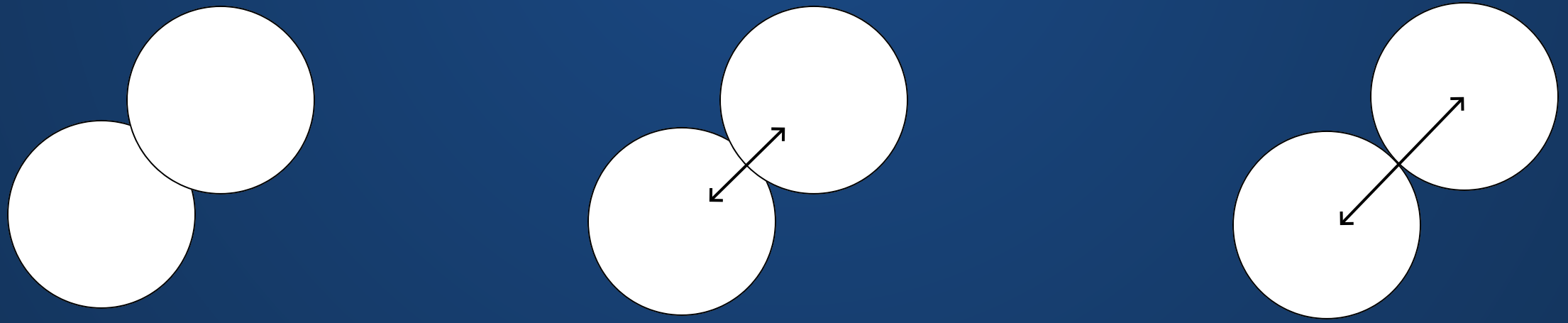
Physical Simulation

Can we apply more complex physics in Processing?

Absolutely. Most physics is controlled by the complexity of your numeric code

How would we make collisions work with lots of particles?



Do PVectors work similarly in 3D?

Yes. In fact, all PVectors are implicitly 3D (but the third component is zeroed out and ignored when you use them in a 2D fashion).

Is array() the best way to access the vectors of PVector?

I might recommend using the .x, .y, and .z members of the vector.

`set()`	Set the components of the vector
`random2D()`	Make a new 2D unit vector with a random direction
`random3D()`	Make a new 3D unit vector with a random direction
`fromAngle()`	Make a new 2D unit vector from an angle
`copy()`	Get a copy of the vector
`mag()`	Calculate the magnitude of the vector
`magSq()`	Calculate the magnitude of the vector, squared
`add()`	Adds x, y, and z components to a vector, one vector to
`sub()`	Subtract x, y, and z components from a vector, one vec
`mult()`	Multiply a vector by a scalar
`div()`	Divide a vector by a scalar
`dist()`	Calculate the distance between two points
`dot()`	Calculate the dot product of two vectors

Why does calling an object draw() from draw() only run it once?

`draw()` is only a special name when it's at the top level of your Processing program.

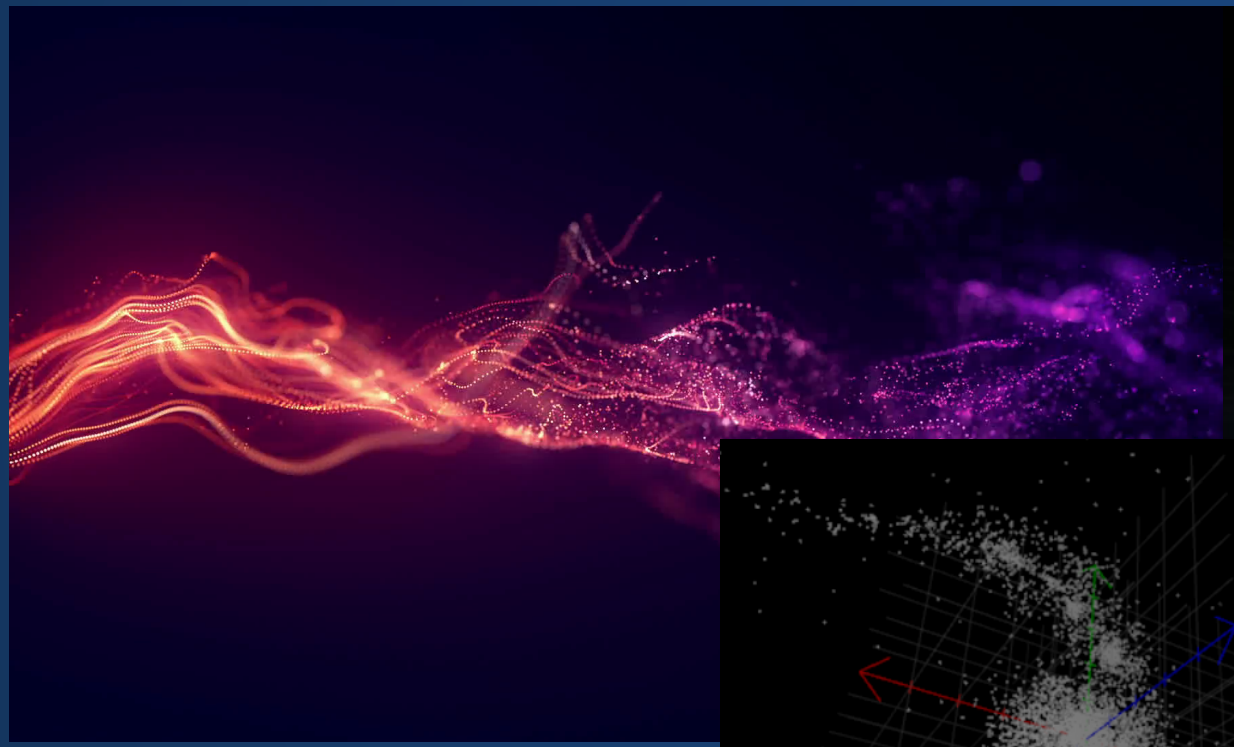
```
1 // This function is special. It is
2 // called in a loop and there can
3 // only be one function named this
4 void draw(){
5
6
7 }
```

