void DdtStateToArray(RigidBody *rb, double *xdot)
{
    /* copy d/dt x(t) = v(t) into xdot */
    *xdot++ = rb->v[0];
    *xdot++ = rb->v[1];
    *xdot++ = rb->v[2];

    /* Compute Rdot(t) = omega(t)*R(t) */
    matrix  Rdot = Star(rb->omega) * rb->R;

    /* copy Rdot(t) into array */
    for(int i = 0; i < 3; i++)
        for(int j = 0; j < 3; j++)
            *xdot++ = Rdot[i,j];

    *xdot++ = rb->force[0];      /* d/dt P(t) = F(t) */
    *xdot++ = rb->force[1];
    *xdot++ = rb->force[2];

    *xdot++ = rb->torque[0];     /* d/dt L(t) = tau(t) */
    *xdot++ = rb->torque[1];
    *xdot++ = rb->torque[2];
}
```

The routine `Star`, used to calculate $\dot{R}(t)$ is defined as

```
matrix   Star(triple a);
```

and returns the matrix

$$\begin{pmatrix} 0 & -a[2] & a[1] \\ a[2] & 0 & -a[0] \\ -a[1] & a[0] & 0 \end{pmatrix}.$$

Given all of the above, actually performing a simulation is simple. Assume that the state variables of all `NBODIES` rigid bodies are initialized by a routine `InitStates`. We'll have our simulation run for 10 seconds, calling a routine `DisplayBodies` every $\frac{1}{24}$th of a second to display the bodies:

```
void RunSimulation()
{
    double  x0[STATE_SIZE * NBODIES],
            xFinal[STATE_SIZE * NBODIES];

    InitStates();
    BodiesToArray(xFinal);
```

```
    for(double t = 0; t < 10.0; t += 1./24.)
    {
        /* copy xFinal back to x0 */
        for(int i = 0; i < STATE_SIZE * NBODIES; i++)
        {
             x0[i] = xFinal[i];

        ode(x0, xFinal, STATE_SIZE * NBODIES,
            t, t+1./24., Dxdt);

        /* copy d/dt X(t+1/24) into state variables */

        ArrayToBodies(xFinal);
        DisplayBodies();
    }
}
```

4 Quaternions vs. Rotation Matrices

There is a better way to represent the orientation of a rigid body than using a 3×3 rotation matrix. For a number of reasons, *unit quaternions*, a type of four element vector normalized to unit length, are a better choice than rotation matrices[16].

For rigid body simulation, the most important reason to avoid using rotation matrices is because of numerical drift. Suppose that we keep track of the orientation of a rigid body according to the formula

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

As we update $R(t)$ using this formula (that is, as we integrate this equation), we will inevitably encounter drift. Numerical error will build up in the coefficients of $R(t)$ so that $R(t)$ will no longer be precisely a rotation matrix. Graphically, the effect would be that applying $R(t)$ to a body would cause a skewing effect.

This problem can be alleviated by representing rotations with unit quaternions. Since quaternions have only four parameters, there is only one extra variable being used to describe the three freedoms of the rotation. In contrast, a rotation matrix uses nine parameters to describe three degrees of freedom; therefore, the degree of redundancy is noticeably lower for quaternions than rotation matrices. As a result, quaternions experience far less drift than rotation matrices. If it does become necessary to account for drift in a quaternion, it is because the quaternion has lost its unit magnitude.[3] This is easily correctable by renormalizing the quaternion to unit length. Because of these two properties, it is desirable to represent the orientation of a body directly as a unit quaternion $q(t)$. We will still express angular velocity as a vector $\omega(t)$. The orientation matrix $R(t)$, which is needed to compute $I^{-1}(t)$, will be computed as an auxiliary variable from $q(t)$.

[3] Any quaternion of unit length corresponds to a rotation, so quaternions deviate from representing rotations only if they lose their unit length. These notes will deal with that problem in a very simplistic way.

We will write a quaternion $s + v_x\mathbf{i} + v_y\mathbf{j} + v_z\mathbf{k}$ as the pair

$$[s, v].$$

Using this notation, quaternion multiplication is

$$[s_1, v_1][s_2, v_2] = [s_1 s_2 - v_1 \cdot v_2, s_1 v_2 + s_2 v_1 + v_1 \times v_2]. \tag{4–1}$$

A rotation of θ radians about a unit axis u is represented by the unit quaternion

$$[\cos(\theta/2), \sin(\theta/2)u].$$

In using quaternions to represent rotations, if q_1 and q_2 indicate rotations, then $q_2 q_1$ represents the composite rotation of q_1 followed by q_2.[4] In a moment, we'll show how to change the routines of section 3 to handle the quaternion representation for orientation. Before we can make these changes though, we'll need a formula for $\dot{q}(t)$. Appendix B derives the formula

$$\dot{q}(t) = \tfrac{1}{2}\omega(t)q(t). \tag{4–2}$$

where the multiplication $\omega(t)q(t)$ is a shorthand for multiplication between the quaternions $[0, \omega(t)]$ and $q(t)$. Note the similarity between equation (4–2) and

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

To actually use a quaternion representation, we'll need to redefine the type `RigidBody`:

```
struct RigidBody {
    /* Constant quantities */
    double  mass;           /* mass M */
    matrix  Ibody,          /* I_body */
            Ibodyinv;       /* I_body^-1 (inverse of I_body) */

    /* State variables */
    triple  x;              /* x(t) */
    quaternion q;           /* q(t) */
    triple  P,              /* P(t) */
            L;              /* L(t) */

    /* Derived quantities (auxiliary variables) */
    matrix  Iinv,           /* I^-1(t) */
            R;              /* R(t) */
    triple  v,              /* v(t) */
            omega;          /* omega(t) */

    /* Computed quantities */
```

[4] This is according to the convention that the rotation of a point p by a quaternion q is qpq^{-1}. Be warned! This is *opposite* the convention for rotation in the original paper Shoemake[16], but it is in correspondence with some more recent versions of Shoemake's article. Writing a composite rotation as $q_2 q_1$ parallels our matrix notation for composition of rotations.

```
        triple  force,          /* F(t) */
                torque;         /* τ(t) */
    };
```

Next, in the routine `StateToArray`, we'll replace the double loop

```
        for(int i = 0; i < 3; i++)      /* copy rotation matrix */
            for(int j = 0; j < 3; j++)
                *y++ = rb->R[i,j];
```

with

```
           /*
            * Assume that a quaternion is represented in
            * terms of elements 'r' for the real part,
            * and 'i', 'j', and 'k' for the vector part.
            */

            *y++ = rb->q.r;
            *y++ = rb->q.i;
            *y++ = rb->q.j;
            *y++ = rb->q.k;
```

A similar change is made in `ArrayToState`. Also, since `ArrayToState` is responsible for computing the auxiliary variable $I^{-1}(t)$, which depends on $R(t)$, `ArrayToState` must also compute $R(t)$ as an auxiliary variable: in the section

```
            /* Compute auxiliary variables... */

            /* v(t) = P(t)/M */
            rb->v = rb->P / mass;

            /* I^-1(t) = R(t) I_body^-1 R(t)^T */
            rb->Iinv = R * Ibodyinv * Transpose(R);

            /* ω(t) = I^-1(t) L(t) */
            rb->omega = rb->Iinv * rb->L;
```

we add the line

```
            rb->R = QuaterionToMatrix(normalize(rb->q));
```

prior to computing `rb->Iinv`. The routine `normalize` returns `q` divided by its length; this unit length quaternion returned by `normalize` is then passed to `QuaterionToMatrix` which returns a 3×3 rotation matrix. Given a quaternion $q = [s, v]$, `QuaterionToMatrix` returns the

matrix

$$\begin{pmatrix} 1-2v_y^2-2v_z^2 & 2v_xv_y-2sv_z & 2v_xv_z+2sv_y \\ 2v_xv_y+2sv_z & 1-2v_x^2-2v_z^2 & 2v_yv_z-2sv_x \\ 2v_xv_z-2sv_y & 2v_yv_z+2sv_x & 1-2v_x^2-2v_y^2 \end{pmatrix}.$$

In case you need to convert from a rotation matrix to a quaternion,

```
quaternion matrixToQuaternion(const matrix &m)
{
    quaternion   q;
    double       tr, s;

    tr = m[0,0] + m[1,1] + m[2,2];

    if(tr >= 0)
    {
        s = sqrt(tr + 1);
        q.r = 0.5 * s;
        s = 0.5 / s;
        q.i = (m[2,1] - m[1,2]) * s;
        q.j = (m[0,2] - m[2,0]) * s;
        q.k = (m[1,0] - m[0,1]) * s;
    }
    else
    {
        int i = 0;

        if(m[1,1] > m[0,0])
            i = 1;
        if(m[2,2] > m[i,i])
            i = 2;

        switch (i)
        {
        case 0:
            s = sqrt((m[0,0] - (m[1,1] + m[2,2])) + 1);
            q.i = 0.5 * s;
            s = 0.5 / s;
            q.j = (m[0,1] + m[1,0]) * s;
            q.k = (m[2,0] + m[0,2]) * s;
            q.r = (m[2,1] - m[1,2]) * s;
            break;
        case 1:
            s = sqrt((m[1,1] - (m[2,2] + m[0,0])) + 1);
            q.j = 0.5 * s;
            s = 0.5 / s;
```

