Differential Equation Basics

Andrew Witkin and David Baraff
School of Computer Science
Carnegie Mellon University

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 $x = e^{-kt}$.

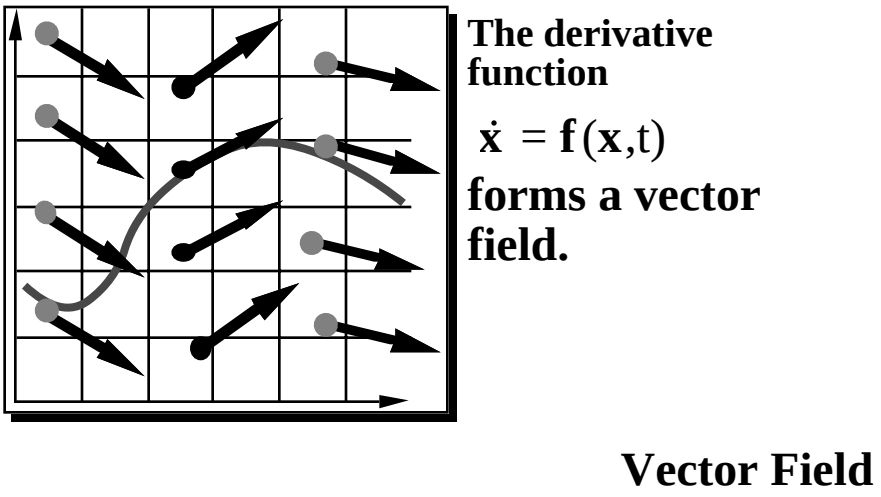


Figure 1: The derivative function $f(\mathbf{x}, t)$. defines a vector field.

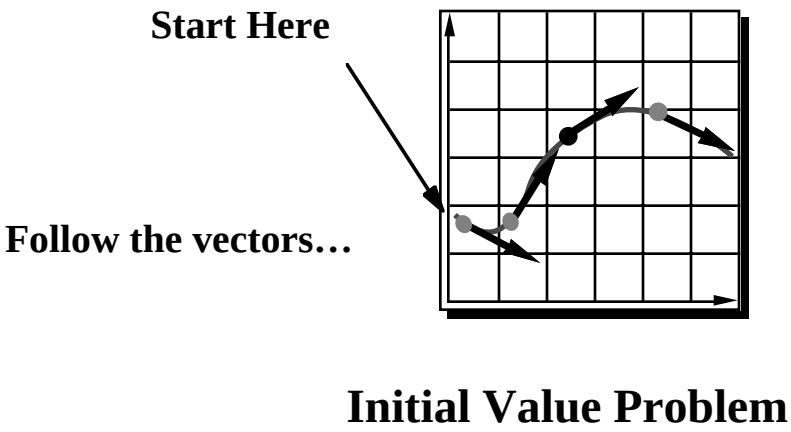
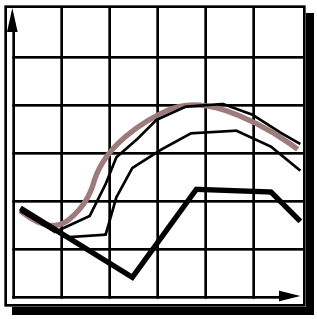


Figure 2: An initial value problem. Starting from a point $\mathbf{x}_0$, move with the velocity specified by the vector field.



- **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.

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 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.

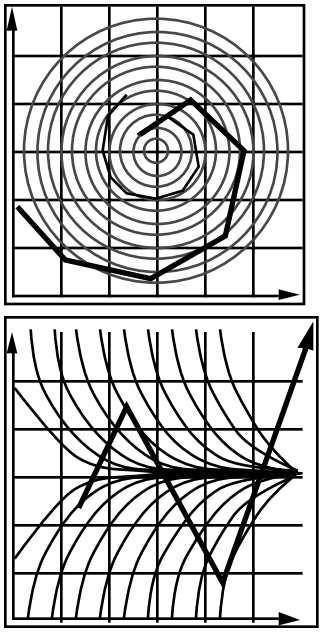
2.1 Euler's Method

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.



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.

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 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

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

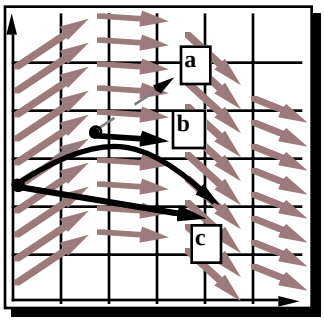
where $\mathbf{x}_0 = \mathbf{x}(t_0)$. We can now multiply both sides by h (turning the $O(h^2)$ term into $O(h^3)$) and rearrange, yielding

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

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



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.

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.

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:

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

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.

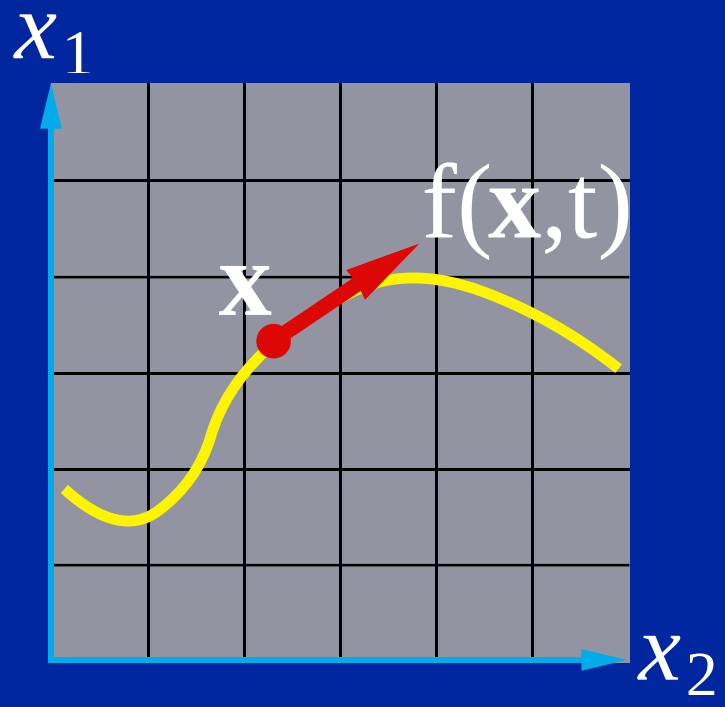
References

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

Differential Equation Basics

Andrew Witkin
Carnegie Mellon University

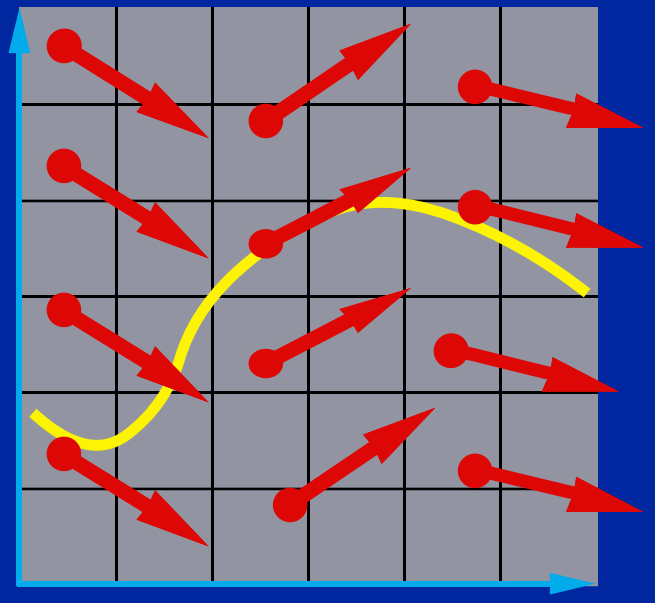
A Canonical Differential Equation



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

- **x(t): a moving point.**
- **f(x,t): x's velocity.**

Vector Field



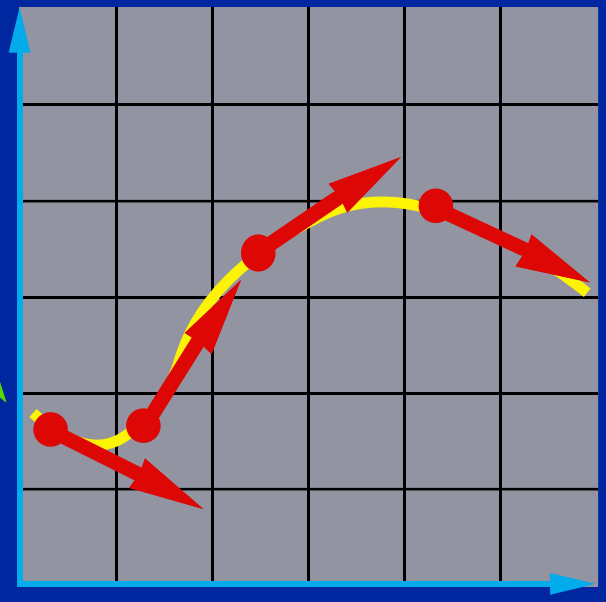
The differential equation

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

defines a vector field over x.