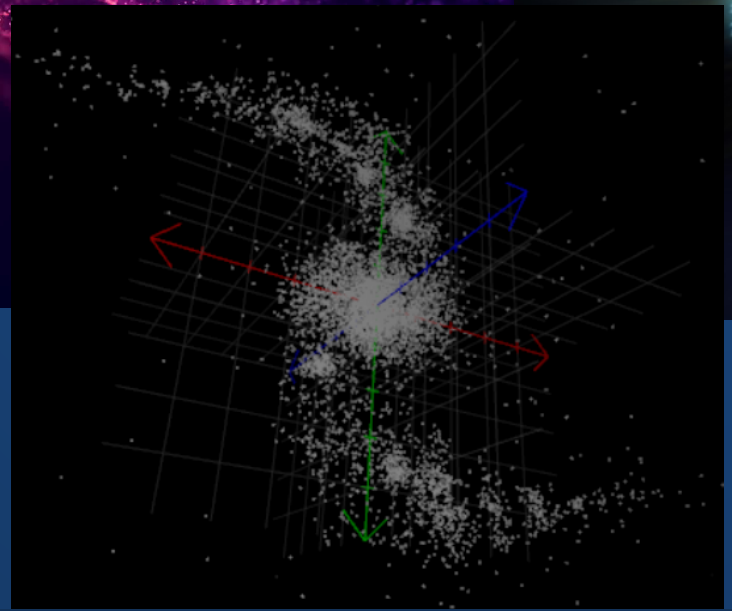
```
 1 // None of these methods are special.
 2 // As far as Processing cares, they're
 3 // just ordinary methods, and could
 4 // be named anything--the name "draw"
 5 // is totally coincidental.
 6 class A {
 7   void draw(){}
 8 }
 9
10 class B {
11   void draw(){}
12   void draw(int x){}
13 }
```

How do we make the acceleration more subtle?

Particle systems dictate the movement and appearance of particles within the world.

Examples: smoke, water, fire, clouds, dust, galaxies, etc.





Particle Systems

Generally, there will be three "stages" in a particle system, though these can be thought more as decisions to make than as stages-in-time.

1. *Emission stage*: we decide where/how particles are generated and their initial states.

2. *Simulation stage*: we decide how particles are going to behave (what forces are going to act on them and whether they'll collide, etc.)

3. *Rendering stage*: we decide how particles are going to be displayed to the user.

What decisions were made for each stage for these particle systems?

