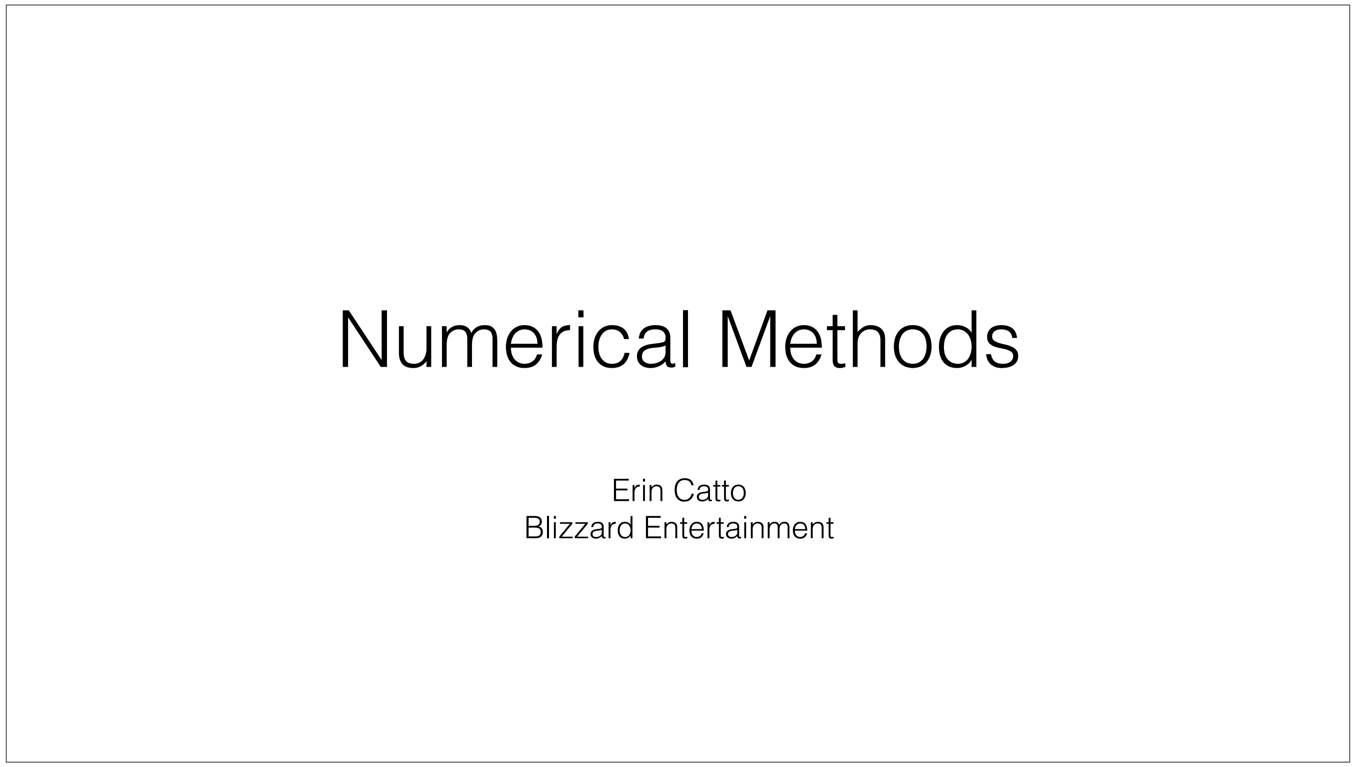


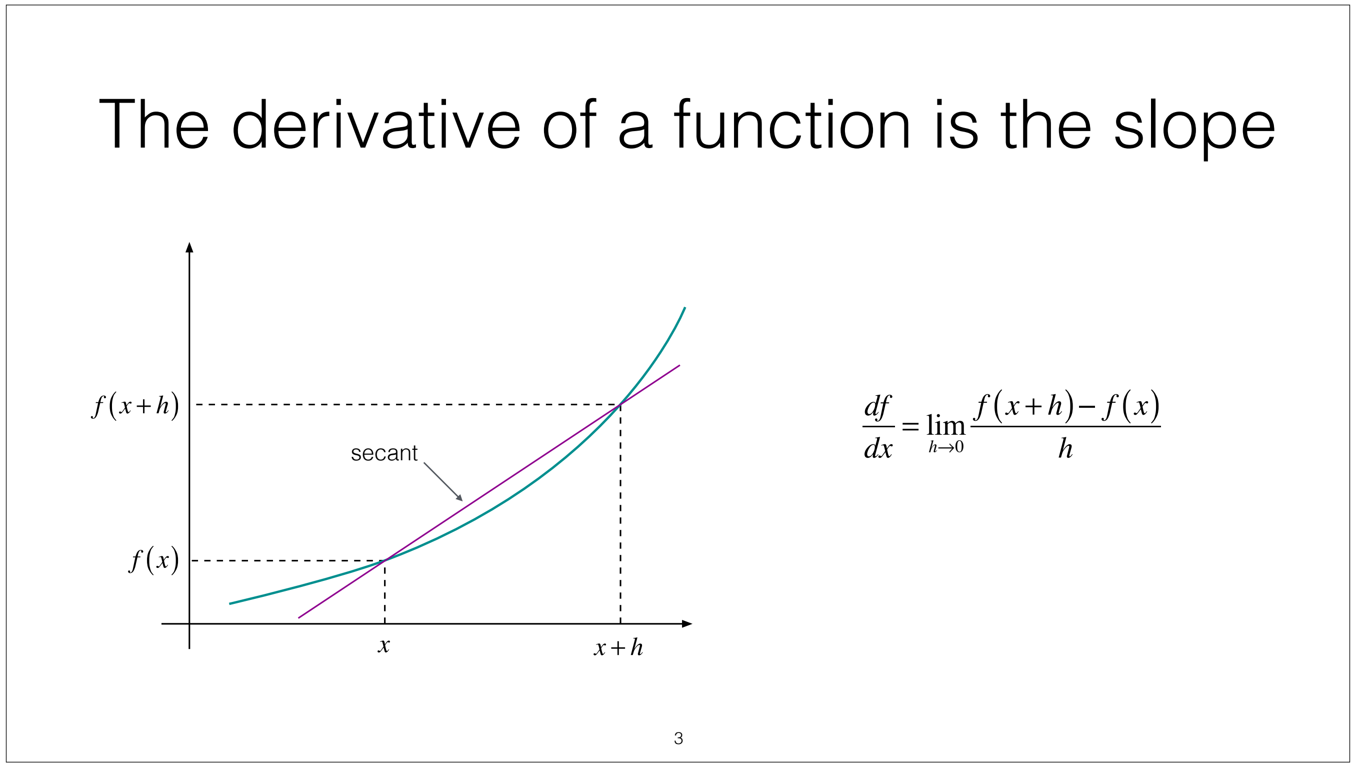
Numerical Methods

Erin Catto
Blizzard Entertainment

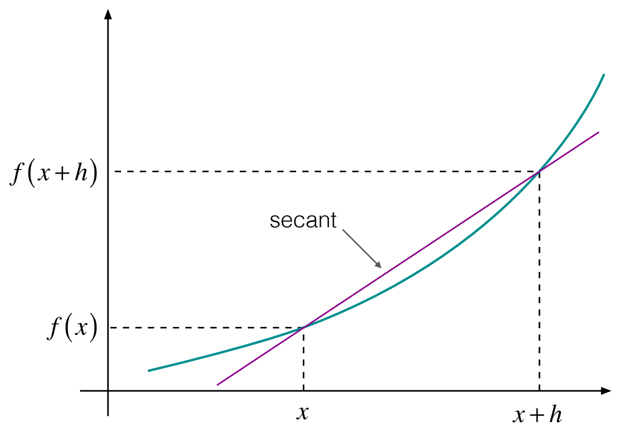
Sometimes the mathematical problems we are faced with in game physics are too difficult to solve exactly. Numerical methods provide a set of tools to get approximate solutions to these difficult problems. In particular, numerical methods help us get approximate solutions to the differential equations that arise in rigid body and point mass physics.
There are many numerical methods available and it isn't always clear which method to use in which situation. What is the right performance versus accuracy trade-off? Will my simulation blow up? What are the common methods used in game physics?
I hope this presentation helps you better understand numerical methods and their application.

First, a little review …

Differential Calculus



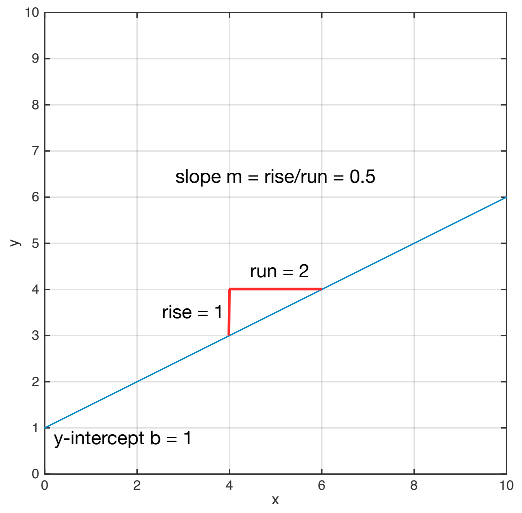
The derivative of a function is the slope



$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

The derivative of a function f(x) is the slope at position x. The derivative is defined as the limit of the secant as the increment h goes to zero.

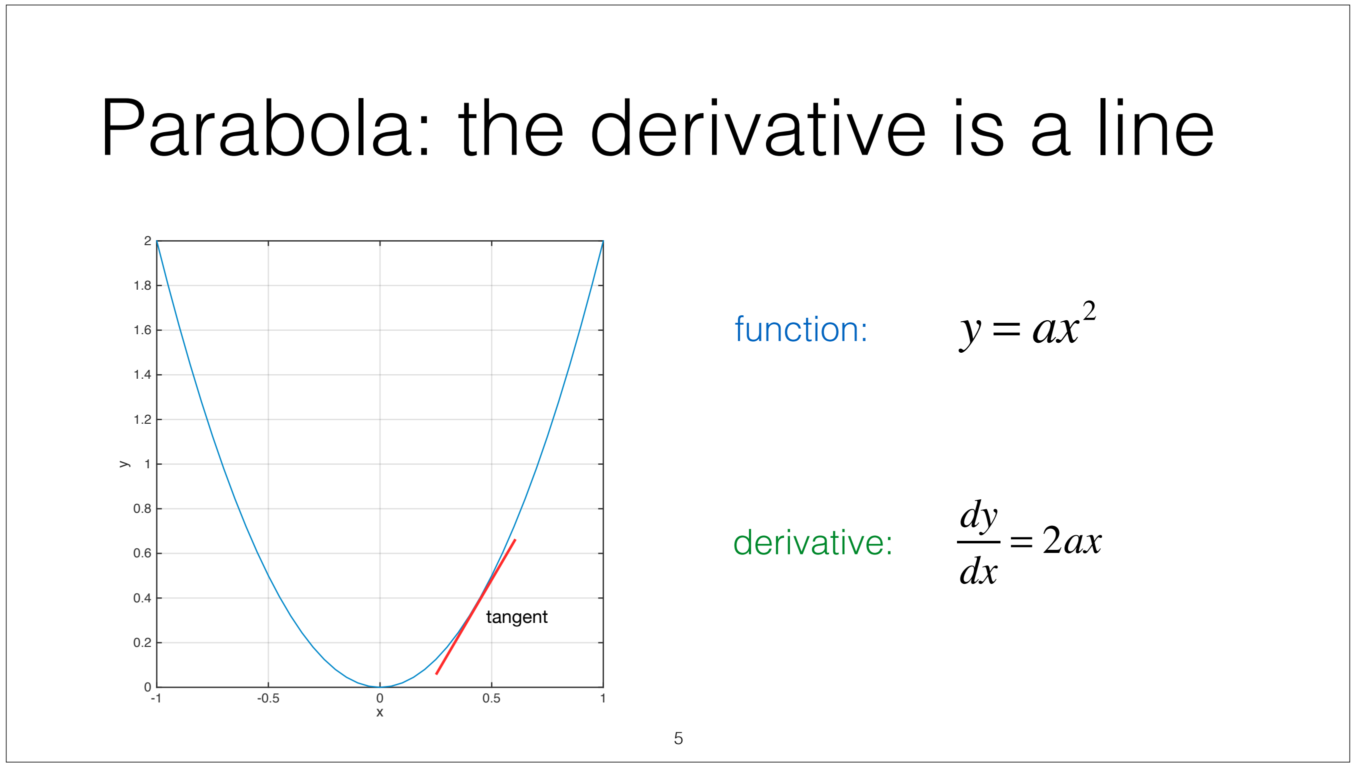
Line: the derivative is constant



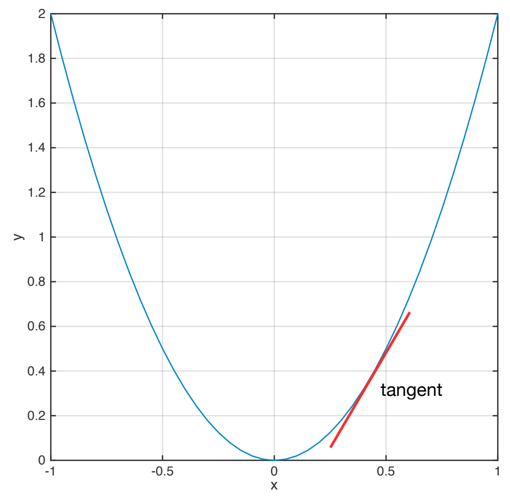
function: $y = mx + b$

derivative: $\dfrac{dy}{dx} = m$

The derivative of a line is the slope, which is the rise over the run. If the line is horizontal, y is constant and the derivative are zero. In other words, y does not change with x.



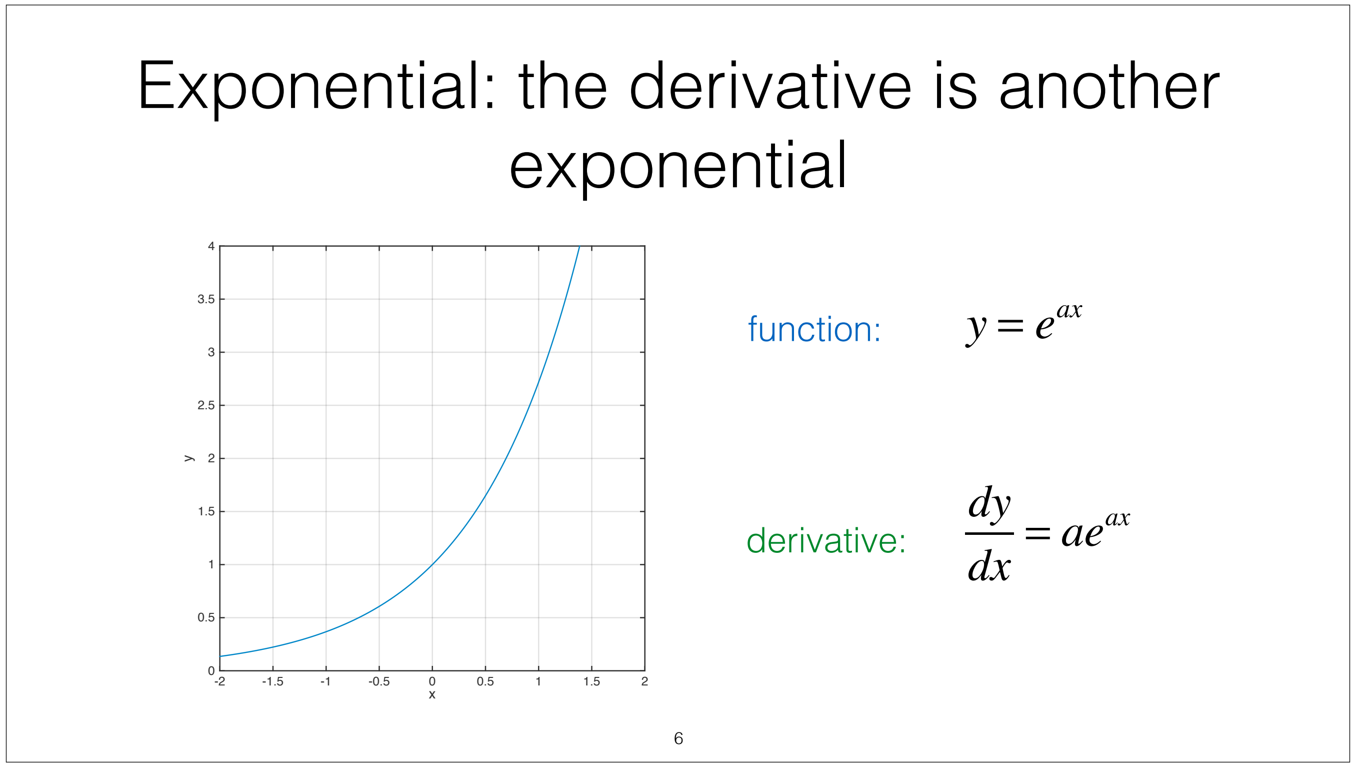
Parabola: the derivative is a line



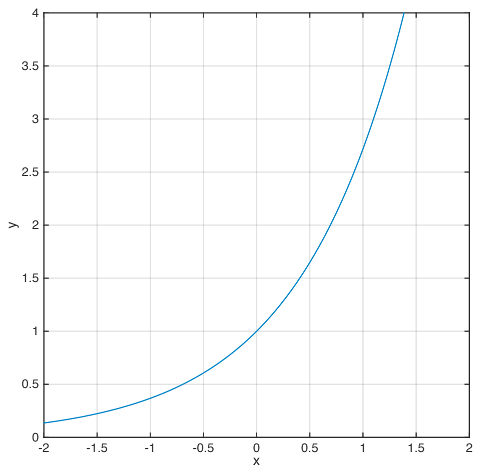
function: $y = ax^2$

derivative: $\frac{dy}{dx} = 2ax$

We can also think of the derivative as the tangent. It is the slope at the current point on the curve. In general the slope changes with the x value.



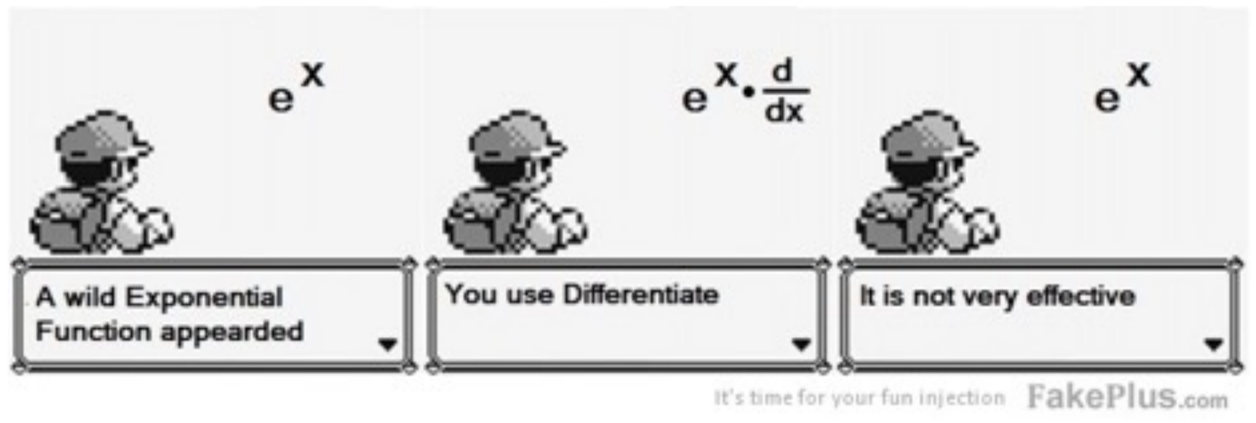
Exponential: the derivative is another exponential



function: $y = e^{ax}$

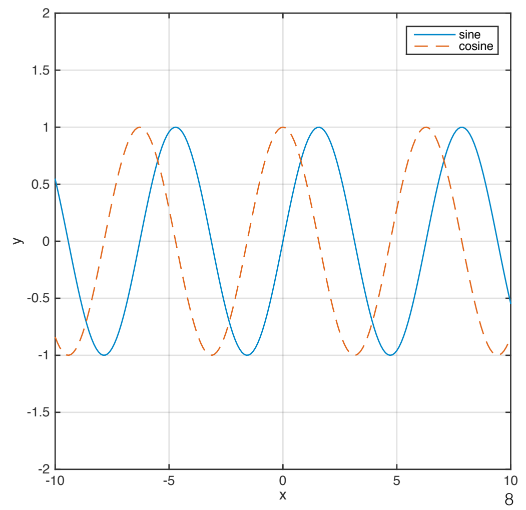
derivative: $\dfrac{dy}{dx} = ae^{ax}$

The exponential shows up a lot in physics especially when there is damping. It has the neat property that its derivative is another exponential.


e^x
A wild Exponential Function appearded
$e^x \cdot \frac{d}{dx}$
You use Differentiate
e^x
It is not very effective
It's time for your fun injection FakePlus.com

Now a bit of Pokemon math.

Sine wave: the derivative is another sine wave



function: $y = \sin x$

derivative: $\dfrac{dy}{dx} = \cos x$

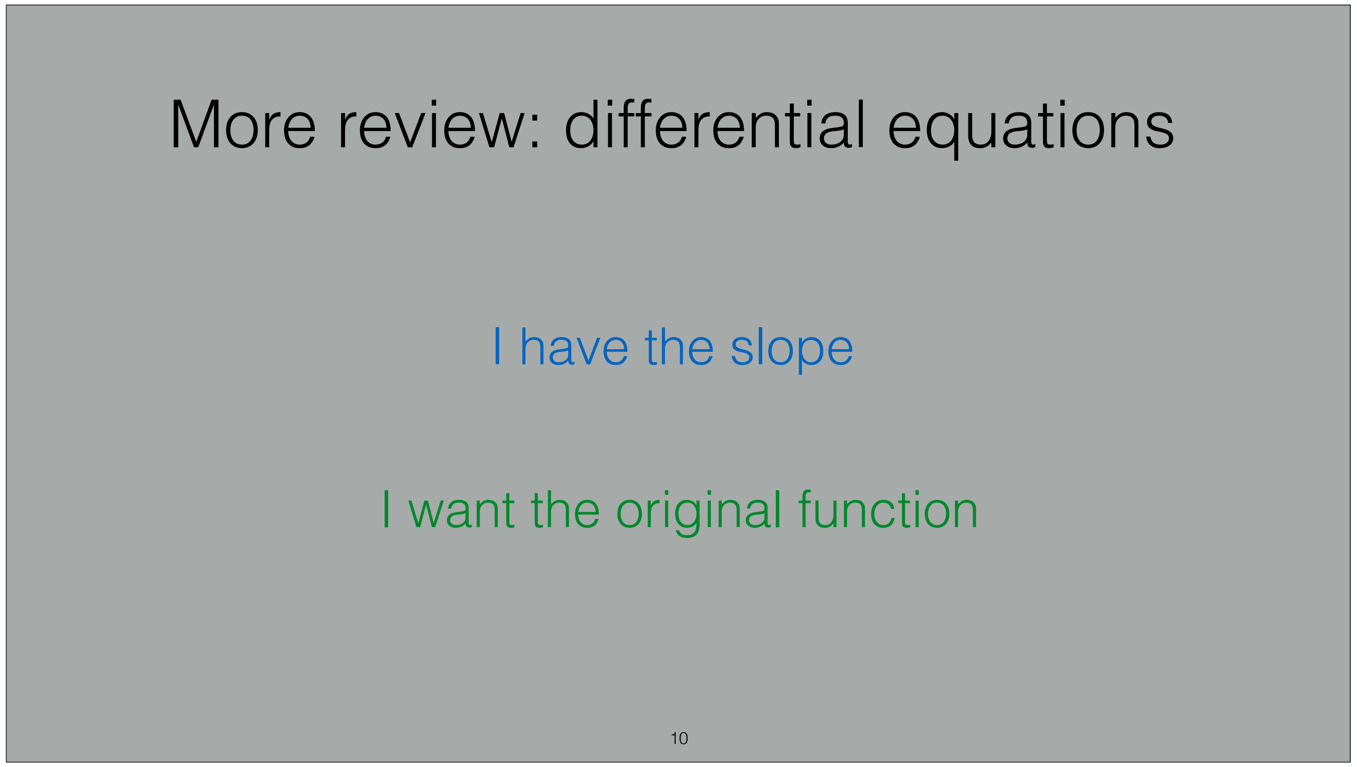
The derivative of a sine wave is a cosine wave.

Functions of time: the derivative is the rate of change

velocity: $\dot{x} = \dfrac{dx}{dt}$

acceleration: $\ddot{x} = \dfrac{d^2x}{dt^2}$

In physics we usually deal with functions of time. If x is position, then x dot is velocity and x double dot is acceleration. I'll be using this dot notation a lot.

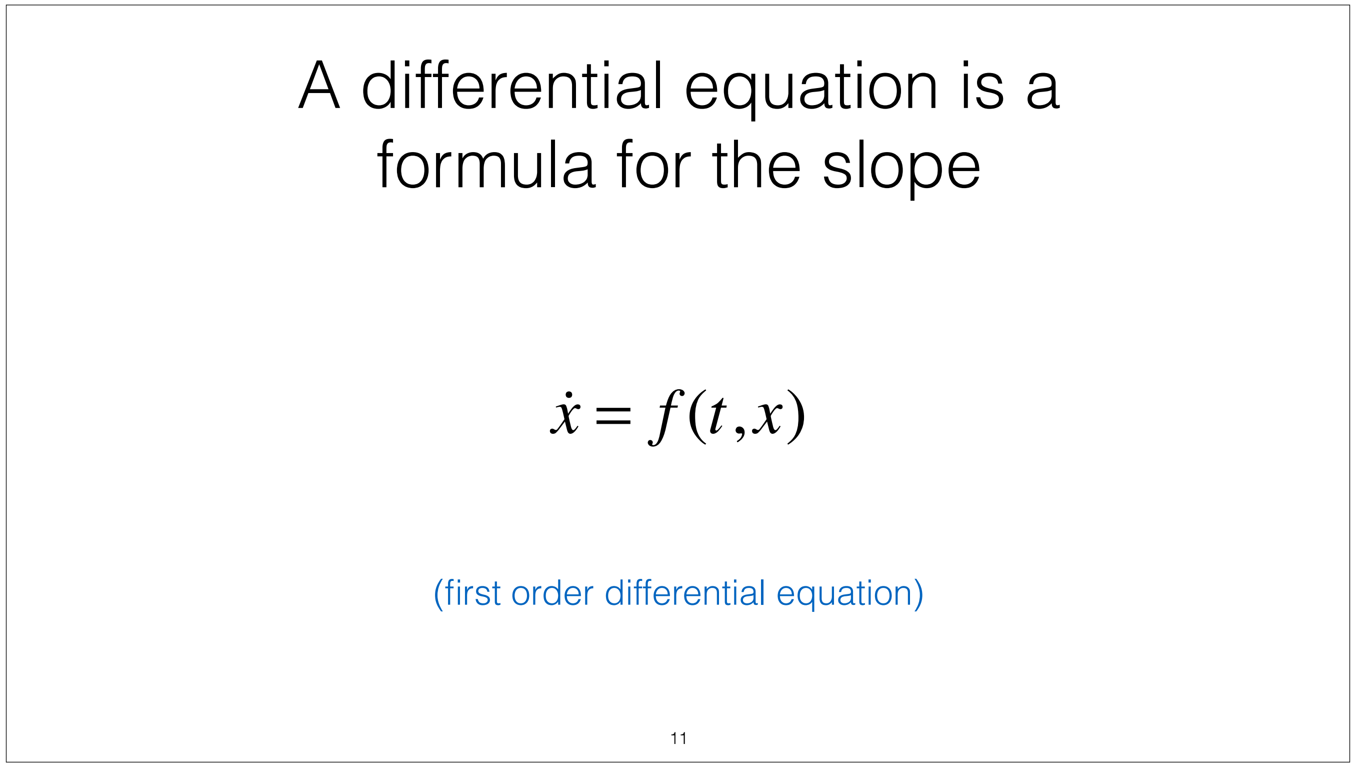


More review: differential equations

I have the slope

I want the original function

I’ve showed a few examples of you can compute a derivative given a function. What if we have the derivative and want to get the function back?