Integral Curves

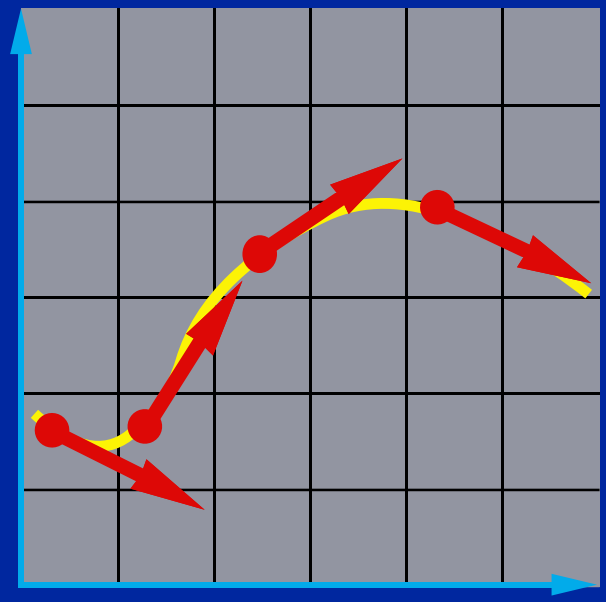
Start Here



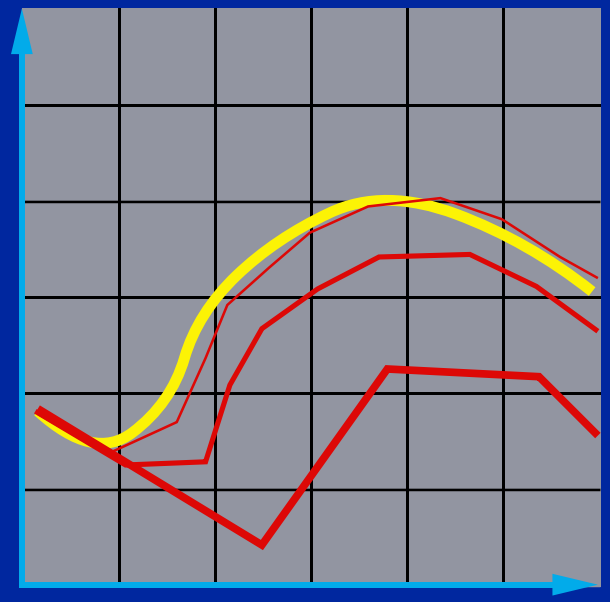
Pick any starting point,
and follow the vectors.

Initial Value Problems

Given the starting point, follow the integral curve.



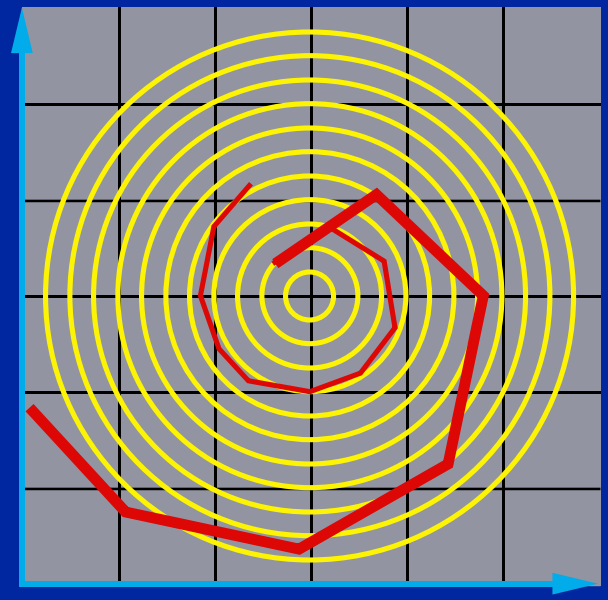
Euler's Method



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

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

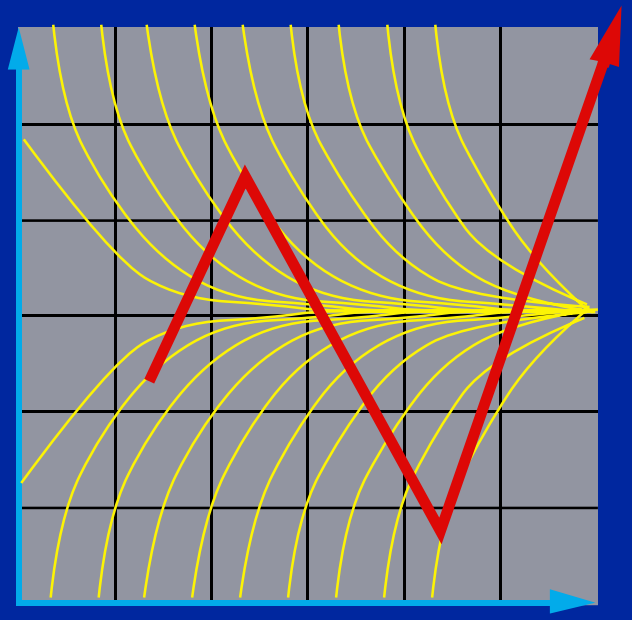
Problem I: Inaccuracy



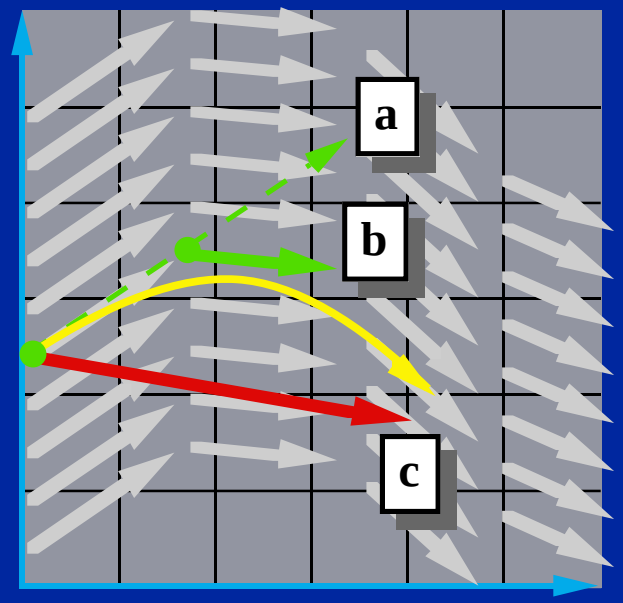
Error turns x(t) from a circle into the spiral of your choice.

Problem II: Instability

to Neptune!



The Midpoint 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

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

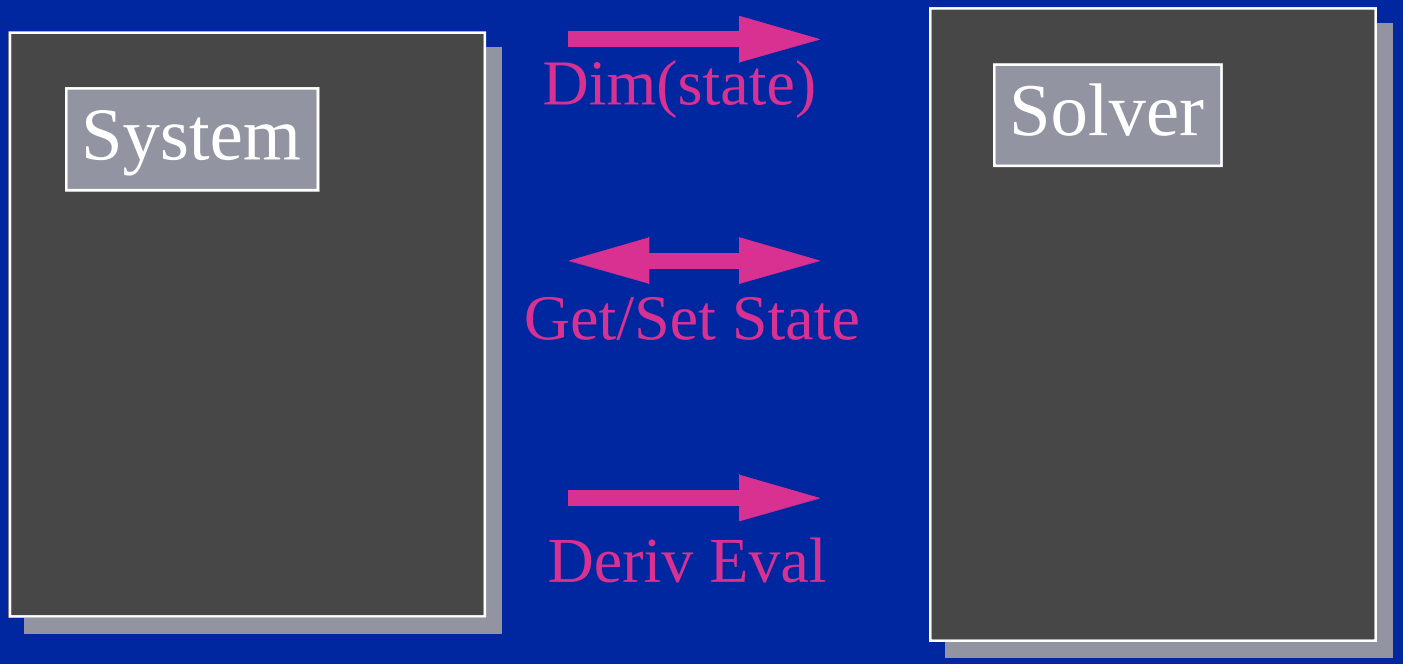
More methods…

- **Euler's method is *1st Order*.**
- **The midpoint method is *2nd Order*.**
- **Just the tip of the iceberg. See *Numerical Recipes* for more.**
- **Helpful hints:**
  - ***Don't*** **use Euler's method (you will anyway.)**
  - ***Do*** **Use adaptive step size.**

Modular Implementation

- Generic operations:
  - Get dim(x)
  - Get/set x and t
  - Deriv Eval at current (x,t)
- Write solvers in terms of these.
  - Re-usable solver code.
  - Simplifies model implementation.