Fireworks

Snowstorm

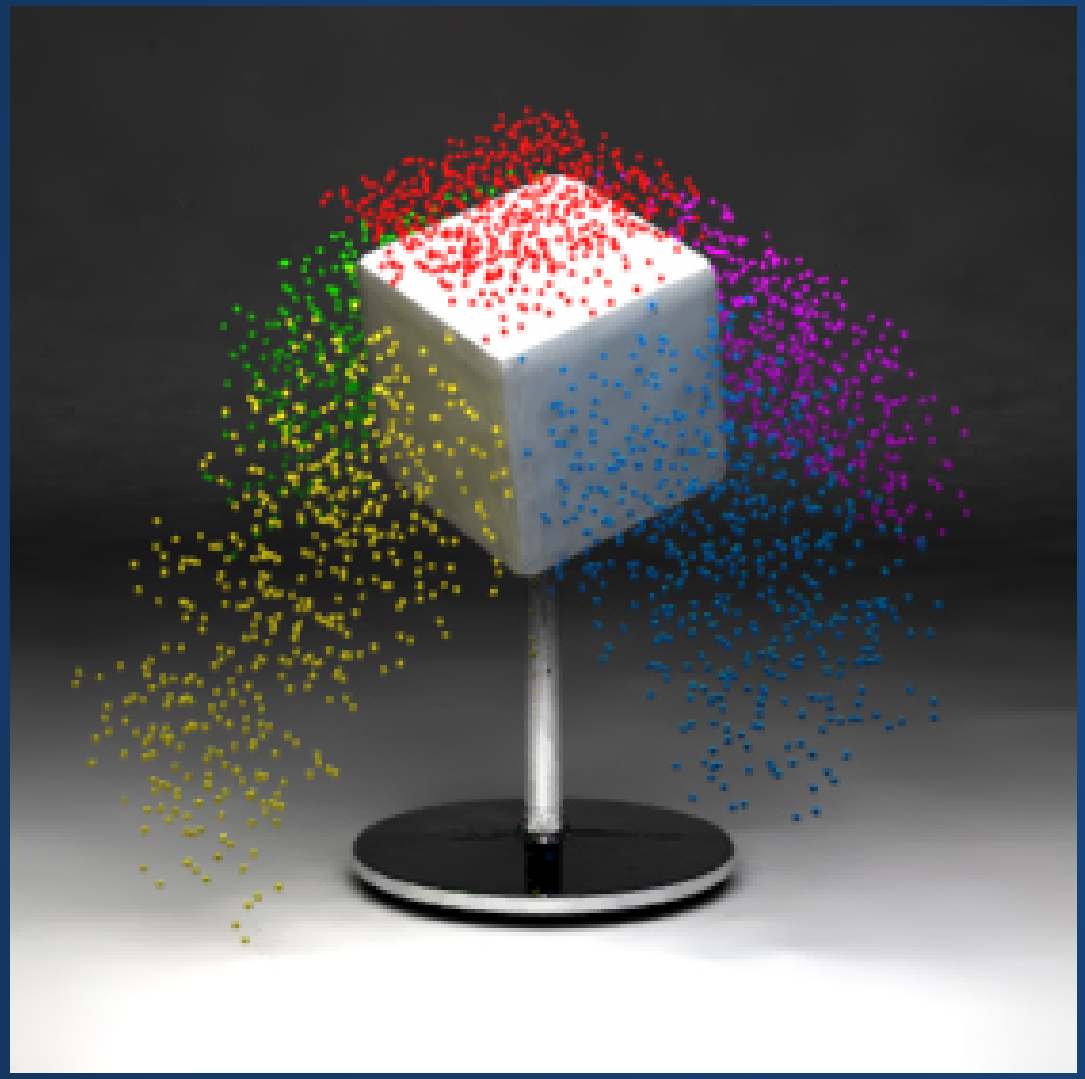
Extending the `Particle`

Additional rules (usually written as methods) can make the behavior of our `Particle` class more complex and interesting:

- Continuous generation of particles
- Changes to particle appearance over time
- Application of additional forces on particles
- Changes in how particles are visualized

These things don't have to be physically based!

Differences in rendering can create a large difference in visual impact!



Linked from https://en.wikipedia.org/wiki/Particle_system#/media/File:Particle_Emitter.jpg

Equations of Motion

Equations of Motion