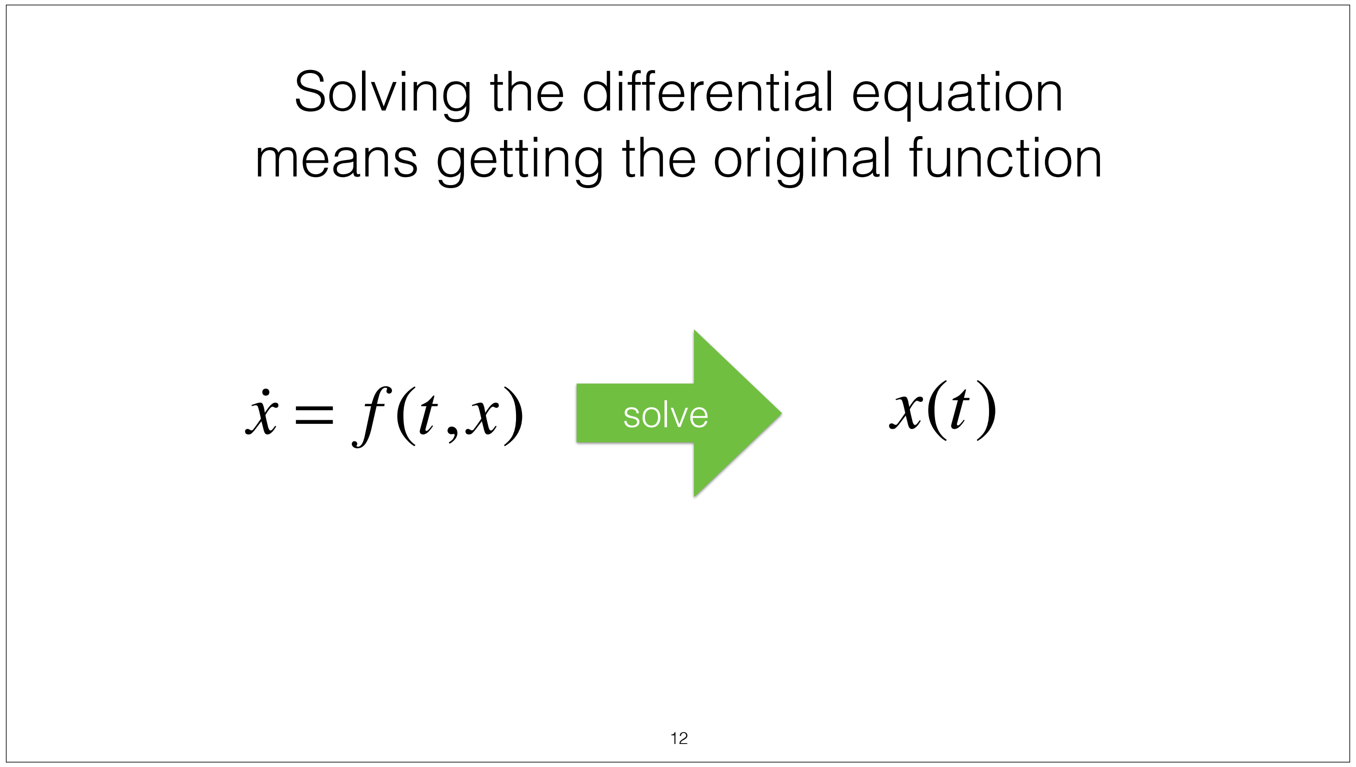


A differential equation is a formula for the slope

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

(first order differential equation)

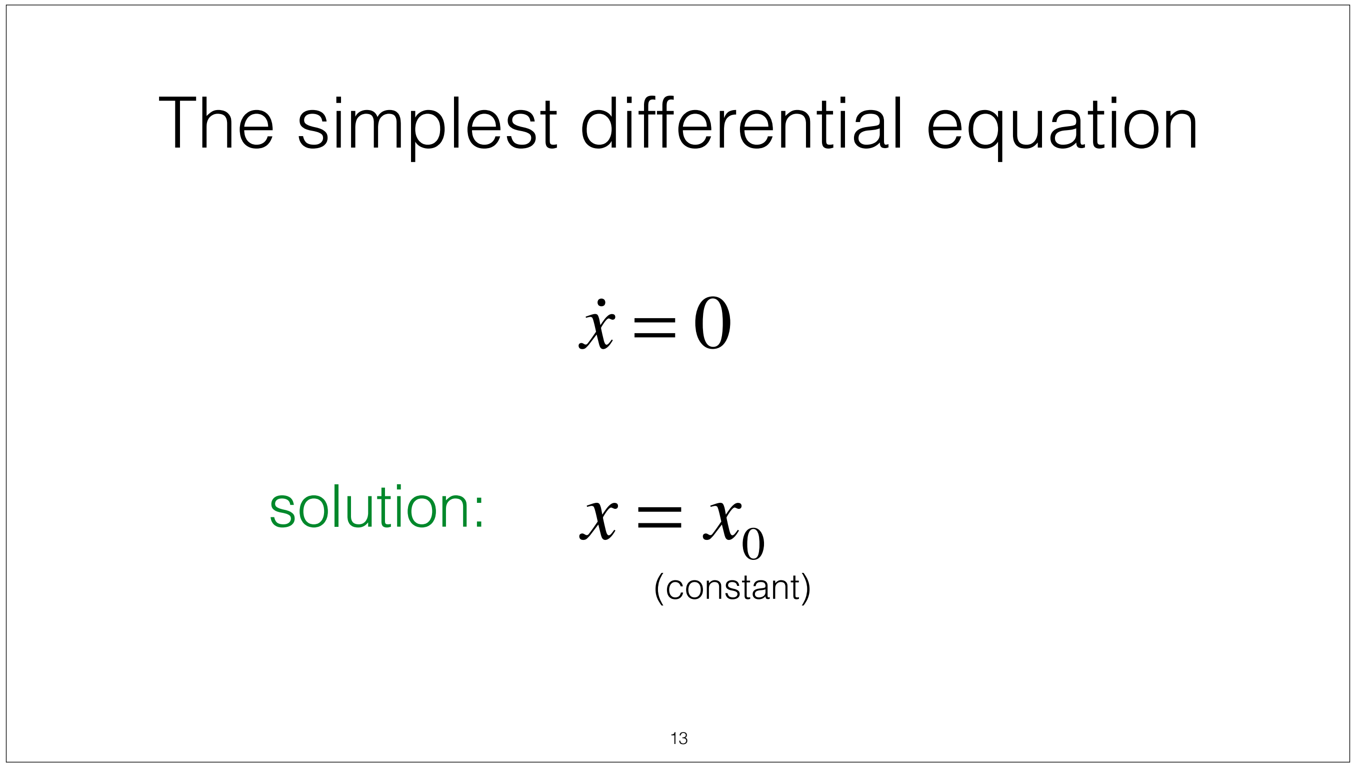
This is the standard form of a first order differential equation in time.



Solving the differential equation means getting the original function

$$\dot{x} = f(t,x) \xrightarrow{\text{solve}} x(t)$$

Solving a differential equation means getting the original function back. If our function f depends only on t, we only need to perform integration. However, if f depends on x, then solution becomes more challenging.



The simplest differential equation

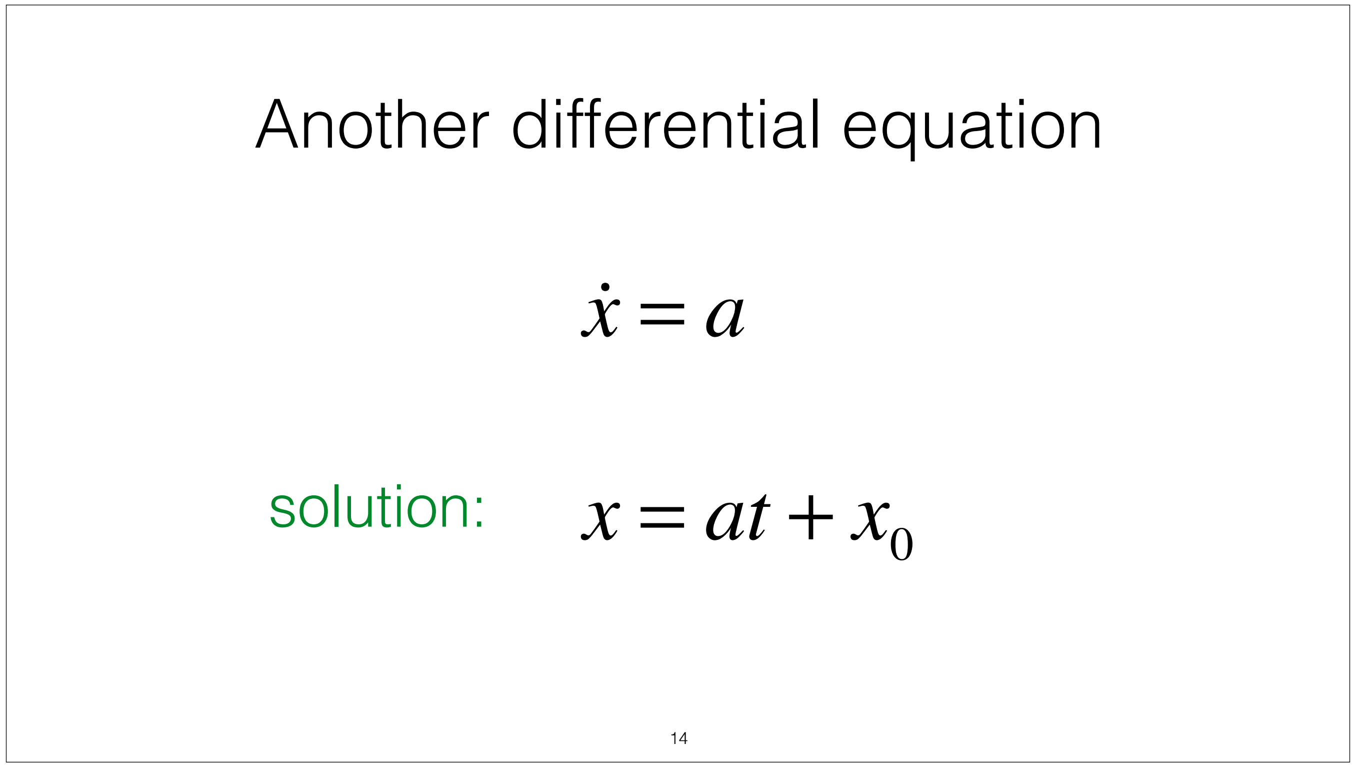
$$\dot{x} = 0$$

solution: $x = x_0$

(constant)

So lets start solving some differential equations!

Here’s is the simplest differential equation I could find and its solution.

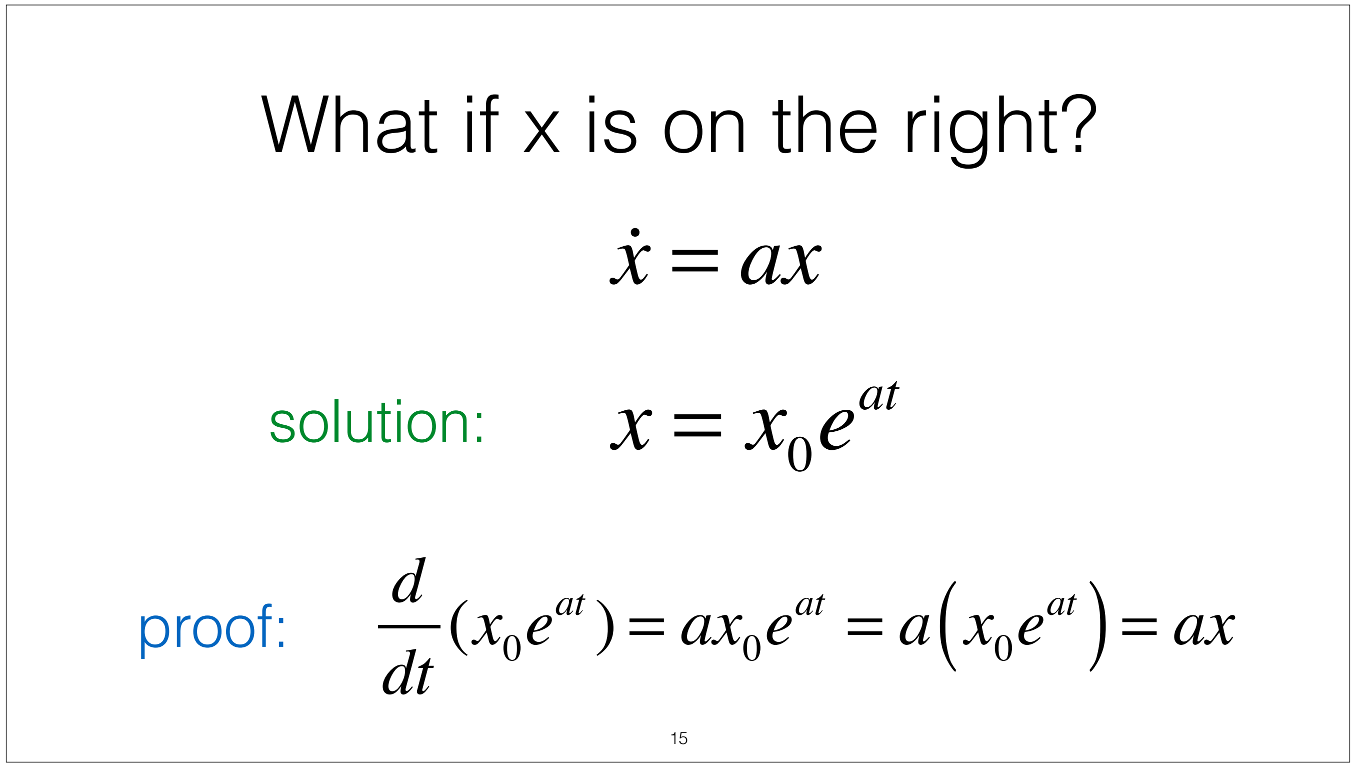


Another differential equation

$$\dot{x} = a$$

solution: $$x = at + x_0$$

This is just a line.



What if x is on the right?

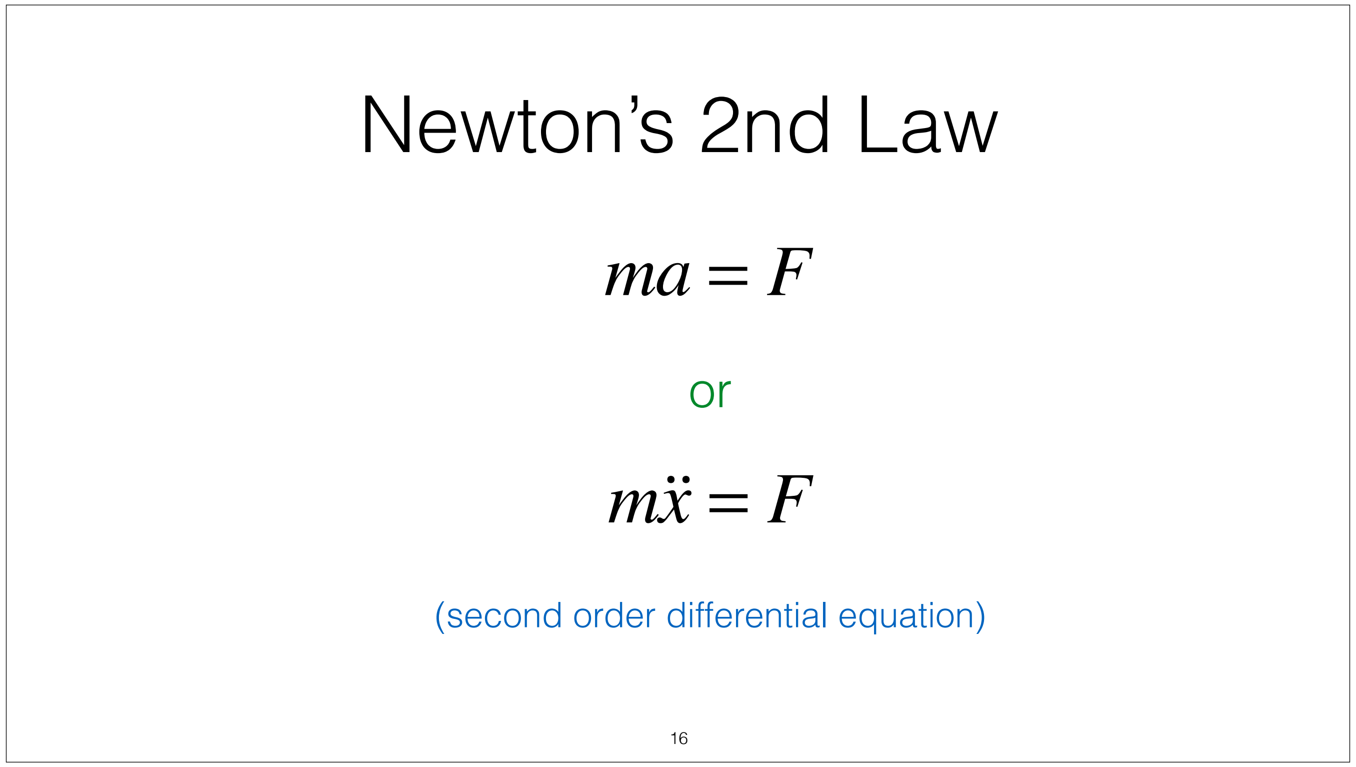
$$\dot{x} = ax$$

solution: $x = x_0 e^{at}$

proof: $\frac{d}{dt}(x_0 e^{at}) = a x_0 e^{at} = a\left(x_0 e^{at}\right) = ax$

The exponential is needed to solve this differential equation.

Remember that the derivative of an exponential is another exponential.



Newton’s 2nd Law

$$ma = F$$

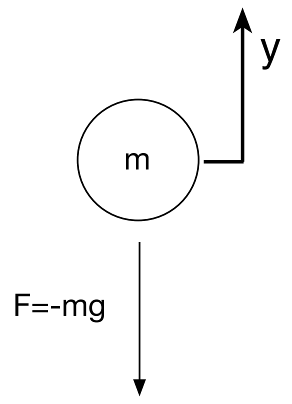
or

$$m\ddot{x} = F$$

(second order differential equation)

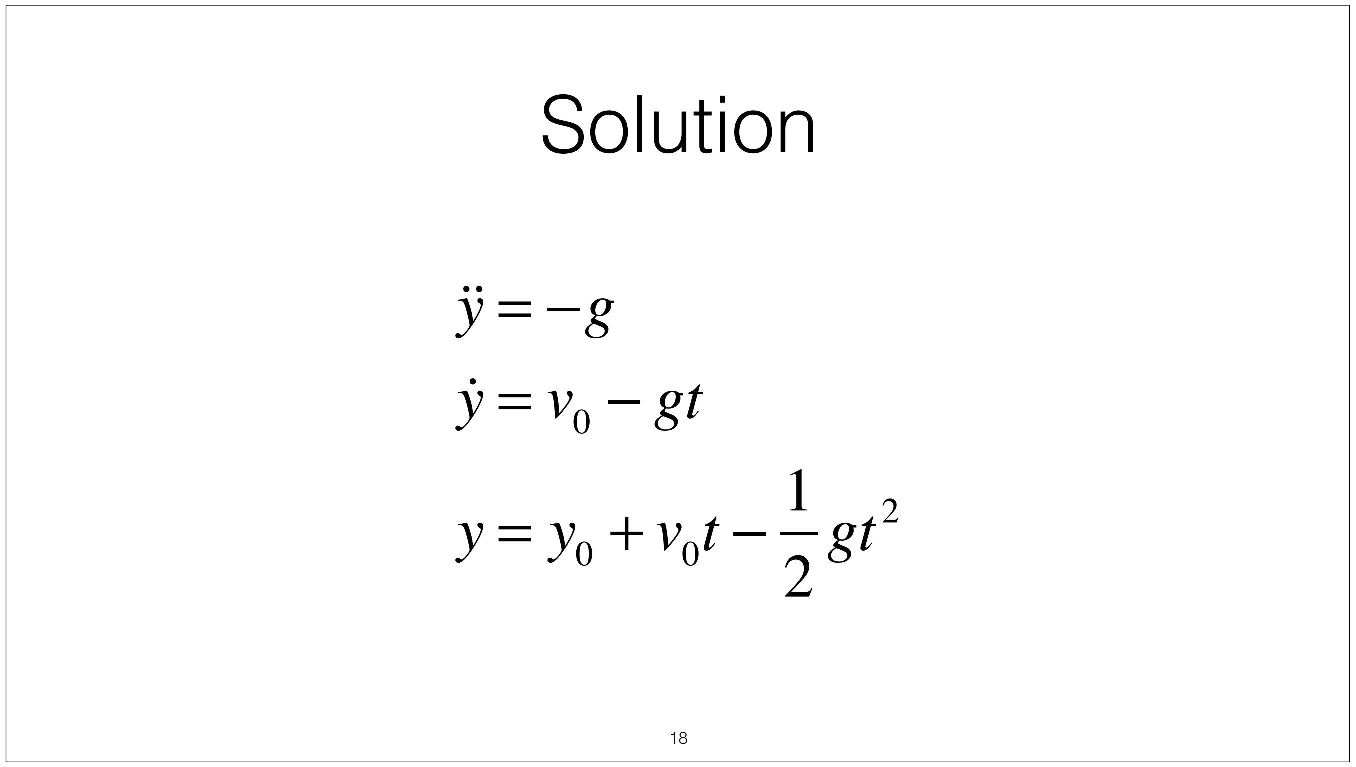
Newton’s law is a differential equation for the acceleration. Because it involves the second derivative, it is a second order differential equation.

Gravity



$$m\ddot{y} = -mg$$

Under gravity the acceleration is constant.



Solution

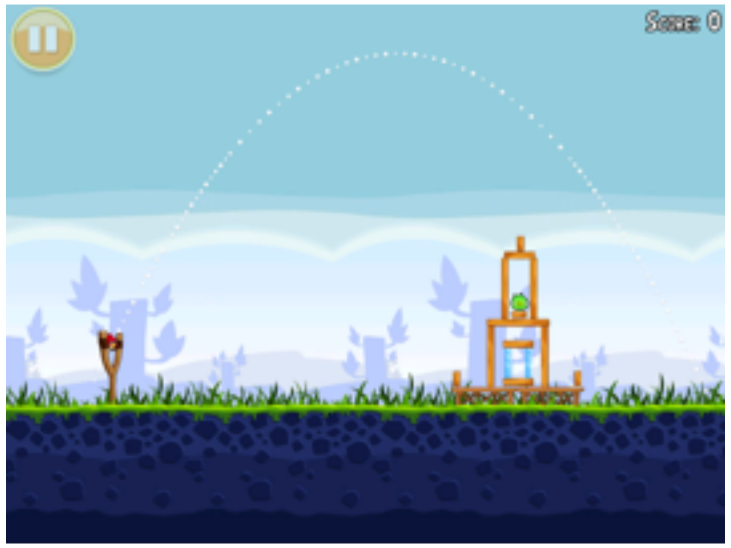
$$\ddot{y} = -g$$

$$\dot{y} = v_0 - gt$$

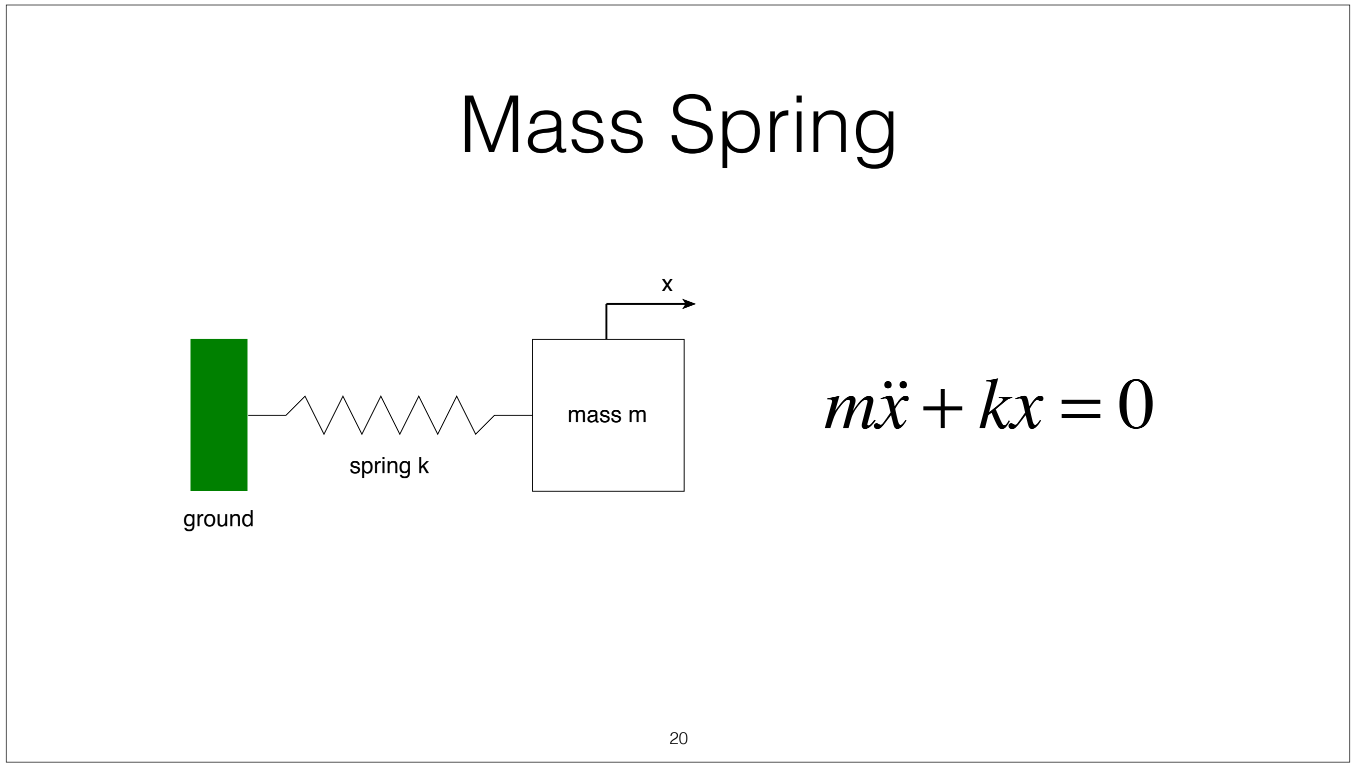
$$y = y_0 + v_0 t - \frac{1}{2} g t^2$$

Solving for a constant force just requires the integration (anti-derivative). I’m also accounting for the initial position y0 and velocity v0.

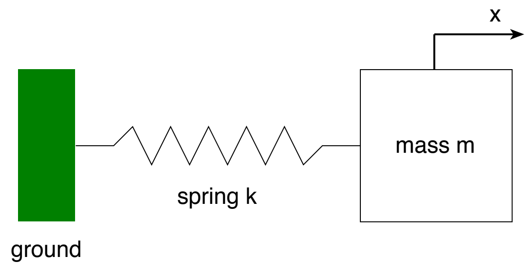
Parabolic motion



This is why projectiles follow a parabolic path.

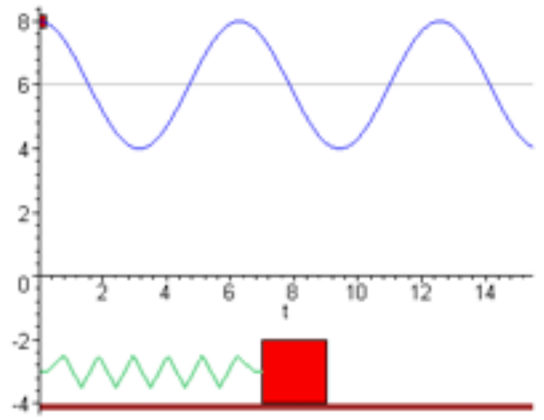


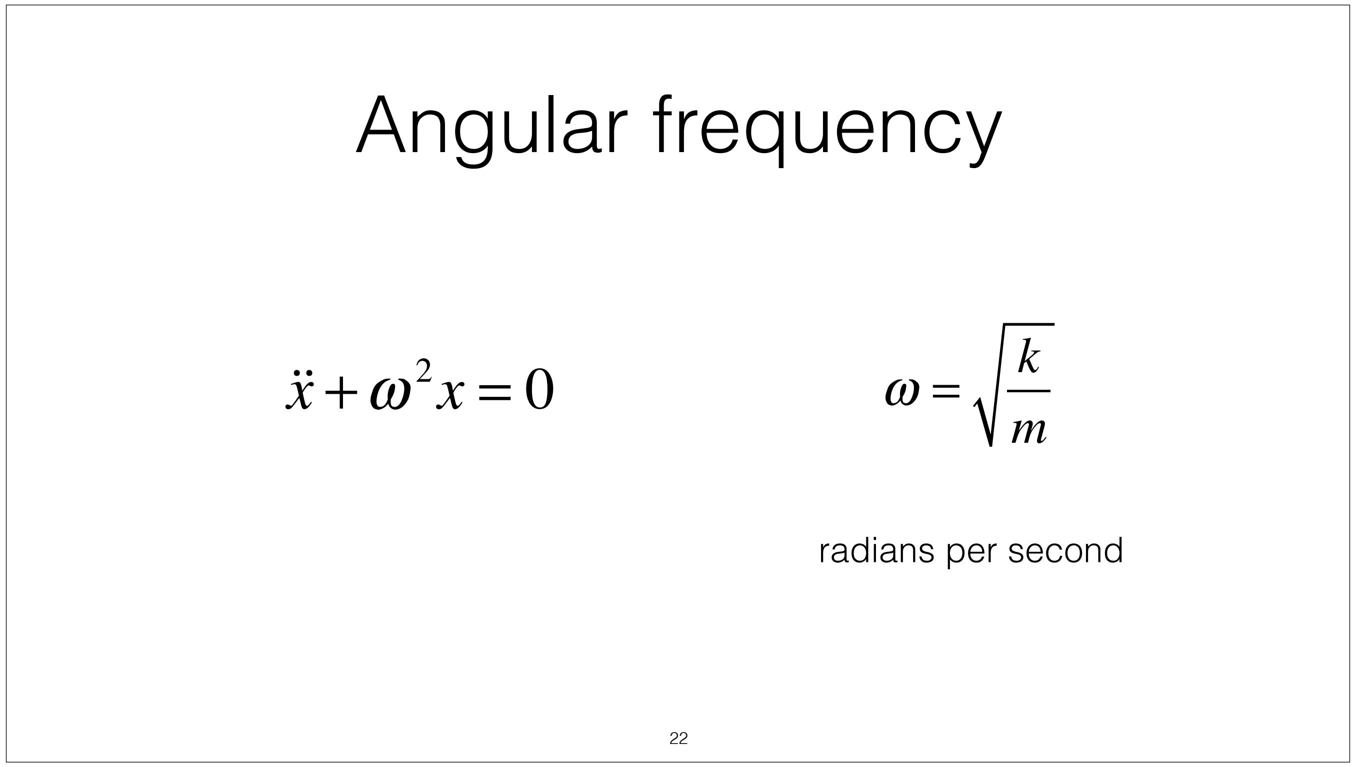
Mass Spring



$$m\ddot{x} + kx = 0$$

The mass spring system has a force that depends on position. The more the spring is stretched, the harder it pulls.


8
6
4
2
0
-2
-4
2
4
6
8
10
12
14
t



Angular frequency

$$\ddot{x} + \omega^2 x = 0$$

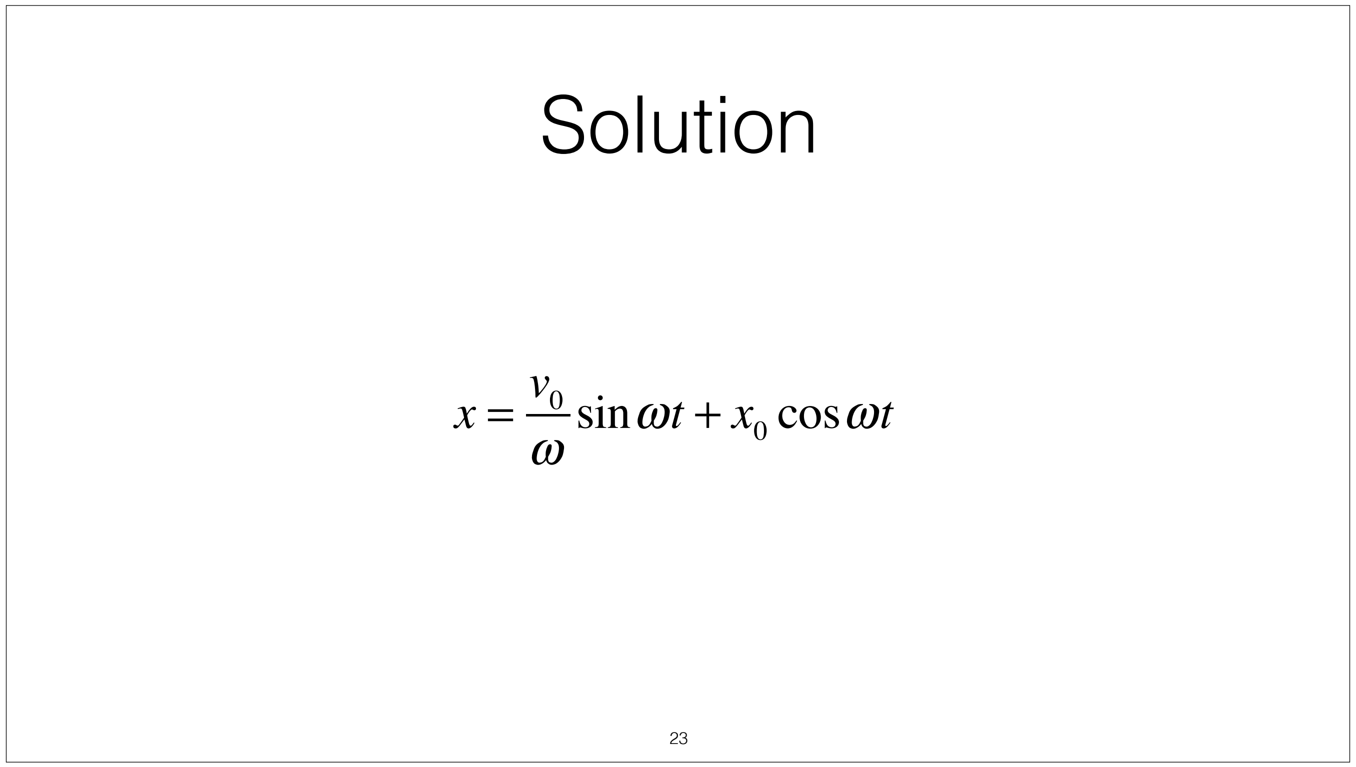
$$\omega = \sqrt{\frac{k}{m}}$$

radians per second

22

I can divide through by the mass to go from two parameters (m,k) to one parameter (omega).