```
            q.k = (m[1,2] + m[2,1]) * s;
            q.i = (m[0,1] + m[1,0]) * s;
            q.r = (m[0,2] - m[2,0]) * s;
            break;
        case 2:
            s = sqrt((m[2,2] - (m[0,0] + m[1,1])) + 1);
            q.k = 0.5 * s;
            s = 0.5 / s;
            q.i = (m[2,0] + m[0,2]) * s;
            q.j = (m[1,2] + m[2,1]) * s;
            q.r = (m[1,0] - m[0,1]) * s;
        }

    }
    return q;
}
```

The matrix `m` is structured so that `m[0,0]`, `m[0,1]` and `m[0,2]` form the first row (not column) of `m`.

The routines `ArrayToBodies` and `BodiesToArray` don't need any changes at all, but note that the constant `STATE_SIZE` changes from 18 to 13, since a quaternion requires five less elements than a rotation matrix. The only other change we need is in `ddtStateToArray`. Instead of

```
matrix  Rdot = Star(rb->omega) * rb->R;

/* copy Ṙ(t) into array */
for(int i = 0; i < 3; i++)
    for(int j = 0; j < 3; j++)
        *xdot++ = Rdot[i,j];
```

we'll use

```
quaternion      qdot = .5 * (rb->omega * rb->q);
*xdot++ = qdot.r;
*xdot++ = qdot.i;
*xdot++ = qdot.j;
*xdot++ = qdot.k;
```

We're assuming here that the multiplication between the triple `rb->omega` and the quaternion `rb->q` is defined to return the quaternion product

$$[0, \texttt{rb->omega}]\texttt{q}.$$

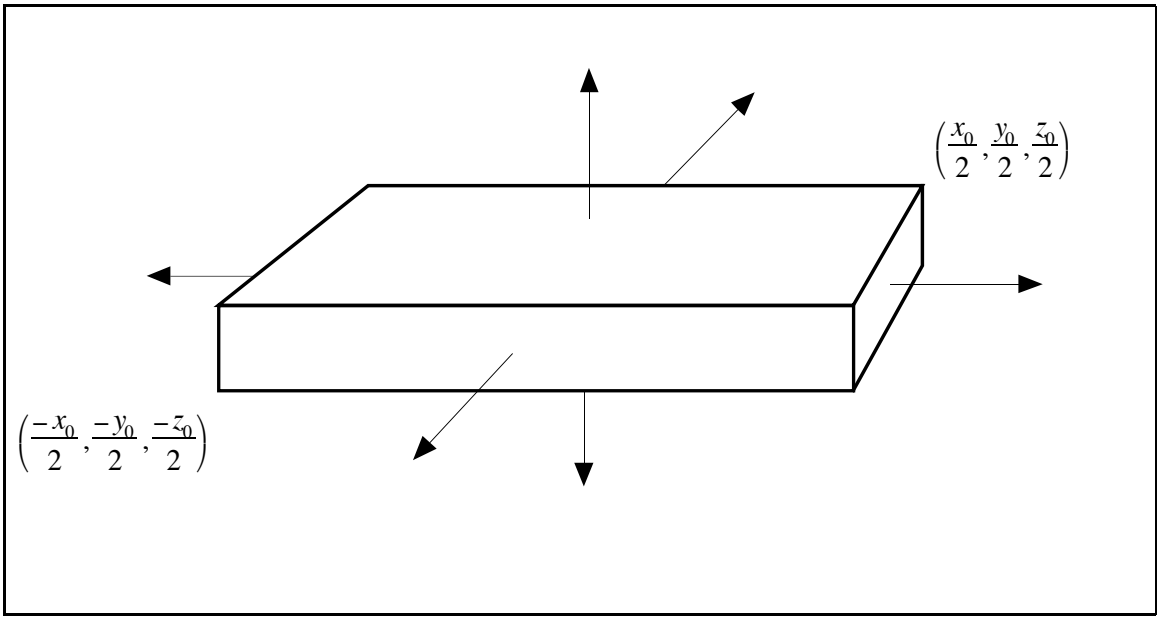


Figure 7: A rectangular block of constant unit density, with center of mass at (0,0,0).

5 Examples

5.1 Inertia Tensor of a Block

Let us calculate the inertia tensor I_{body} of the rectangular block in figure 7. The block has dimensions $x_0 \times y_0 \times z_0$. As required, the center of mass of the block is at the origin. Thus, the extent of the block is from $-\frac{x_0}{2}$ to $\frac{x_0}{2}$ along the x axis, and similarly for the y and z axes. To calculate the inertia tensor, we must treat the sums in equation (2–32) as integrals over the volume of the block. Let us assume that the block has constant unit density. This means that the density function $\rho(x, y, z)$ is always one. Since the block has volume $x_0 y_0 z_0$, the mass M of the block is $M = x_0 y_0 z_0$. Then, in

body space,

$$
\begin{aligned}
I_{xx} &= \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \int_{\frac{-y_0}{2}}^{\frac{y_0}{2}} \int_{\frac{-z_0}{2}}^{\frac{z_0}{2}} \rho(x,y,z)(y^2+z^2)\,dx\,dy\,dz = \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \int_{\frac{-y_0}{2}}^{\frac{y_0}{2}} \int_{\frac{-z_0}{2}}^{\frac{z_0}{2}} y^2+z^2\,dx\,dy\,dz \\
&= \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \int_{\frac{-y_0}{2}}^{\frac{y_0}{2}} y^2 z + \frac{z^3}{3}\Big|_{z=\frac{-z_0}{2}}^{z=\frac{z_0}{2}} dx\,dy \\