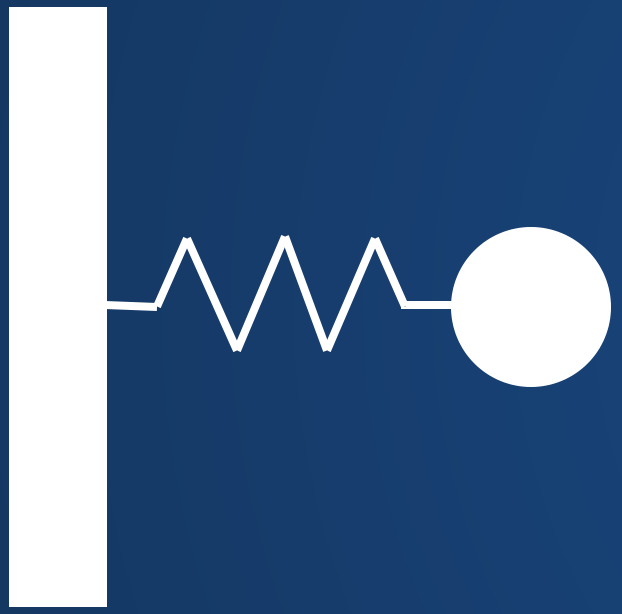
Equations that describe motions over time.

Newton's Second Law

$$F = ma$$

This is not solvable exactly on a computer for any interesting set of forces.

Not exactly solvable?



1. Pull slightly on ball
2. Start simulation
3. Spring applies a restoring force
4. Ball accelerates toward wall
5. As soon as ball moves the slightest, the force on the ball changes. (???)

But we don't have a way to express "infinitely small amount" on a computer!

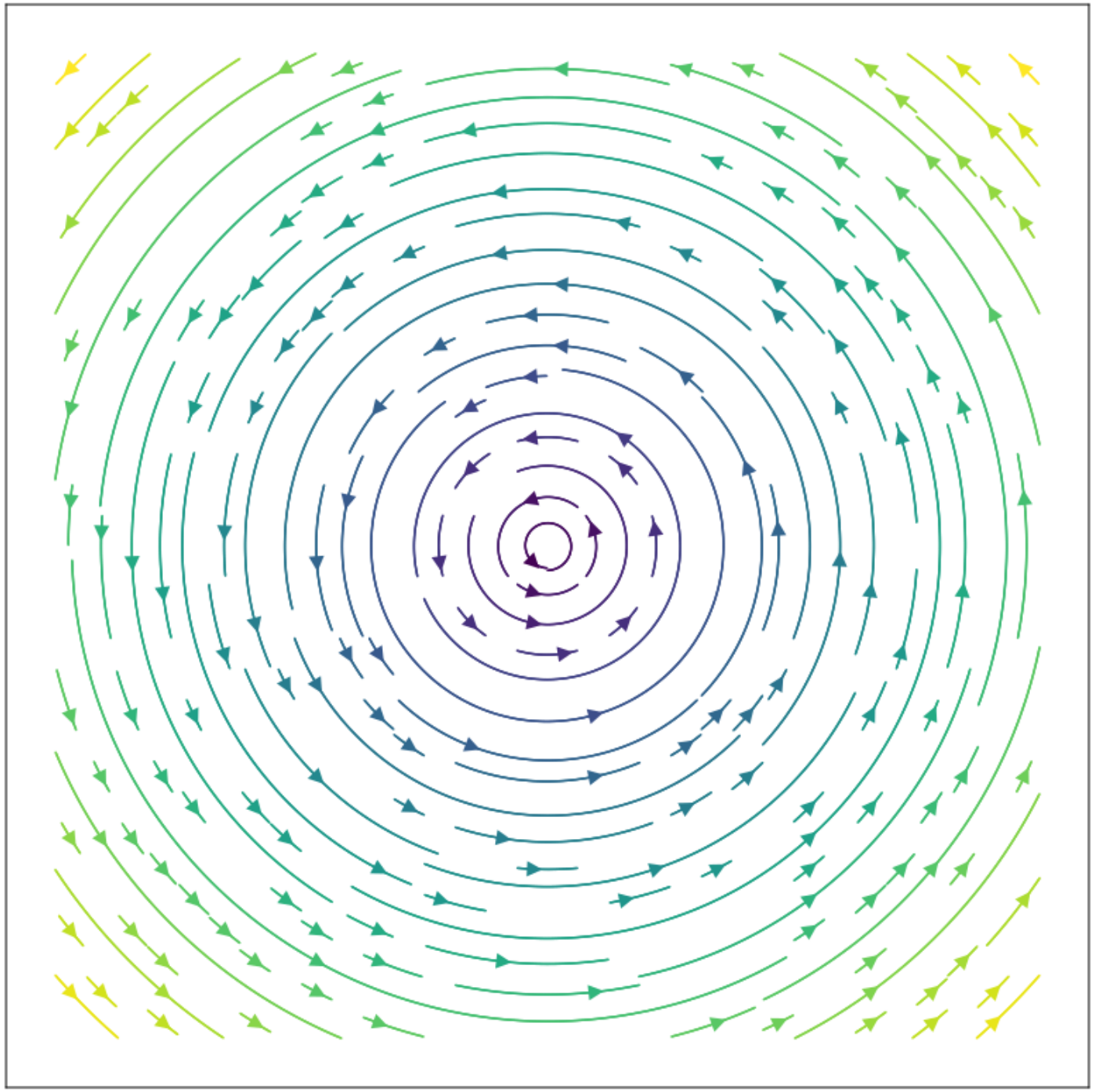
We need some way to *discretize* the simulation into *timesteps* and decide what happens in each timestep.

