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

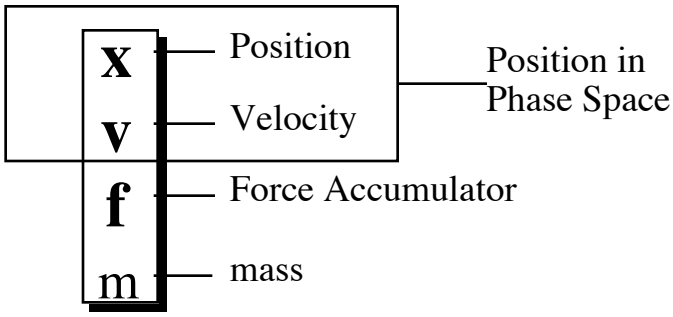

Figure 1: A particle may be represented by a structure containing its position, velocity, force, and mass. The six-vector formed by concatenating the position and velocity comprises the point's position in phase space.

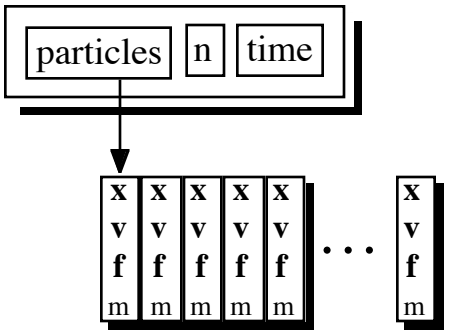

Figure 2: A bare particle system is essentially just a list of particles.

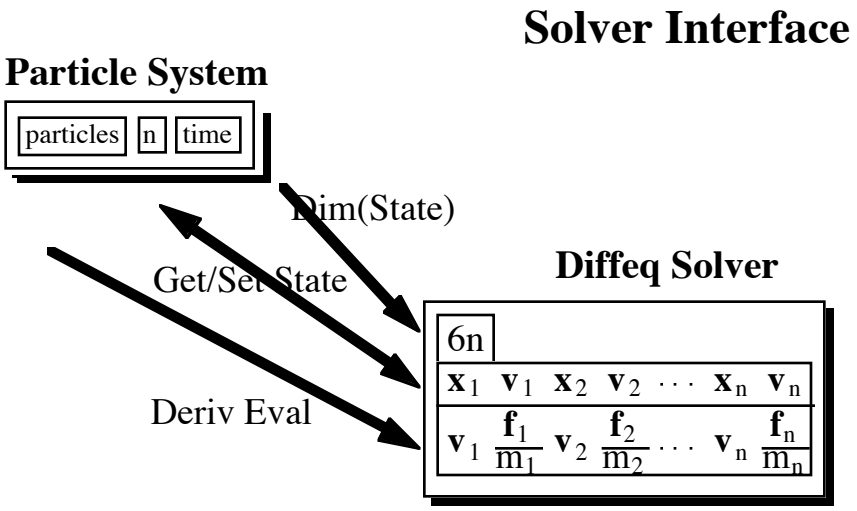

Figure 3: The relation between a particle system and a differential equation solver.

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

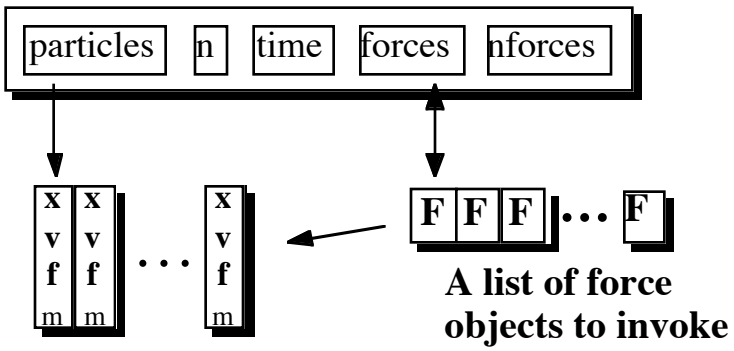

Figure 4: A particle system augmented to contain a list of *force objects*. Each force object points at the particles that it influences, and contains a function that knows how to compute the force on each affected particle.