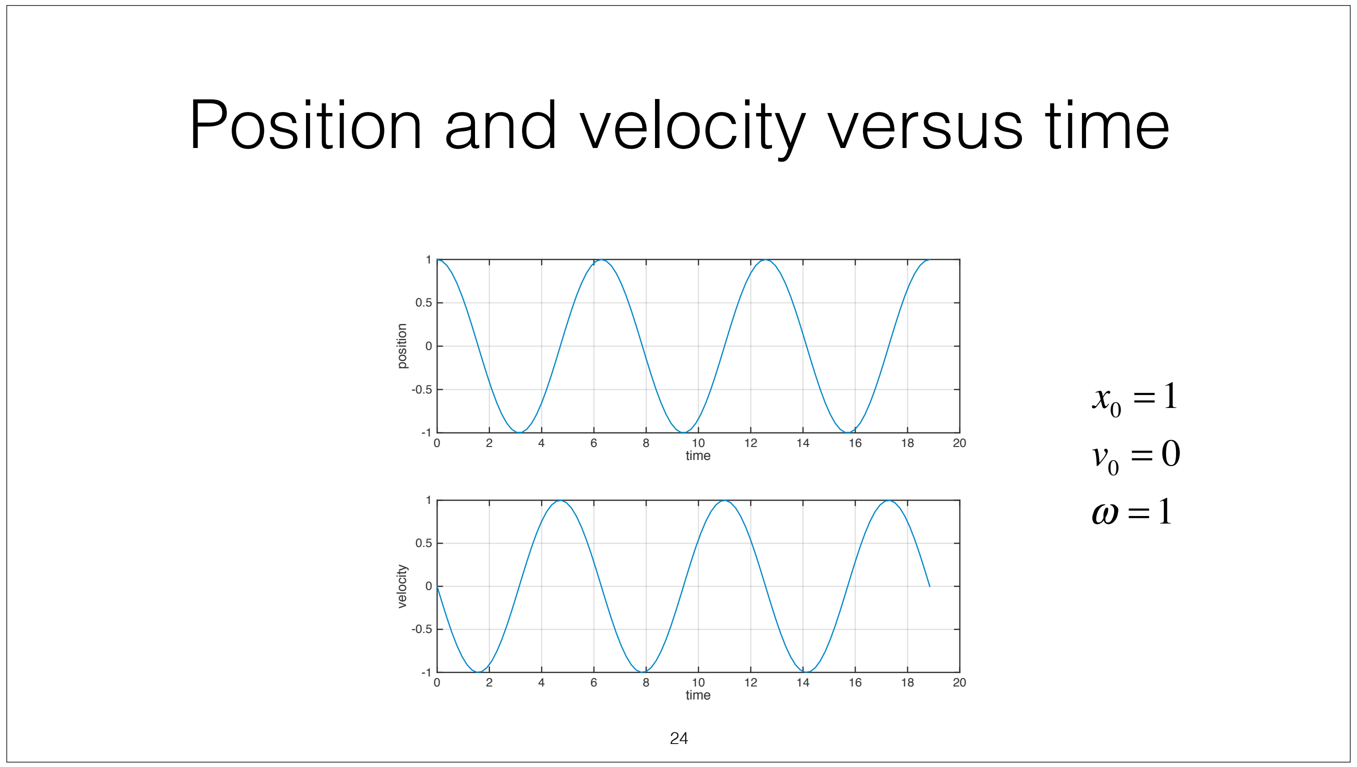
Omega is the angular frequency of the mass-spring system. Notice that the angular frequency increases with the spring stiffness and decreases with the mass.



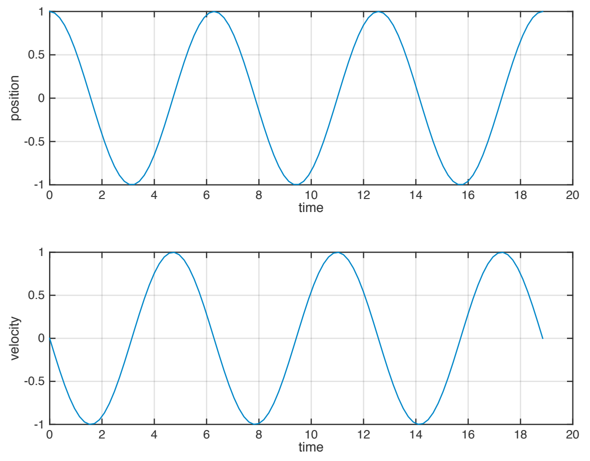
Solution

$$x = \frac{v_0}{\omega}\sin\omega t + x_0\cos\omega t$$

The solution is a sine wave with angular frequency omega. We need to combine a sine wave and cosine wave to account for initial conditions.

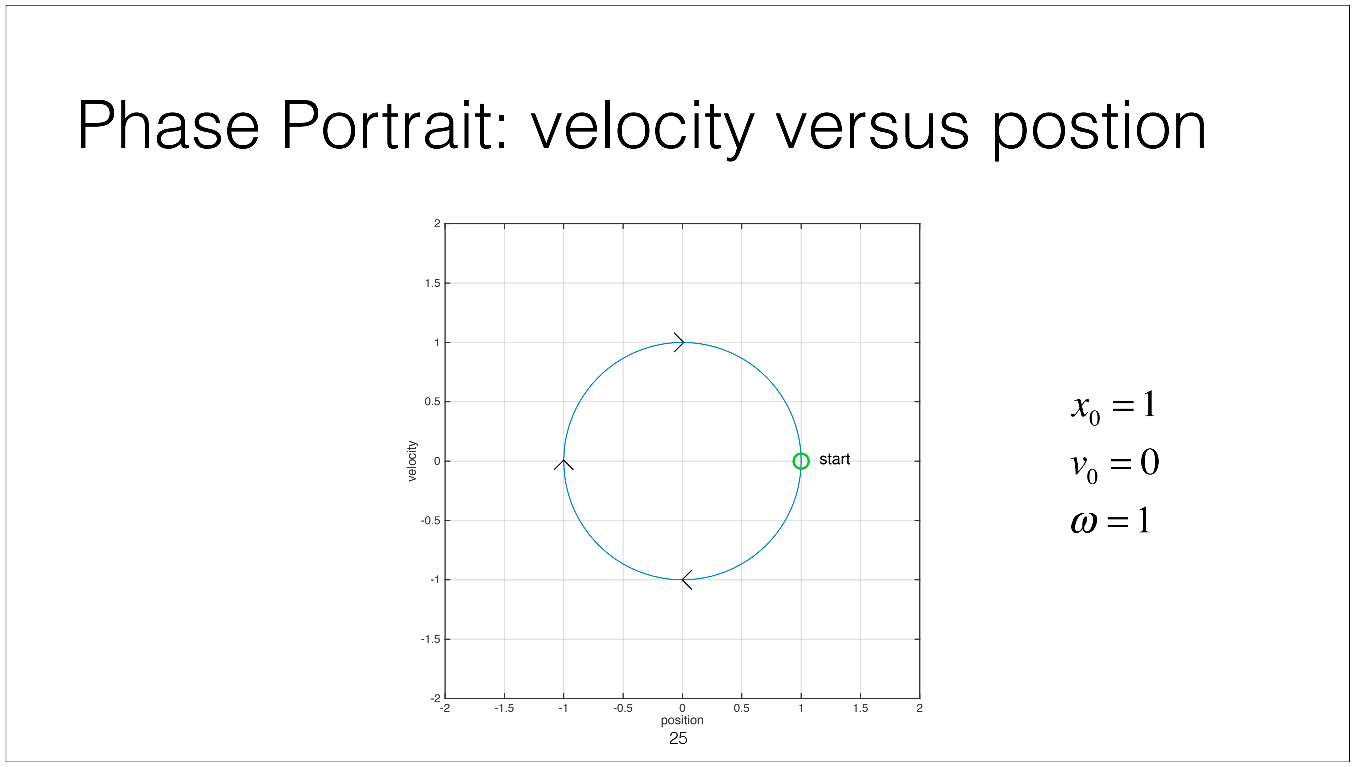


Position and velocity versus time

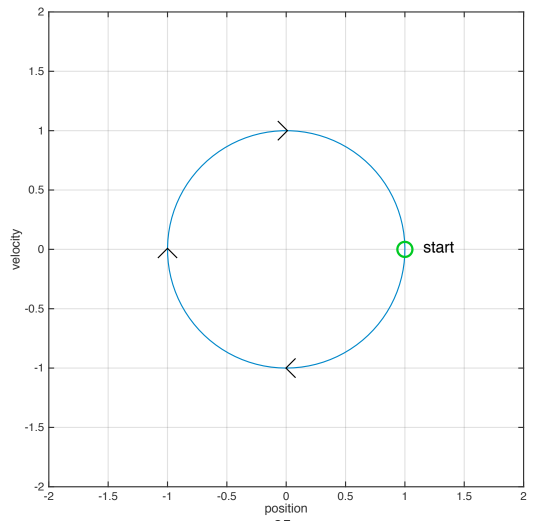


$$x_0 = 1$$
$$v_0 = 0$$
$$\omega = 1$$

These graphs show position and velocity versus time. Notice that position and velocity are 90 degrees out of phase. So the position is a cosine wave and the velocity is a sine wave.



Phase Portrait: velocity versus postion

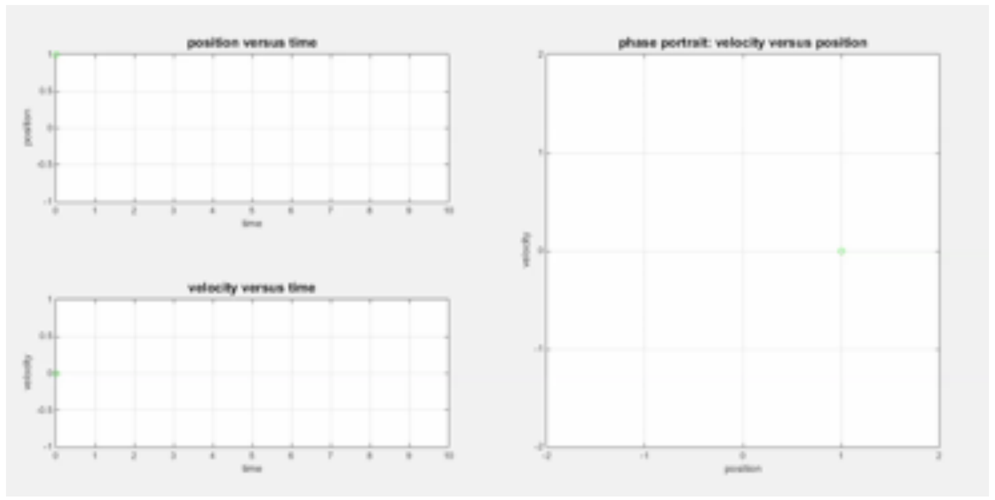


$x_0 = 1$
$v_0 = 0$
$\omega = 1$

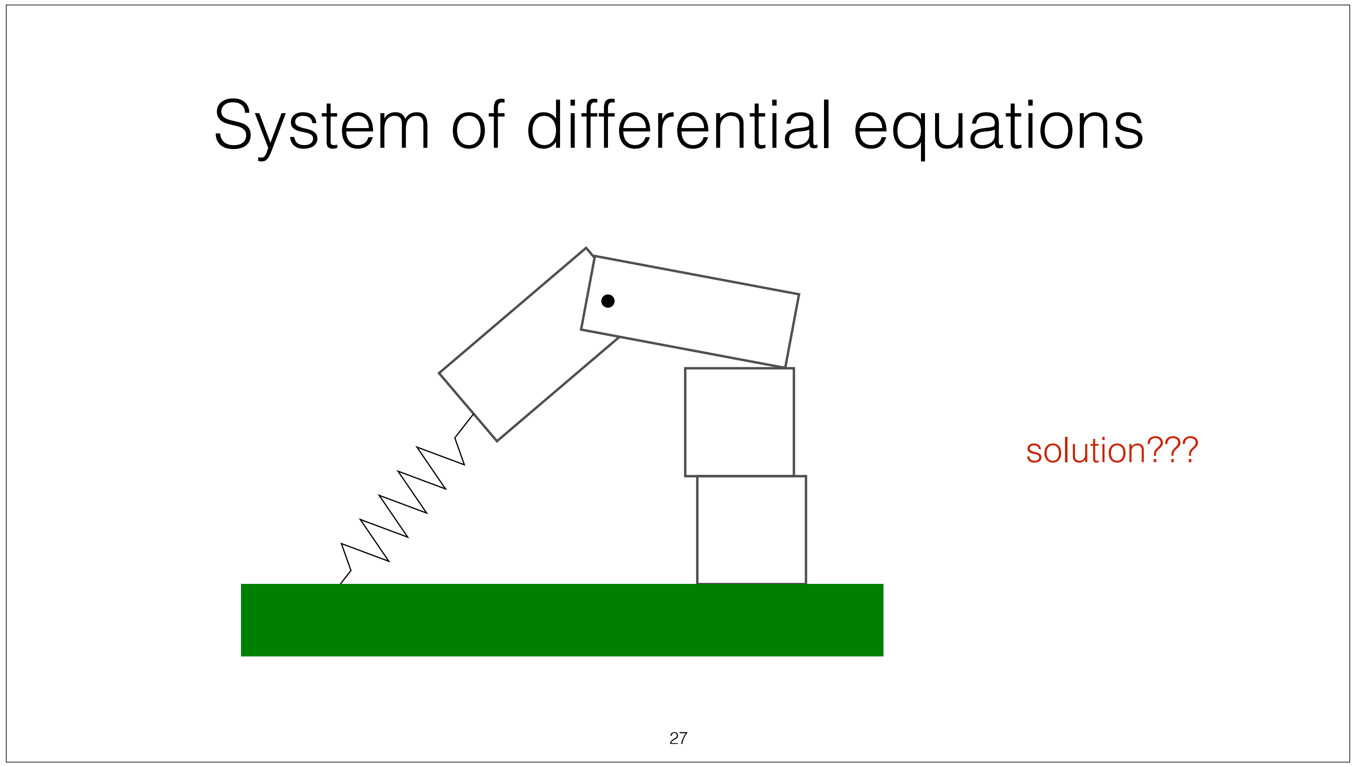
A nice way to visualize the oscillation is to plot velocity versus position. You can view this as a parametric curve with time as the curve parameter. This is drawn here as a unit circle, but keep in mind that position and velocity have different units. The arrows show the direction of increasing time. The position and velocity start at the green circle and progressing clockwise around the circle. And the state continues around the circle forever. The angular frequency controls how fast the state goes around the circle as time moves forward.

This graph is sometimes called the *phase plot* or *phase portrait*. This plot shows all possible states (phases) of the system as time goes forward.

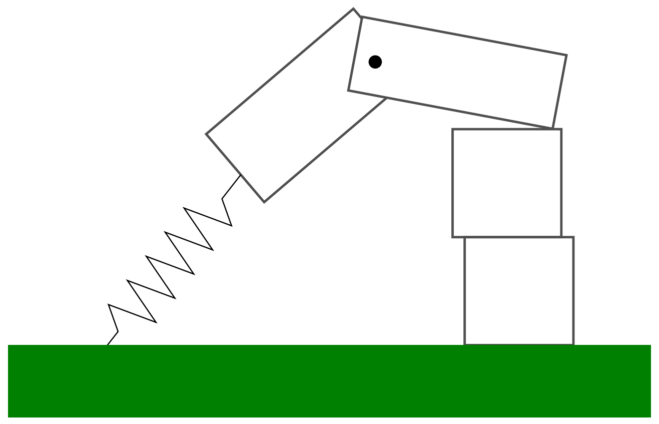
Phase portrait: velocity versus postion



This animation shows position and velocity versus time along with the phase portrait.



System of differential equations

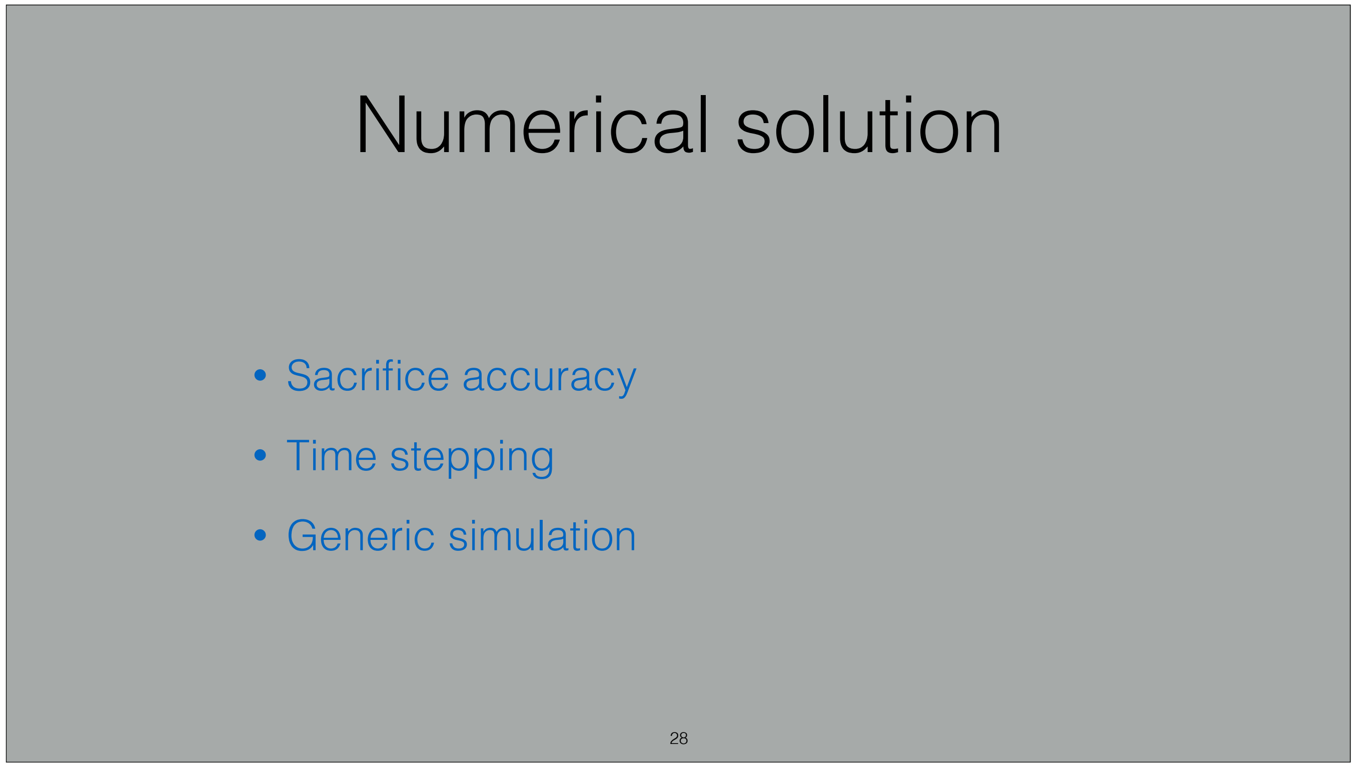


solution???

So far I have showed some simple physical systems, such as a projectile and a mass-spring system.

The simulations in modern games are more complex. Often there are multiple rigid bodies colliding with friction. This means that there are multiple mutually dependent differential equations. We have little hope of finding the exact solutions for these complex systems of differential equations.

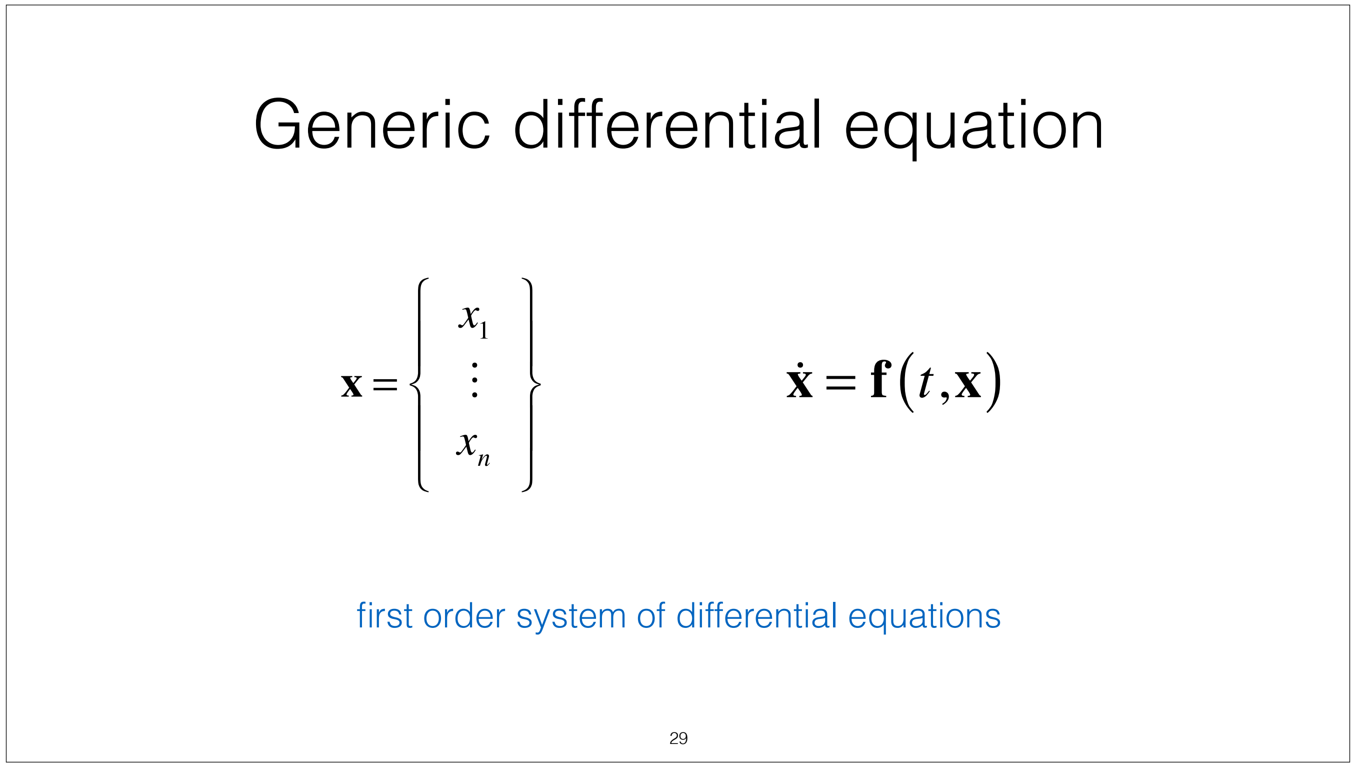
Therefore we need to turn to numerical methods.



Numerical solution

- Sacrifice accuracy
- Time stepping
- Generic simulation

We can simulate systems of differential equations using numerical methods. This means we are abandoning exact answers. But we are making games, so this is okay. You will see that numerical solution works well with typical game loops. Numerical solution also creates a nice abstraction that allows us to solve differential equations in a generic manner. These methods can handle any number of rigid bodies and any number of forces (as memory and processor power allow).



Generic differential equation

$$\mathbf{x} = \left\{ \begin{matrix} x_1 \\ \vdots \\ x_n \end{matrix} \right\} \qquad \dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x})$$

first order system of differential equations

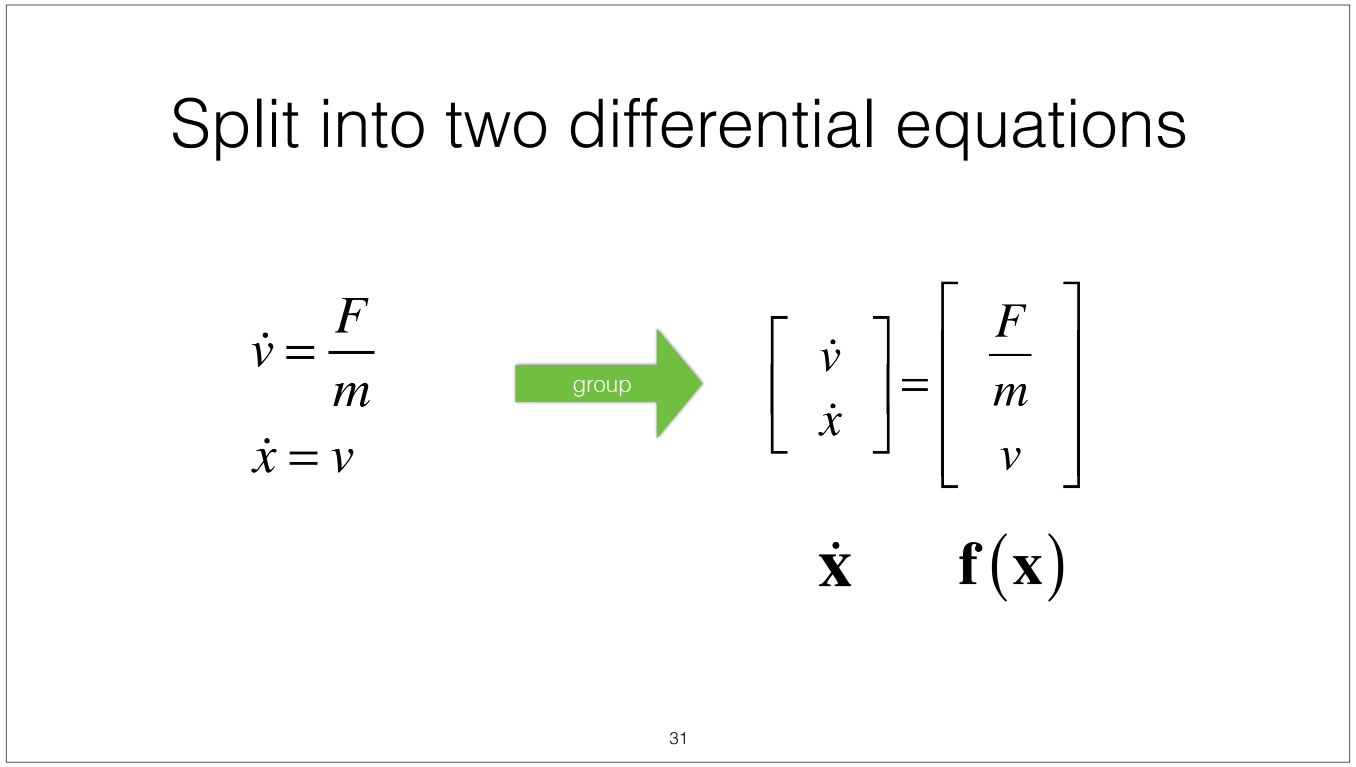
Numerical methods like to have differential equations in a standard form. We can have multiple differential equations assembled into an array. That way we can solve them all at once. This also allows us to account for objects that are interacting and have mutually dependent solutions, such as stacking of rigid bodies.

Handling second order differential equations

have: $m\ddot{x} = F$

want: $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$

I will be dealing with Newton's law and I need a way to get this into the first order form needed by numerical methods.

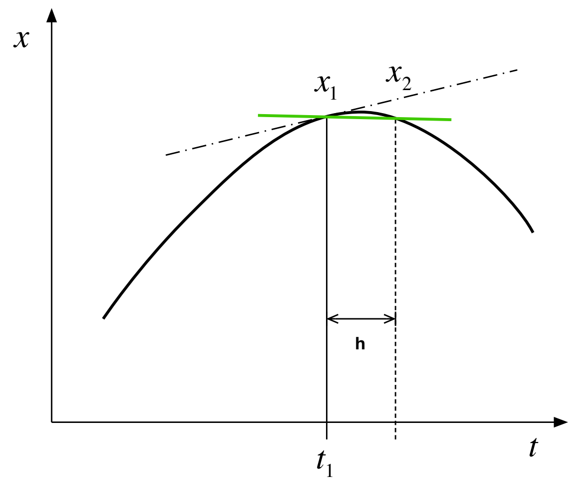


Split into two differential equations

$$\begin{aligned}\dot{v} &= \frac{F}{m}\\ \dot{x} &= v\end{aligned} \quad \xrightarrow{\text{group}} \quad \underbrace{\begin{bmatrix}\dot{v}\\ \dot{x}\end{bmatrix}}_{\dot{\mathbf{x}}} = \underbrace{\begin{bmatrix}\frac{F}{m}\\ v\end{bmatrix}}_{\mathbf{f}(\mathbf{x})}$$

The idea is to convert a single second order differential equation into two first order differential equations. This is accomplished by introducing the velocity as an independent state. Then I get a first order DE for the velocity and a first order DE for the position.