&= \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \int_{\frac{-y_0}{2}}^{\frac{y_0}{2}} y^2 z_0 + \frac{z_0^3}{12}\,dx\,dy \\
&= \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \frac{y_0^3}{3} z_0 + \frac{z_0^3}{12} y_0\Big|_{y=\frac{-y_0}{2}}^{y=\frac{y_0}{2}} dx \\
&= \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \frac{y_0^3 z_0}{12} + \frac{z_0^3 y_0}{12}\,dx \\
&= \frac{y_0^3 z_0}{12} + \frac{z_0^3 y_0}{12}\Big|_{x=\frac{-x_0}{2}}^{x=\frac{x_0}{2}} \frac{y_0^3 z_0 x_0}{12} + \frac{z_0^3 y_0 x_0}{12} = \frac{x_0 y_0 z_0}{12}(y_0^2+z_0^2) = \frac{M}{12}(y_0^2+z_0^2).
\end{aligned}
\tag{5–1}
$$

Similarly, $I_{yy} = \frac{M}{12}(x_0^2+z_0^2)$ and $I_{zz} = \frac{M}{12}(x_0^2+y_0^2)$. Now, the off-diagonal terms, such as I_{xy}, are

$$
I_{xy} = \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \int_{\frac{-y_0}{2}}^{\frac{y_0}{2}} \int_{\frac{-z_0}{2}}^{\frac{z_0}{2}} \rho(x,y,z)(xy)\,dx\,dy\,dz = \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \int_{\frac{-y_0}{2}}^{\frac{y_0}{2}} \int_{\frac{-z_0}{2}}^{\frac{z_0}{2}} xy\,dx\,dy\,dz = 0 \tag{5–2}
$$

(and similarly for the others) because the integrals are all symmetric. Thus, the inertia tensor of the block is

$$
I_{body} = \frac{M}{12}\begin{pmatrix} y_0^2+z_0^2 & 0 & 0 \\ 0 & x_0^2+z_0^2 & 0 \\ 0 & 0 & x_0^2+y_0^2 \end{pmatrix}. \tag{5–3}
$$

5.2 A Uniform Force Field

Suppose a uniform force acts on each particle of a body. For example, we typically describe a gravitational field as exerting a force $m_i g$ on each particle of a rigid body, where g is a vector pointing downwards. The net force F_g acting due to gravity on the body then is

$$
F_g = \sum m_i g = Mg \tag{5–4}
$$

which yields an acceleration of $\frac{Mg}{g} = g$ of the center of mass, as expected. What is the torque due to the gravitational field? The net torque is the sum

$$
\sum (r_i(t) - x(t)) \times m_i g = \left(\sum m_i (r_i(t) - x(t))\right) \times g = \mathbf{0} \tag{5–5}
$$

by equation (2–20). We see from this that a uniform gravitational field can have no effect on the angular momentum of a body. Furthermore, the gravitational field can be treated as a single force Mg acting on the body at its center of mass.

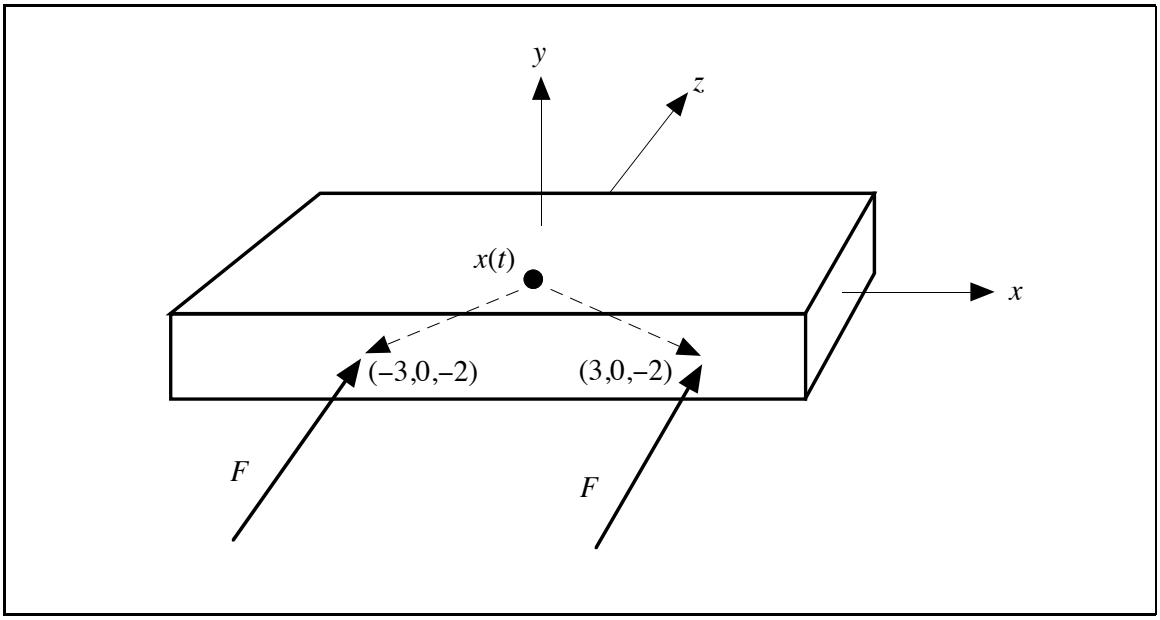


Figure 8: A block acted on by two equal forces F at two different points.

5.3 Rotation Free Movement of a Body

Now, let us consider some forces acting on the block of figure 8. Suppose that an external force $F = (0, 0, f)$ acts on the body at points $x(t) + (-3, 0, -2)$ and $x(t) + (3, 0, -2)$. We would expect that this would cause the body to accelerate linearly, without accelerating angularly. The net force acting on the body is $(0, 0, 2f)$, so the acceleration of the center of mass is