Solver Interface



A Code Fragment

```
void euler_step(sys,h){
   float time;
   get_state(sys,temp1, &time);
   deriv_eval(sys,temp2);
   vtimes(h,temp2);
   vadd(temp2,temp1);*/
   set_state(sys,temp1,time + h);
   }
```

Particle System Dynamics

Andrew Witkin
School of Computer Science
Carnegie Mellon University

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

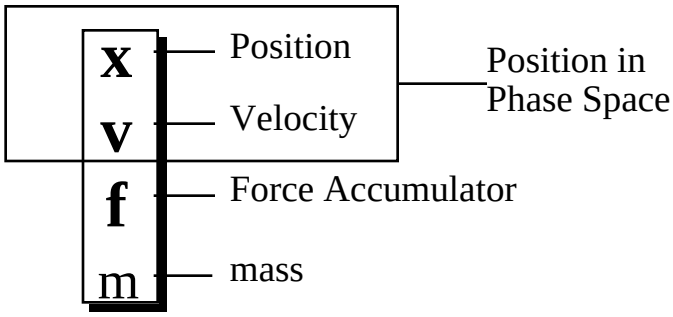


Figure 1: A particle may be represented by a structure containing its position, velocity, force, and mass. The six-vector formed by concatenating the position and velocity comprises the point's position in phase space.

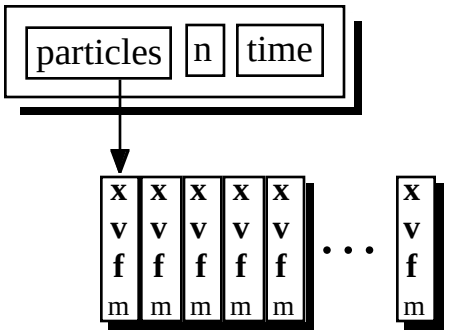


Figure 2: A bare particle system is essentially just a list of particles.

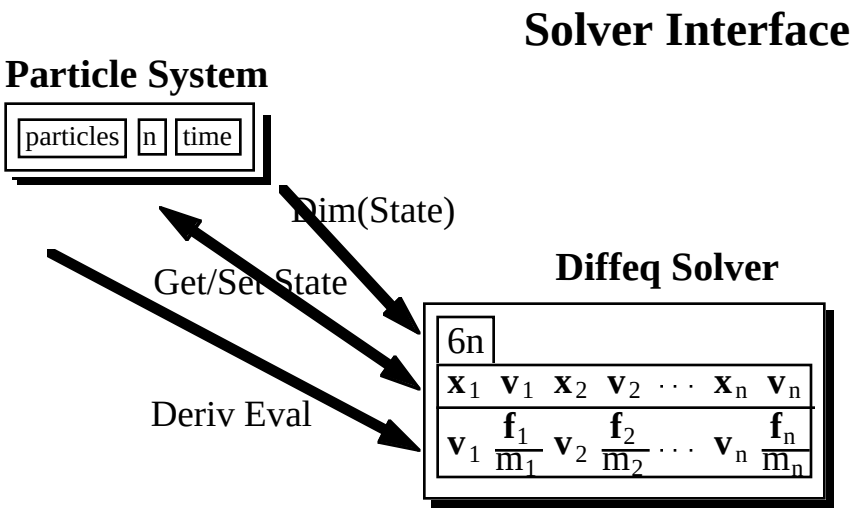


Figure 3: The relation between a particle system and a differential equation solver.

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

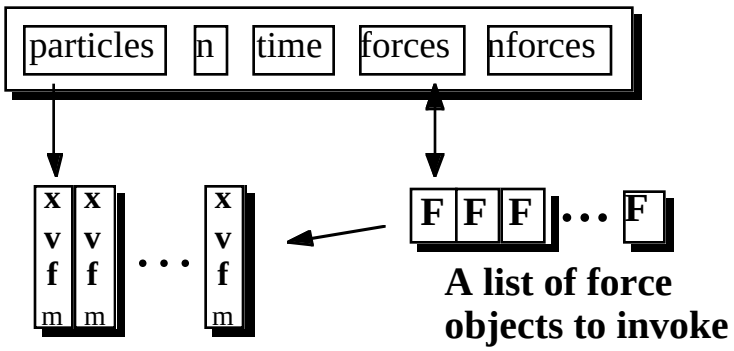


Figure 4: A particle system augmented to contain a list of *force objects.* Each force object points at the particles that it influences, and contains a function that knows how to compute the force on each affected particle.

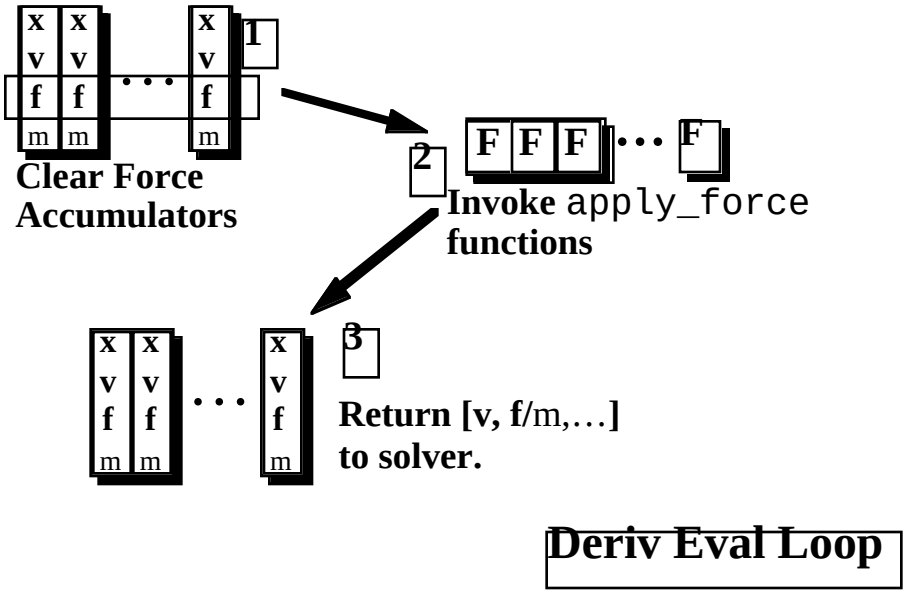
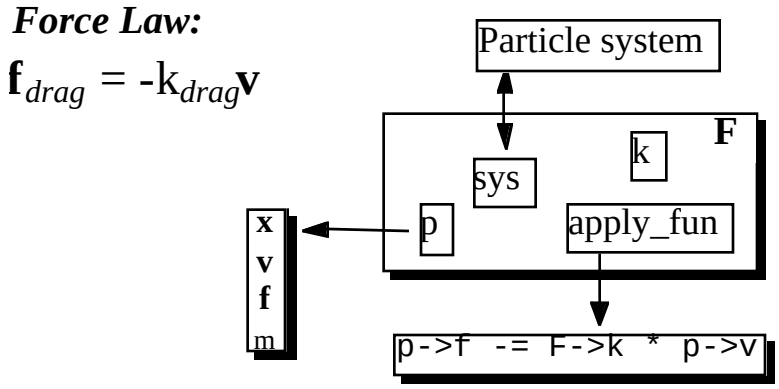


Figure 5: The derivative evaluation loop for a particle system with force objects.



A Force Object: Viscous Drag

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

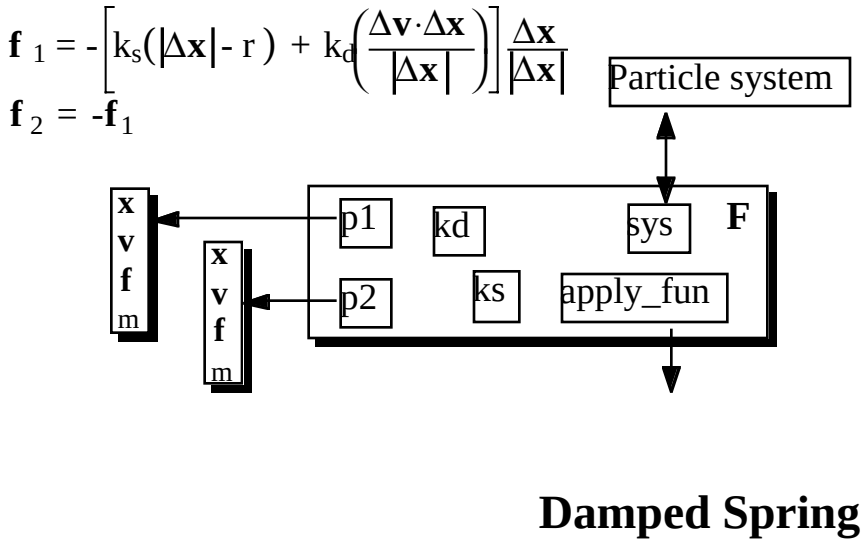


Figure 7: A schematic view of a force object implementing a damped spring that attaches particles p_1 and p_2.

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.

Particle Dynamics

Andrew Witkin
Carnegie Mellon University

Overview

- One Lousy Particle
- Particle Systems
- Forces: gravity, springs …
- Implementation and Interaction
- Simple collisions

A Newtonian Particle

- **Differential equation: f = ma**
- **Forces can depend on:**
  - **Position, Velocity, Time**

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

Second Order Equations

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

$$\begin{cases} \dot{\mathbf{x}} = \mathbf{v} \\ \dot{\mathbf{v}} = \mathbf{f}/m \end{cases}$$

Not in our standard form because it has 2nd derivatives

Add a new variable, **v,** to get a pair of coupled 1st order equations.