Finite Difference

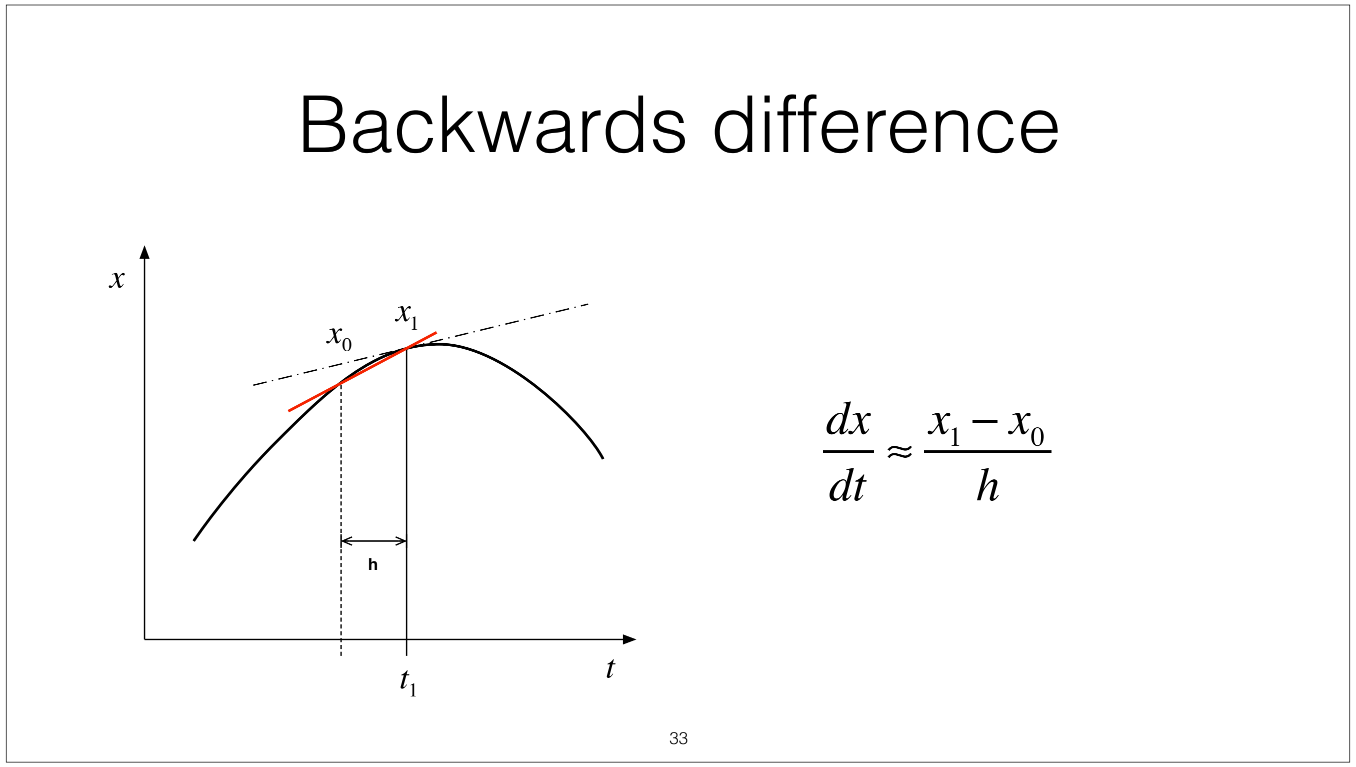


$$\frac{dx}{dt} \approx \frac{x_2 - x_1}{h}$$

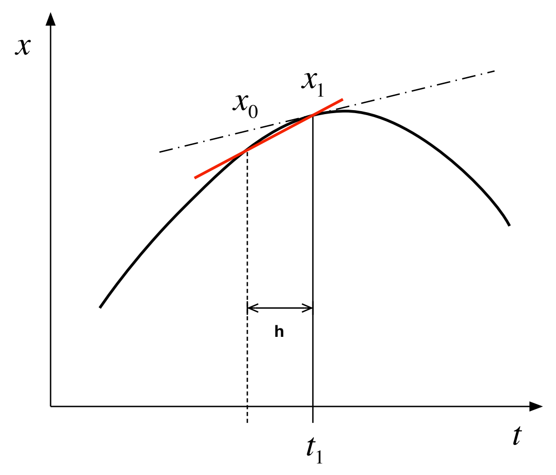
forward difference

The basic approach of numerical solution is to use the finite difference. This allows me to convert a differential equation into an algebraic equation. There are many kinds of finite difference formulas. This one is the forward difference. It approximates the derivative as the next value x2 minus the current value x1 over the time step h.

You can see by the green line that this really is an approximation of the slope at x1. The approximation gets better by decreasing the time step h. However, taking smaller time steps is more expensive because you need to take more of them.



Backwards difference



$$\frac{dx}{dt} \approx \frac{x_1 - x_0}{h}$$

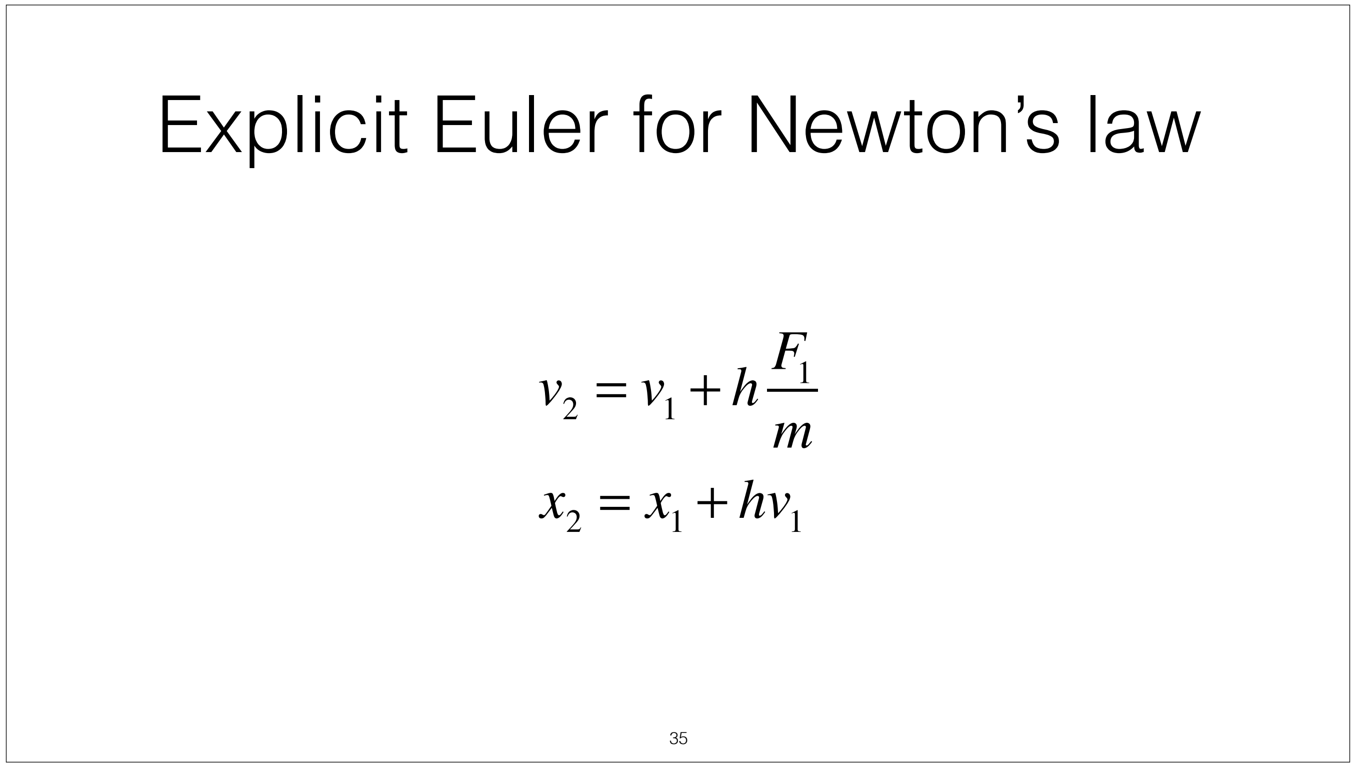
We can also flip things around and approximate the derivative based on the previous value. This is called the backwards difference.

Numerical solution via explicit Euler

$$\dot{x} = f(t,x) \xrightarrow{\text{forward difference}} \frac{x_2 - x_1}{h} = f(t_1, x_1)$$

$$\boxed{x_2 = x_1 + hf(t_1, x_1)}$$ explicit Euler

Lets take our standard first order differential equation and apply the forward difference formula to remove the derivative. Notice that this converts the differential equation into an algebraic equation. This leads to the explicit Euler method: we can simply plug in the current value for x1 and t1 to compute x2.



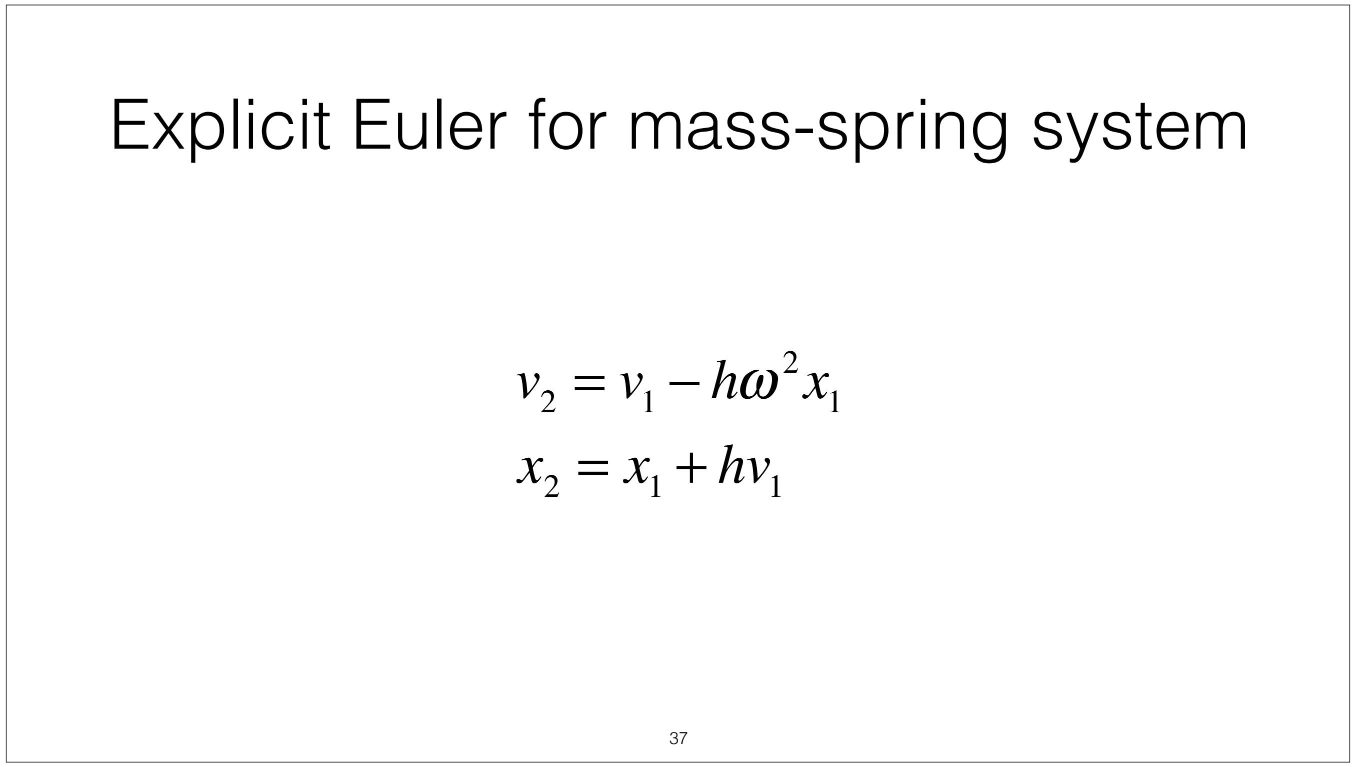
Explicit Euler for Newton's law

$$v_2 = v_1 + h\frac{F_1}{m}$$
$$x_2 = x_1 + hv_1$$

I can apply explicit Euler to Newton's law. Velocity is extrapolated based on the current force and position is extrapolated based on the current velocity.

Pseudocode

```
struct State
{
  float t, x, v;
}

State ExplicitEuler(float h, State state1)
{
  float force = ComputeForce(state1);

  State state2;
  state2.t = state1.t + h;
  state2.v = state1.v + h * force / mass;
  state2.x = state1.x + h * state1.v;

  return state2;
}
```



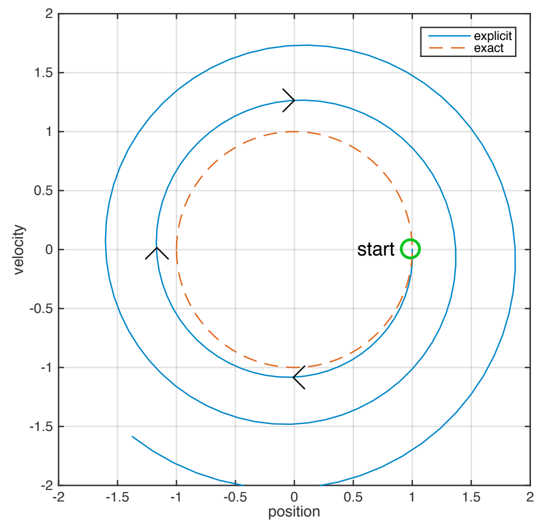
Explicit Euler for mass-spring system

$$v_2 = v_1 - h\omega^2 x_1$$
$$x_2 = x_1 + hv_1$$

37

Here is explicit Euler for the mass-spring system. So here omega is the angular frequency in radians per second. Notice how explicit Euler gives us a way to simulate an oscillation without using trig functions.

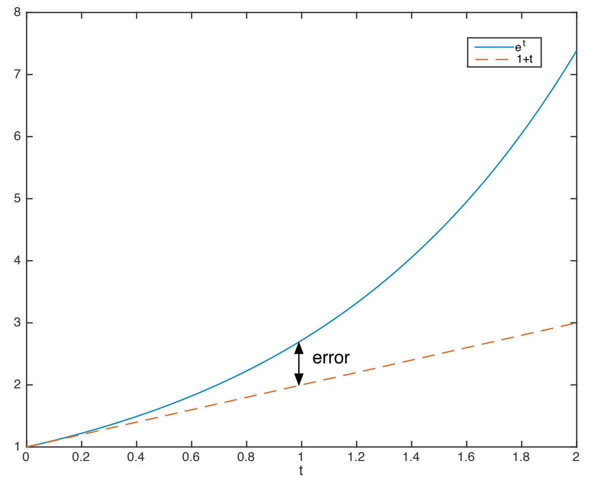
Explicit Euler phase portrait



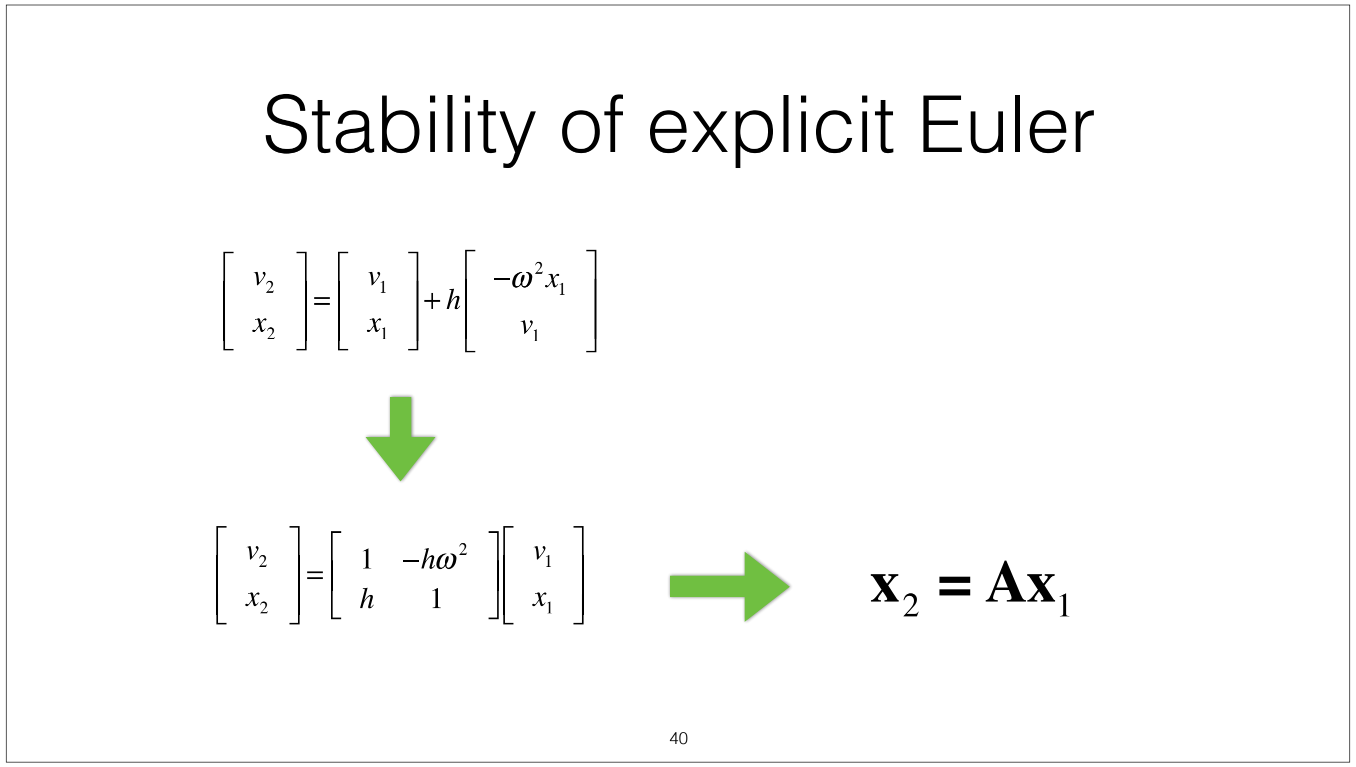
$x_0 = 1$
$v_0 = 0$
$\omega = 1$
$h = 0.1$

Here is a Matlab simulation of the mass-spring system using explicit Euler. As before it is a plot of velocity versus position. The exact solution is a circle. But explicit Euler is diverging and the amplitude is growing unbounded as time progresses.

Explicit Euler is extrapolation



Explicit Euler is essentially extrapolation. Intuitively extrapolation is bad. If the function is rapidly changing, the extrapolation has poor accuracy and can lead to a numerical instability.



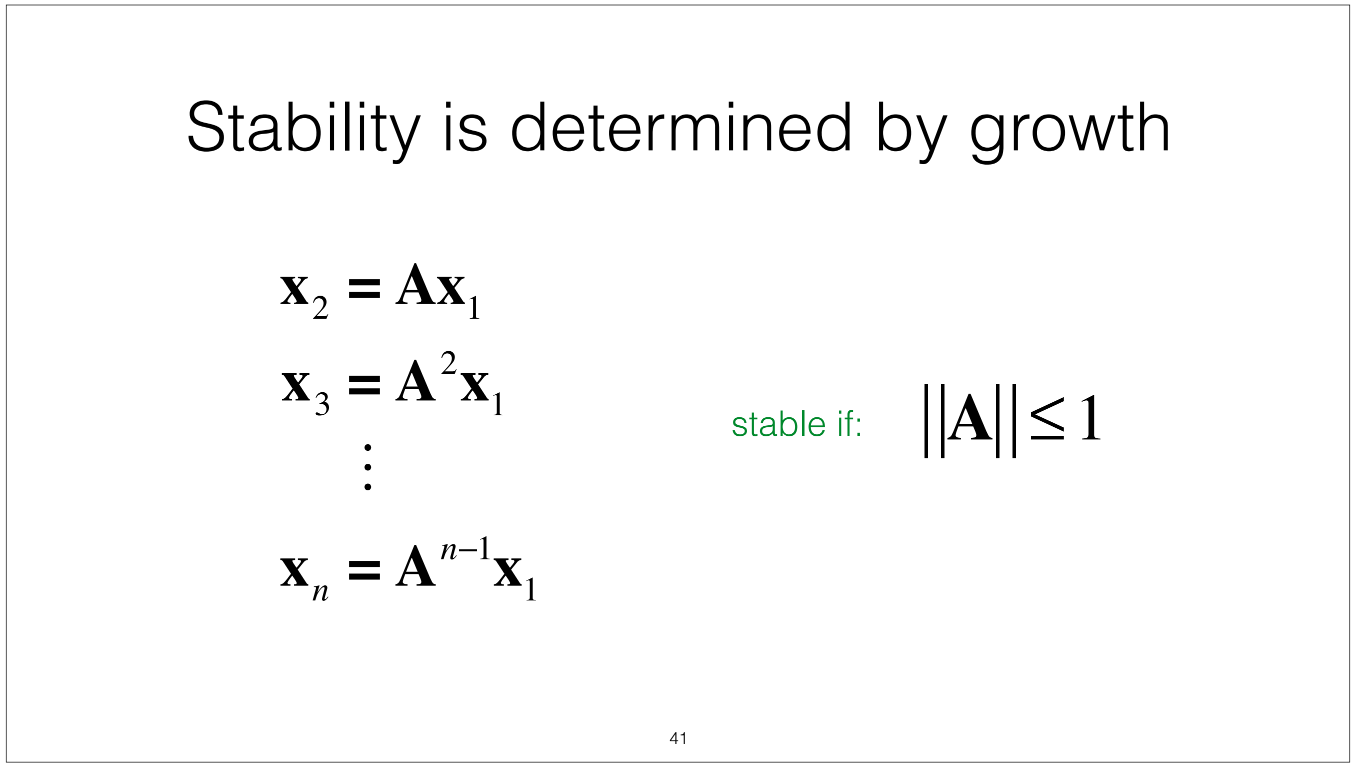
Stability of explicit Euler

$$\begin{bmatrix} v_2 \\ x_2 \end{bmatrix} = \begin{bmatrix} v_1 \\ x_1 \end{bmatrix} + h \begin{bmatrix} -\omega^2 x_1 \\ v_1 \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} v_2 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 & -h\omega^2 \\ h & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ x_1 \end{bmatrix} \Rightarrow \mathbf{x}_2 = \mathbf{A}\mathbf{x}_1$$

The Matlab simulation showed that explicit Euler is unstable. We can also understand the instability by looking at the explicit Euler equations in detail. For the mass-spring system, the time stepping process is equivalent to multiplying the previous state vector by a matrix A.



Stability is determined by growth

$$\mathbf{x}_2 = \mathbf{A}\mathbf{x}_1$$

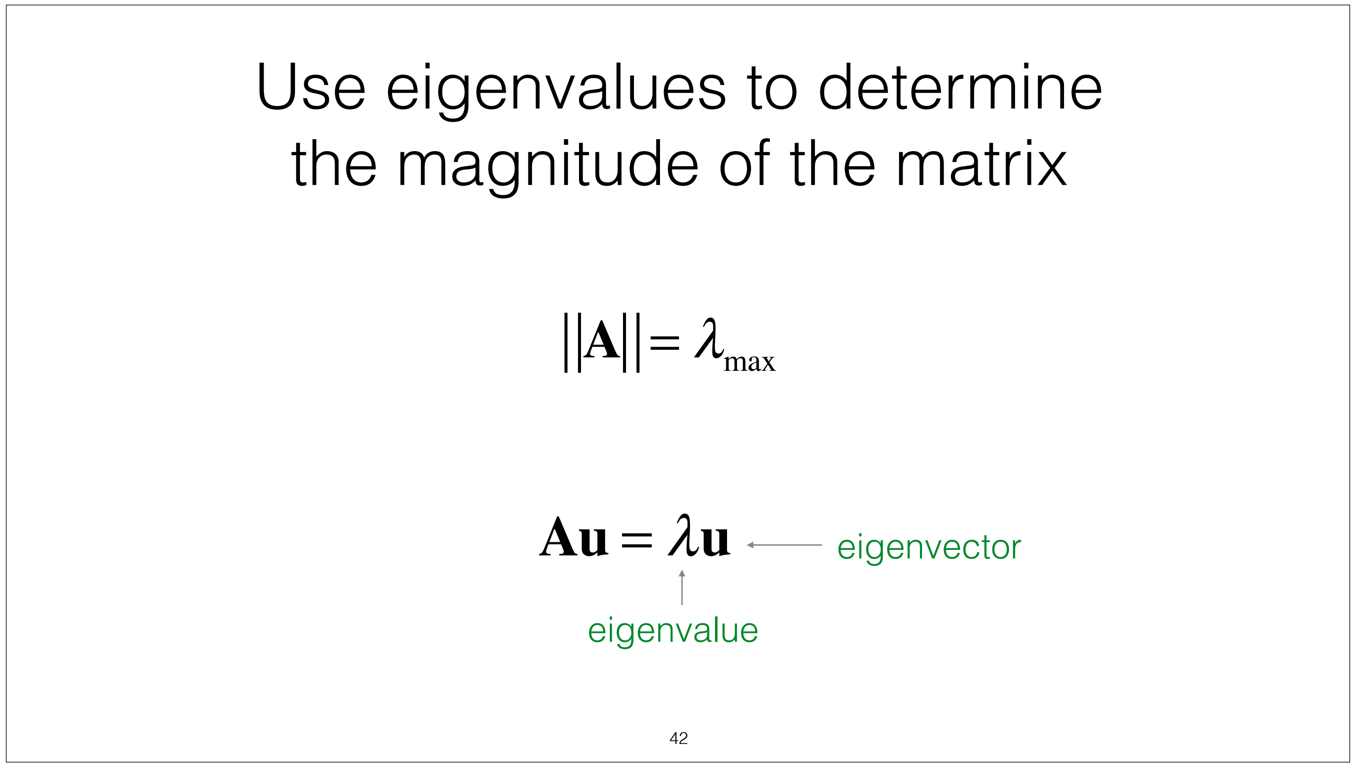
$$\mathbf{x}_3 = \mathbf{A}^2\mathbf{x}_1$$

$$\vdots$$

$$\mathbf{x}_n = \mathbf{A}^{n-1}\mathbf{x}_1$$

stable if: $||\mathbf{A}|| \leq 1$

As time marches forward, I am just multiply the previous state by the same matrix over and over. The magnitude of this matrix must be less than one, otherwise the solution will blow up.



Use eigenvalues to determine the magnitude of the matrix

$$\|\mathbf{A}\| = \lambda_{\max}$$

$$\mathbf{Au} = \underset{\substack{\uparrow \\ \text{eigenvalue}}}{\lambda}\mathbf{u} \longleftarrow \text{eigenvector}$$

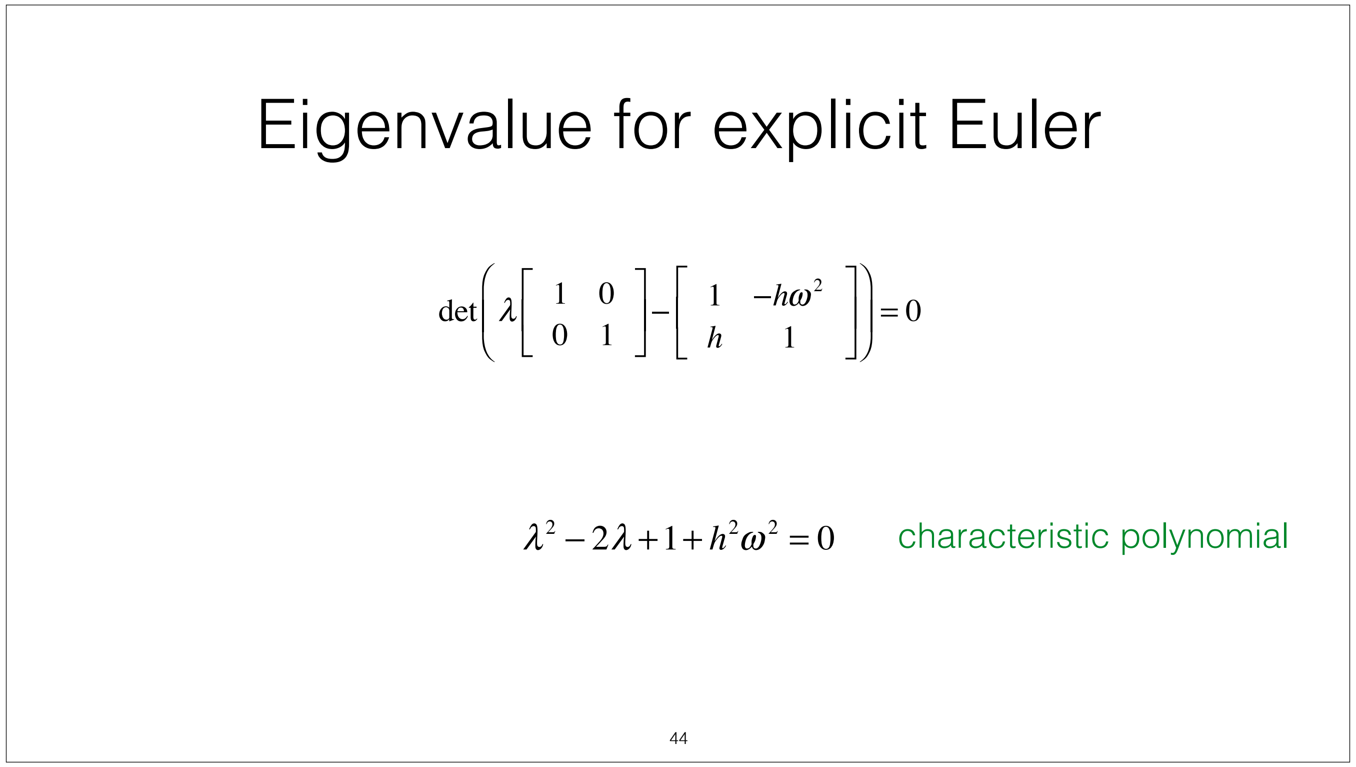
The magnitude of a matrix can be represent by its eigenvalue lambda, which is a scalar. Please see Wikipedia for more information on eigenvalues.

Use the determinant to compute the eigenvalue

$$(\lambda \mathbf{I} - \mathbf{A})\mathbf{u} = 0$$

$$\det(\lambda \mathbf{I} - \mathbf{A}) = 0$$

I can compute the eigenvalue of a matrix using the determinant.