$$\frac{2f}{M}$$

along the z axis. The torque due to the force acting at $x(t) + (-3, 0, -2)$ is

$$((x(t) + \begin{pmatrix} -3 \\ 0 \\ -2 \end{pmatrix}) - x(t)) \times F = \begin{pmatrix} -3 \\ 0 \\ -2 \end{pmatrix} \times F$$

while the torque due to the force acting at $x(t) + (3, 0, -2)$ is

$$((x(t) + \begin{pmatrix} 3 \\ 0 \\ -2 \end{pmatrix}) - x(t)) \times F = \begin{pmatrix} 3 \\ 0 \\ -2 \end{pmatrix} \times F.$$

The total torque τ is therefore

$$\tau = \begin{pmatrix} -3 \\ 0 \\ -2 \end{pmatrix} \times F + \begin{pmatrix} 3 \\ 0 \\ -2 \end{pmatrix} \times F = (\begin{pmatrix} -3 \\ 0 \\ -2 \end{pmatrix} + \begin{pmatrix} 3 \\ 0 \\ -2 \end{pmatrix}) \times F = \begin{pmatrix} 0 \\ 0 \\ -2 \end{pmatrix} \times F.$$

But this gives

$$\tau = \begin{pmatrix} 0 \\ 0 \\ -2 \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ f \end{pmatrix} = \mathbf{0}.$$

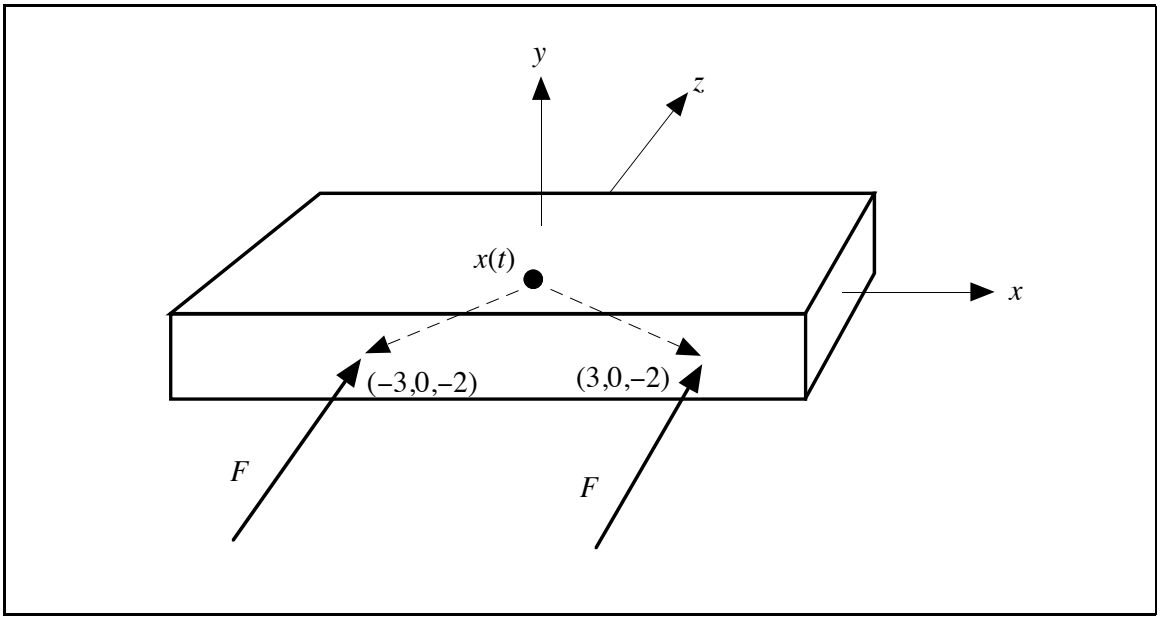


Figure 9: A block acted on by two opposite forces F_1 and $F_2 = -F_1$, at two different points.

As expected then, the forces acting on the block impart no angular acceleration to the block.

5.4 Translation Free Movement of a Body

Suppose now that an external force $F_1 = (0, 0, f)$ acts on the body at point $x(t) + (-3, 0, -2)$ and an external force $F_2 = (0, 0, -f)$ acts on the body at point $x(t) + (3, 0, 2)$ (figure 9). Since $F_1 = -F_2$, the net force acting on the block is $F_1 + F_2 = \mathbf{0}$, so there is no acceleration of the center of mass. On the other hand, the net torque is

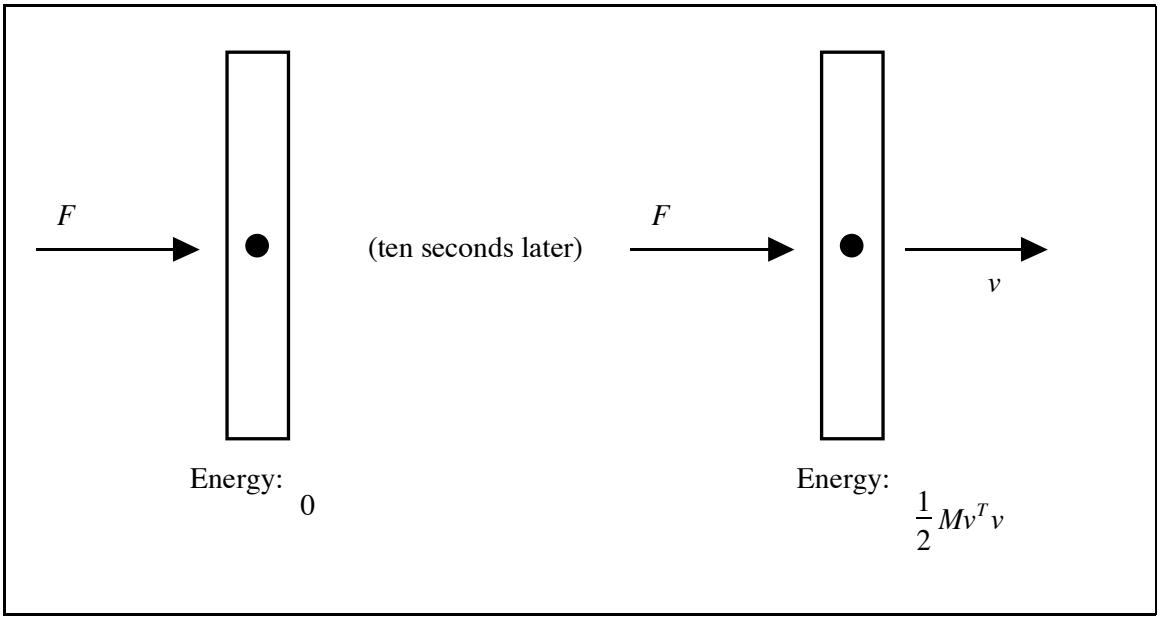


Figure 10: A rectangular block acted on by a force through its center of mass.

$$
\begin{aligned}
((x(t)+\begin{pmatrix}-3\\0\\-2\end{pmatrix})-x(t))\times F_1+&\\
((x(t)+\begin{pmatrix}3\\0\\2\end{pmatrix})-x(t))\times F_2 &= \begin{pmatrix}-3\\0\\-2\end{pmatrix}\times\begin{pmatrix}0\\0\\f\end{pmatrix}+\begin{pmatrix}3\\0\\-2\end{pmatrix}\times\begin{pmatrix}0\\0\\-f\end{pmatrix} \qquad (5\text{–}6)\\
&= \begin{pmatrix}0\\3f\\0\end{pmatrix}+\begin{pmatrix}0\\3f\\0\end{pmatrix}=\begin{pmatrix}0\\6f\\0\end{pmatrix}.
\end{aligned}
$$

Thus, the net torque is $(0, 6f, 0)$, which is parallel to the y axis. The final result is that the forces acting on the block cause it to angularly accelerate about the y axis.

5.5 Force vs. Torque Puzzle

In considering the effect of a force acting at a point on a body, it sometimes seems that the force is being considered twice. That is, if a force F acts on a body at a point $r + x(t)$ in space, then we first consider F as accelerating the center of mass, and then consider F as imparting a spin to the body.