Phase Space

$$\begin{bmatrix} \mathbf{x} \\ \mathbf{v} \end{bmatrix}$$

Concatenate **x** and **v** to make a 6-vector: *Position in Phase Space.*

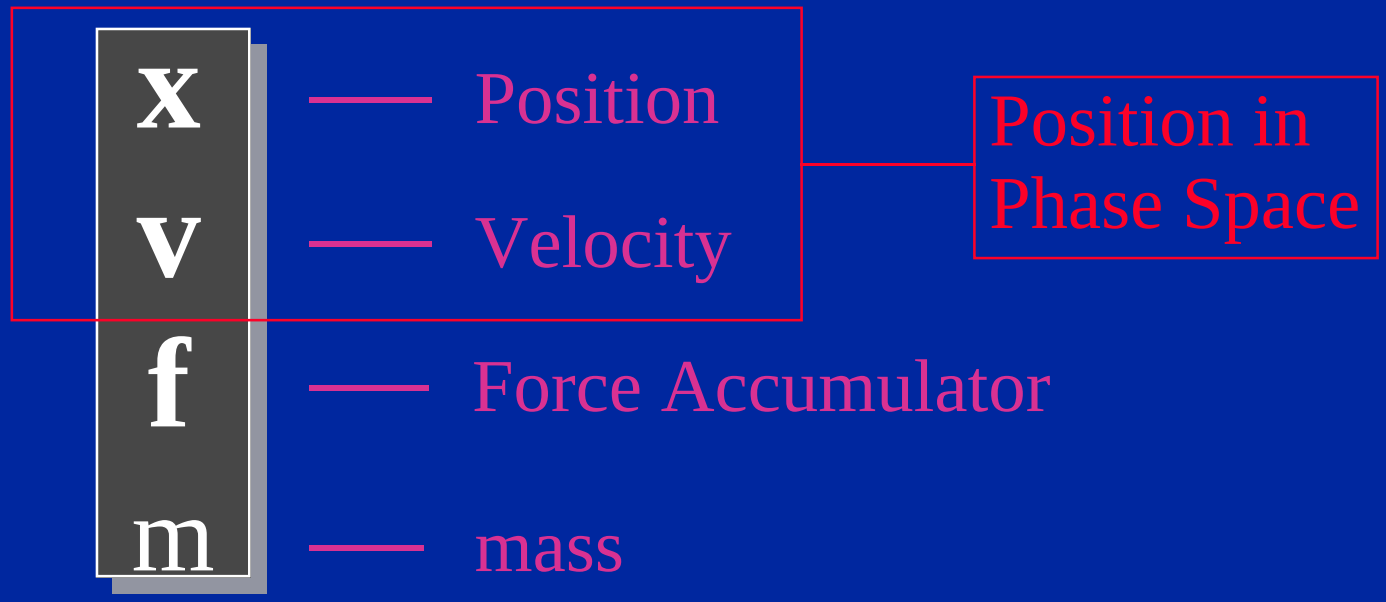
$$\begin{bmatrix} \dot{\mathbf{x}} \\ \dot{\mathbf{v}} \end{bmatrix}$$

Velocity in Phase Space: another 6-vector.

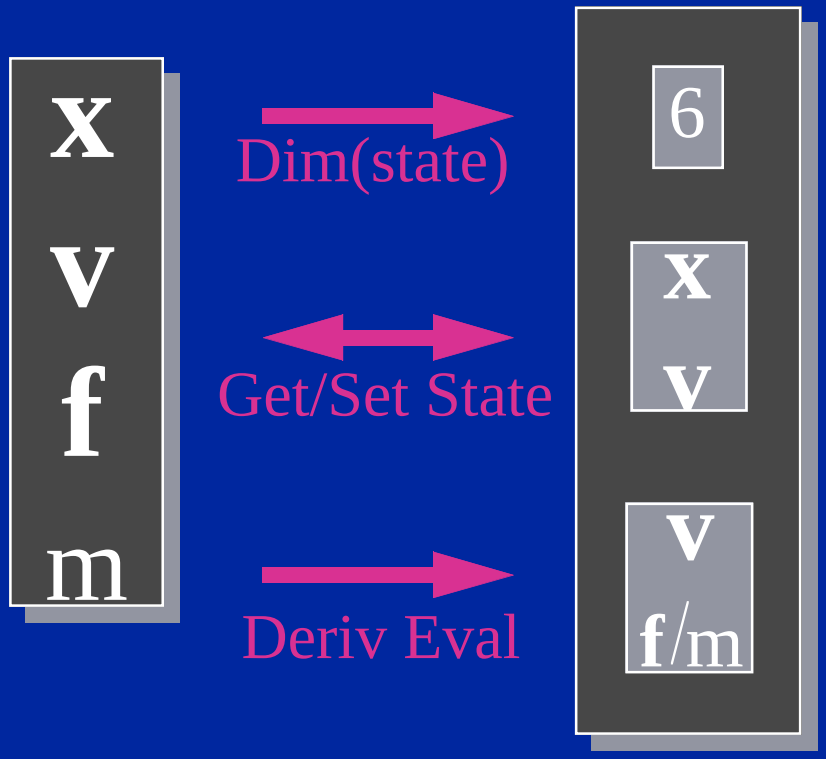
$$\begin{bmatrix} \dot{\mathbf{x}} \\ \dot{\mathbf{v}} \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ \mathbf{f}/m \end{bmatrix}$$

A vanilla 1st-order differential equation.

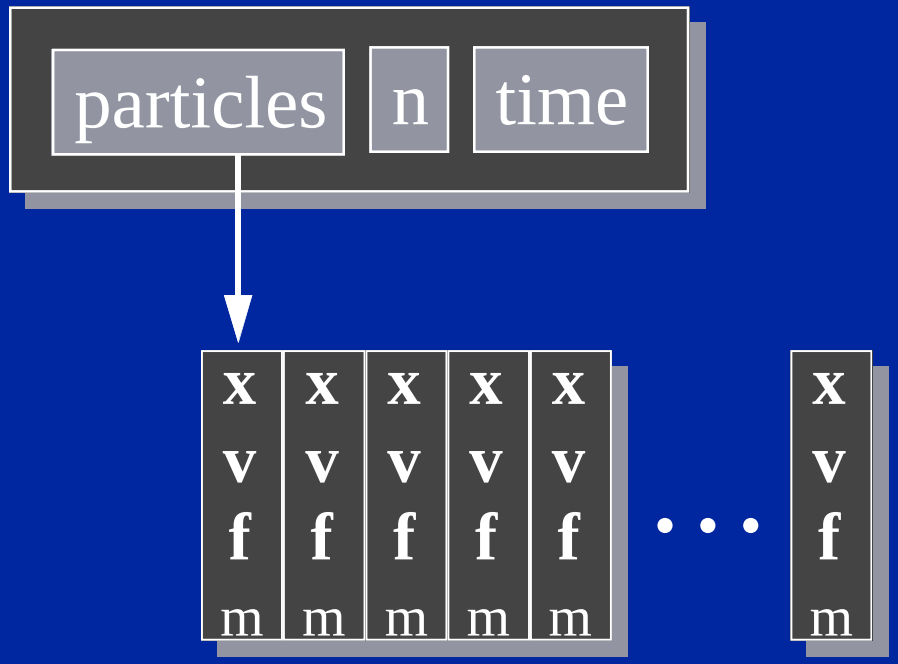
Particle Structure



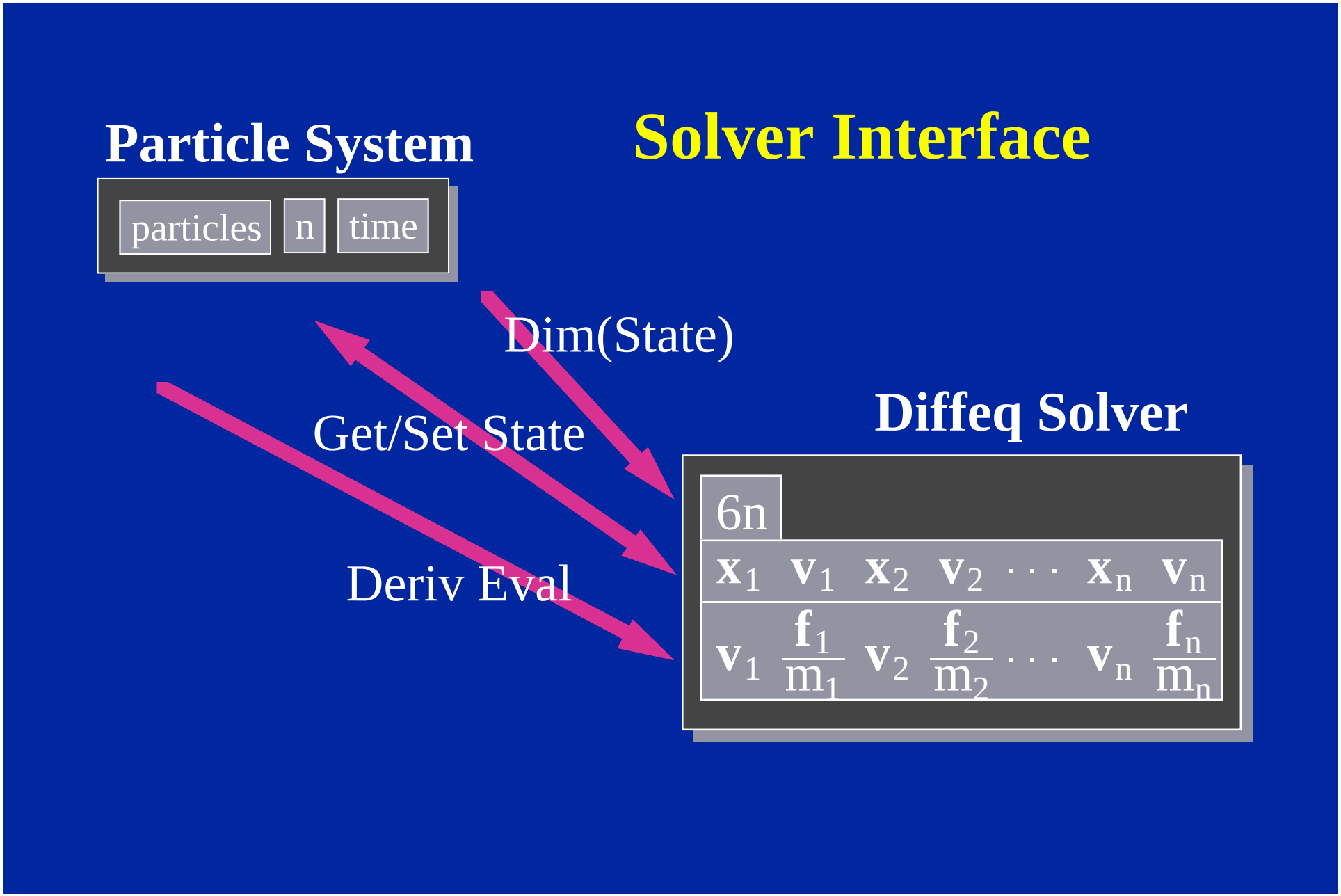
Solver Interface



Particle Systems



Solver Interface



Deriv Eval Loop

- **Clear forces**
  - Loop over particles, zero force accumulators.
- **Calculate forces**
  - Sum all forces into accumulators.
- **Gather**
  - Loop over particles, copying **v** and **f**/m into destination array.