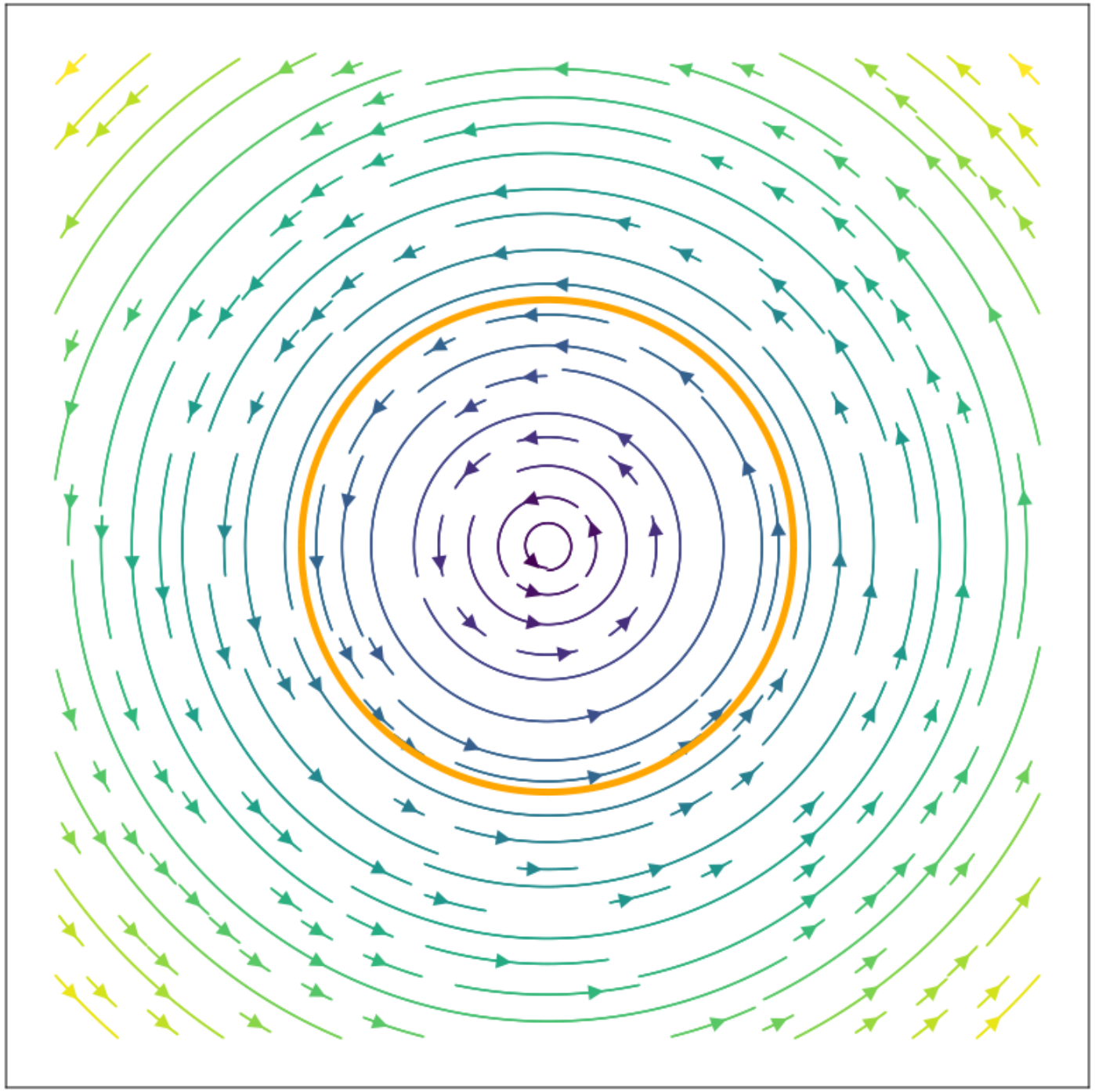
(Explicit) Euler's Method

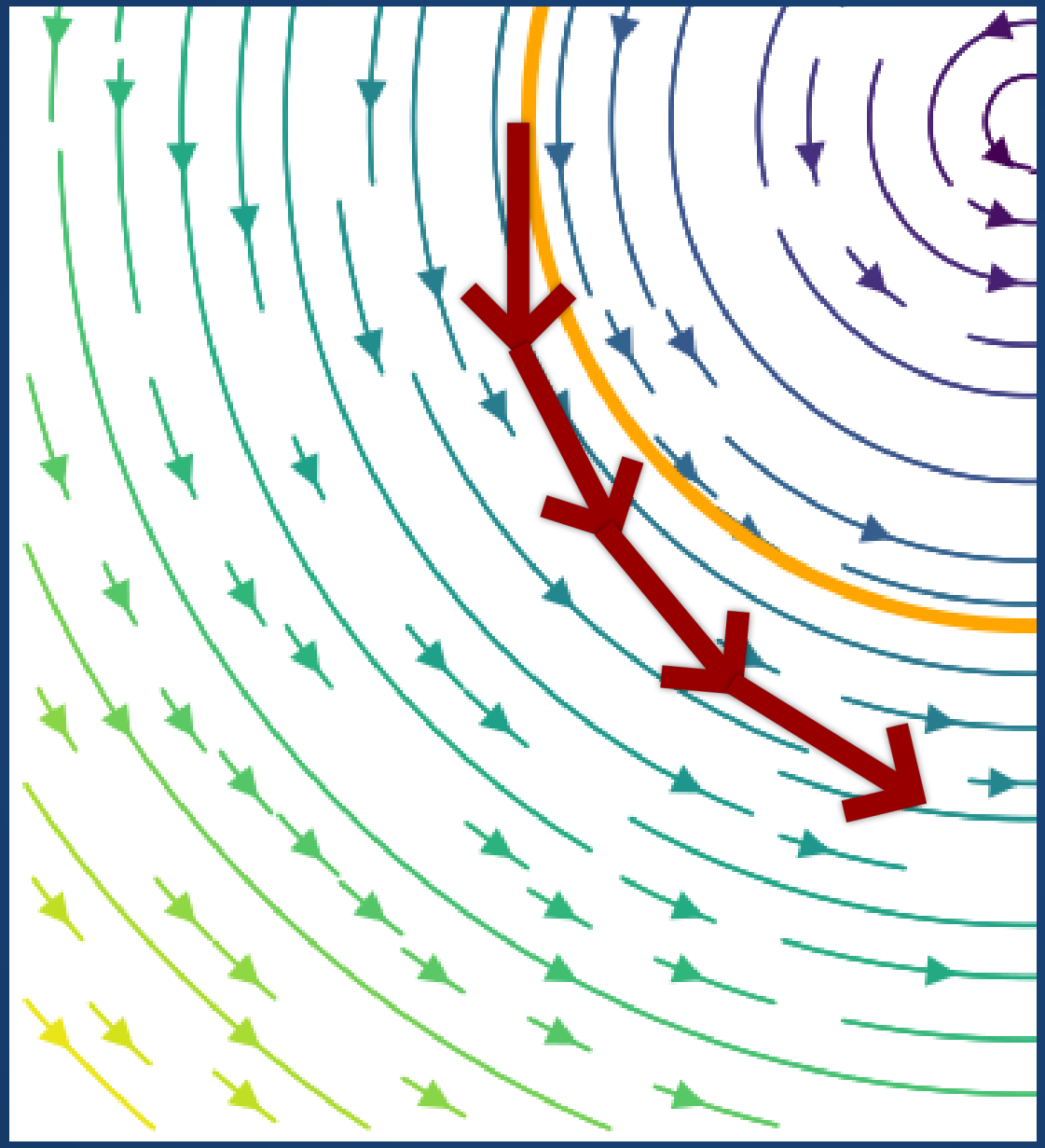
1. We are at $\vec{x}$ with velocity $\vec{v}$.
2. Compute the update to our position
   - Take current velocity and multiply it by timestep duration
   - This is the step we'll take during the next timestep
3. Compute the update to our velocity
   - Compute forces at our current location
   - Multiply acceleration by timestep
   - This is how much the velocity will change over the next timestep.