This gives rise to what at first seems a paradox: Consider the long horizontal block of figure 10 which is initially at rest. Suppose that a force F acts on the block at the center of mass for some period of time, say, ten seconds. Since the force acts at the center of mass, no torque is exerted on the body. After ten seconds, the body will have acquired some linear velocity v. The body will not have acquired any angular velocity; thus the kinetic energy of the block will be $\frac{1}{2}M|v|^2$.

Now suppose that the same force F is applied off-center to the body as shown in figure 11. Since the force acting on the body is the same, the acceleration of the center of mass is the same. Thus,

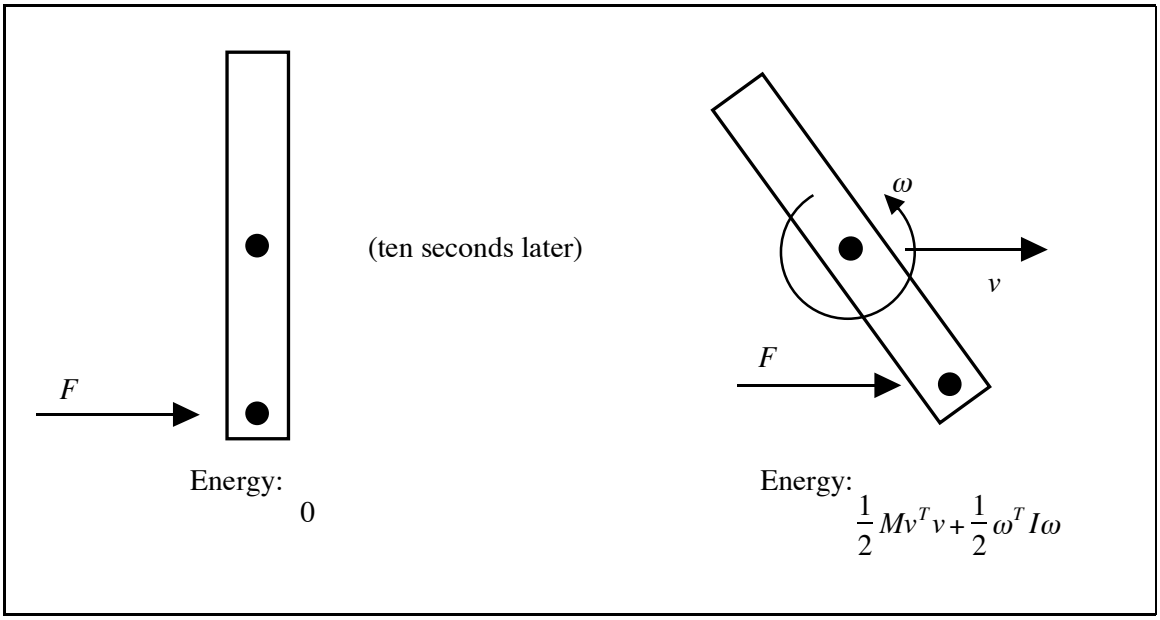


Figure 11: A block acted on by a force, off-center of the center of mass.

after ten seconds, the body will again have linear velocity v. However, after ten seconds, the body will have picked up some angular velocity ω, since the force F, acting off center, now exerts a torque on the body. Since the kinetic energy is (see appendix C)

$$\tfrac{1}{2}M|v|^2+\tfrac{1}{2}\omega^TI\omega$$

the kinetic energy of the block is higher than when the force acted through the center of mass. But if identical forces pushed the block in both cases, how can the energy of the block be different? Hint: Energy, or work, is the integral of force over distance.

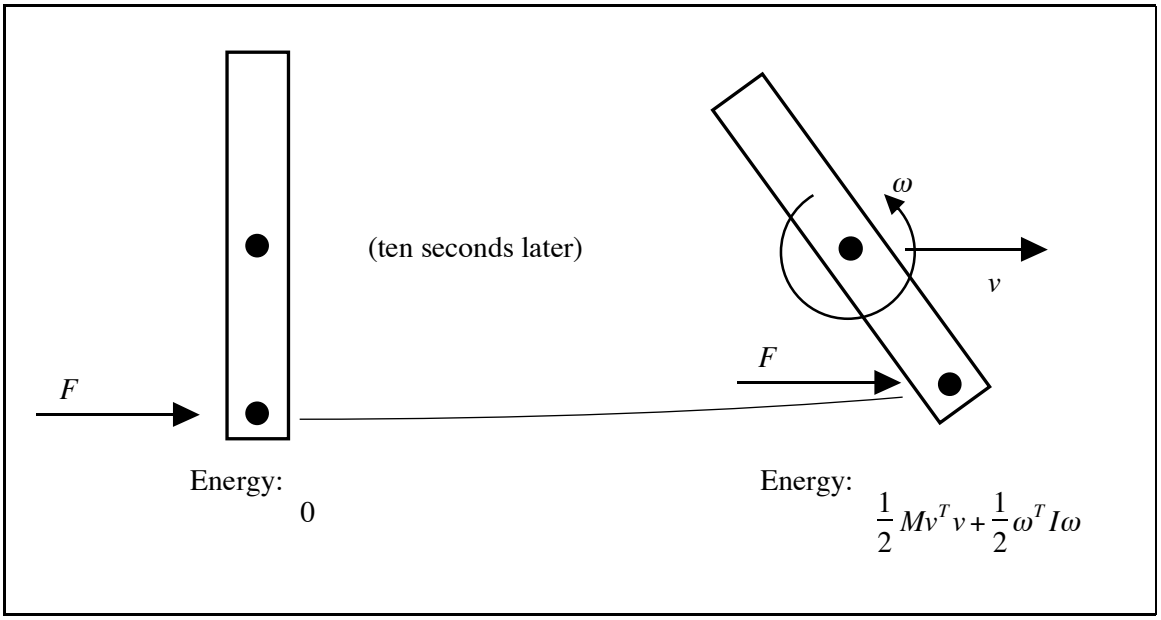


Figure 12: The path the force acts over is longer than in figure 10. As a result, the force does more work, imparting a larger kinetic energy to the block.

Figure 12 shows why the force acting off center results in a higher kinetic energy. The kinetic energy of the block is equivalent to the work done by the force. The work done by the force is the integral of the force over the path traveled in applying that force. In figure 11, where the force acts off the center of mass, consider the path traced out by the point where the force is applied. This path is clearly longer than the path taken by the center of mass in figure 10. Thus, when the force is applied off center, more work is done because the point p at which the force is applied traces out a longer path then when the force is applied at the center of mass.

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 $\mathbf{X}(t)$ always varies smoothly. Clearly, requiring $\mathbf{X}(t)$ 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, $\mathbf{X}(t_c)$, and compute how the velocities of bodies involved in the collision must change. We'll call the state reflecting these new velocities $\mathbf{X}(t_c)^+$. Note that $\mathbf{X}(t_c)$ and $\mathbf{X}(t_c)^+$ 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 $\mathbf{X}(t_c)$, 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 `ComputeForceAndTorque`.