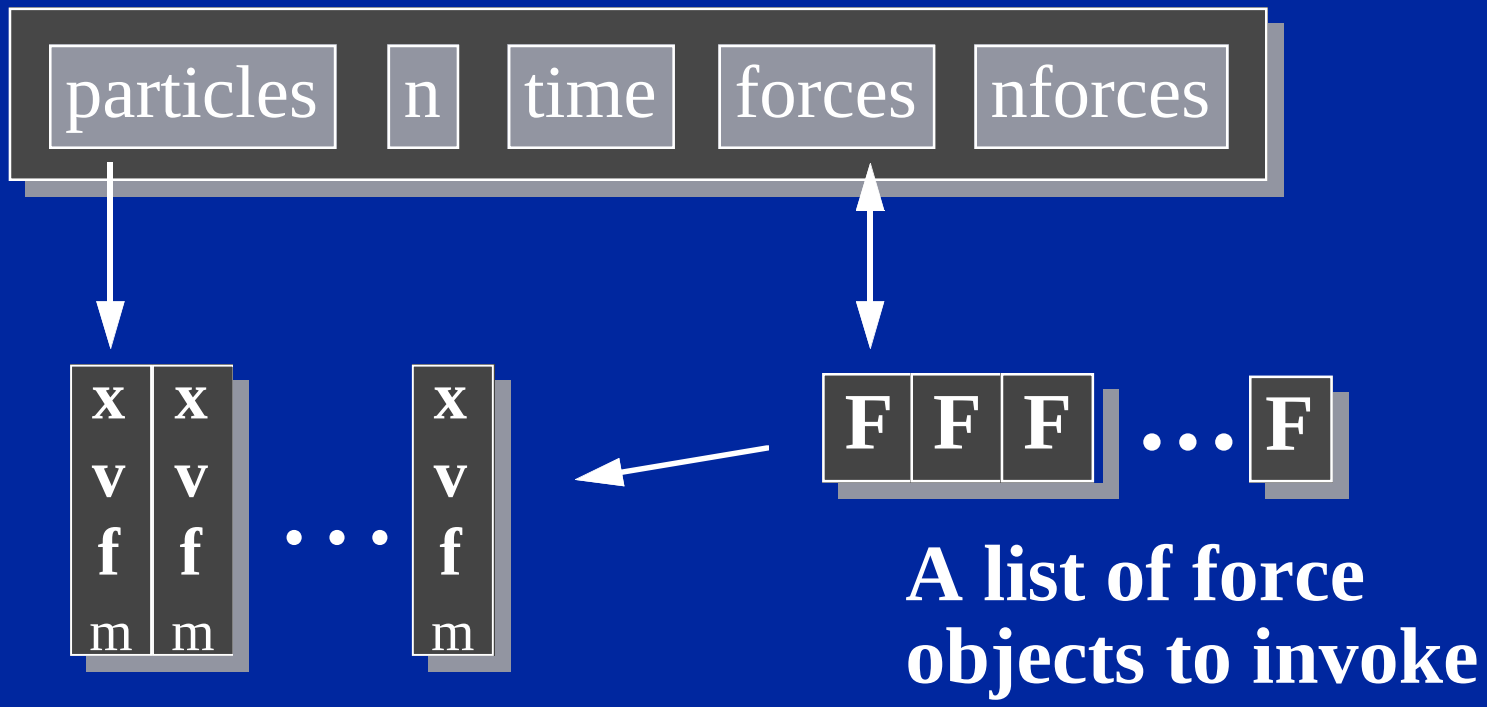
Forces

- **Constant** — **gravity**
- **Position/time dependent** — **force fields**
- **Velocity-Dependent** — **drag**
- **n-ary** — **springs**

Force Structures

- **Unlike particles, forces are heterogeneous.**
- **Force Objects:**
  - black boxes
  - point to the particles they influence
  - add in their own forces (type dependent)
- **Global force calculation:**
  - loop, invoking force objects

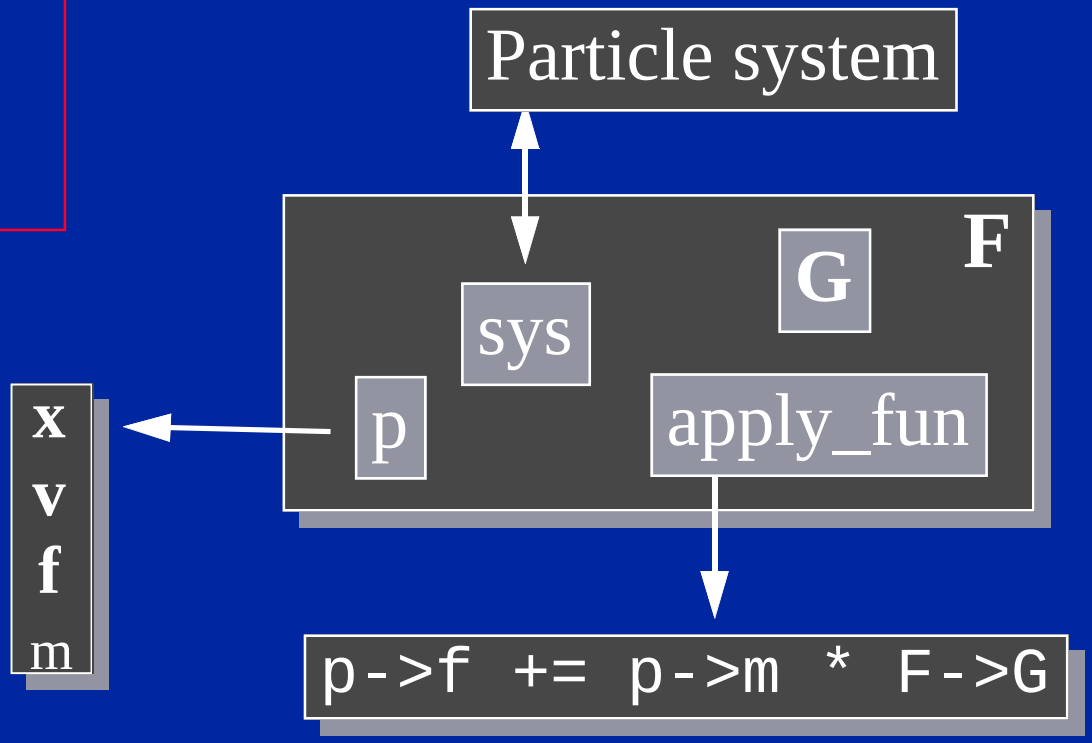
Particle Systems, with forces



Gravity

Force Law:

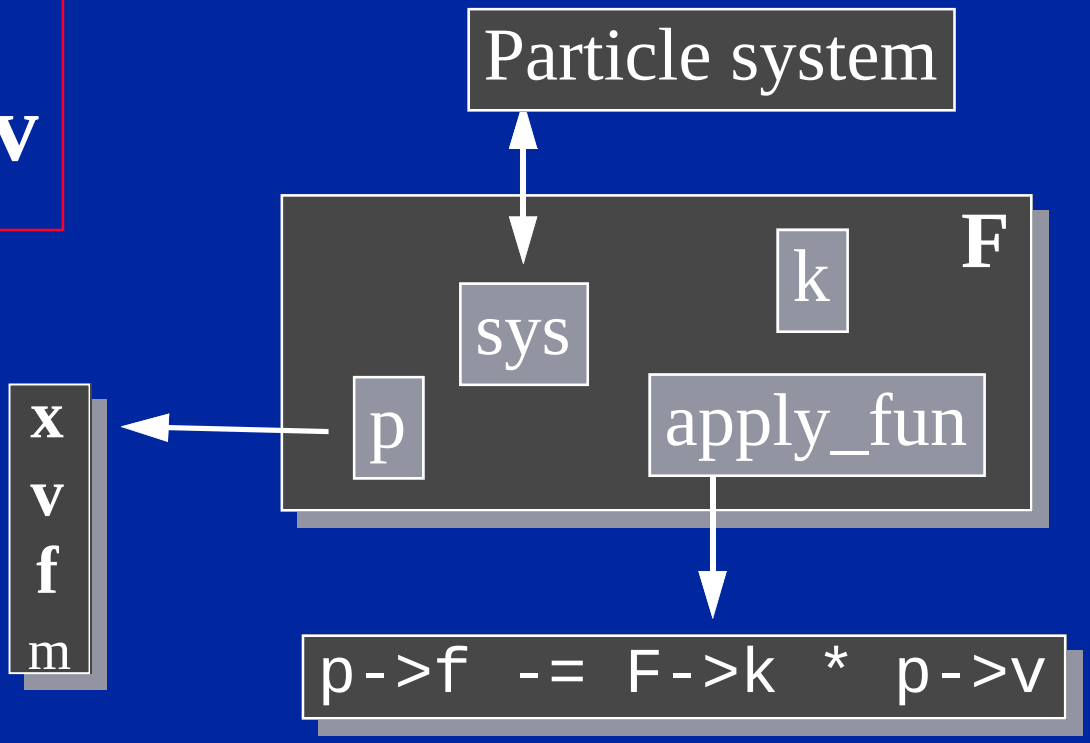
$$\mathbf{f}_{grav} = m\mathbf{G}$$



Viscous Drag

Force Law:

$$\mathbf{f}_{drag} = -k_{drag}\mathbf{v}$$

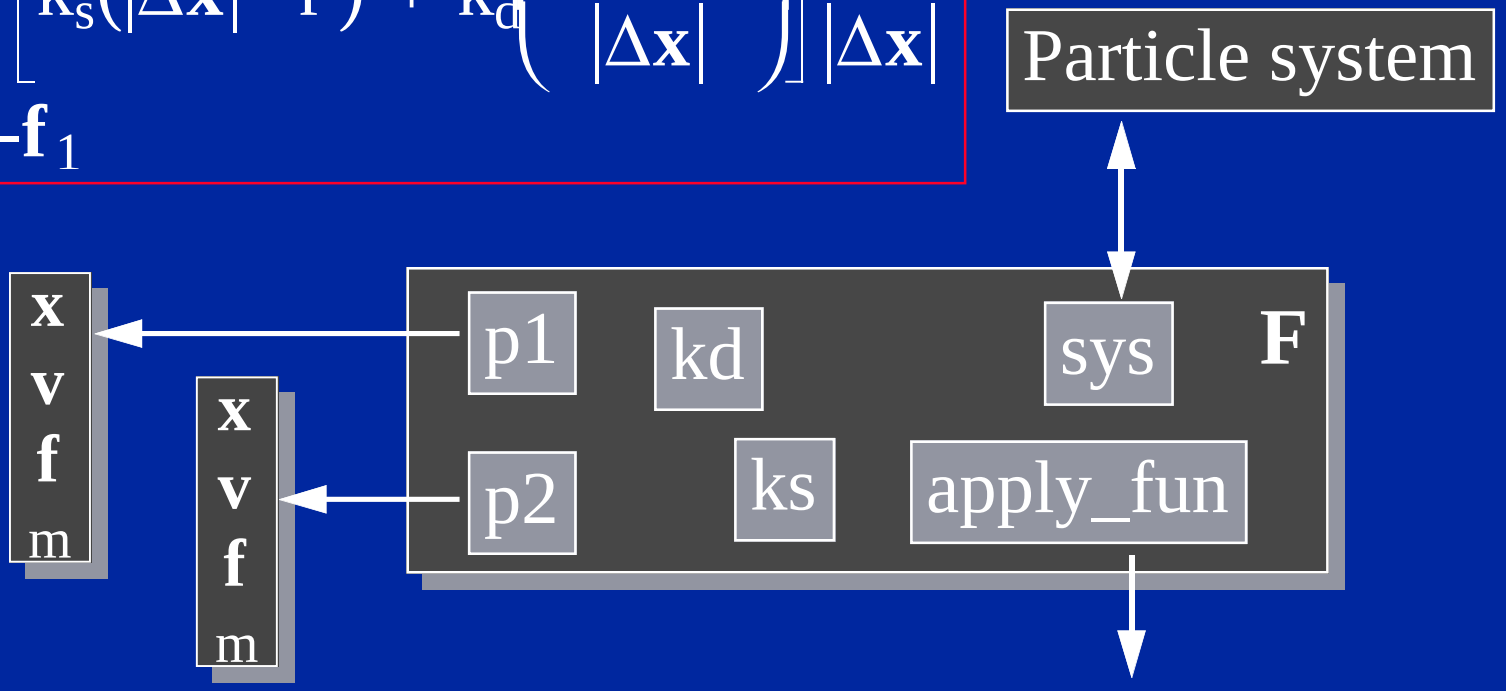


Damped Spring

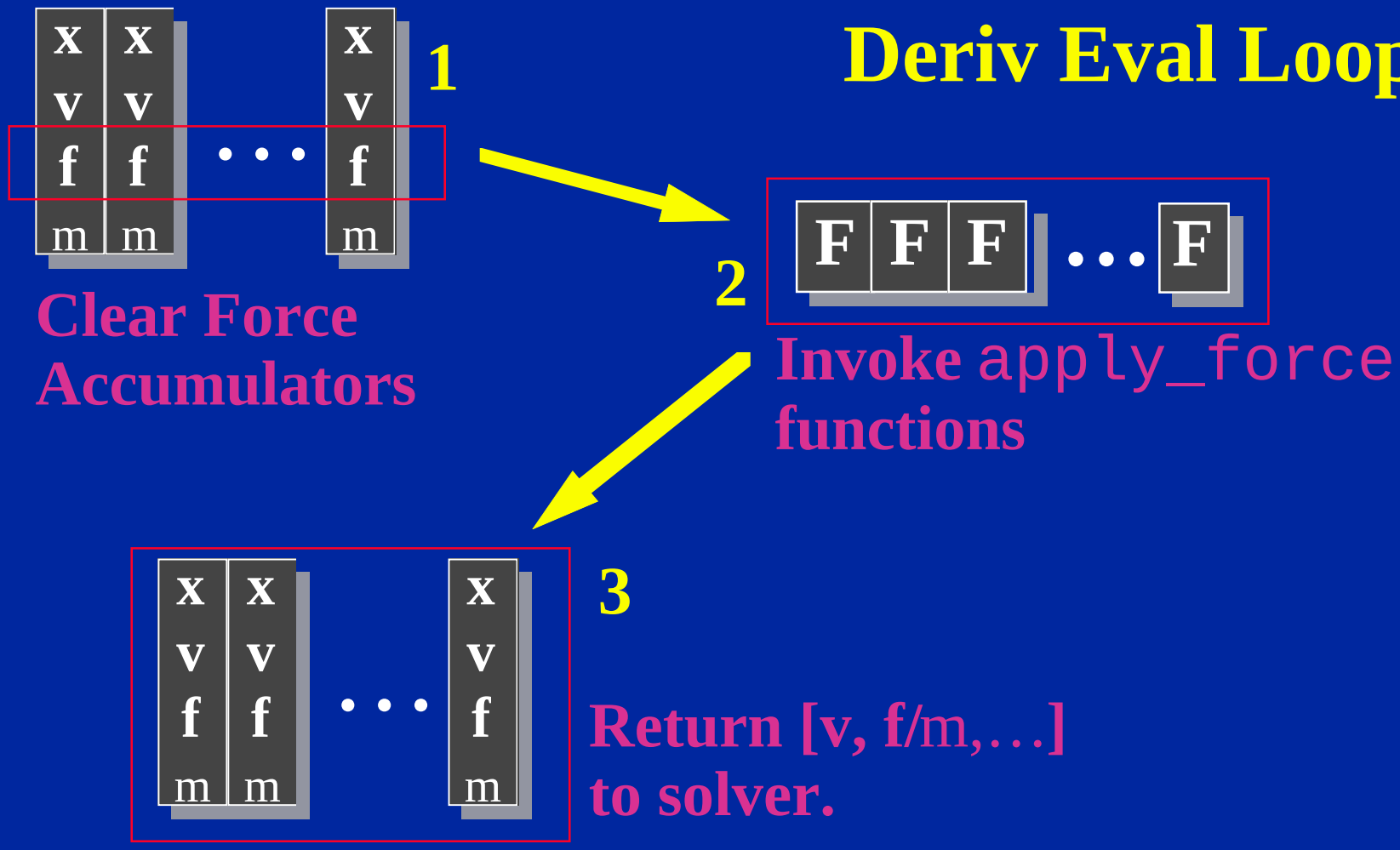
Force Law:

$$\mathbf{f}_1 = -\left[k_s(|\Delta\mathbf{x}| - r) + k_d\left(\frac{\Delta\mathbf{v}\cdot\Delta\mathbf{x}}{|\Delta\mathbf{x}|}\right)\right]\frac{\Delta\mathbf{x}}{|\Delta\mathbf{x}|}$$