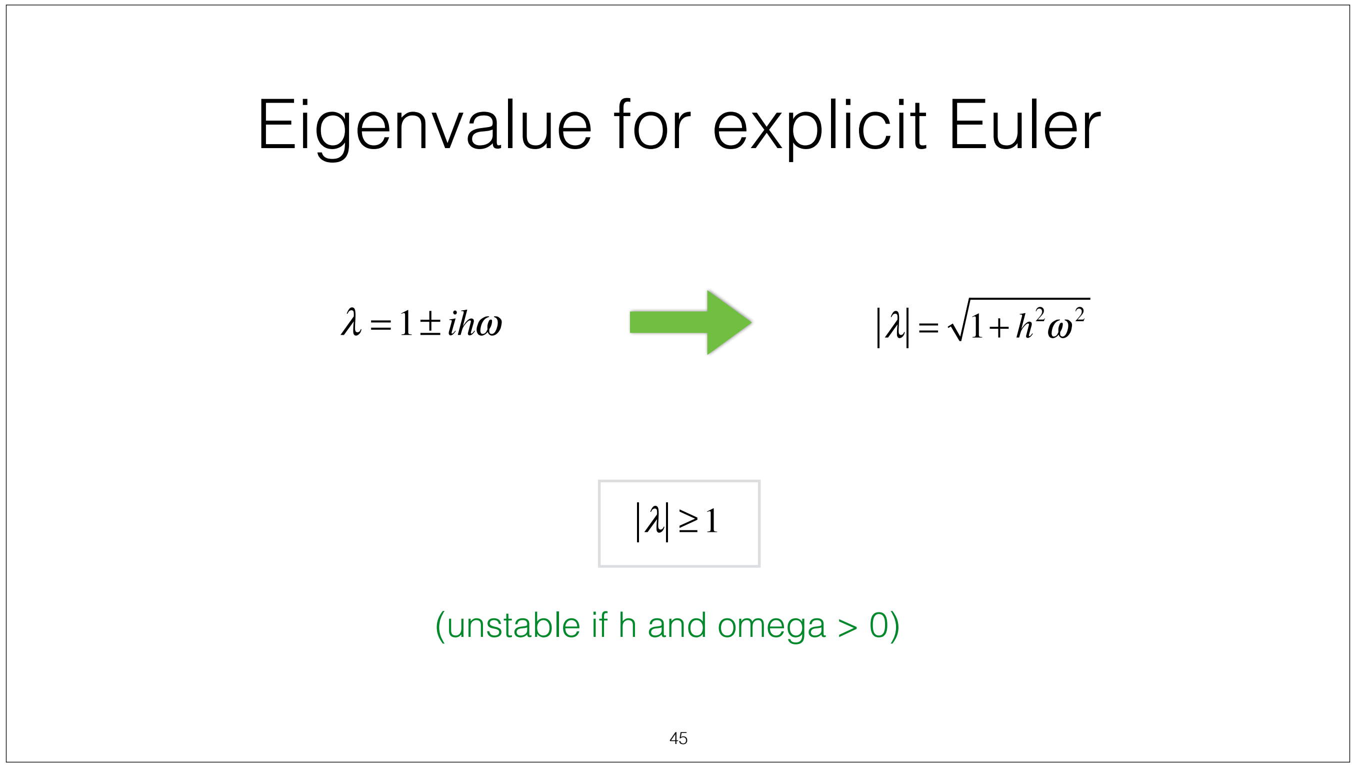


Eigenvalue for explicit Euler

$$\det\left(\lambda\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}-\begin{bmatrix}1 & -h\omega^2\\ h & 1\end{bmatrix}\right)=0$$

$$\lambda^2-2\lambda+1+h^2\omega^2=0$$ characteristic polynomial

The determinant produces a polynomial in lambda. This is called the characteristic polynomial. You can compute the roots of the polynomial to determine the values of lambda. In this case there are two values of lambda, but I just care about the value with the largest magnitude.



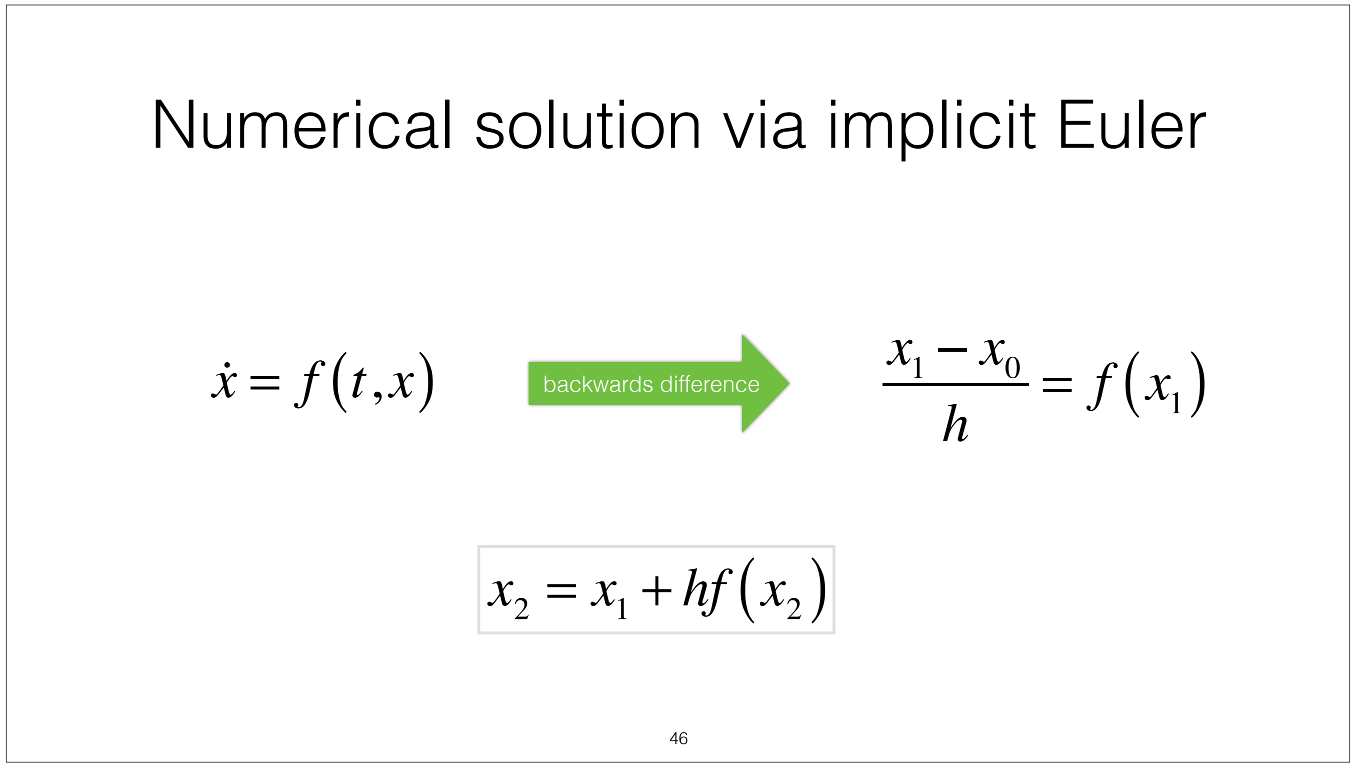
Eigenvalue for explicit Euler

$$\lambda = 1 \pm ih\omega \quad \Rightarrow \quad |\lambda| = \sqrt{1+h^2\omega^2}$$

$$\boxed{|\lambda| \geq 1}$$

(unstable if h and omega > 0)

The eigenvalue is a complex number, but its magnitude is greater than one if the spring stiffness is positive. This shows that explicit Euler is unstable. You can get explicit Euler to be stable by adding some damping, but as you will see, there are better alternatives.

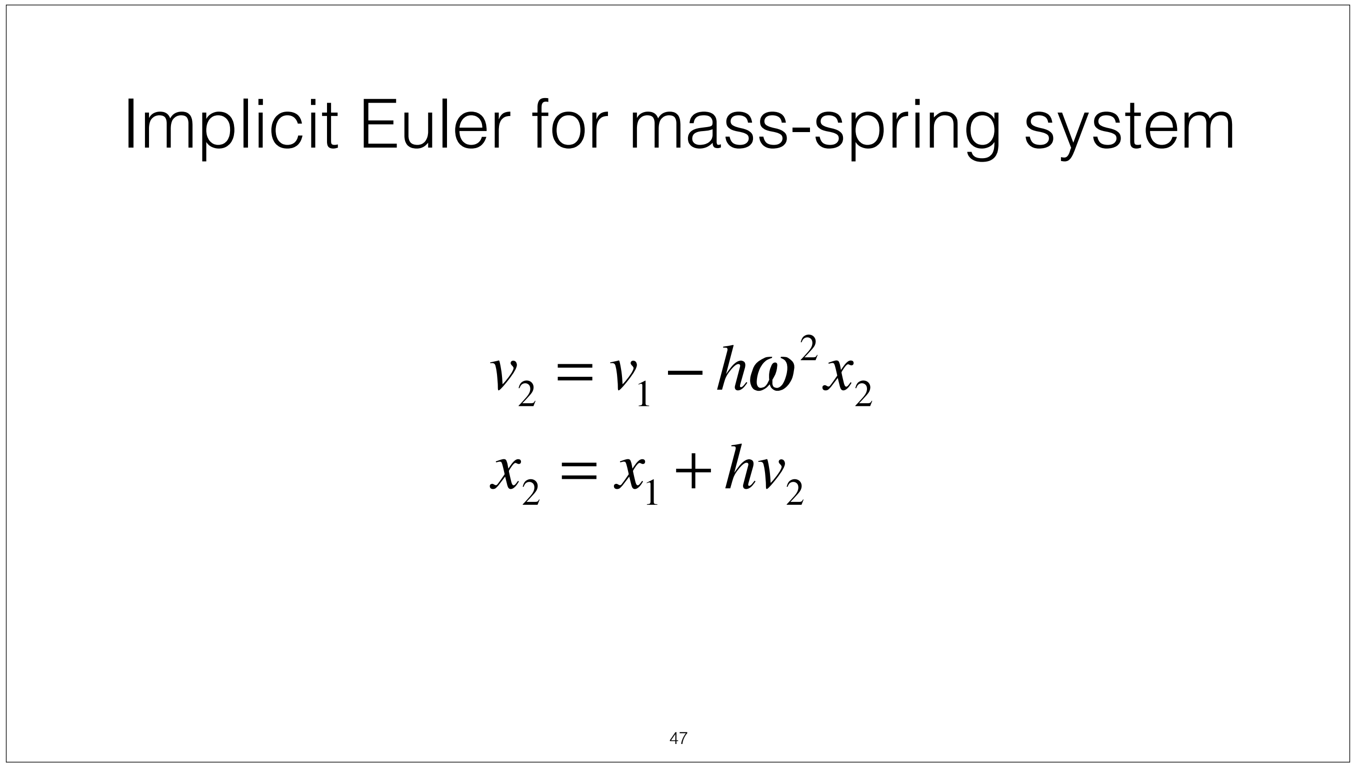


Numerical solution via implicit Euler

$$\dot{x} = f(t,x) \xrightarrow{\text{backwards difference}} \frac{x_1 - x_0}{h} = f(x_1)$$

$$\boxed{x_2 = x_1 + hf(x_2)}$$

Now lets use the backwards difference instead of the forward difference. After shifting the indices, this leads to an implicit function for x2 and the method is called implicit Euler. To actually compute x2 we have to solve an algebraic equation. It may be nonlinear, so we may have to use numerical root finding, such as the Newton-Raphson method.

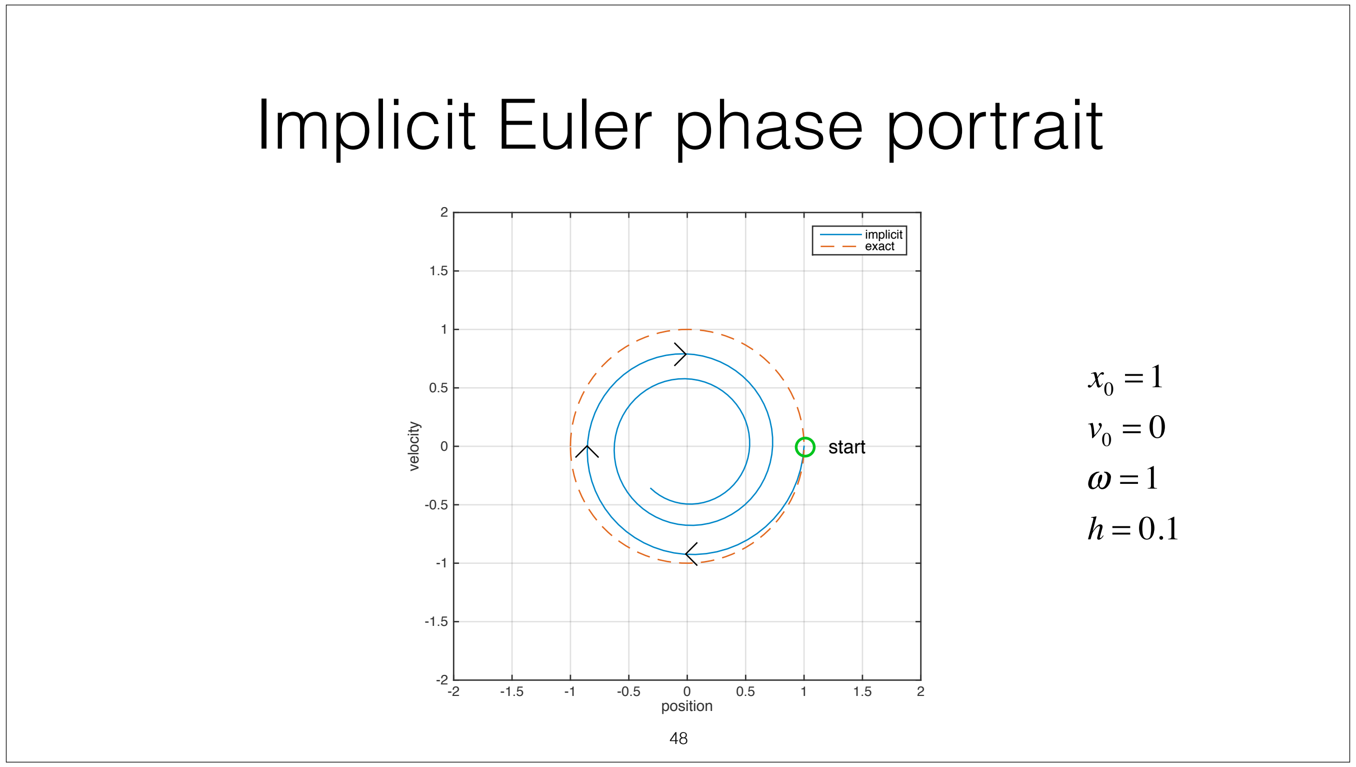


Implicit Euler for mass-spring system

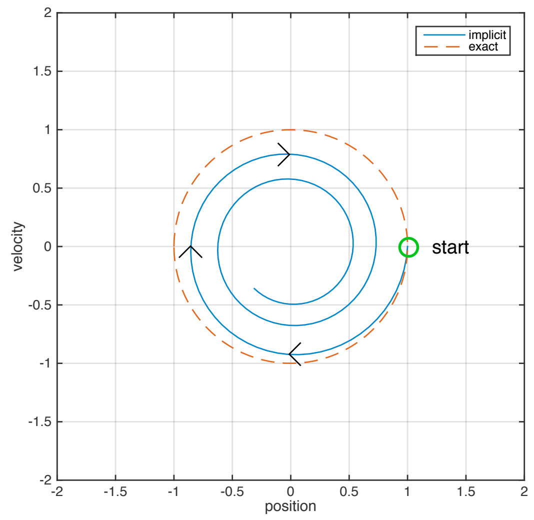
$$v_2 = v_1 - h\omega^2 x_2$$
$$x_2 = x_1 + hv_2$$

Here is implicit Euler for the mass-spring system. Notice that x2 and v2 appear on both sides of the formulas. This means I have to solve for x2 and v2 to advance time.

I can solve these equations easily since they are linear and only have two variables. Larger simulations and non-linearities can make implicit Euler much more challenging to implement.

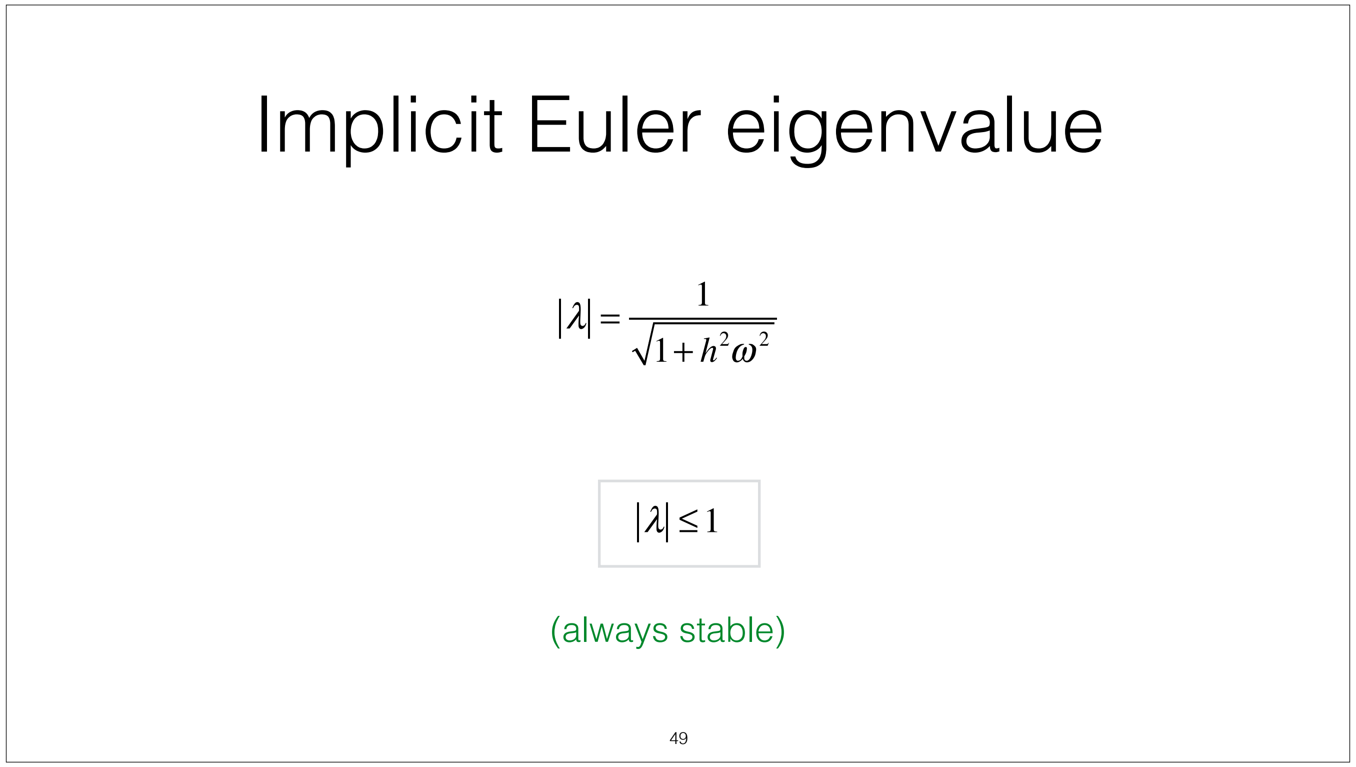


Implicit Euler phase portrait



$x_0 = 1$
$v_0 = 0$
$\omega = 1$
$h = 0.1$

The phase portrait shows that implicit Euler is stable. Also, implicit Euler is damped. So even though the mass should oscillate forever, implicit Euler damps the motion and brings the mass to rest. I call this numerical damping.
If you compare this figure to explicit Euler, you should see that they look they look similar, except that explicit Euler is growing. This is because implicit Euler gives the same result as running explicit Euler in reverse.



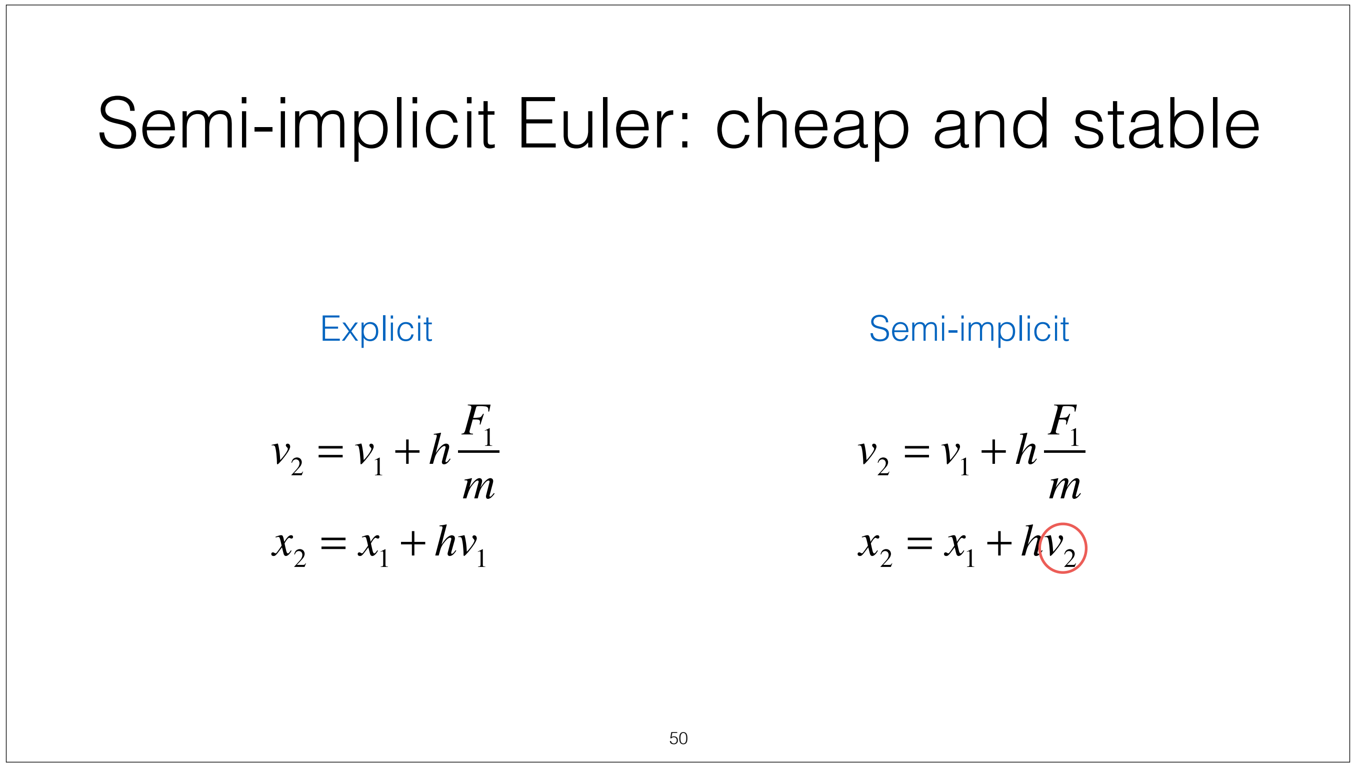
Implicit Euler eigenvalue

$$|\lambda| = \frac{1}{\sqrt{1+h^2\omega^2}}$$

$$\boxed{|\lambda| \le 1}$$

(always stable)

I performed a similar computation for implicit Euler and the result shows that implicit Euler is stable for any positive value of omega.



Semi-implicit Euler: cheap and stable

Explicit

$$v_2 = v_1 + h\frac{F_1}{m}$$
$$x_2 = x_1 + hv_1$$

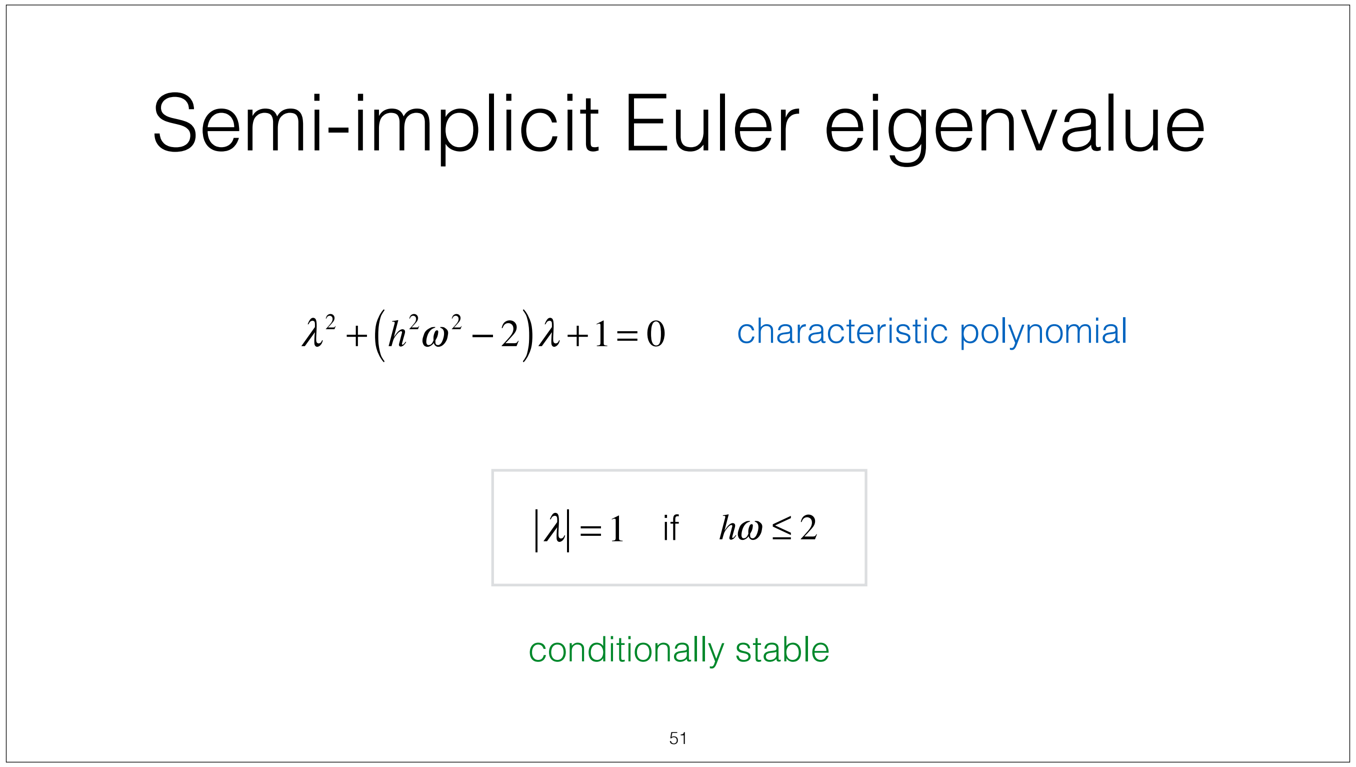
Semi-implicit

$$v_2 = v_1 + h\frac{F_1}{m}$$
$$x_2 = x_1 + h\boxed{v_2}$$

Explicit Euler is efficient but has poor stability. Implicit Euler has great stability but can be expensive. There is a 3rd method that strikes a balance between these two methods, giving good stability and performance.

If I use the current velocity in the position update, then explicit Euler becomes semi-implicit Euler. This small difference makes a huge difference in stability. Yet the computational cost is just as good as explicit Euler.

Semi-implicit Euler is often called symplectic Euler because it has some good energy conservation properties. You'll see that shortly.



Semi-implicit Euler eigenvalue

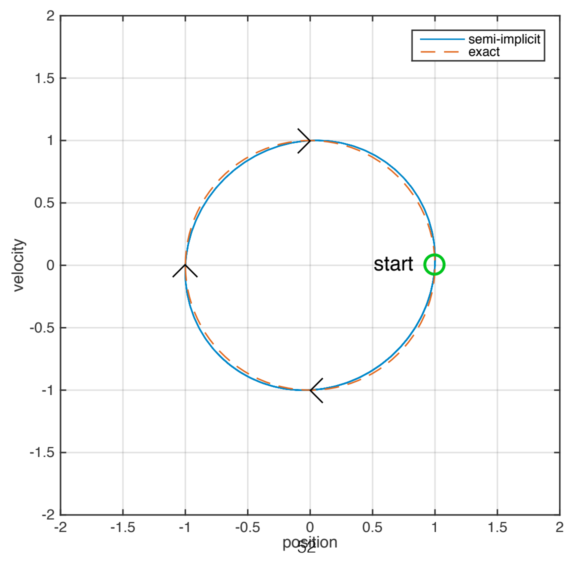
$$\lambda^2 + \left(h^2\omega^2 - 2\right)\lambda + 1 = 0$$ characteristic polynomial

$$\boxed{|\lambda| = 1 \quad \text{if} \quad h\omega \le 2}$$

conditionally stable

An eigenvalue analysis shows that semi-implicit Euler is conditionally stable. This is much better than explicit Euler and in practice we can push semi-implicit Euler quite hard before it goes unstable.

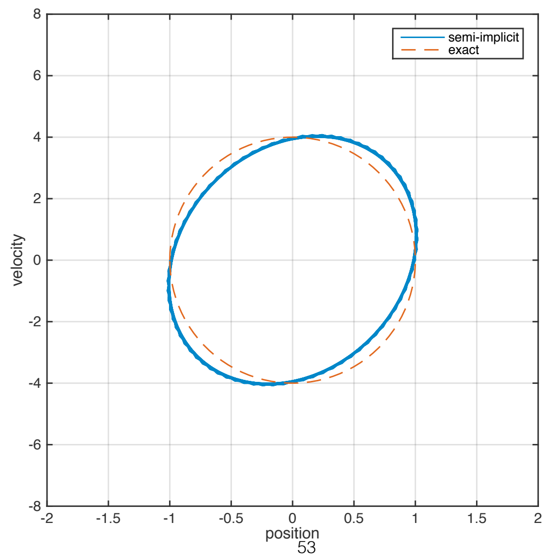
Semi-implicit Euler is conditionally stable



$x_0 = 1$
$v_0 = 0$
$\omega = 1$
$h = 0.1$

At omega = 1, semi-implicit Euler is stable and accurate. Additionally it conserves energy on average.