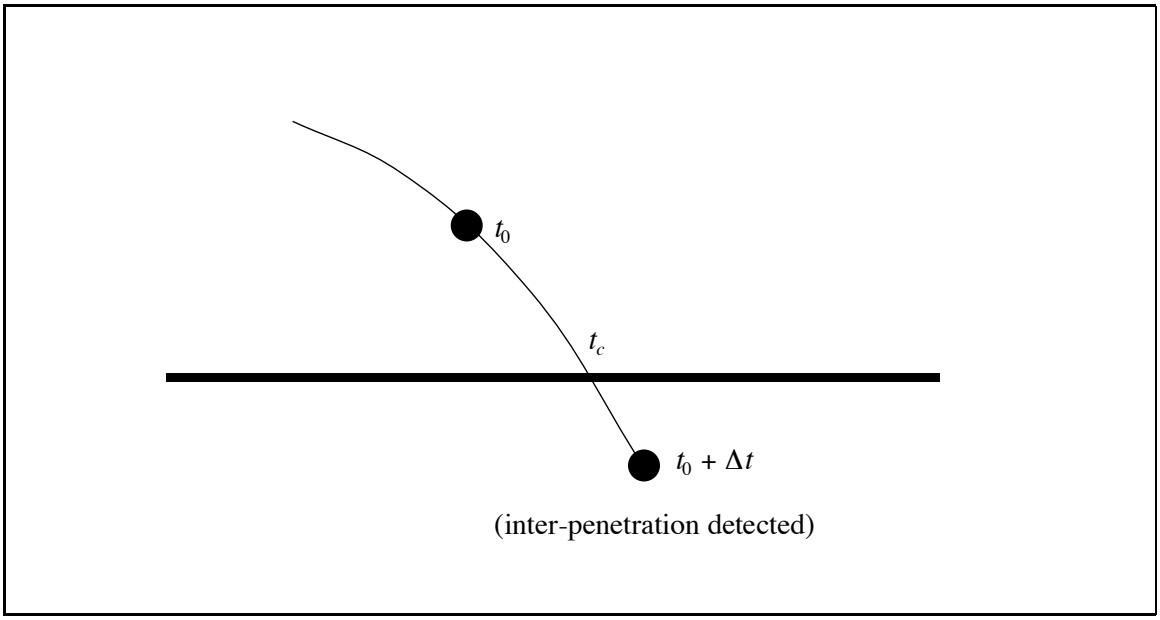


Figure 13: At time $t_0 + \Delta t$, the particle is found to lie below the floor. Thus, the actual time of collision t_c lies between the time of the last known legal position, t_0, and $t_0 + \Delta t$.

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 $\mathbf{X}(t_0)$ and $\mathbf{X}(t_0 + \Delta t)$. 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.

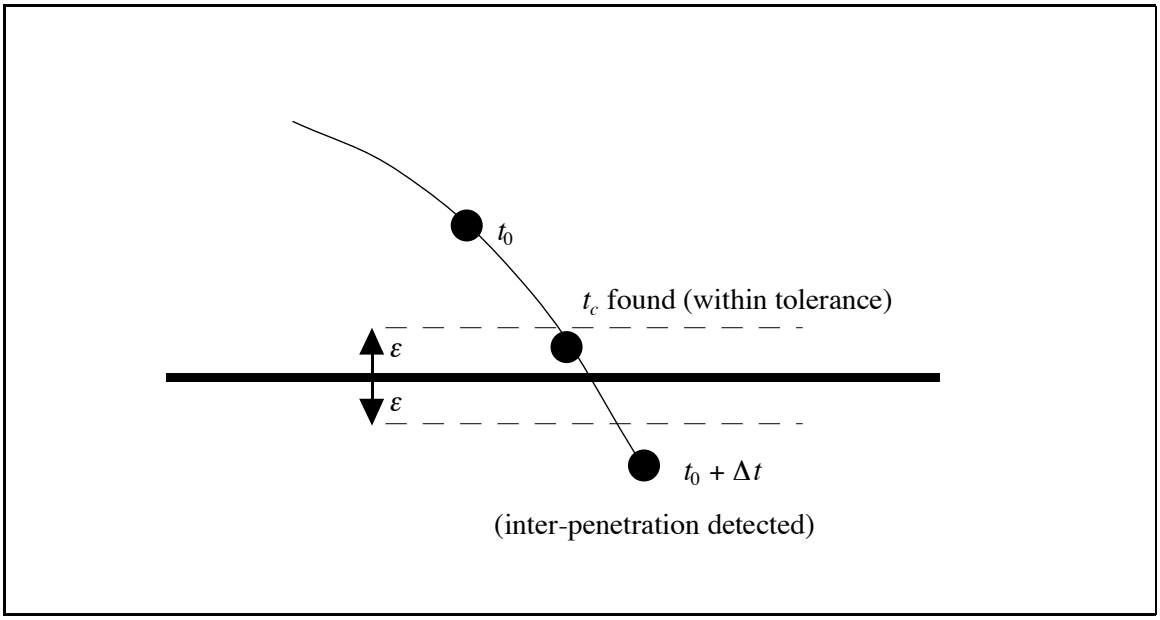


Figure 14: When the particle is found to be within some tolerance ϵ of contacting the floor, then t_c is considered to have been computed to within sufficient accuracy.

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 $\mathbf{X}(t)$ 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, $\frac{d}{dt}\mathbf{X}(t)$, 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 $\mathbf{X}$ 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-

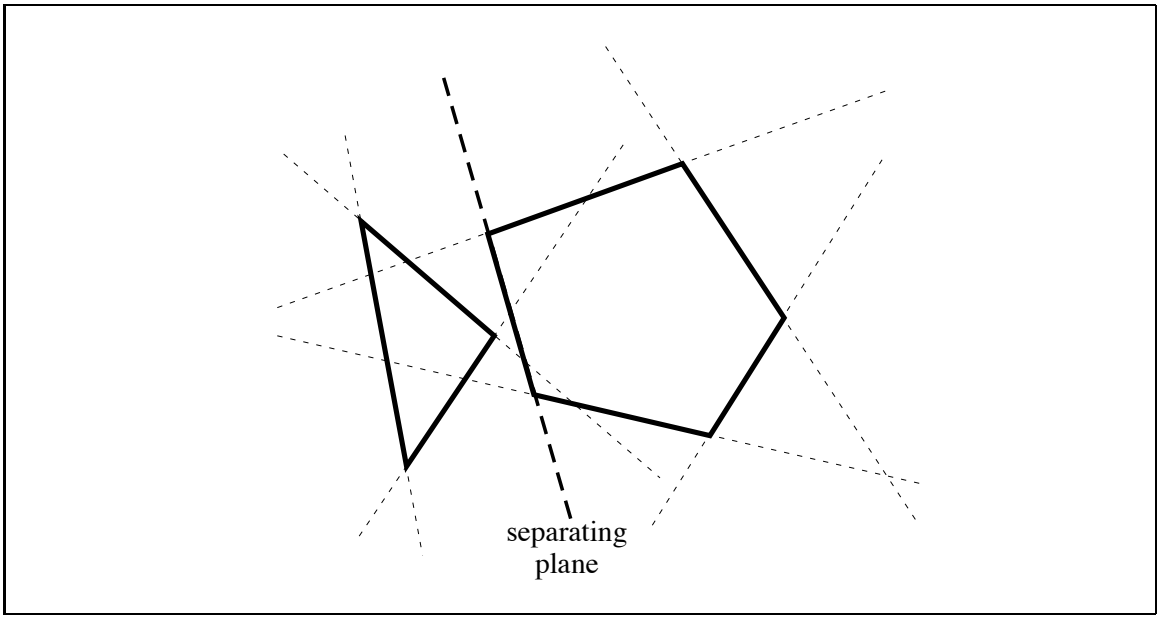


Figure 15: Exhaustive search for a separating plane. Only one face of the two polygons forms a separating plane.

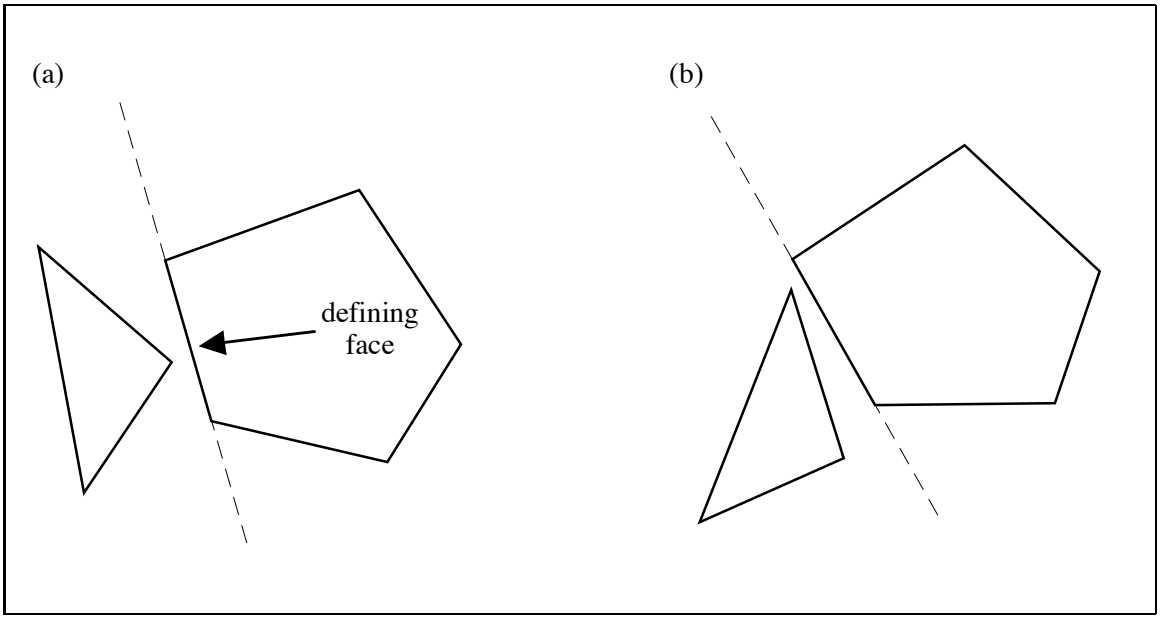


Figure 16: (a) At this time step, the separating plane is defined by a face of one of the polygons. (b) At the next time step, the polygons have moved, but the same face still defines a separating plane.

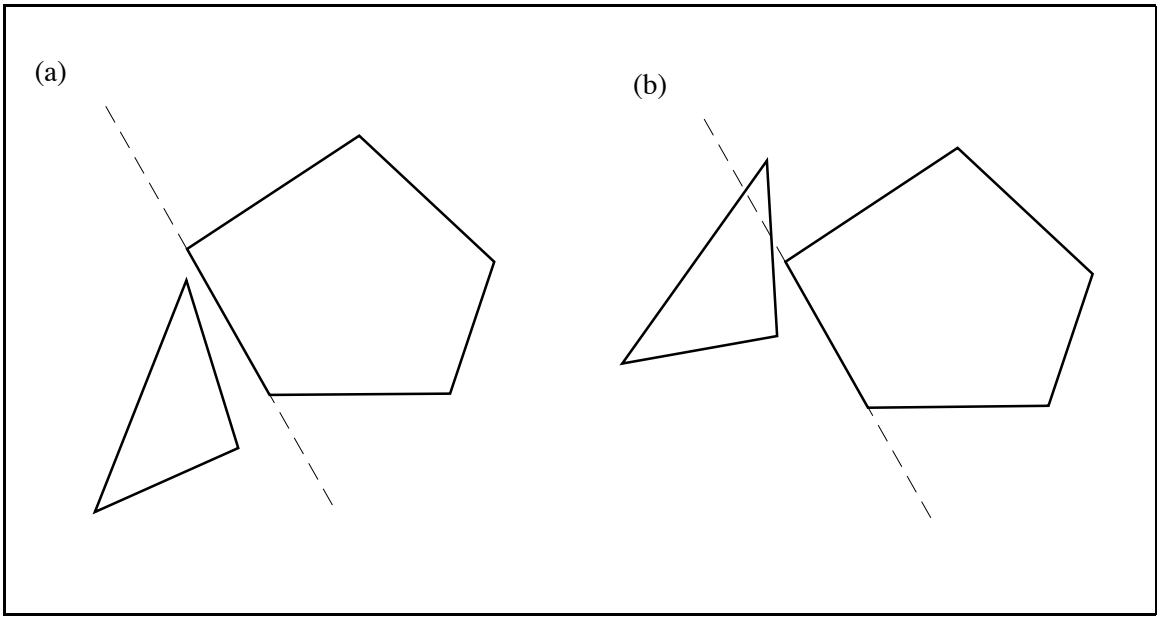


Figure 17: The face that has been defining a separating plane no longer does so, and a new separating plane must be found.

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 $\frac{d}{dt}\mathbf{X}(t)$ 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.

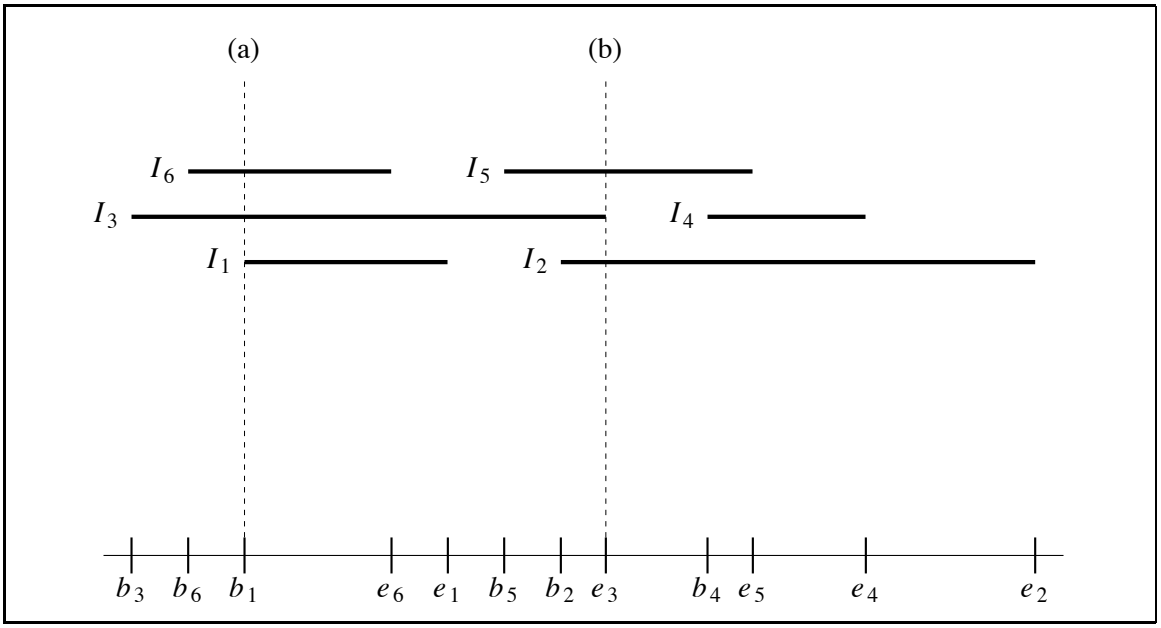


Figure 18: The sweep/sort algorithm. (a) When b_1 is encountered, the active list contains intervals 3 and 6; interval 1 is reported to overlap with these two intervals. Interval 1 is added to the active list and the algorithm continues. (b) When e_3 is encountered, the active list contains intervals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

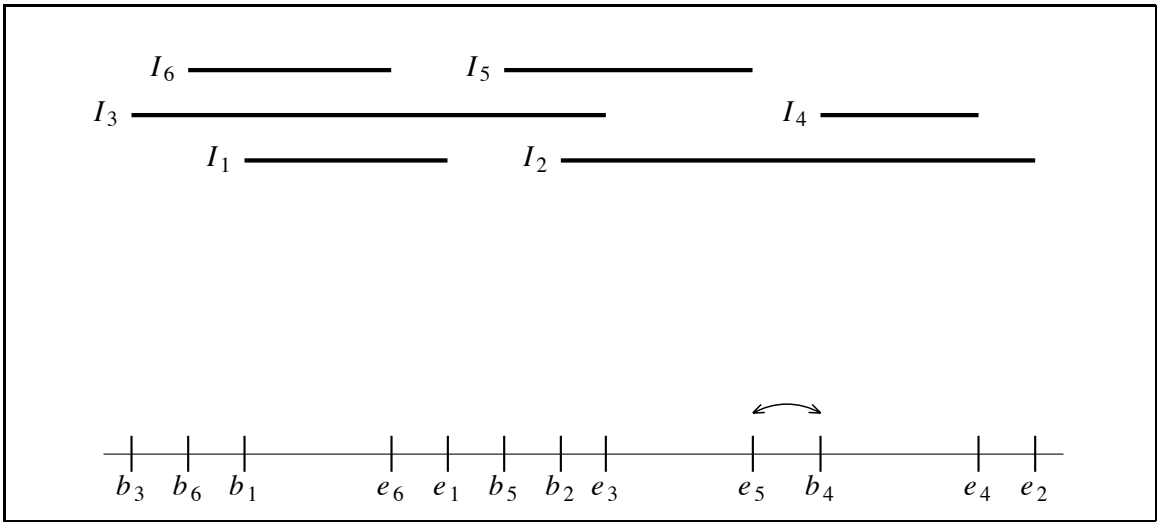


Figure 19: A coherence-based method of detecting overlaps. The order produced in figure 18 is nearly correct for this arrangement of intervals. Only b_4 and e_5 need to be exchanged. When the exchange occurs, the change in overlap status between interval 4 and 5 is detected.

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 interpenetrating, 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 */
    bool            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 edge of body A (pointed to by `a`). Similarly, `eb` is a unit vector giving the direction that the contact

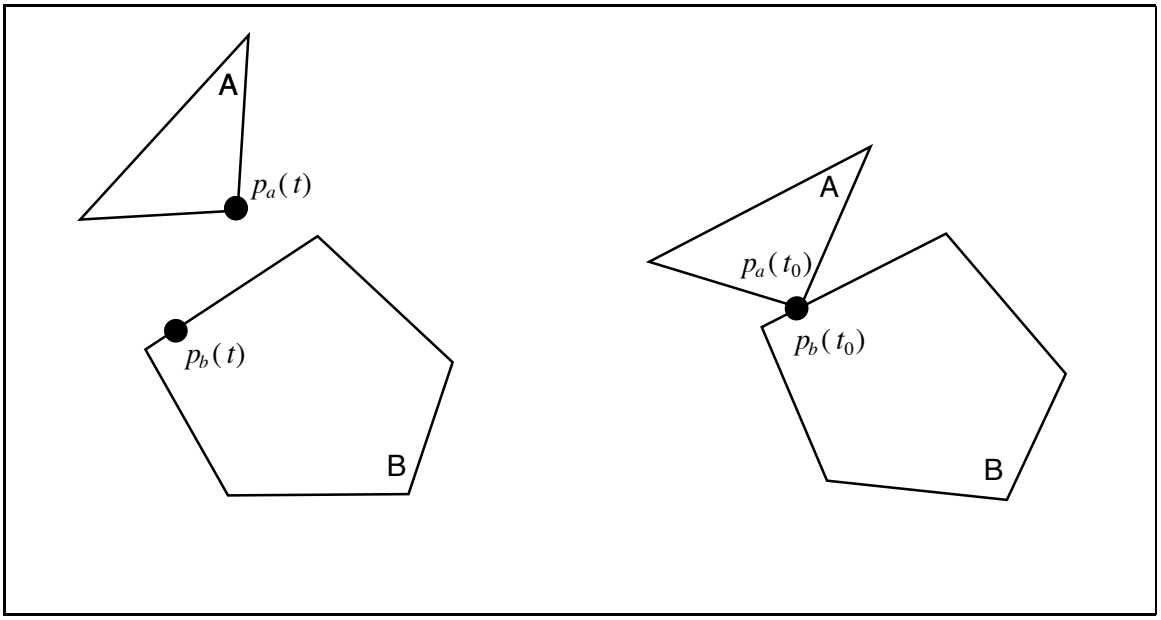