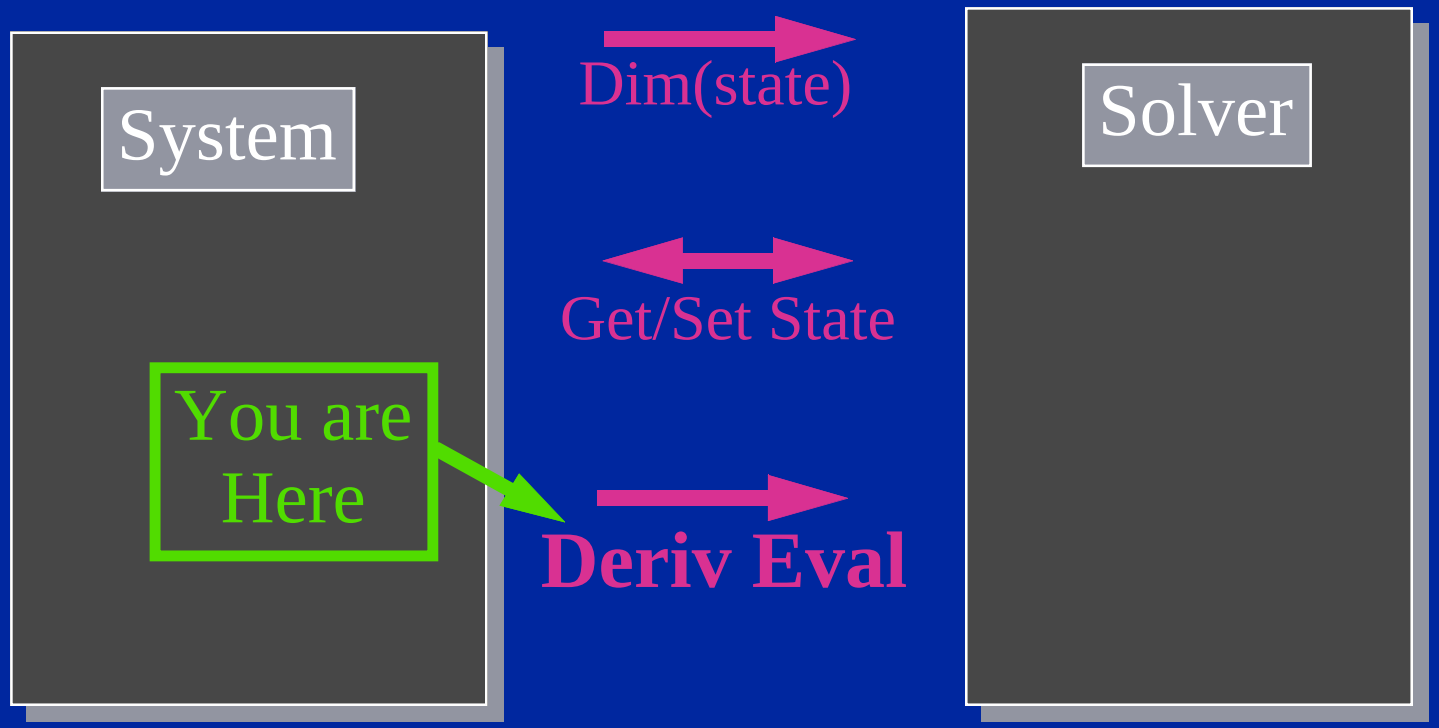
$$\mathbf{f}_2 = -\mathbf{f}_1$$



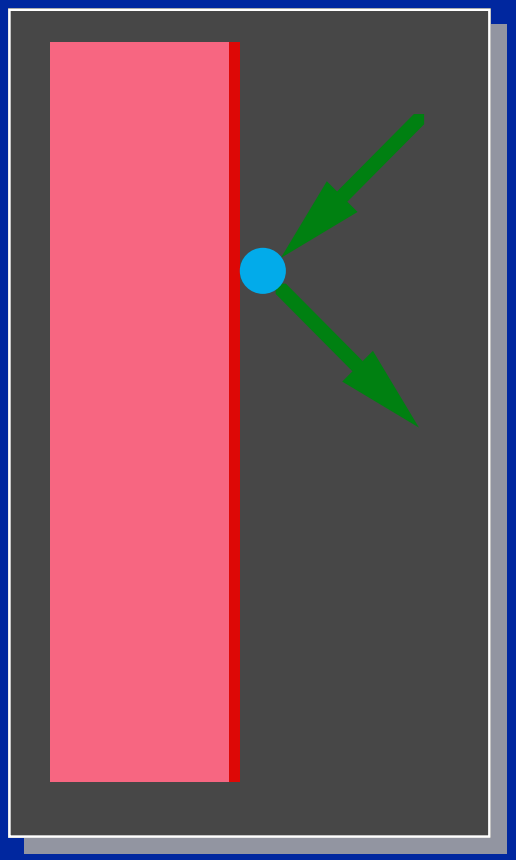
Deriv Eval Loop



Solver Interface

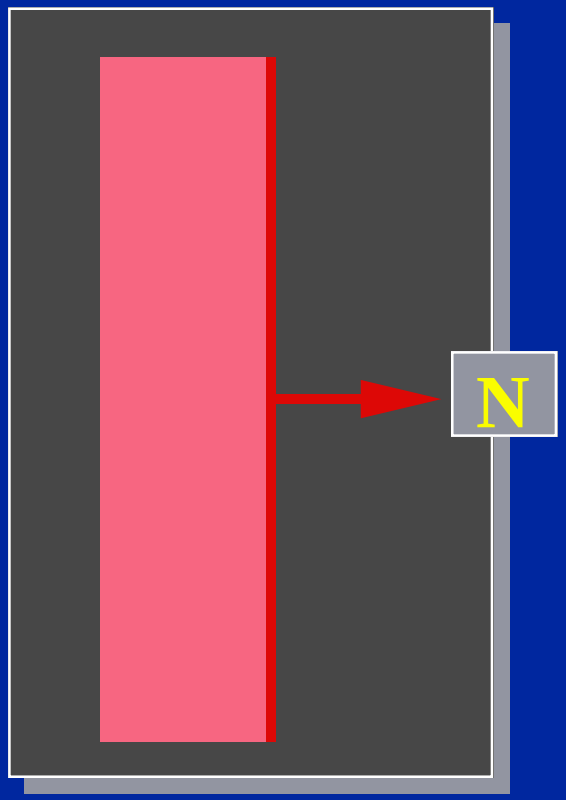


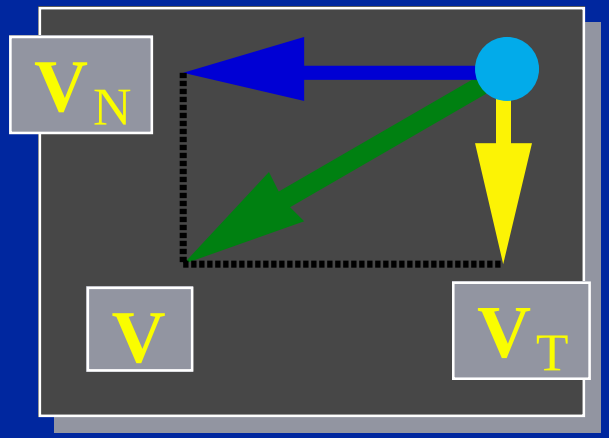
Bouncing off the Walls



- Later: rigid body collision and contact.
- For now, just simple point-plane collisions.
- Add-ons for a particle simulator.

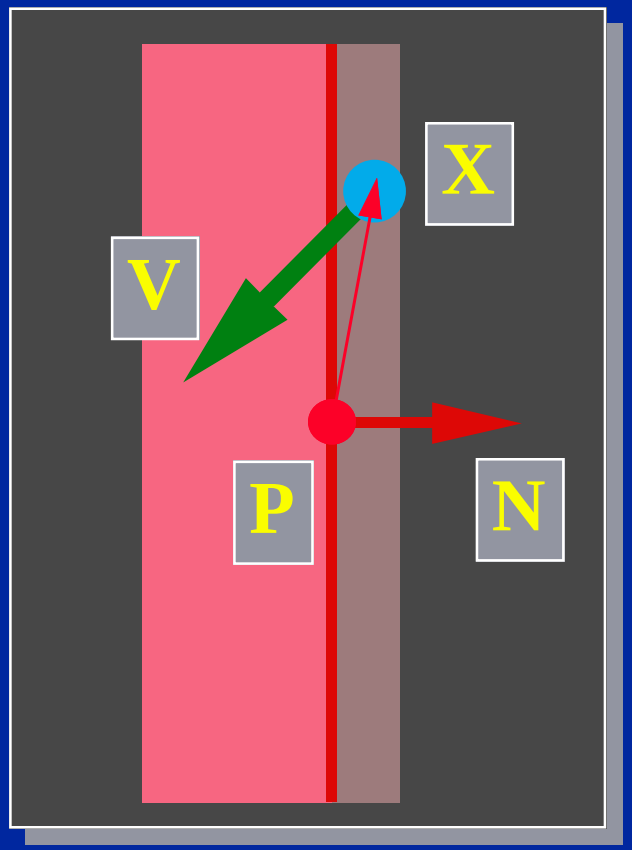
Normal and Tangential Components





$$\mathbf{V}_N = (\mathbf{N} \cdot \mathbf{V})\mathbf{N}$$
$$\mathbf{V}_T = \mathbf{V} - \mathbf{V}_N$$