Increasing the frequency

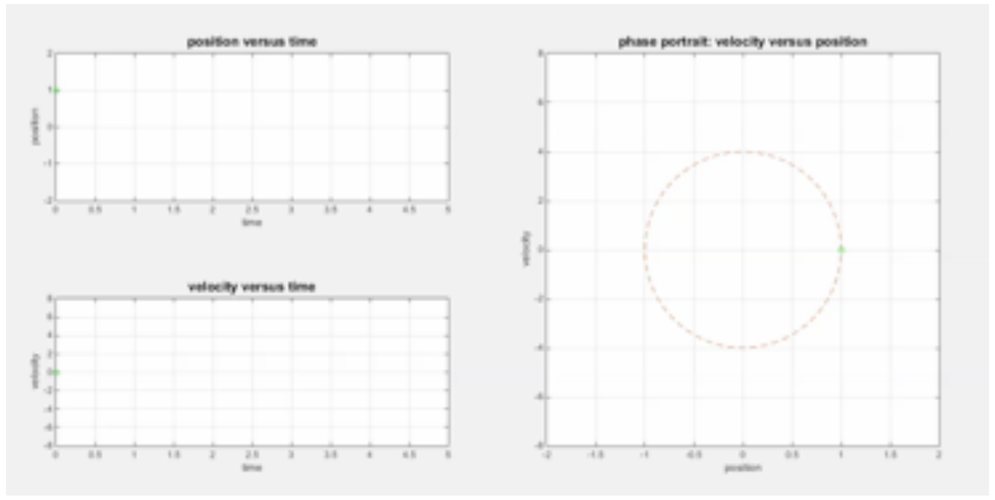


$\omega = 4$

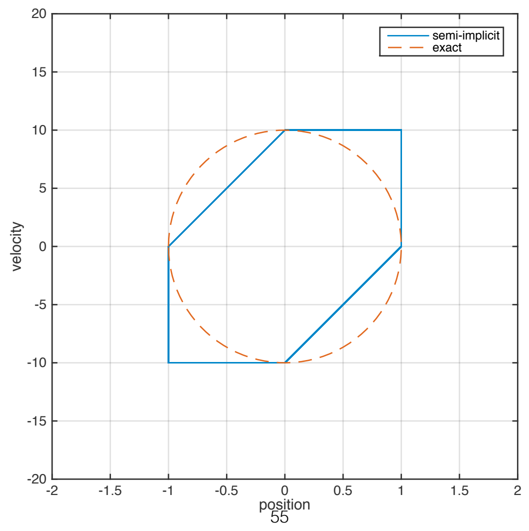
$h = 0.1$

Increasing the angular frequency reveals some deviation from the exact solution. But it is still stable and still conserves energy on average.

$$\omega = 4$$


position versus time
position
time
velocity versus time
velocity
time
phase portrait: velocity versus position
velocity
position

Increasing the frequency



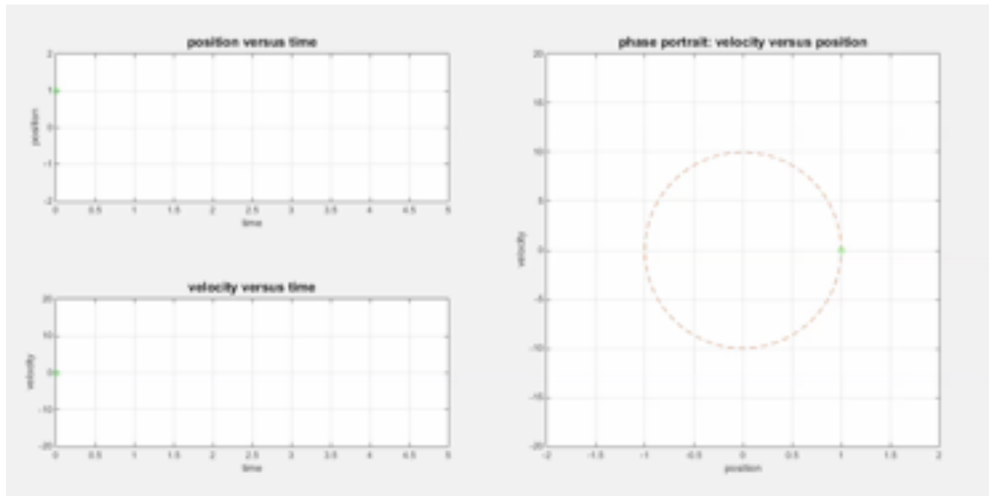
$\omega = 10$

$h = 0.1$

$\boxed{h\omega = 1}$

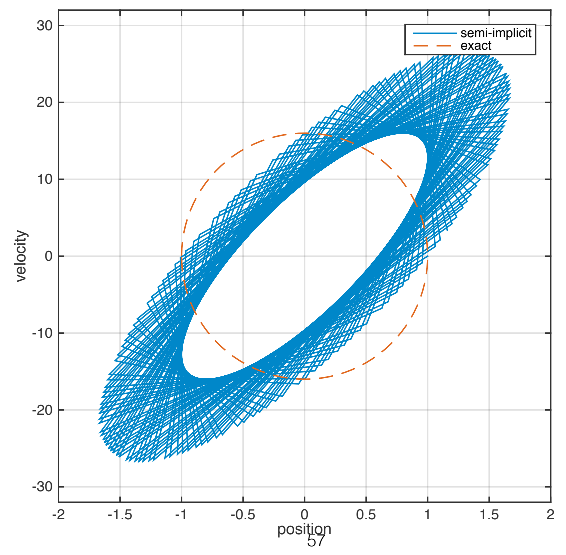
Half the stability limit, things get strange, but it is still stable.

$$\omega = 10$$



position versus time
position
time
velocity versus time
velocity
time
phase portrait: velocity versus position
velocity
position

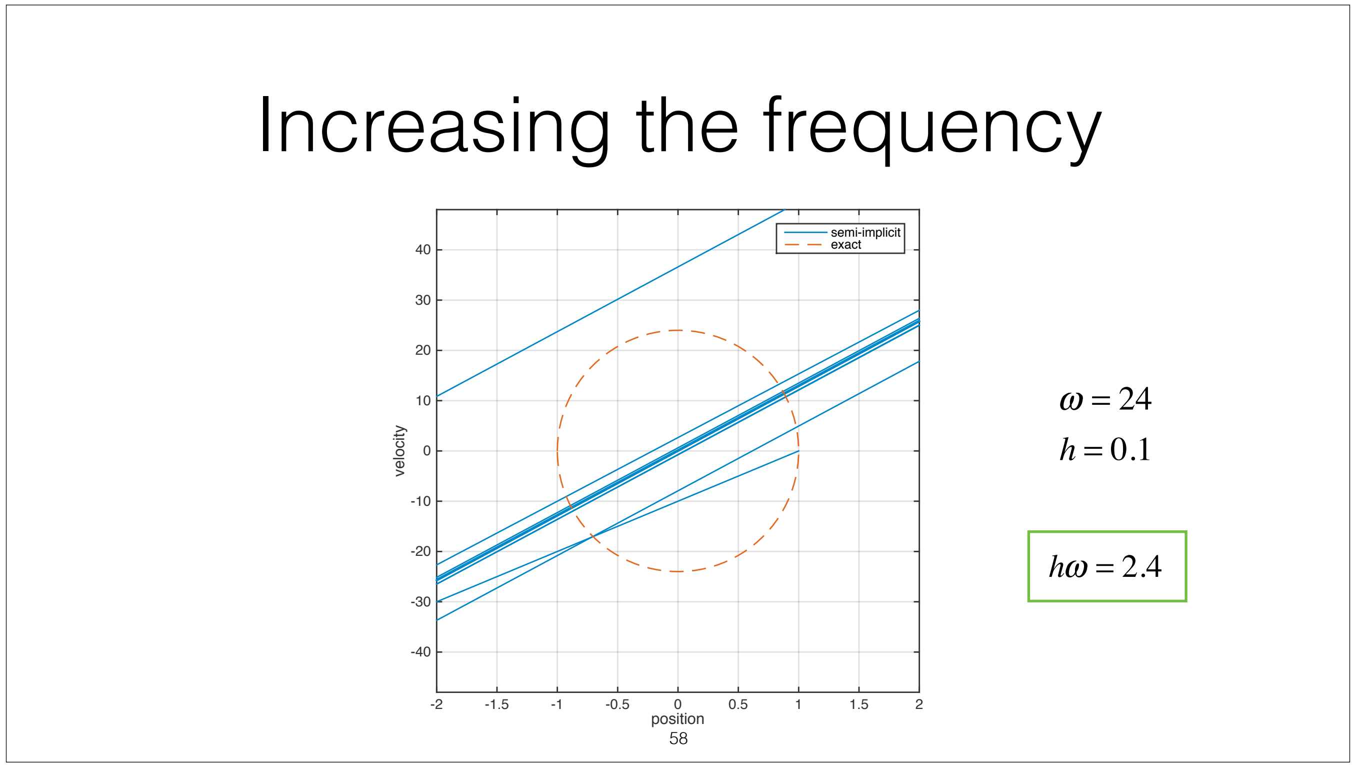
Increasing the frequency



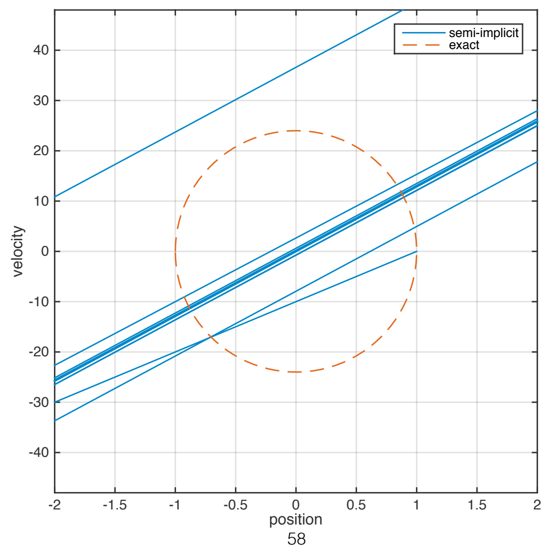
$\omega = 16$

$h = 0.1$

This get really wacky at larger frequencies. Yet it didn’t blow up yet.



Increasing the frequency



$\omega = 24$

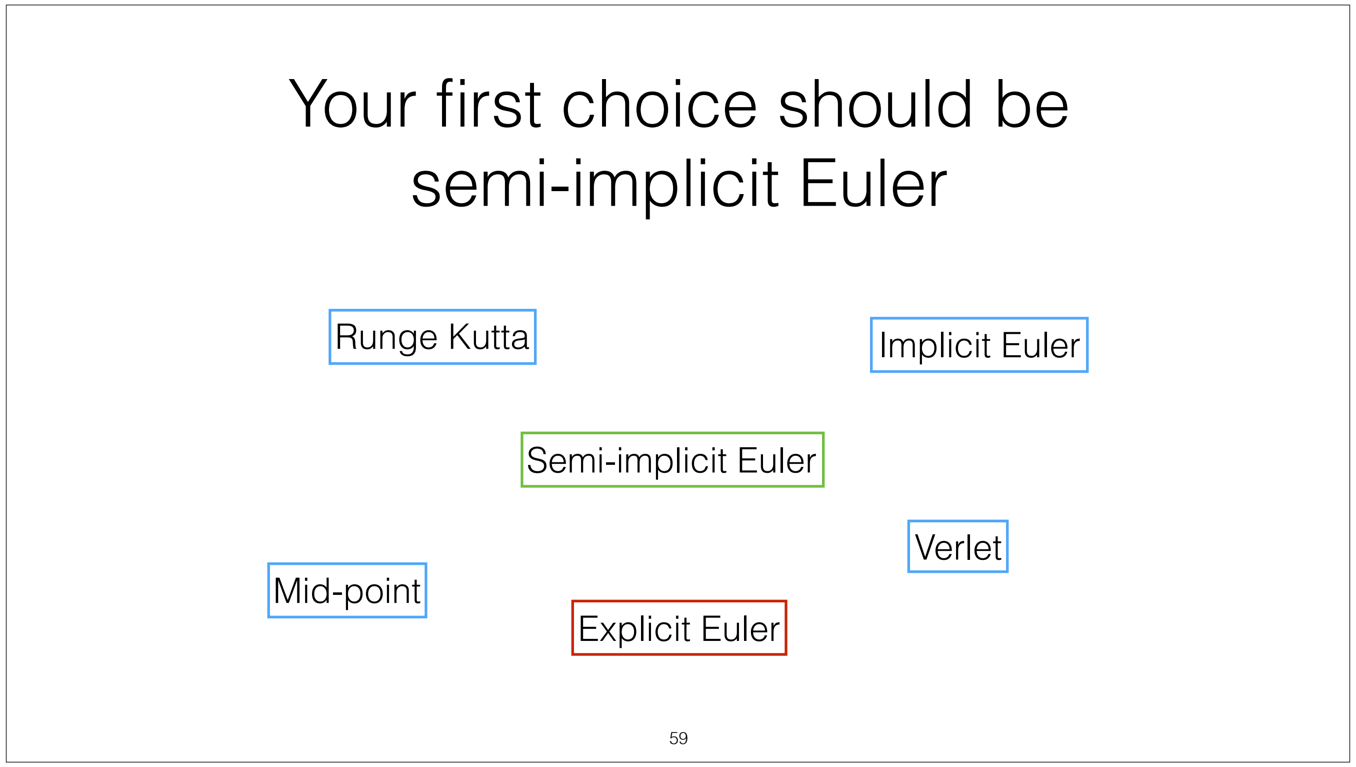
$h = 0.1$

$\boxed{h\omega = 2.4}$

58

This shows that eventually semi-implicit Euler will blow up.

Rule of thumb: aim for at least 4 time steps per spring oscillation.



Your first choice should be semi-implicit Euler

Runge Kutta

Implicit Euler

Semi-implicit Euler

Verlet

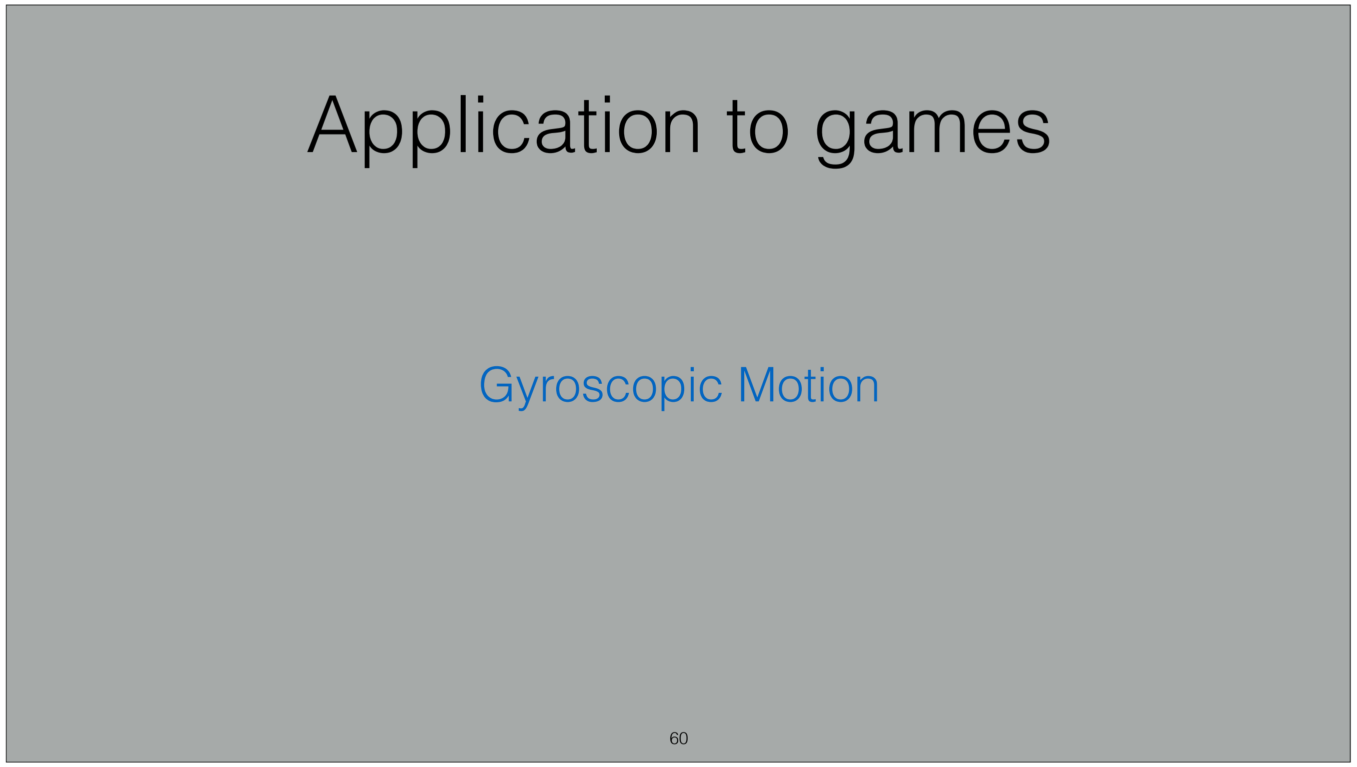
Mid-point

Explicit Euler

59

So far I have shown 3 numerical methods for differential equations: explicit Euler, implicit Euler, and semi-implicit Euler. There are many more algorithms out there, such as Runge-Kutta, mid-point, predictor-corrector, and more. For games, your first choice should be semi-implicit Euler. It is fast, reasonably stable, and easy to implement.

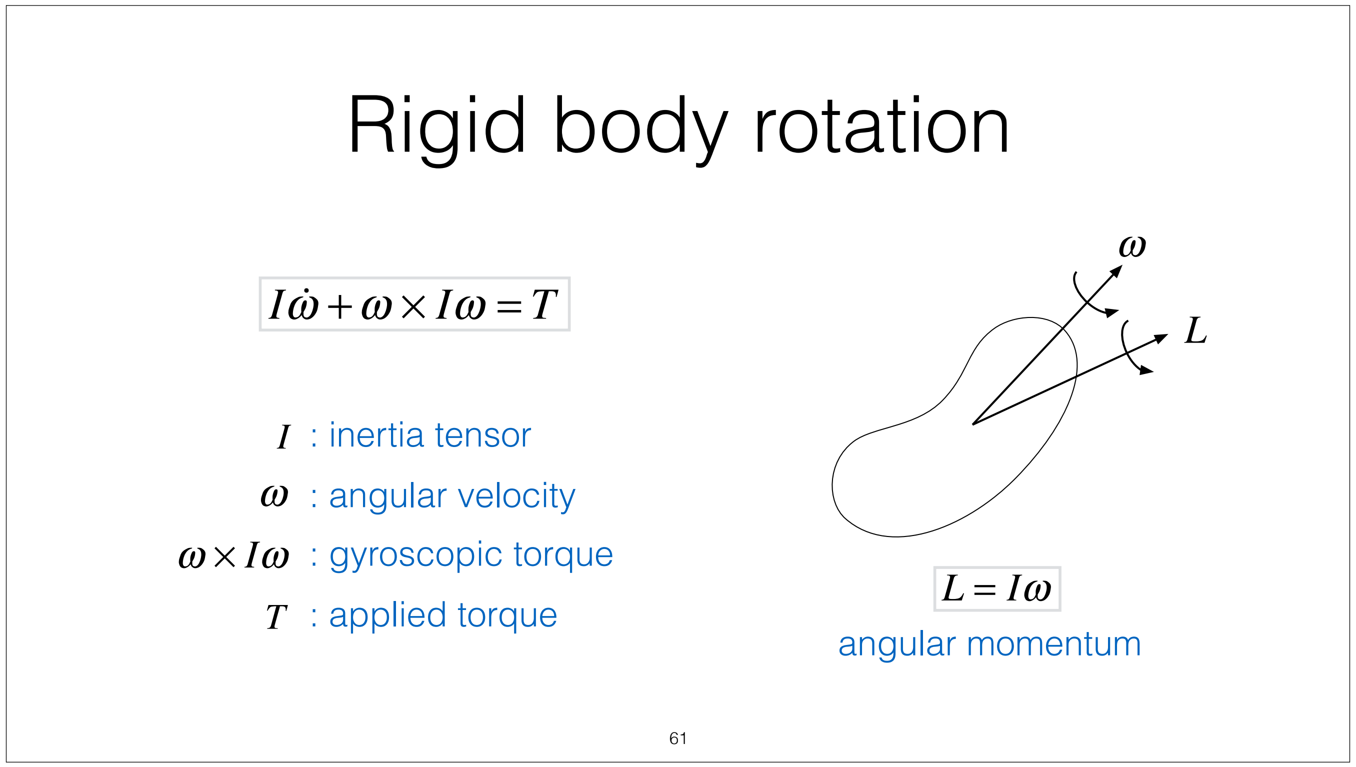
The increased accuracy of Runge-Kutta, such as RK4, may seem appealing. However, any simulation with rigid body collisions is already wildly inaccurate. What matters more is stability and performance. In this department semi-implicit Euler delivers.



Application to games

Gyroscopic Motion

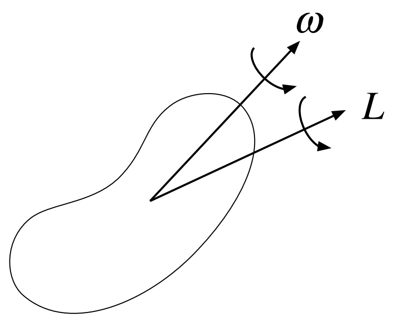
Now I would like to show an example of applying numerical methods to solve a difficult problem in rigid body dynamics. In particular we are going to look at gyroscopic motion of rigid bodies.



Rigid body rotation

$$\boxed{I\dot{\omega} + \omega \times I\omega = T}$$

I	: inertia tensor
ω	: angular velocity
$\omega \times I\omega$	: gyroscopic torque
T	: applied torque



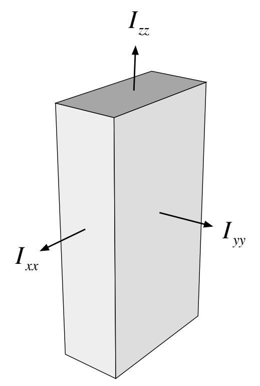
$$\boxed{L = I\omega}$$

angular momentum

This equation is known as Euler’s equations of motion. You can view it as Newton’s law for rotation. The inertia tensor is represented by a 3-by-3 matrix. The angular velocity and the applied torque are both 3D column vectors. Note that now omega represents angular velocity, not angular frequency! Curiously, they both have the same units: radians per second.

This equation looks like ma = F, except there is a middle term that depends on angular velocity. This is the gyroscopic torque. It accounts for non-spherical shapes that may wobble as they spin freely in the air. Specifically, it arises because angular velocity and angular momentum may point in different directions.

The gyroscopic torque is due to a non-uniform inertia tensor



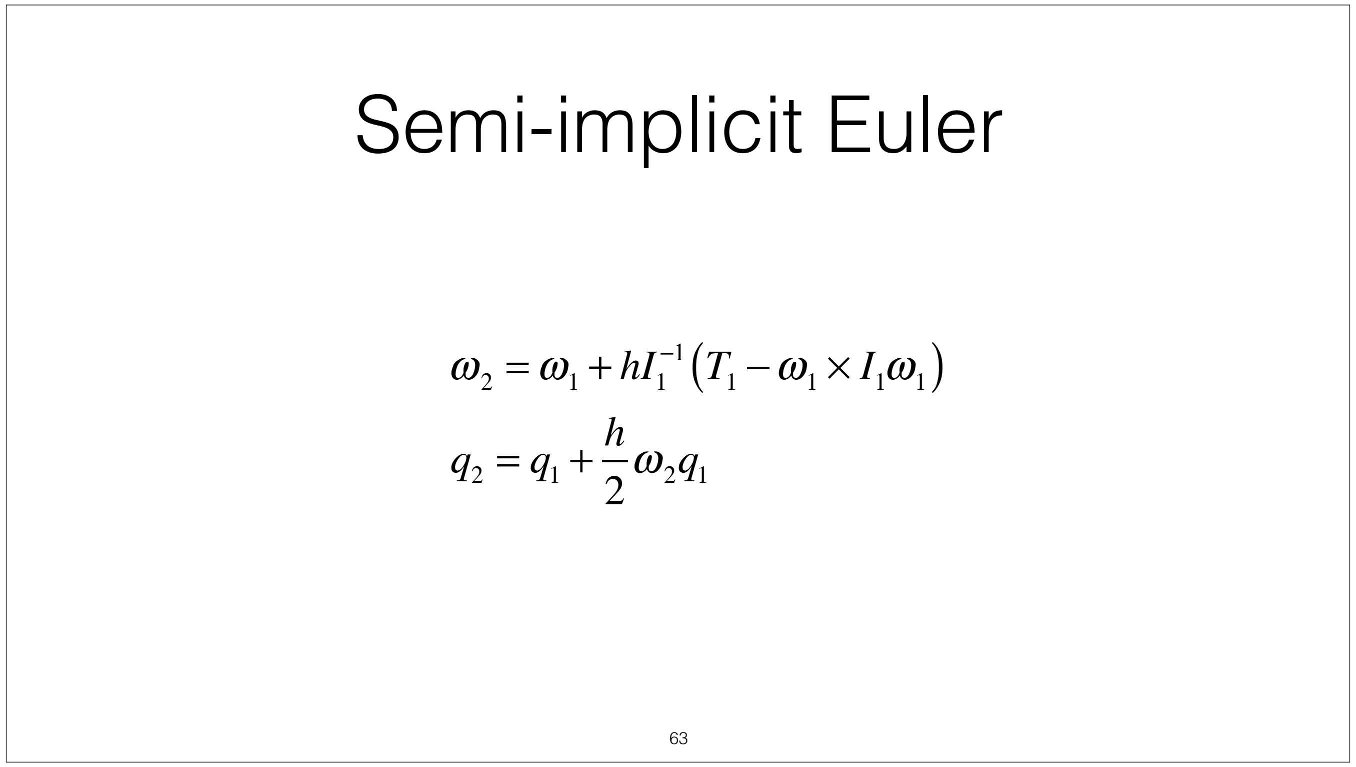
$$I_{xx} \neq I_{yy} \neq I_{zz}$$

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$$

$$L = I\omega = \begin{bmatrix} I_{xx}\omega_x \\ I_{yy}\omega_y \\ I_{zz}\omega_z \end{bmatrix}$$

$$\boxed{\omega \times I\omega = stuff}$$

The inertia tensor for a box with non-equal sides is non-uniform. So apply the inertia tensor to the angular velocity results in an angular momentum that points in a different direction than the angular velocity. So the cross product between the angular velocity and the angular momentum is non-zero.



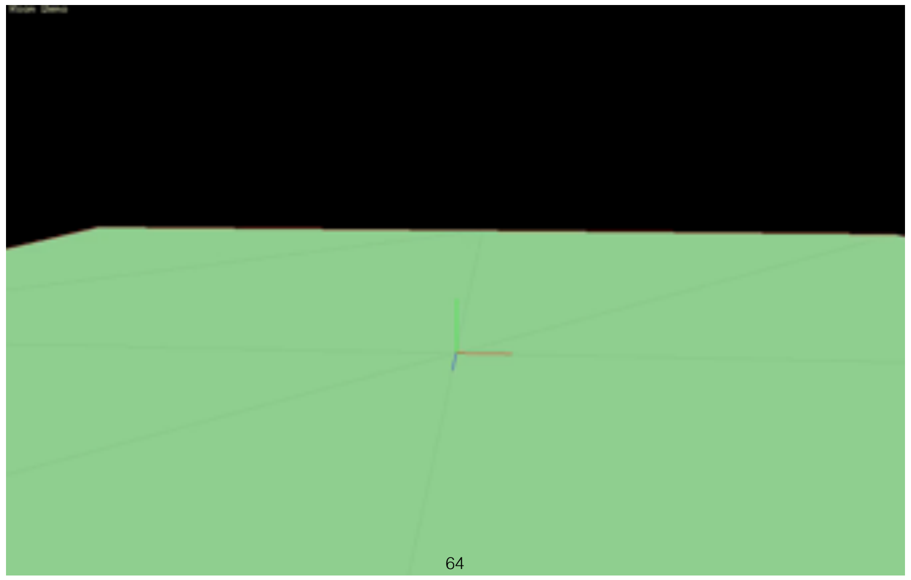
Semi-implicit Euler

$$\omega_2 = \omega_1 + h I_1^{-1}\left(T_1 - \omega_1 \times I_1 \omega_1\right)$$

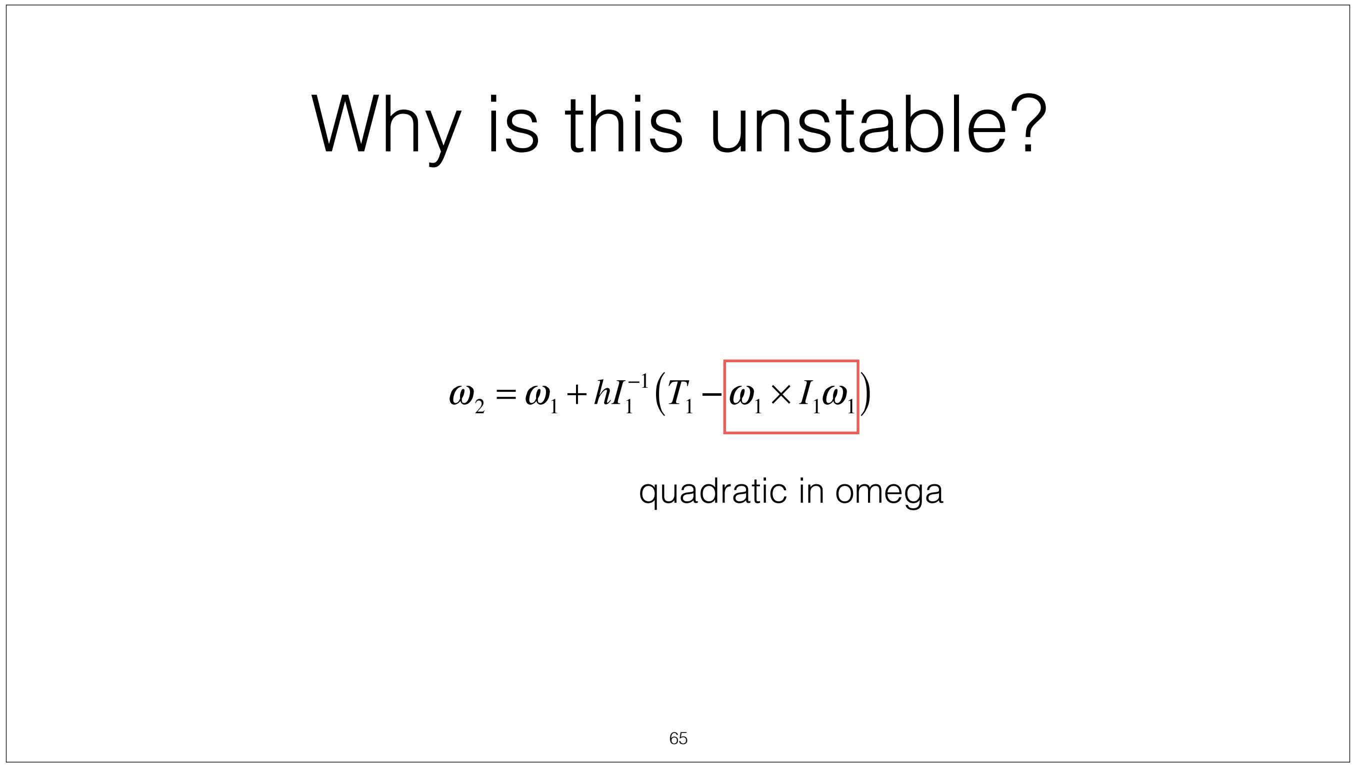
$$q_2 = q_1 + \frac{h}{2}\omega_2 q_1$$

This is semi-implicit Euler applied to the rotational equation. I'm using a special formula to update the quaternion. See the references for details.

Capsule spin test



Here is a simulation of a spinning capsule using semi-implicit Euler. This video was created using the Blizzard physics engine called Domino.