Figure 20: (a) The points $p_a(t)$ and $p_b(t)$ for a vertex/face contact. (b) At time t_0, the bodies come into contact at $p_a(t_0) = p_b(t_0)$.

edge 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

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

Let's examine the quantity

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

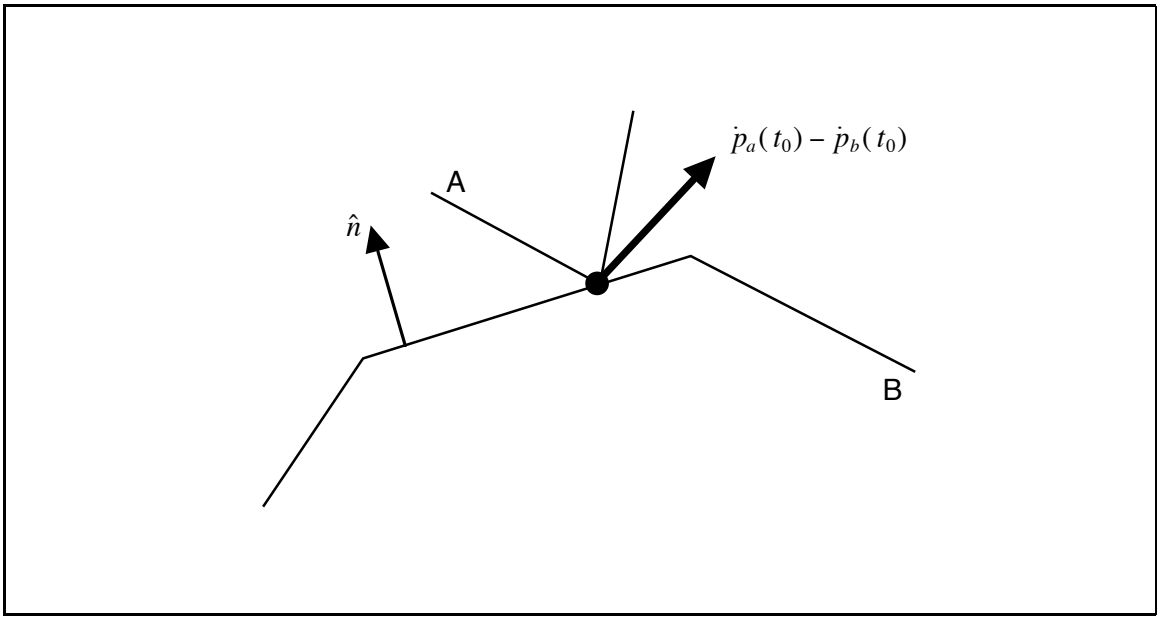


Figure 21: The vector $\dot{p}_a(t_0) - \dot{p}_b(t_0)$ points in the same direction as $\hat{n}(t_0)$; the bodies are separating.

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.

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

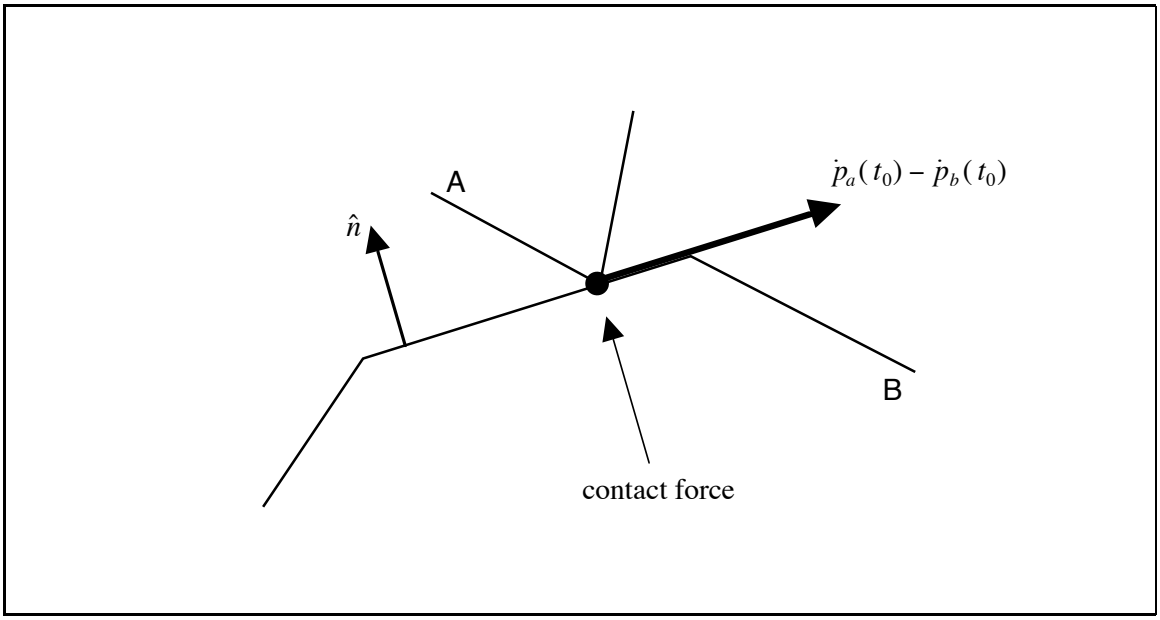


Figure 22: The vector $\dot{p}_a(t_0) - \dot{p}_b(t_0)$ is perpendicular to $\hat{n}(t_0)$; the bodies are in resting contact. A contact force may be necessary to prevent bodies from accelerating towards each other.

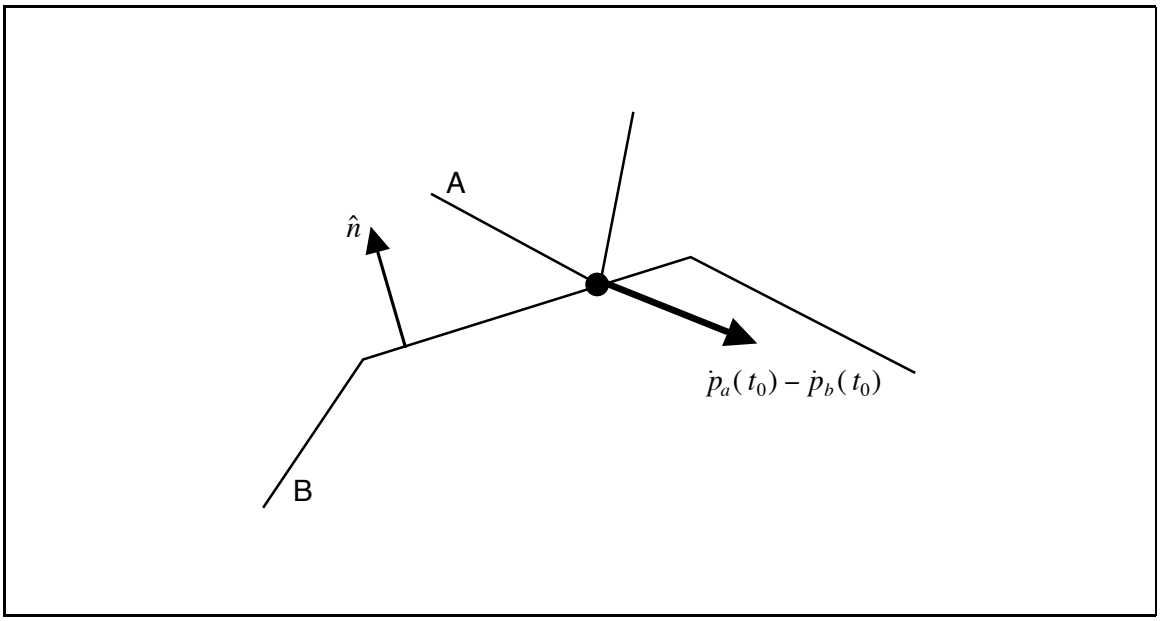


Figure 23: Colliding contact. The relative velocity $\dot{p}_a(t_0) - \dot{p}_b(t_0)$ is directed inwards, opposite $\hat{n}(t_0)$. Unless the relative velocity is abruptly changed, inter-penetration will occur immediately after time t_0.

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 $0 \leq \epsilon \leq 1$. 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

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

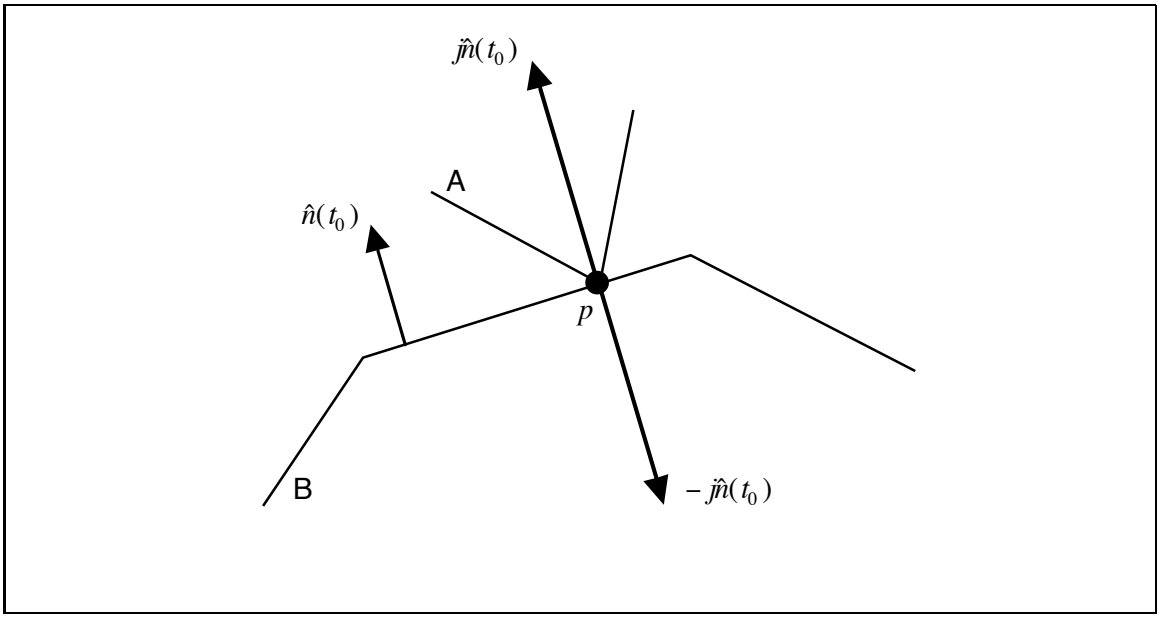


Figure 24: The impulse between two bodies at a contact point. An impulse of $j\hat{n}(t_0)$ acts on A, while an impulse of $-j\hat{n}(t_0)$ acts on B.

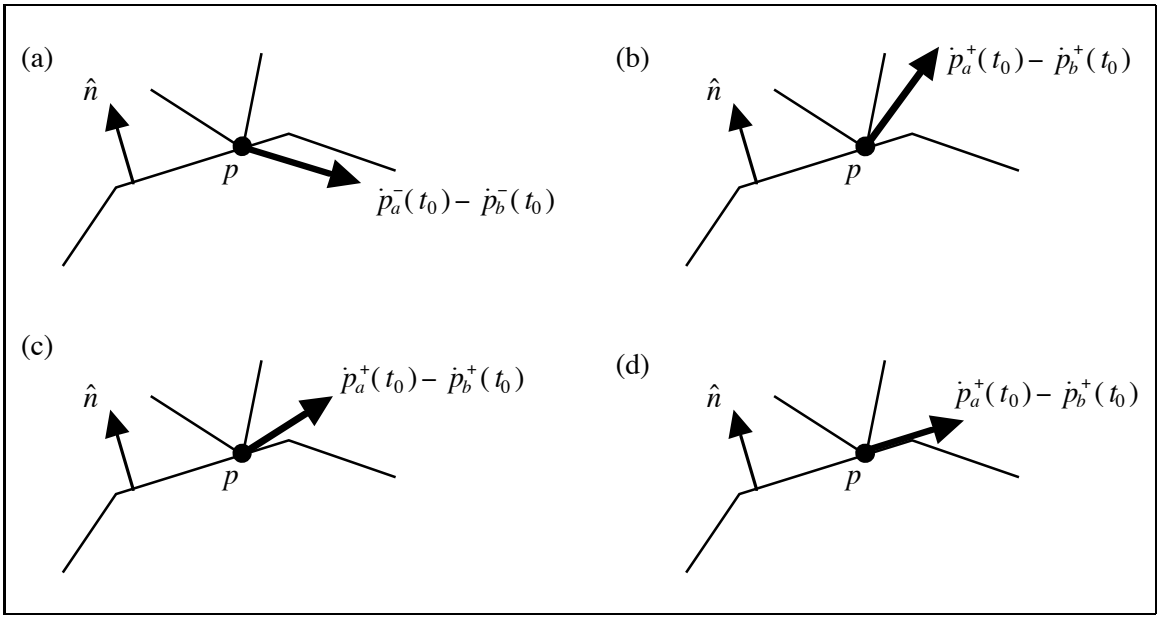


Figure 25: (a) The relative velocity before application of the impulse. (b) The component of the relative velocity in the $\hat{n}(t_0)$ direction is reversed for an $\epsilon = 1$ collision. The relative velocity perpendicular to $\hat{n}(t_0)$ remains the same. (c) A collision with $0 < \epsilon < 1$. 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

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

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.
 */
bool colliding(Contact *c)
{
    triple  padot = pt_velocity(c->a, p),   /* pdot_a^-(t0) */
            pbdot = pt_velocity(c->b, p);   /* pdot_b^-(t0) */
    double  vrel = c->n * (padot - pbdot);  /* v_rel^- */

    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, c->p), /* pdot_a^-(t0) */
            pbdot = pt_velocity(c->b, c->p), /* pdot_b^-(t0) */
            n = c->n,                        /* n^(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 FindAllCollisions(Contact contacts[], int ncontacts)
    {
        bool    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 `FindAllCollisions()`, 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