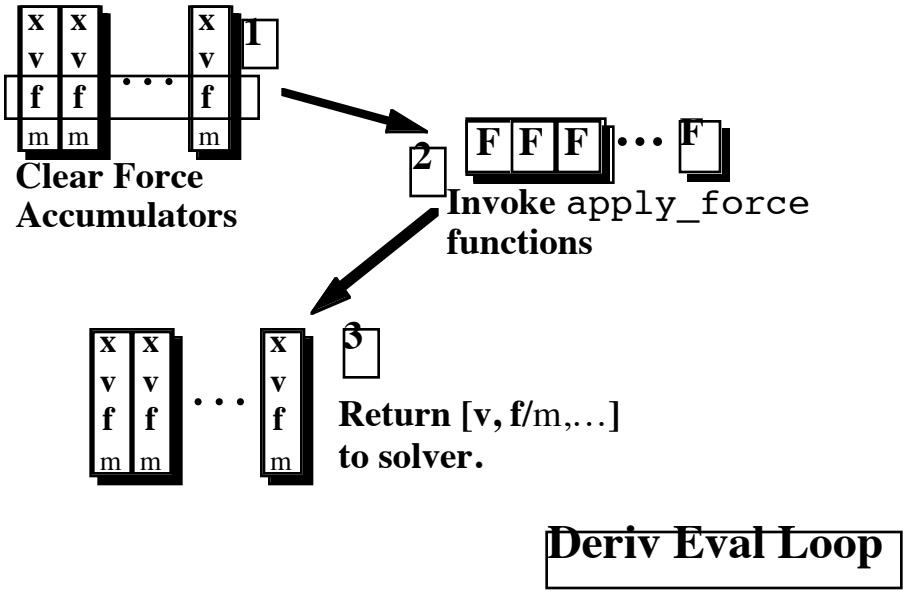

Figure 5: The derivative evaluation loop for a particle system with force objects.

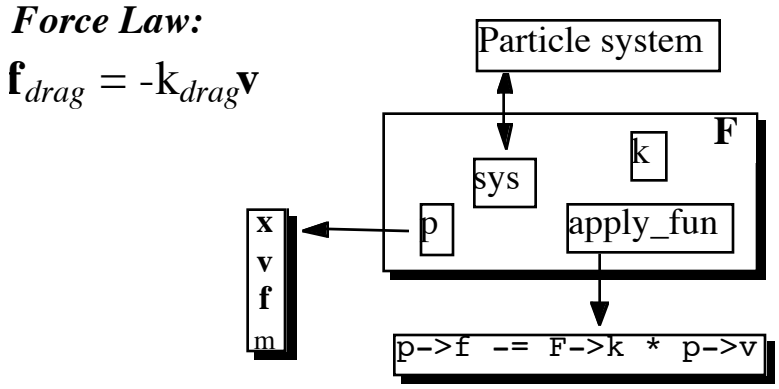

Figure 6: Schematic view of a force object implementing viscous drag. The object points at the particle to which drag is being applied, and also points to a function that implements the force law for drag.

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

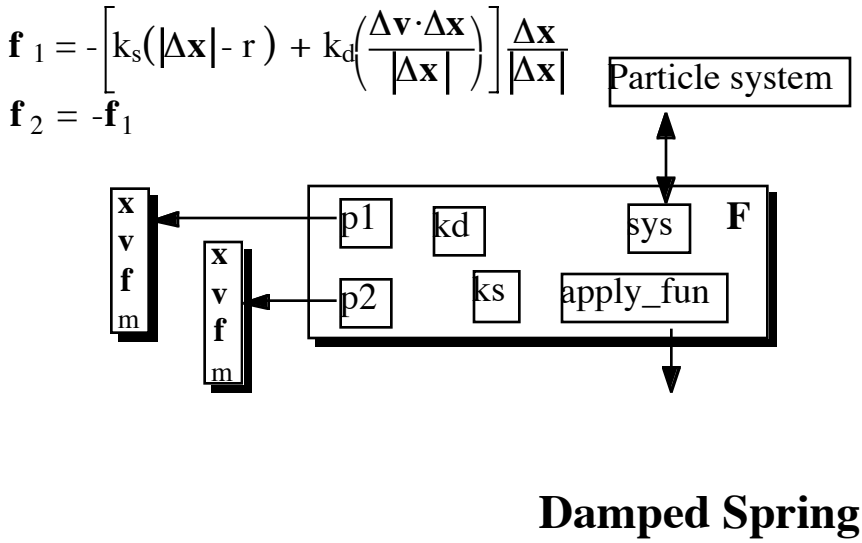

Figure 7: A schematic view of a force object implementing a damped spring that attaches particles p_1 and p_2.

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.