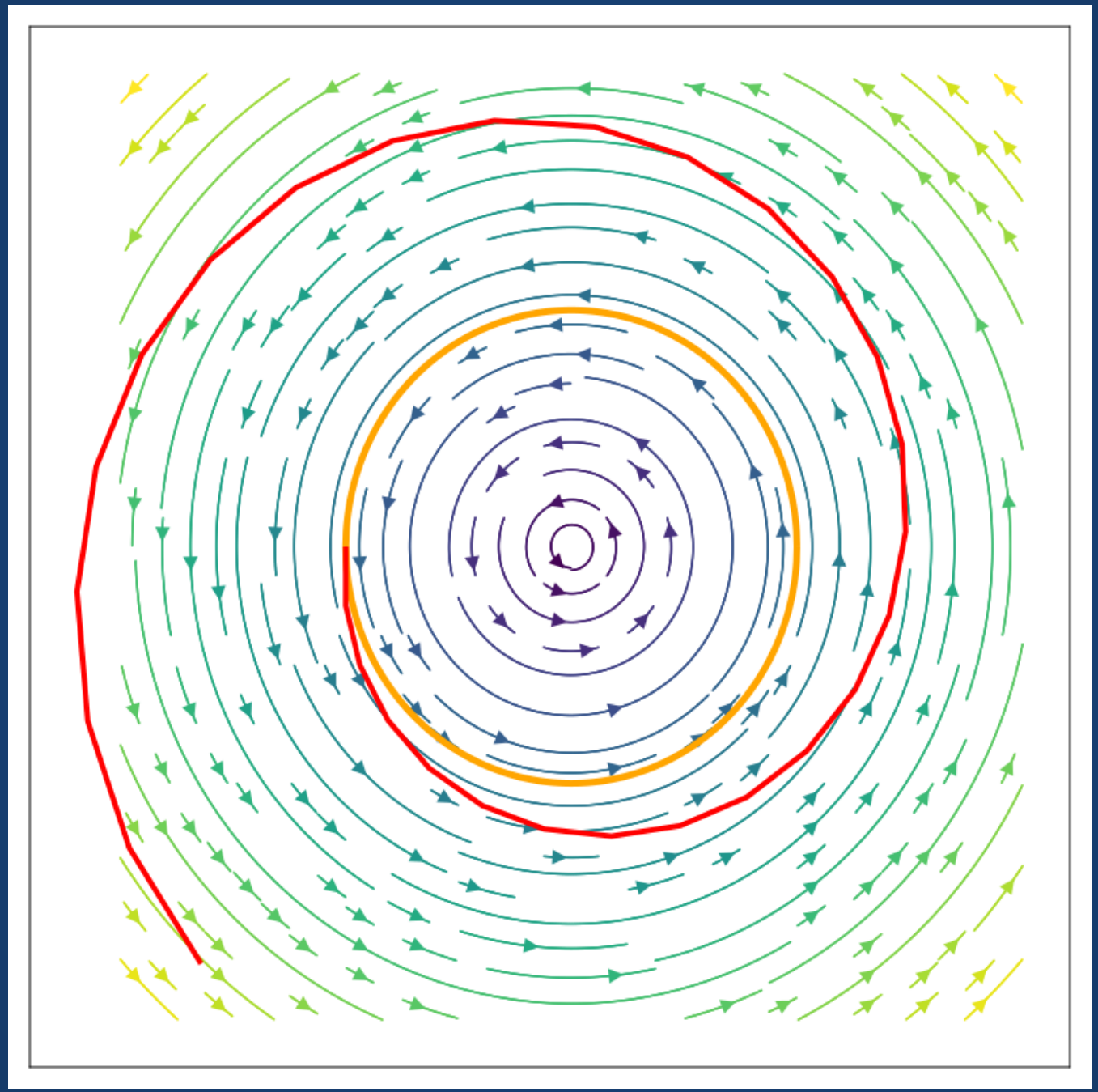
"Look at where we'll go next based off of where we are now"

$$\vec{x}^{i+1} = \vec{x}^{i} + \Delta t \cdot \vec{v}^{i}$$
$$\vec{v}^{i+1} = \vec{v}^{i} + \Delta t \cdot \vec{F}^{i}$$









Issues with Explicit Euler

- Requires very small timesteps to work.
  - Have to do much more computation to simulate the same amount of "real world" time.
- Error tends to accumulate resulting in violation of conservation of energy (tends to lead to explodey systems).
- Better, more stable methods exist, so explicit Euler is rarely used in practice
- But it's simple to understand, and works well enough for our class!

Velocity Verlet

One of the methods often used in practice: it is not much more accurate, but it's still simple and doesn't suffer energy growth like the explicit Euler method does.

Velocity Verlet

$$\vec{x}^{i+1} = \vec{x}^i + \Delta t \cdot \vec{v}^i + \frac{1}{2}\vec{a}^i \Delta t^2$$

$$\vec{v}^{i+1} = \vec{v}^i + \frac{\vec{a}^i + \vec{a}^{i+1}}{2} \Delta t$$

where $\vec{a}$ is the acceleration vector.

Explicit Euler

$$\vec{x}^{i+1} = \vec{x}^i + \Delta t \cdot \vec{v}^i$$

$$\vec{v}^{i+1} = \vec{v}^i + \Delta t \cdot \vec{F}^i$$

Other common methods: Runge-Kutta methods, Midpoint Euler

Accounting for Mass

We still haven't dealt with mass!

Our particle has a mass of m.

The particle experiences a force of $\vec{f}$.

$$\vec{f} = m\vec{a}$$

Solution: we need to divide the force by the mass to get the acceleration.

Particle with Mass

```
1 class Particle {
2   float x;
3   float vx;
4   float mass;
5
6  // ...
```

```
1 void advanceTime(float force, float dt){
2   float accel = force / this.mass;
3
4   this.x += this.vx * dt;
5   this.vx += accel * dt;
6 }
```