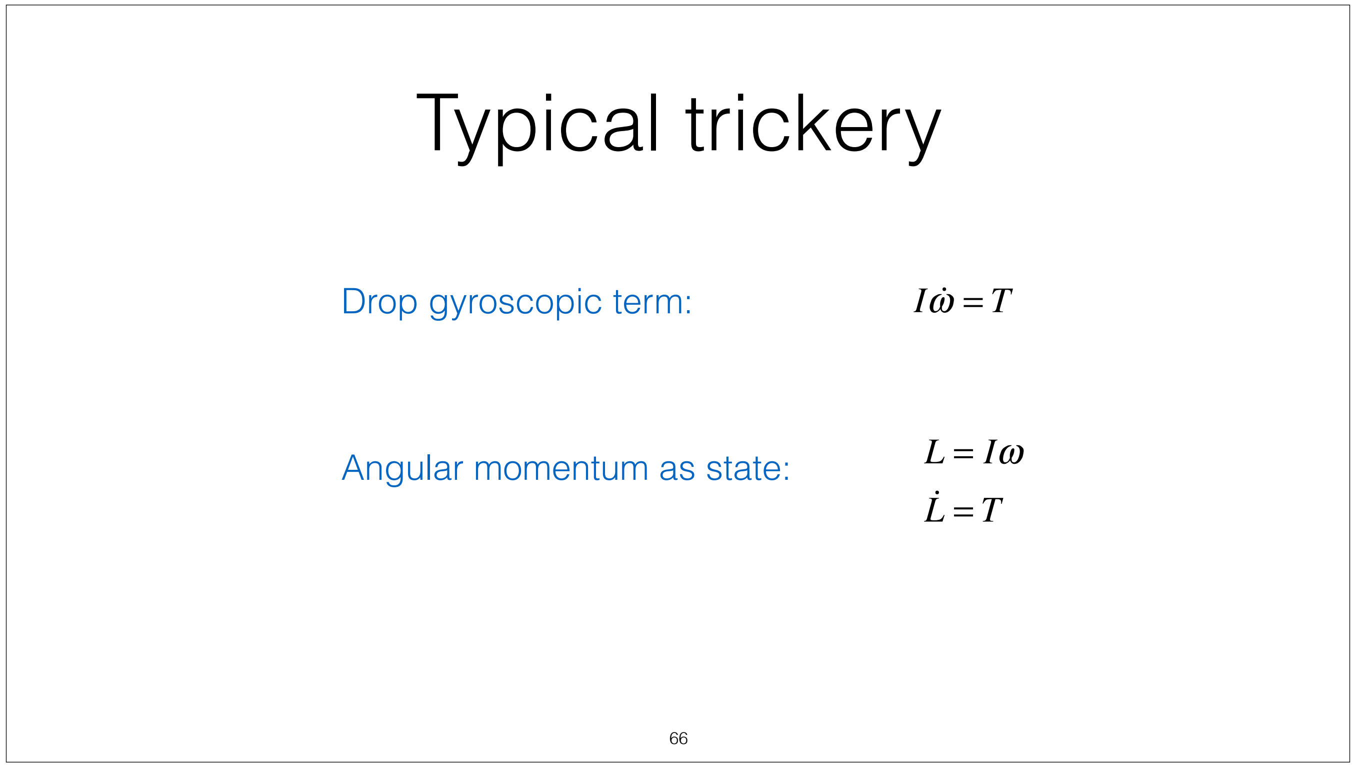


Why is this unstable?

$$\omega_2 = \omega_1 + h I_1^{-1}\left(T_1 - \boxed{\omega_1 \times I_1 \omega_1}\right)$$

quadratic in omega

Here is the velocity update from semi-implicit Euler. This blows up because semi-implicit Euler extrapolates velocity. Since the gyroscopic torque is quadratic in the angular velocity, the extrapolation easily diverges.

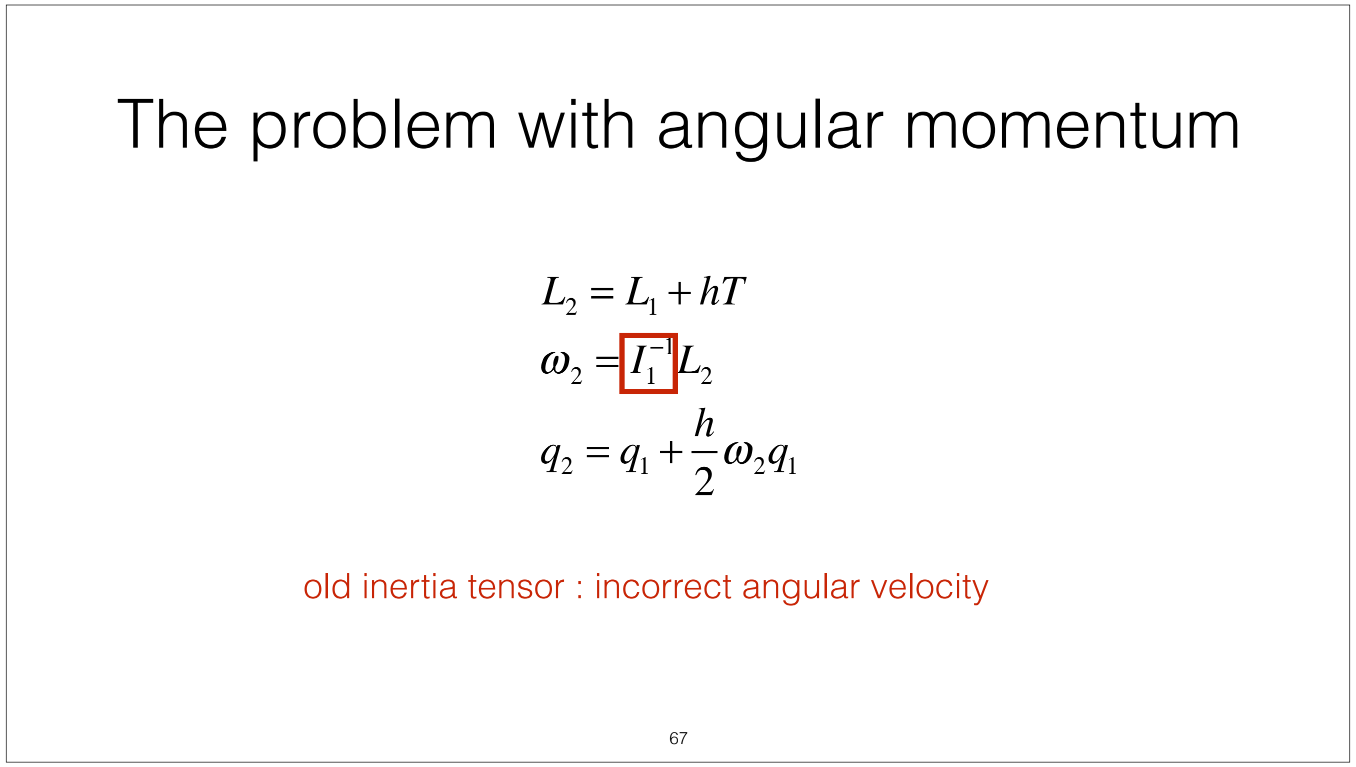


Typical trickery

Drop gyroscopic term: $I\dot{\omega} = T$

Angular momentum as state:
$$L = I\omega$$
$$\dot{L} = T$$

There are a couple tricks used to avoid the gyroscopic term. Many physics engines simply drop the gyroscopic term. This is what I have done for many years.
Some folks like to avoid the gyroscopic term by using the angular momentum L rather than the angular velocity. This makes the gyroscopic term disappear from the rigid body equations of motion.



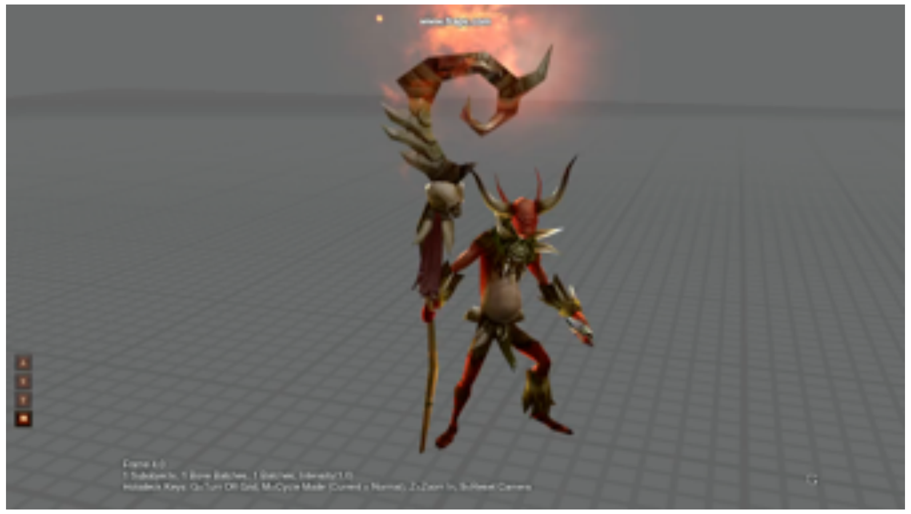
The problem with angular momentum

$$L_2 = L_1 + hT$$
$$\omega_2 = \boxed{I_1^{-1}} L_2$$
$$q_2 = q_1 + \frac{h}{2}\omega_2 q_1$$

old inertia tensor : incorrect angular velocity

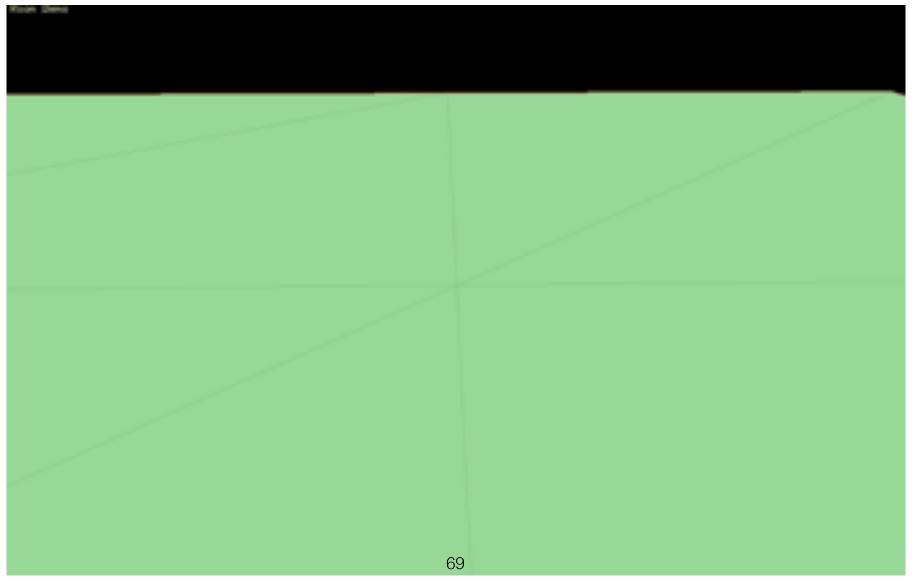
I can propagate the angular momentum with good stability using semi-implicit Euler. I need the updated angular velocity to compute the new orientation (quaternion). To compute the new angular velocity I need to apply the inverse inertia tensor to the angular momentum. In general the inertia tensor depends on orientation, so I am forced to use the old inertia tensor. This loses the correct rotational behavior.

Lack of gyroscopic term in Diablo 3



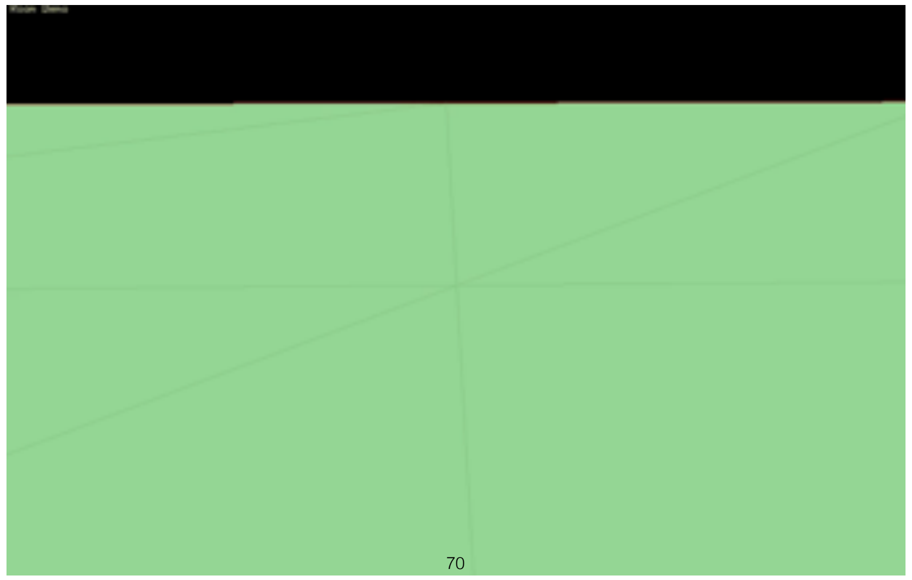
In the past I’ve run into real game scenarios where dropping the gyroscopic term caused some odd behavior. In this case this creature’s staff is a slender box that spins a bit too much.

Dropping the gyroscopic term

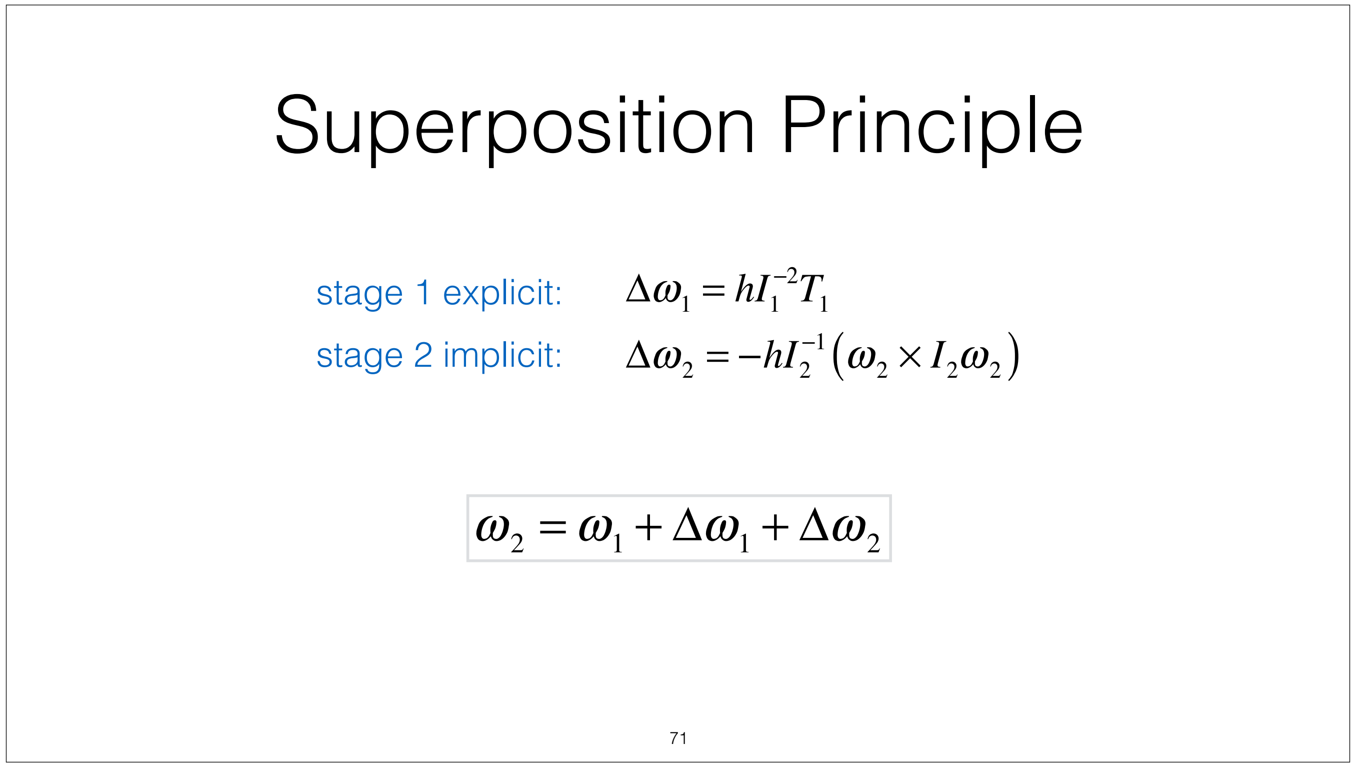


The capsule test is now stable. But it is too stable! The capsule spins forever.

… adding rolling resistance



Adding rolling resistance results in a simulation that is still stable, but not plausible.



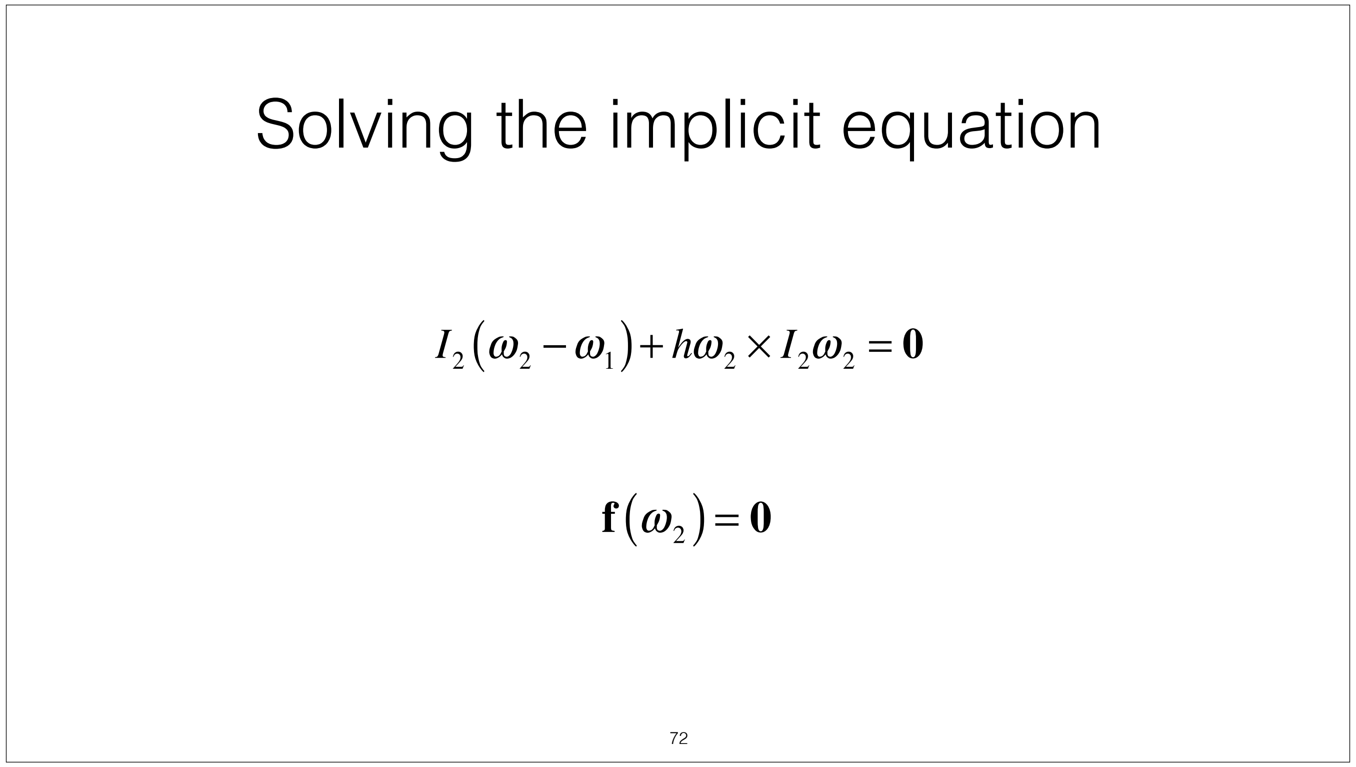
Superposition Principle

stage 1 explicit: $\Delta\omega_1 = hI_1^{-2}T_1$

stage 2 implicit: $\Delta\omega_2 = -hI_2^{-1}\left(\omega_2 \times I_2\omega_2\right)$

$$\boxed{\omega_2 = \omega_1 + \Delta\omega_1 + \Delta\omega_2}$$

I can include the gyroscopic torque by using superposition principle which says that the effect of multiple separate torques is additive. This lets me use implicit Euler on the gyroscopic torque while using explicit Euler on all the torques.



Solving the implicit equation

$$I_2\left(\omega_2-\omega_1\right)+h\omega_2\times I_2\omega_2=\mathbf{0}$$

$$\mathbf{f}\left(\omega_2\right)=\mathbf{0}$$

For implicit solution of the gyroscopic term, I need to solve for omega. However, this is a nonlinear equation in 3 variables (xyz rotation). So I solve this using Newton-Raphson iteration. Therefore I put the problem into a standard form of some function of omega equals zero.

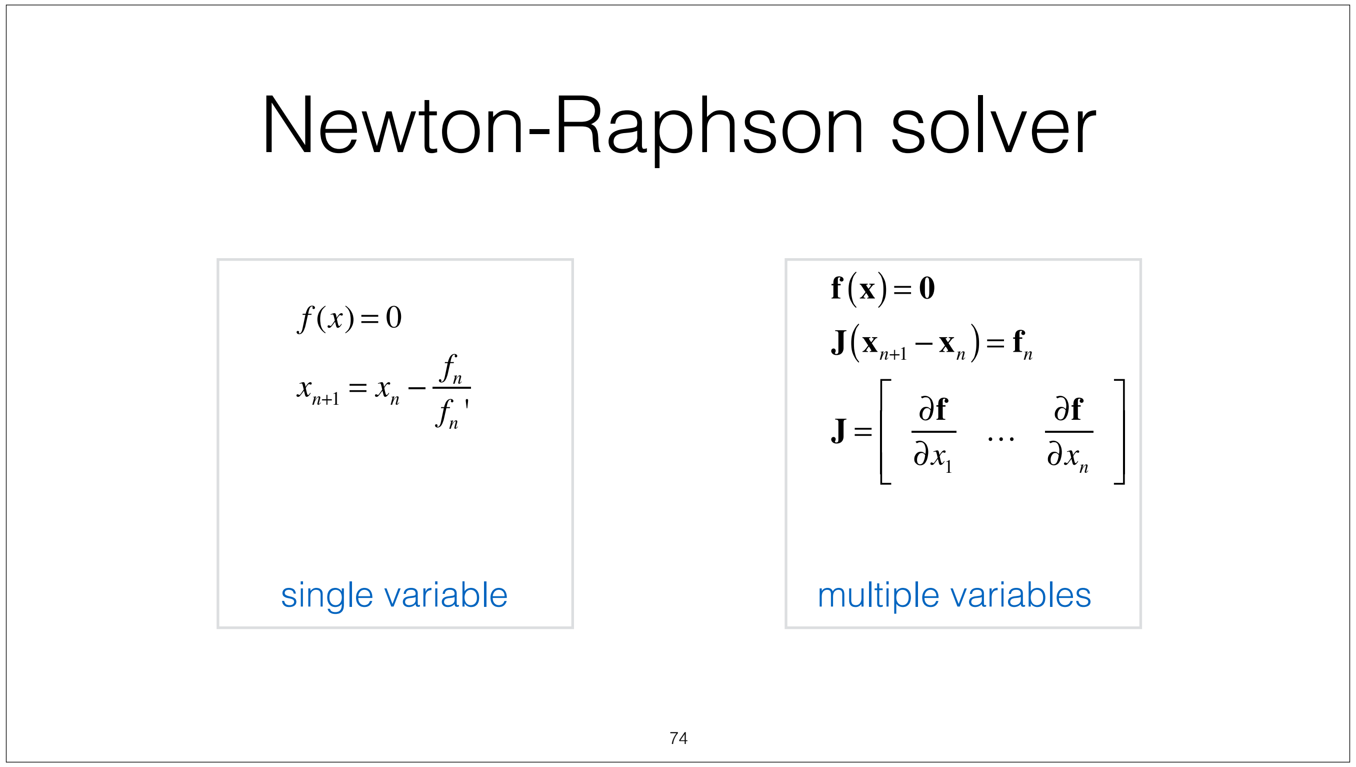
Problem: inertia tensor

I don't have I_2

Solution: work in local coordinates

The inertia tensor is **constant** in local coordinates

The inertia tensor in world coordinates depends on the orientation of the rigid body. So we cannot compute I2. By working in local coordinates, I can side-step this problem.



Newton-Raphson solver

$$f(x) = 0$$

$$x_{n+1} = x_n - \frac{f_n}{f_n'}$$

single variable

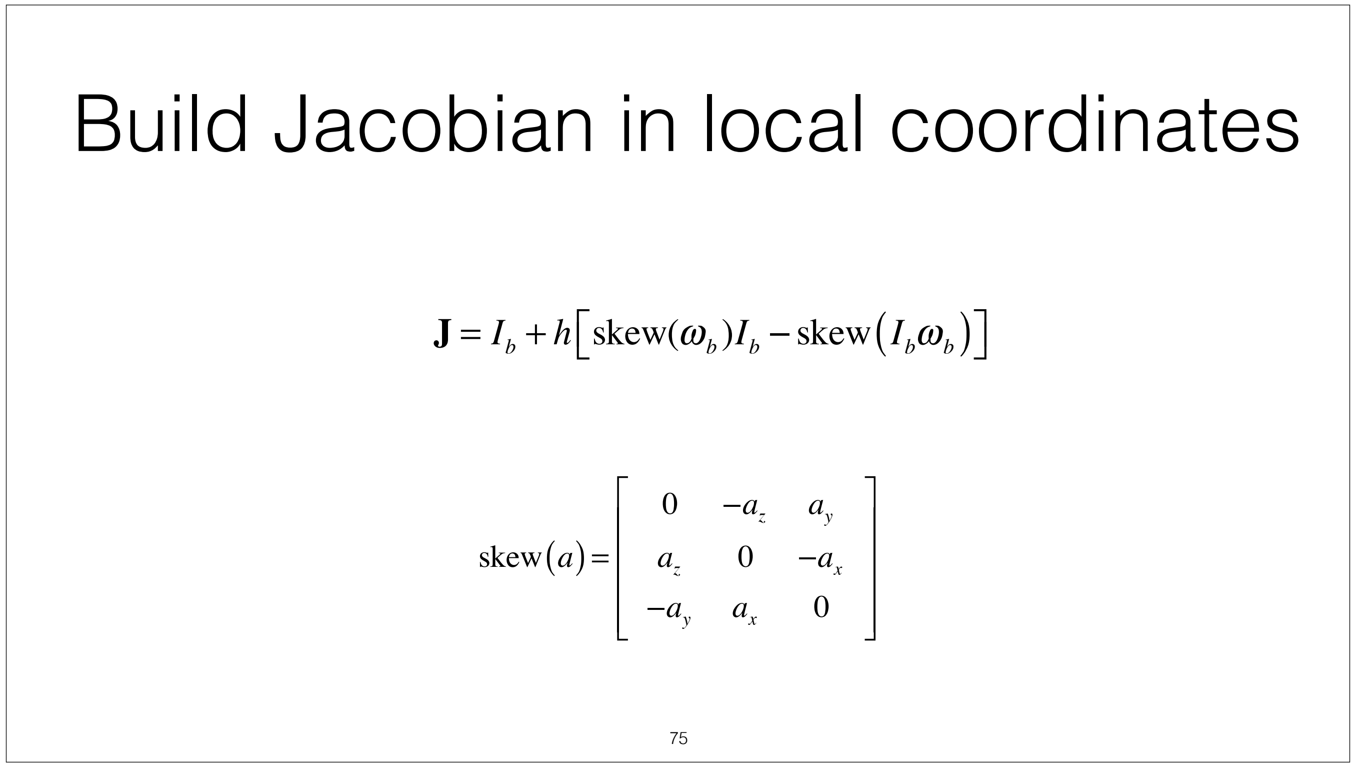
$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

$$\mathbf{J}\left(\mathbf{x}_{n+1} - \mathbf{x}_n\right) = \mathbf{f}_n$$

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix}$$

multiple variables

Here is a summary of Newton-Raphson. Typically it is described as a method for solving a single scalar equation. However, it can be extended to solve multiple simultaneous equations. To do this I need to compute the Jacobian of the function.



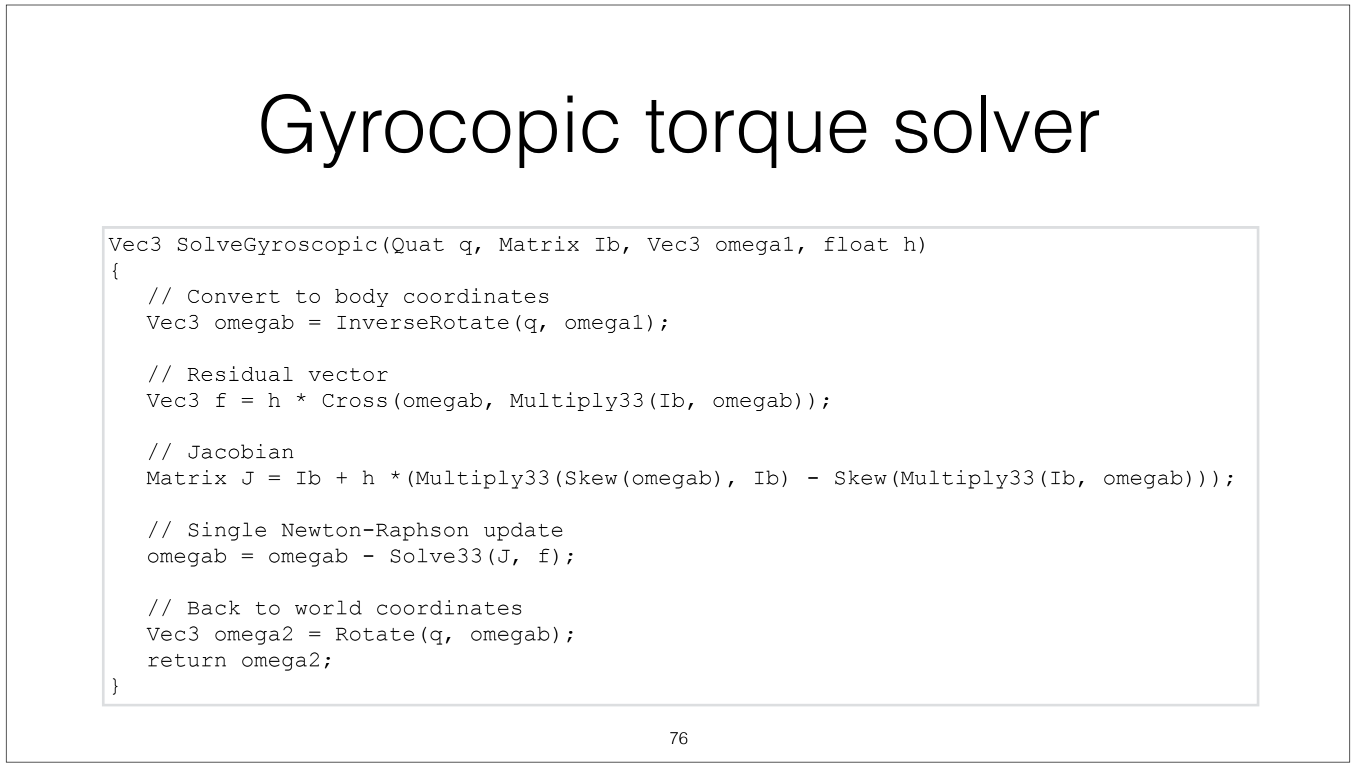
Build Jacobian in local coordinates

$$\mathbf{J} = I_b + h\left[\text{skew}(\omega_b) I_b - \text{skew}\left(I_b \omega_b\right)\right]$$

$$\text{skew}(a) = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}$$

75

Normally the inertia tensor is a function of orientation. However, I can build the implicit function for omega in local coordinates (b for body coordinates). The inertia tensor is constant in local coordinates so it does not need to be differentiated when I compute the Jacobian. Here the skew function takes a vector and builds the corresponding skew symmetric matrix. This vastly simplifies the solution.