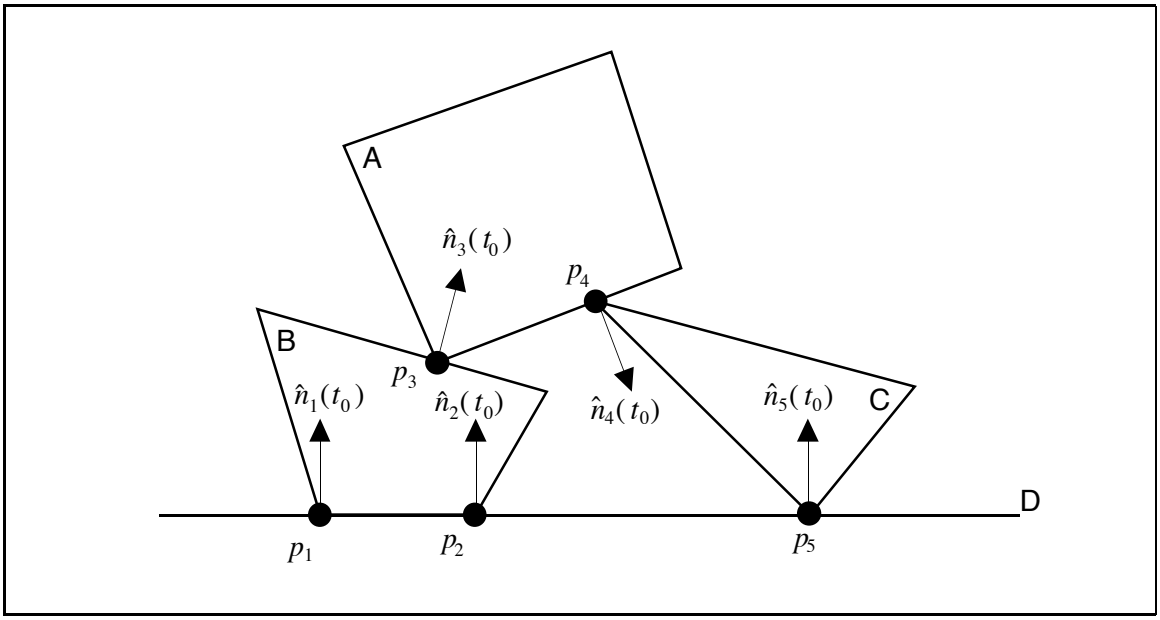


Figure 26: Resting contact. This configuration has five contact points; a contact force acts between pairs of bodies at each contact point.

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 $\hat{n}_i(t)$

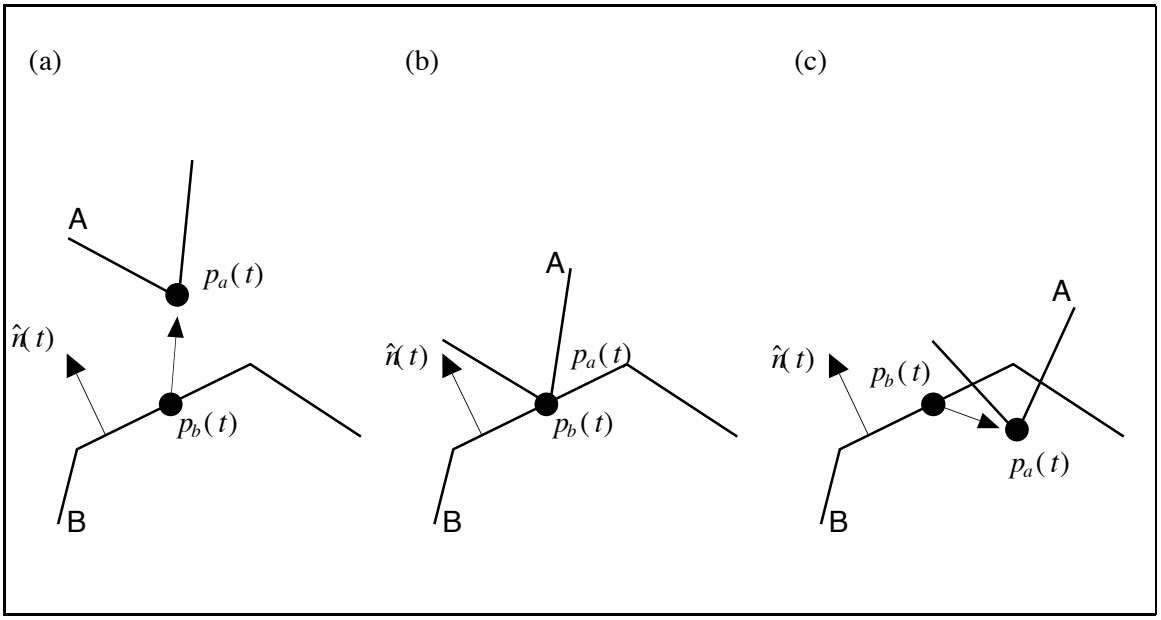


Figure 27: (a) The displacement $p_a(t) - p_b(t)$, indicated by an arrow, points in the same direction as $\hat{n}(t)$. Thus, the distance function $d(t)$ would be positive. (b) The distance function $d(t)$ is zero. (c) The displacement $p_a(t) - p_b(t)$ points in the opposite direction as $\hat{n}(t)$.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

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

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 `ComputeForceAndTorque()`, after `FindAllCollisions()` has been called.

```
void ComputeContactForces(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
    *
    *       ComputeExternalForceAndTorqueForAllBodies(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,

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

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 right-most 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 $(I_a^{-1}(t_0)(\text{Ł}_a(t_0) \times \omega_a(t_0))) \times r_a$. 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

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

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 computeNdot(Contact *c)
{
    if(c->vf)     /* vertex/face contact */
    {
        /* The vector 'n' is attached to B, so... */
        return c->b->omega ^ 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 ^ ea,     /* ėa */
                ebdot = c->b->omega ^ eb;     /* ėb */
                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,          /* n̂i(t0) */
                ra = c->p - A->x,     /* p − xa(t0) */
                rb = c->p - B->x;     /* p − xb(t0) */


        /* 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: '^' 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 p̈_a(t_0) due to the external
           force and torque, and similarly for p̈_b(t_0). */

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

        /* Compute the part of p̈_a(t_0) due to velocity,
           and similarly for p̈_b(t_0). */

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

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


        /* Combine the above results, and dot with n̂_i(t_0) */
        double  k1 = n * ((a_ext_part + a_vel_part) -
                          (b_ext_part + b_vel_part));
        triple  ndot = computeNdot(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.