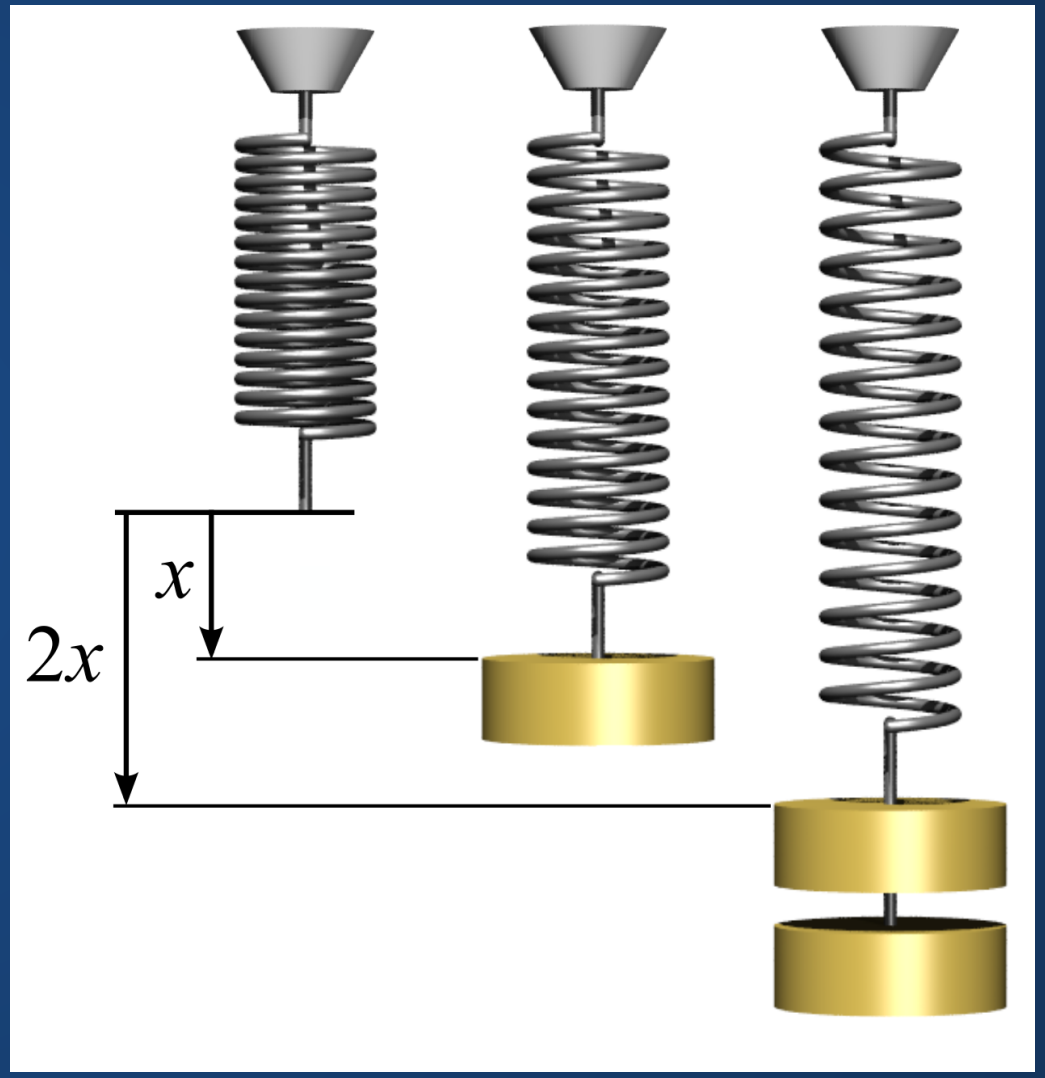
Springs

Spring Forces

The force a spring exerts on its endpoints is based on:

- Spring stiffness (k)
- Stretch from resting position (x)

Hooke's Law tells us that the force is $F = -kx$.



Spring Example

```
1  float y = 100.0;
2  float vy = 0.0;
3  float m = 1.0;
4  float ry = 250;
5  float ks = 1.0;
6  float dt = 0.1;
7
8  void setup() {
9    size(500, 500);
10   rectMode(CENTER);
11 }
```

```
1  void draw() {
2    background(210);
3    fill(0);
4    ellipse(200, ry, 50, 50);
5    float f = -(ks * (y - ry));
6    float a = f/m;
7    vy = vy + dt*a;
8    y += dt*vy;
9    line(200,ry,200,y);
10   rect(200, y, 100, 20);
11 }
```

Question: what does `ry` represent in this code?

Damping

This spring system oscillates forever (the spring doesn't dissipate any energy).

That's not necessarily bad!

Sometimes, we want vibrations to "damp" out, i.e. get weaker and weaker until they stop.

How can we introduce damping into our spring?

Damping

$$F = -k_s X - k_d v$$

Spring force: tries to restore spring to its resting length.

Damping force: acts against motion of the spring

Damping

```
float y;
float vy;
float m = 1.0;
float ry = 250;
float ks = 0.1;
float kd = 0.1;

void setup() {
  size(500, 500);
}
```

```
void draw() {
  background(210);

  float f = -((ks * (y - ry)) + kd*vy);
  float a = f/m;
  vy = vy + a;
  y += vy;

  rect(200, y, 100, 20);
}
```

Extensions of Springs

All our springs so far have been zero restlength. Realistic springs have restlength of greater than zero.

Sequences of springs can simulate:

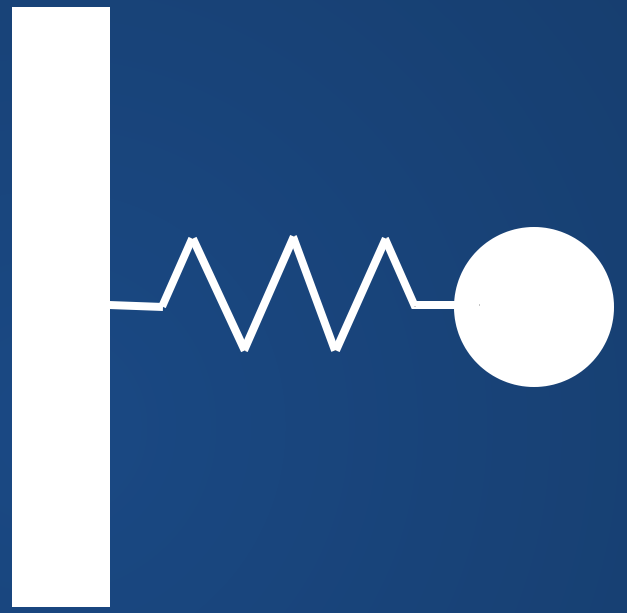
- Hair
- Rope
- Grass

Networks of springs can simulate cloth.