Collision Detection

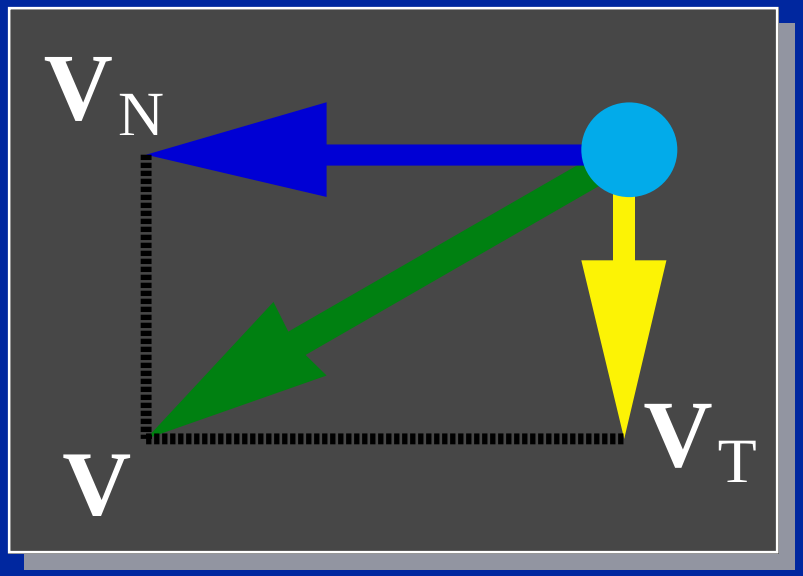


$$(\mathbf{X} - \mathbf{P}) \cdot \mathbf{N} < \varepsilon$$

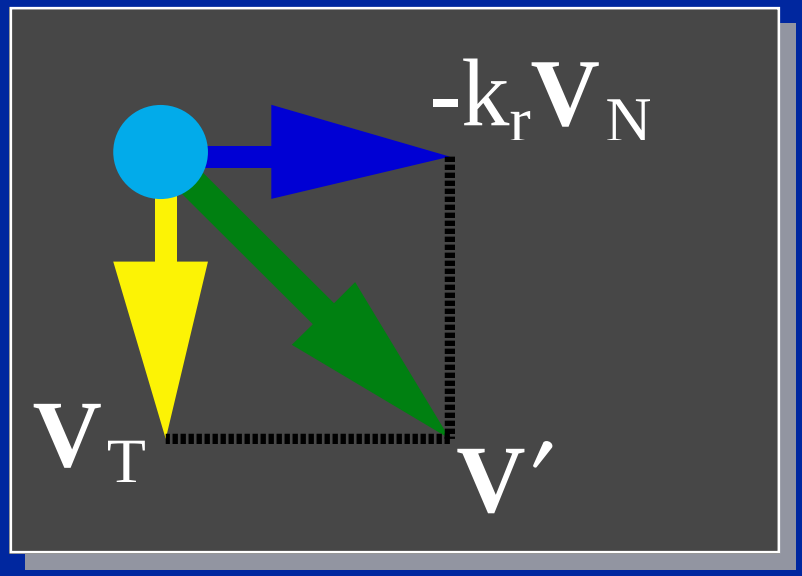
$$\mathbf{N} \cdot \mathbf{V} < 0$$

- Within ε of the wall.
- Heading in.

Collision Response



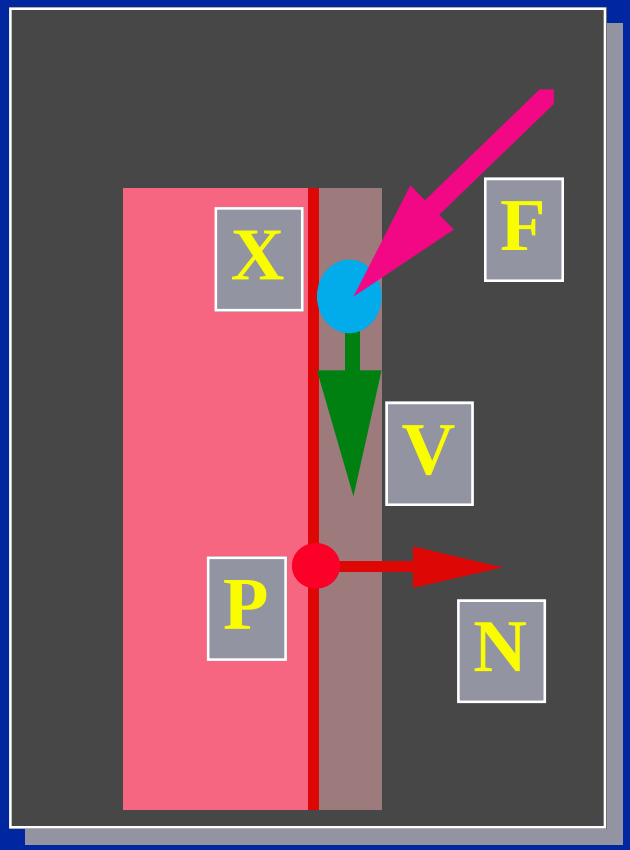
Before



After

$$\mathbf{V}' = \mathbf{V}_T - k_r\mathbf{V}_N$$

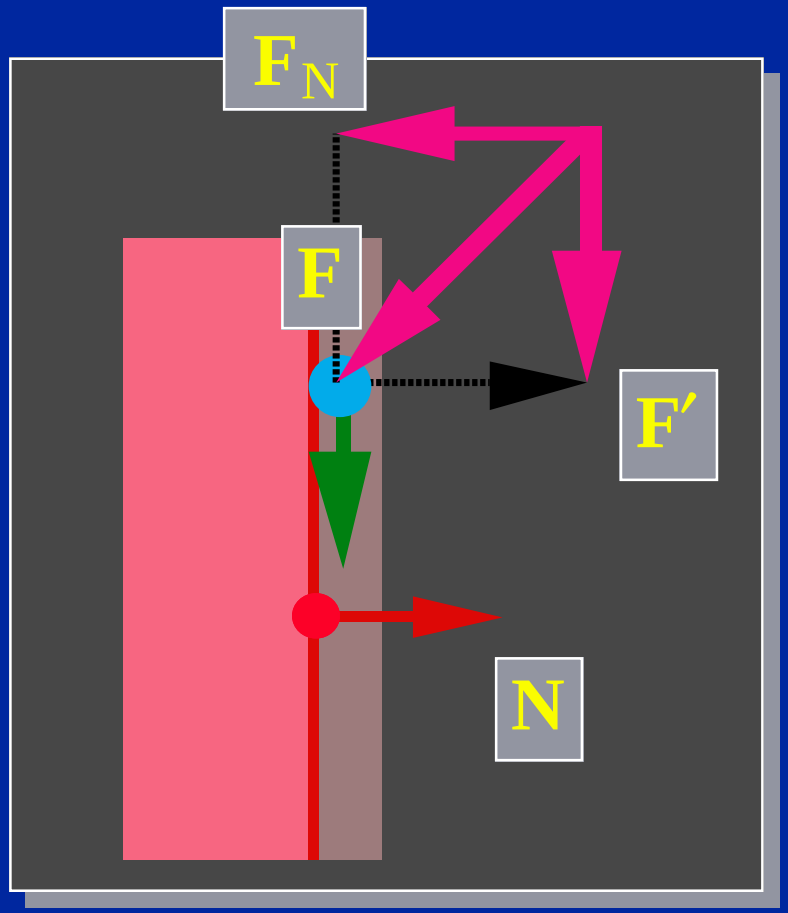
Conditions for Contact



$$|(\mathbf{X} - \mathbf{P}) \cdot \mathbf{N}| < \varepsilon$$

$$|\mathbf{N} \cdot \mathbf{V}| < \varepsilon$$

- On the wall
- Moving along the wall
- Pushing against the wall



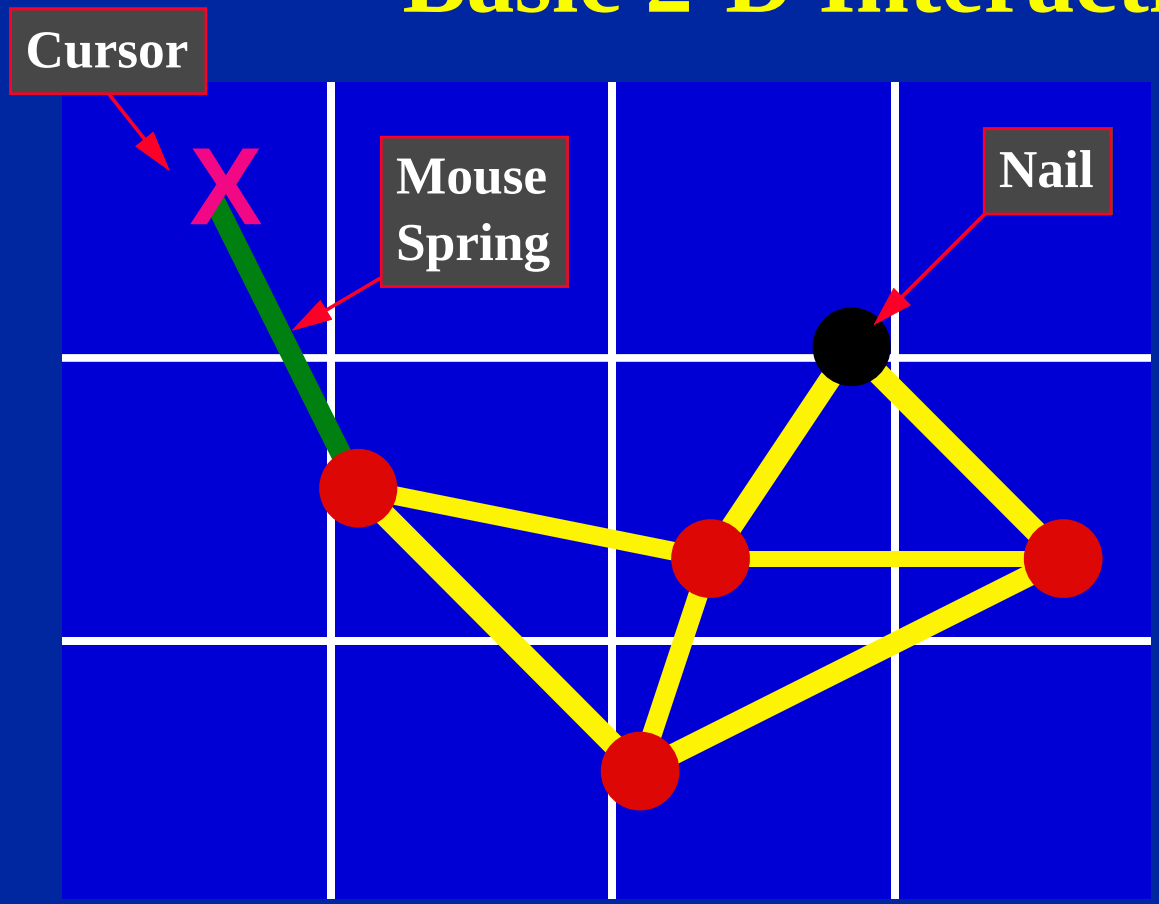
Contact Force

$$F' = F_T$$

The wall pushes back, cancelling the normal component of F.

(An example of a constraint force.)

Basic 2-D Interaction



Operations:

- Create
- Attach
- Drag
- Nail

Try this at home!

The notes give you everything you need to build a basic interactive mass/spring simulator—try it.