https://andrew.wang-hoyer.com/experiments/cloth/

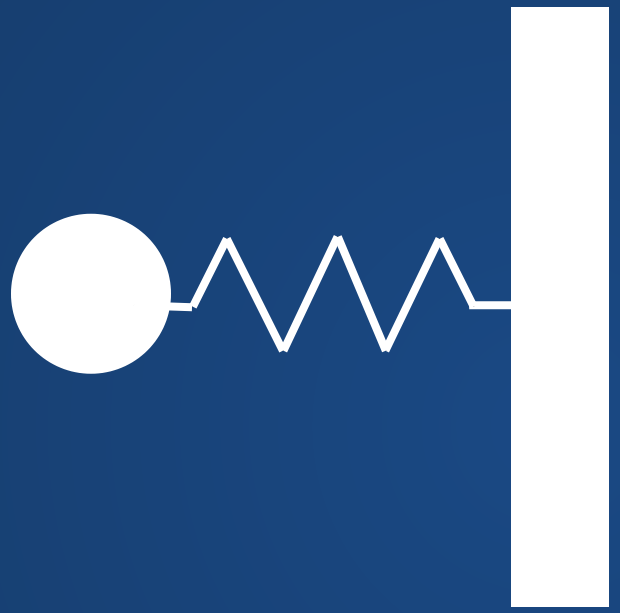
Timestepping and Stiffness

Consider the following



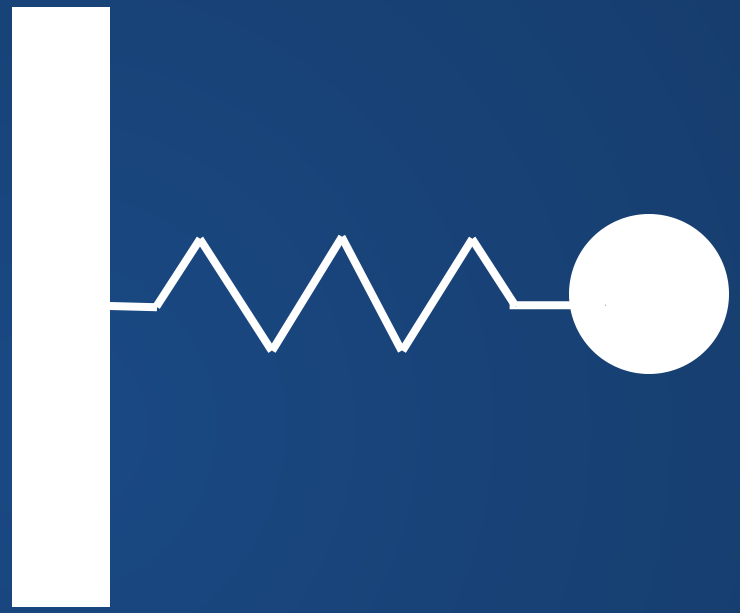
Timestep 0

Consider the following



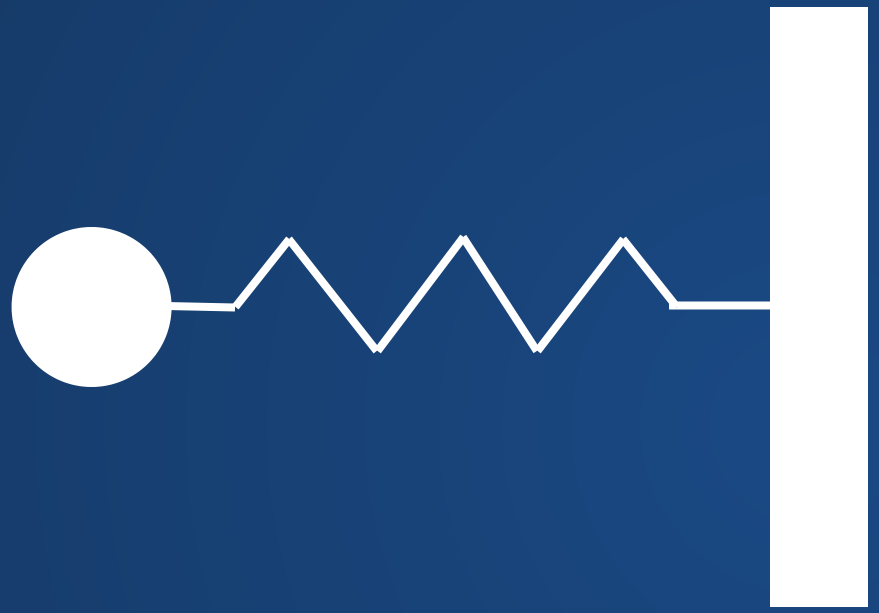
Timestep 1

Consider the following



Timestep 2

Consider the following



Timestep 3

Consider the following

Timestep 4

Consider the following

Uh oh….

If the timestep is too large

We can enter a cycle where our forces spiral out of control.

Will *always* happen with a sufficiently large timestep, though things like the integrator and spring stiffness will play a role.

```
 1 float y;
 2 float vy;
 3 float m = 1.0;
 4 float ry = 250;
 5 float ks = 100.0;
 6 float dt = 2.0;
 7
 8 void setup() {
 9   size(500, 500);
10 }
```

```
1 void draw() {
2   background(210);
3   float f = -(ks * (y - ry)) + 0
4   float a = f/m;
5   vy = vy + dt*a;
6   y += dt*vy;
7   rect(200, y, 100, 20);
8 }
```

Hands-On: Physical Sim

1. Double-check your simulation from yesterday: the `applyForce()` method should **set** the force acting on the particle (not modify it!), while the `advanceTime()` method should modify the velocity and position vectors using the explicit Euler method.

2. Modify your particle to account for mass. You can do this either in `applyForce()` or `advanceTime()`.

3. Attach your particle to a spring (pick a set of parameters that looks nice) and animate this by passing the force to `applyForce()`.

4. OPTIONAL: Make a bungee cord by linking a set of springs together. For extra fun, add gravity and allow the user to stretch the cord by clicking + dragging on the endpoint.

Index Cards!

1. Your name and EID.

2. One thing that you learned from class today. You are allowed to say "nothing" if you didn't learn anything.

3. One question you have about something covered in class today. You *may not* respond "nothing".

4. (Optional) Any other comments/questions/thoughts about today's class.