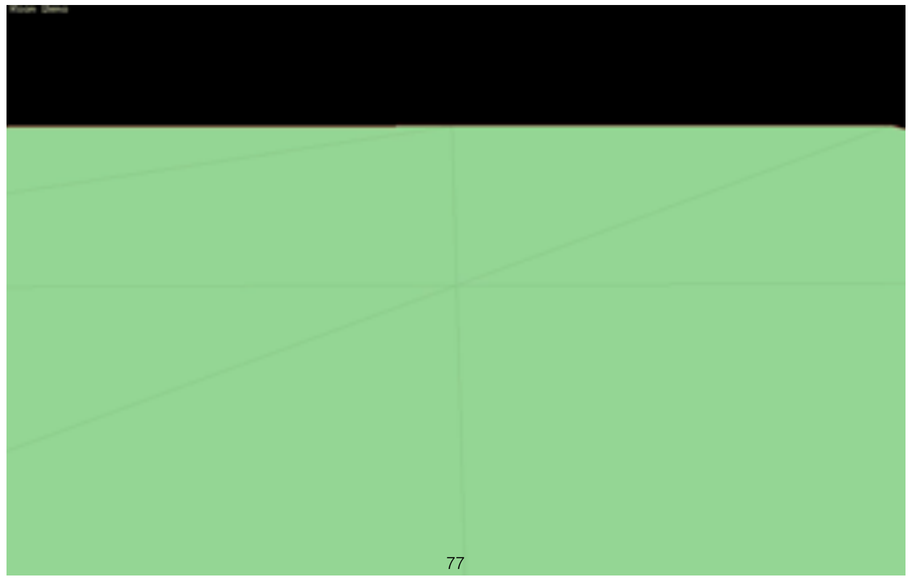


Gyrocopic torque solver

```
Vec3 SolveGyroscopic(Quat q, Matrix Ib, Vec3 omega1, float h)
{
   // Convert to body coordinates
   Vec3 omegab = InverseRotate(q, omega1);

   // Residual vector
   Vec3 f = h * Cross(omegab, Multiply33(Ib, omegab));

   // Jacobian
   Matrix J = Ib + h *(Multiply33(Skew(omegab), Ib) - Skew(Multiply33(Ib, omegab)));

   // Single Newton-Raphson update
   omegab = omegab - Solve33(J, f);

   // Back to world coordinates
   Vec3 omega2 = Rotate(q, omegab);
   return omega2;
}
```

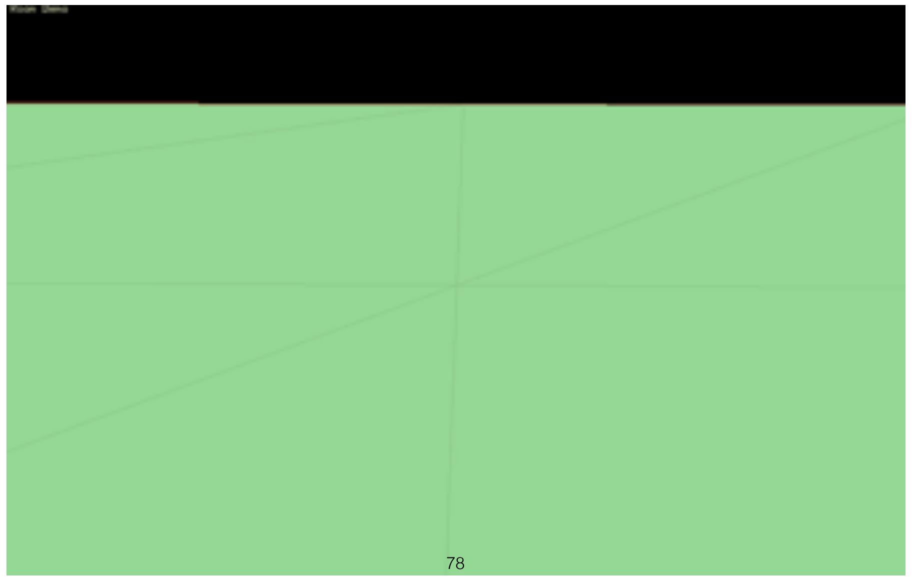
Here is the implicit gyroscopic force solver. It uses body coordinates because I do not have the new world inertia tensor yet.

Capsule implicit solution



Here is the result of using implicit solution of the gyroscopic torque. I only used one Newton-Raphson iteration.

With rolling resistance



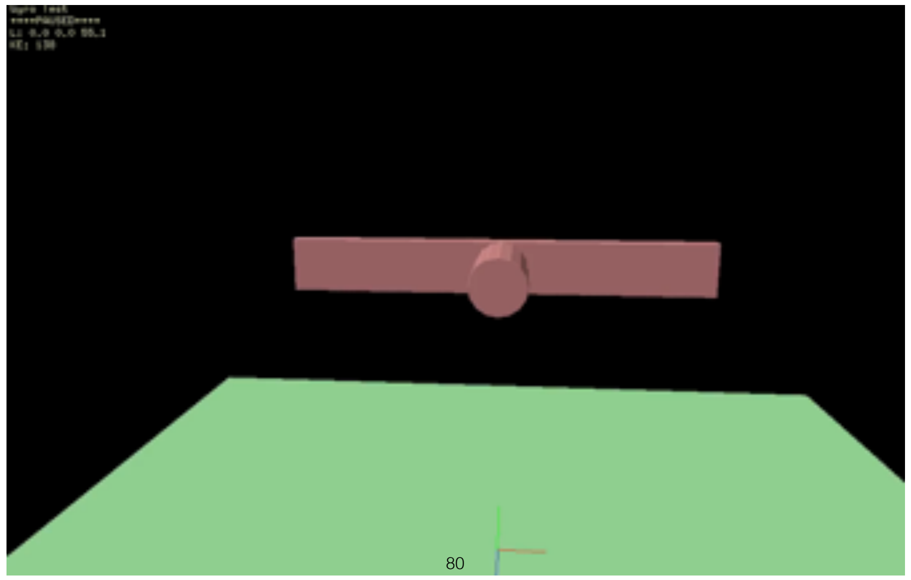
78

Adding rolling resistance makes the solution even better.

The Dzhanibekov Effect

Here is a video of the Dzhanibekov effect shown on the ISS. Similar effects can be seen in tennis rackets, white board erasers, and mobile phones (when they slip out of our hands, that is why they are so hard to catch).

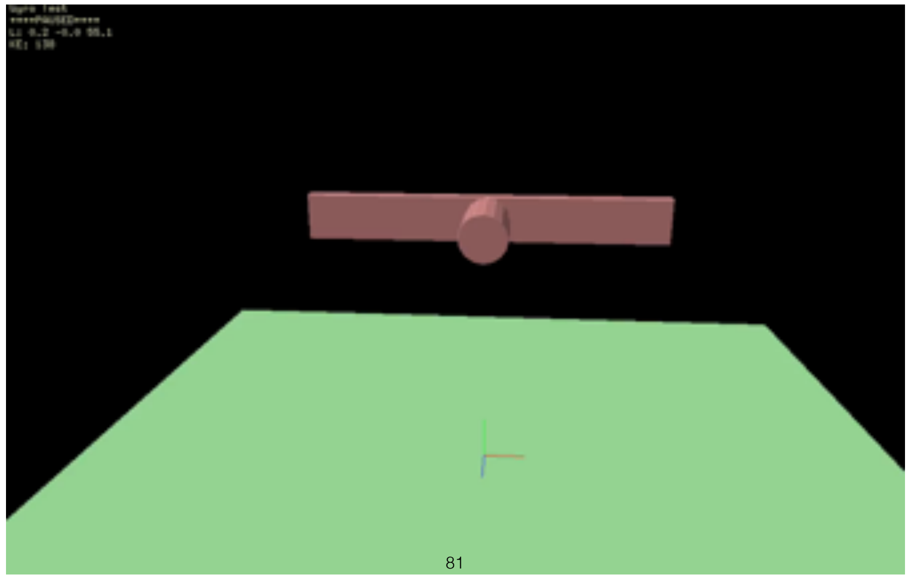
Explicit gyroscopic integration



```
Gyro Test
****PAUSED****
L: 0.0 0.0 55.1
KE: 130
```

I can the experiment using explicit Euler. Unfortunately the kinetic energy grows unbounded.

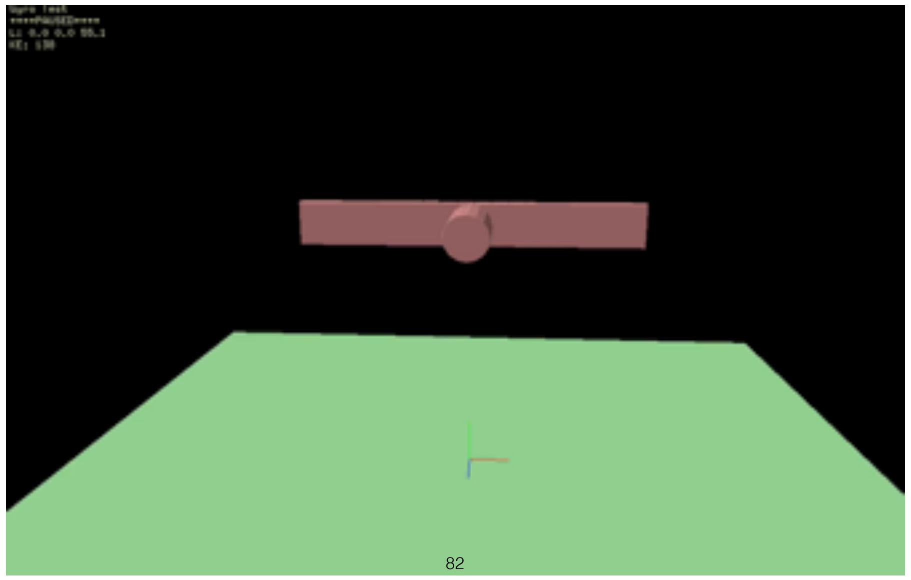
Dropping gyroscopic term



```
Gyro Test
****PAUSED****
L: 0.2 -0.0 55.1
KE: 130
```

Here I dropped the gyroscopic term and you can see there is no tumbling motion.

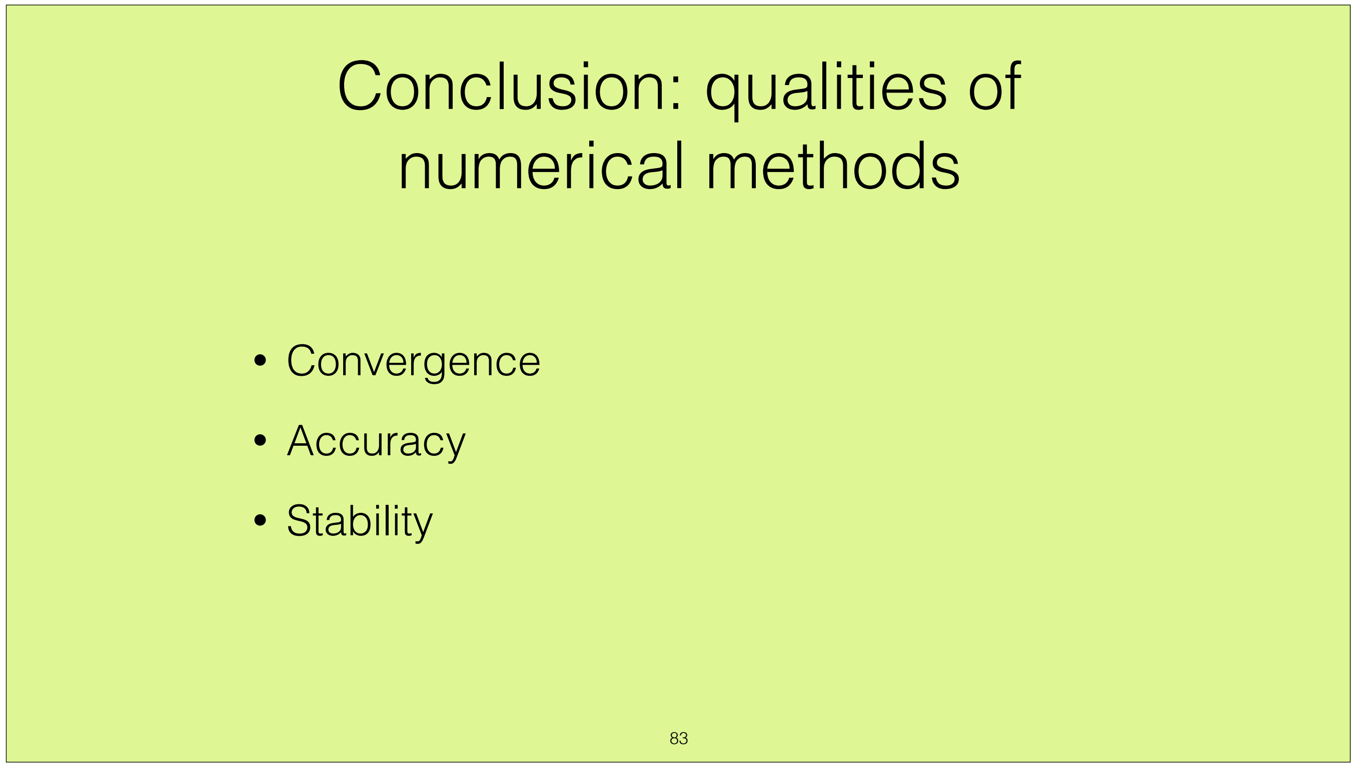
Implicit solution



```
Gyro Test
****PAUSED****
L: 0.0 0.0 55.1
KE: 138
```

In this video, I have included the implicit gyroscopic solver using one Newton-Raphson iteration. The tumbling motion is reproduced. You can see that the kinetic energy dissipates over time.

This sort of tumbling motion can make destruction and explosions look better in games, especially when the are long skinny objects.



Conclusion: qualities of numerical methods

- Convergence
- Accuracy
- Stability

There are few qualities of numerical methods that are used to find the best match for what our games need.
Convergence: as the time step gets smaller, the numerical result approaches the exact solution. For example, a numerical method that always returns 42 is not convergent for a harmonic oscillator.
Accuracy: how well the solution matches the solution for a given time step. In games we can set the accuracy bar low. We aren’t building satellites.
Stability: the numerical result grows unbounded even though the exact solution is unbounded

References

http://en.wikipedia.org/wiki/Semi-implicit_Euler_method

http://en.wikipedia.org/wiki/Euler's_equations_(rigid_body_dynamics)

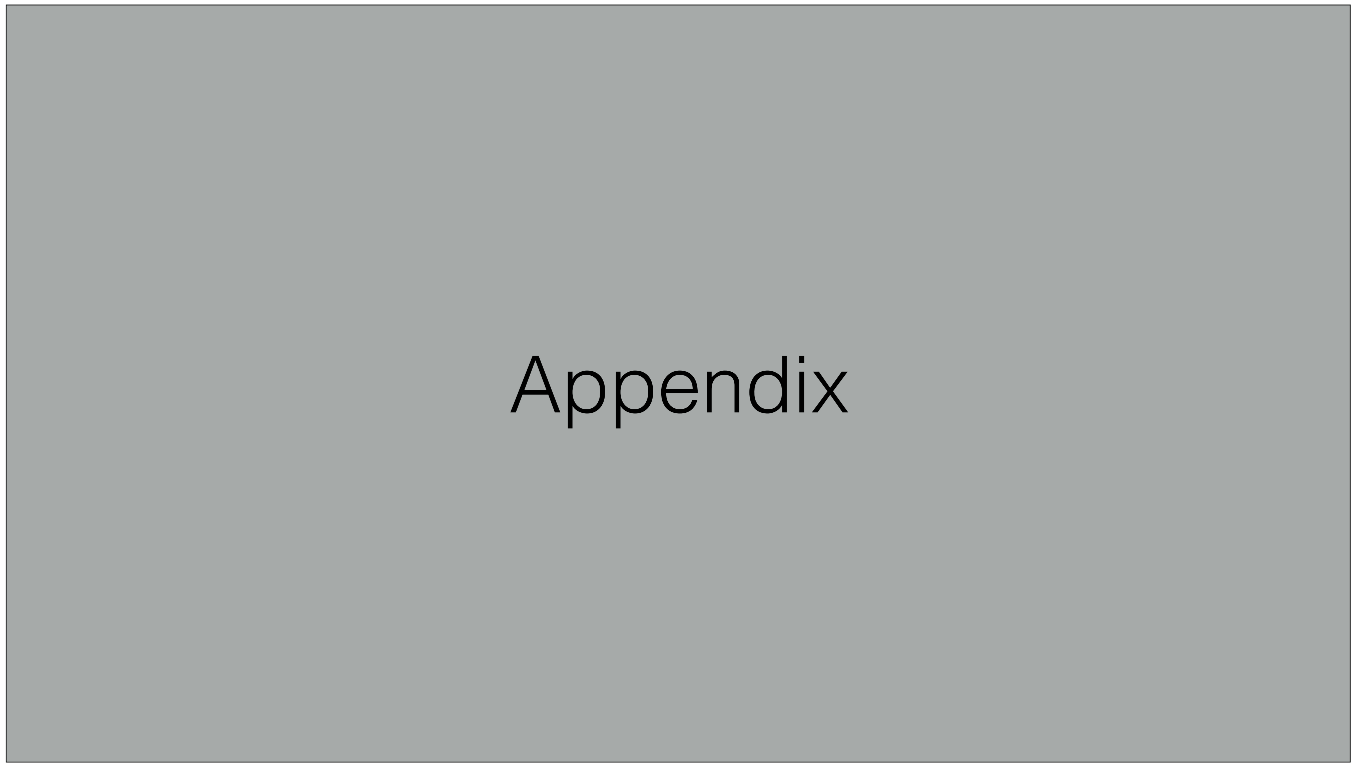
https://fgiesen.wordpress.com/2012/08/24/quaternion-differentiation/

http://en.wikipedia.org/wiki/Tennis_racket_theorem

Stabilizing Gyroscopic Forces In Rigid Multibody Simulations, Claude Lacoursie`re

http://www.av8n.com/physics/symplectic-integrator.htm

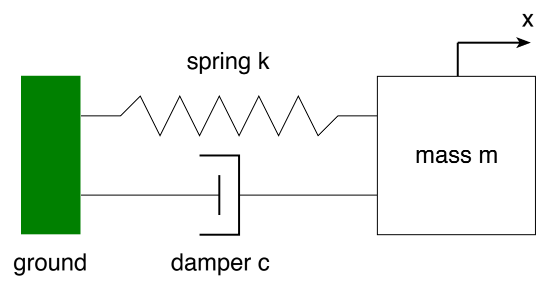
box2d.org @erin_catto



Appendix

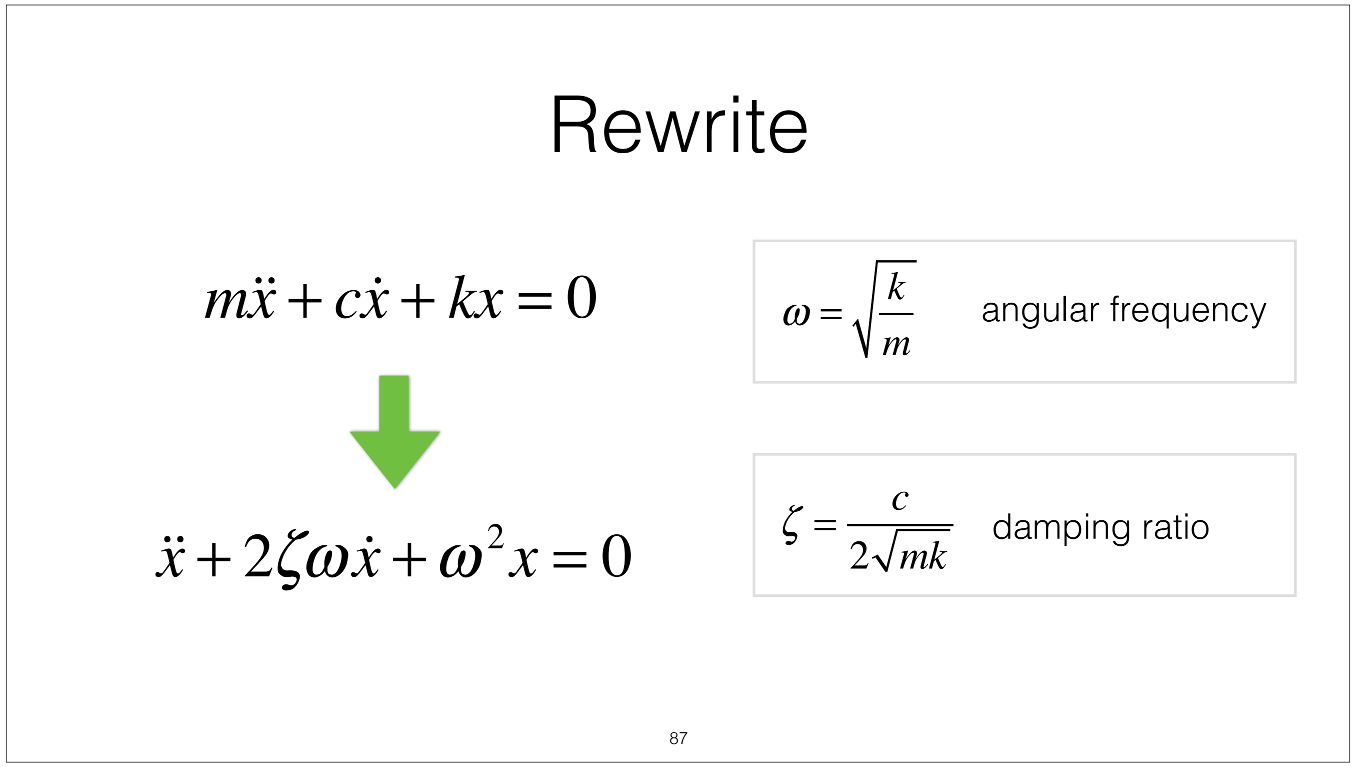
Here is stuff that didn’t fit into a one hour presentation.

Mass-Spring-Damper



$$m\ddot{x} + c\dot{x} + kx = 0$$

It is also interesting to add damping to see the effect on the phase portrait.



Rewrite

$$m\ddot{x} + c\dot{x} + kx = 0$$

$$\Downarrow$$

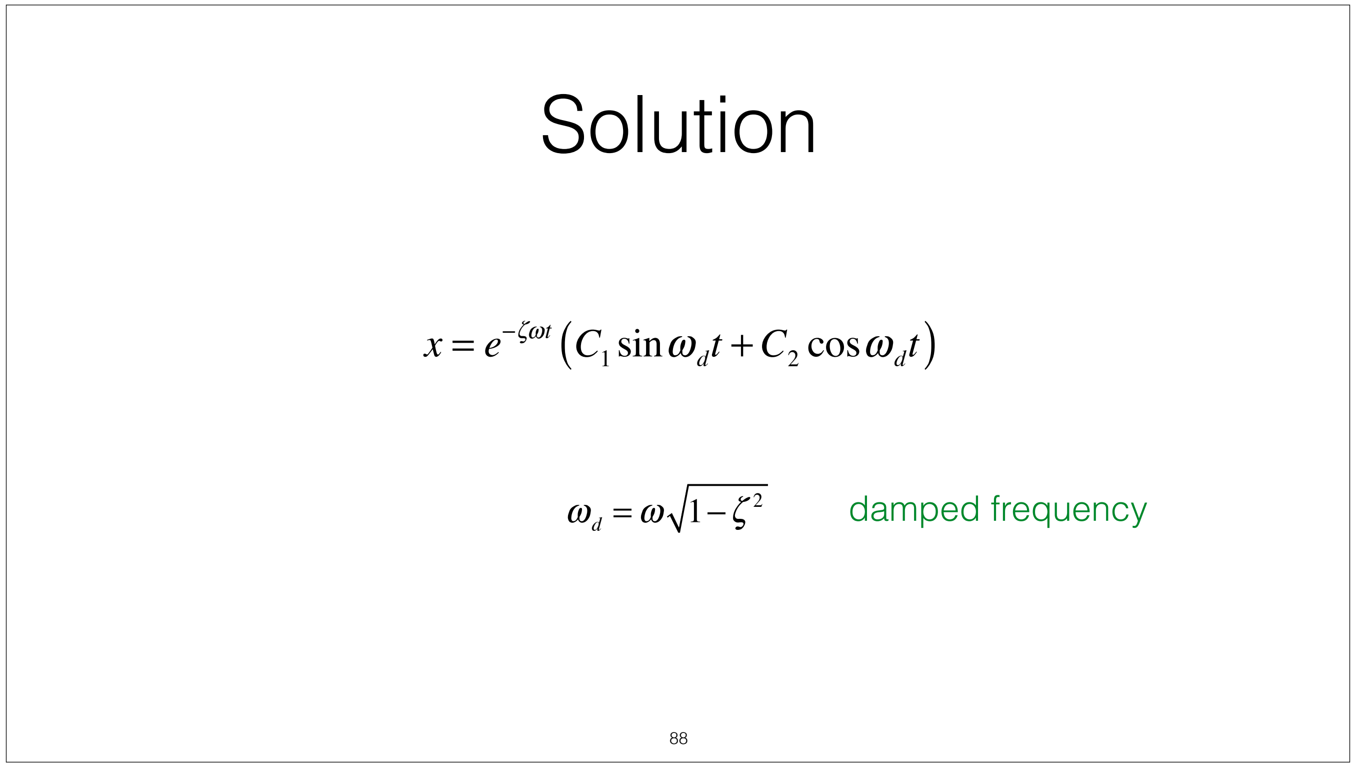
$$\ddot{x} + 2\zeta\omega\dot{x} + \omega^2 x = 0$$

$\omega = \sqrt{\frac{k}{m}}$ angular frequency

$\zeta = \frac{c}{2\sqrt{mk}}$ damping ratio

We can reduced from 3 parameters (m,k,c) to 2 parameters (omega, zeta) by dividing through by the mass. The new parameter zeta is called the damping ratio.

If zeta is zero, there is no damping. Increasing zeta increases damping. Once zeta reaches 1, the damping is so strong that all oscillation is eliminated.



Solution

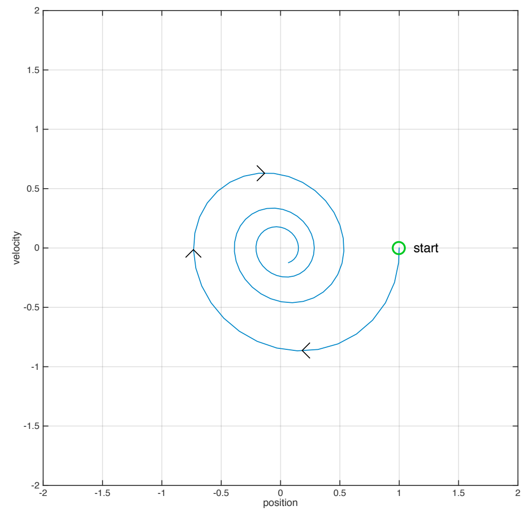
$$x = e^{-\zeta\omega t}\left(C_1 \sin\omega_d t + C_2 \cos\omega_d t\right)$$

$$\omega_d = \omega\sqrt{1-\zeta^2}$$ damped frequency

Here is the oscillatory solution to the mass-spring-damper. The constants C1 and C2 depend on the initial conditions.

When zeta is larger than one, then this solution no longer works. In that case the solution is just a decaying exponential.

Phase portrait



$x_0 = 1$

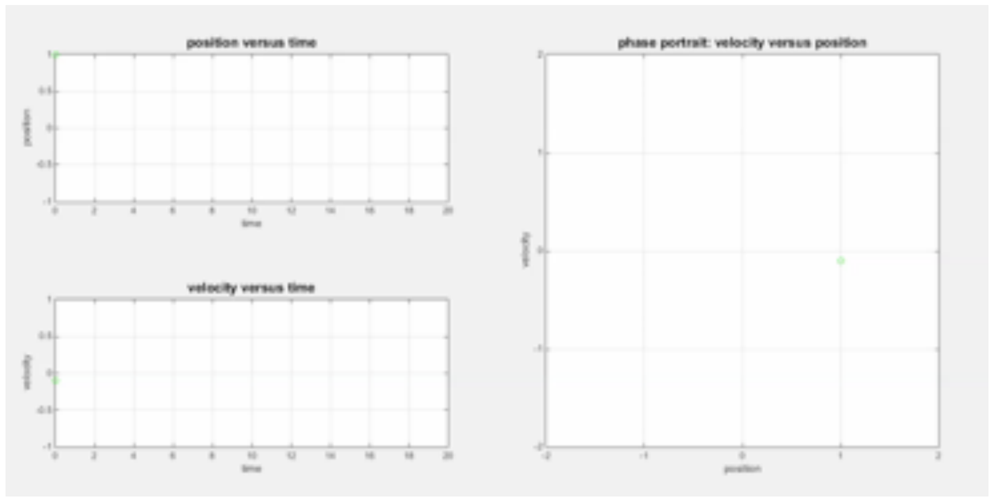
$v_0 = 0$

$\omega = 1$

$\zeta = 0.1$

Here is the phase portrait with a small amount of damping. As the time moves forward the position and velocity both decay to zero until the mass comes to rest.

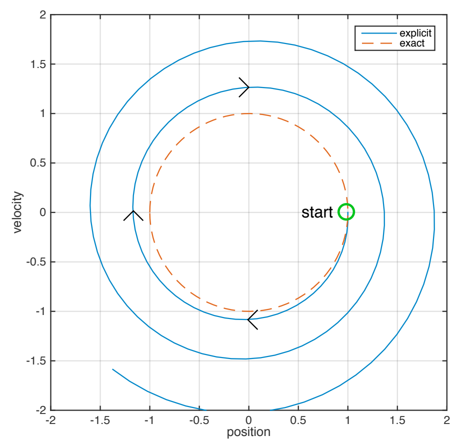
Phase portrait: velocity versus postion



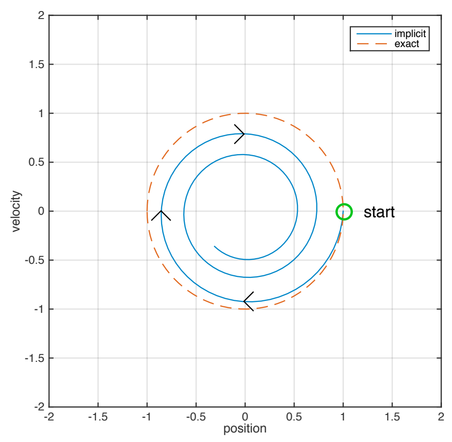
Phase portrait animation for mass-spring-damper system

Similarity of explicit and implicit Euler

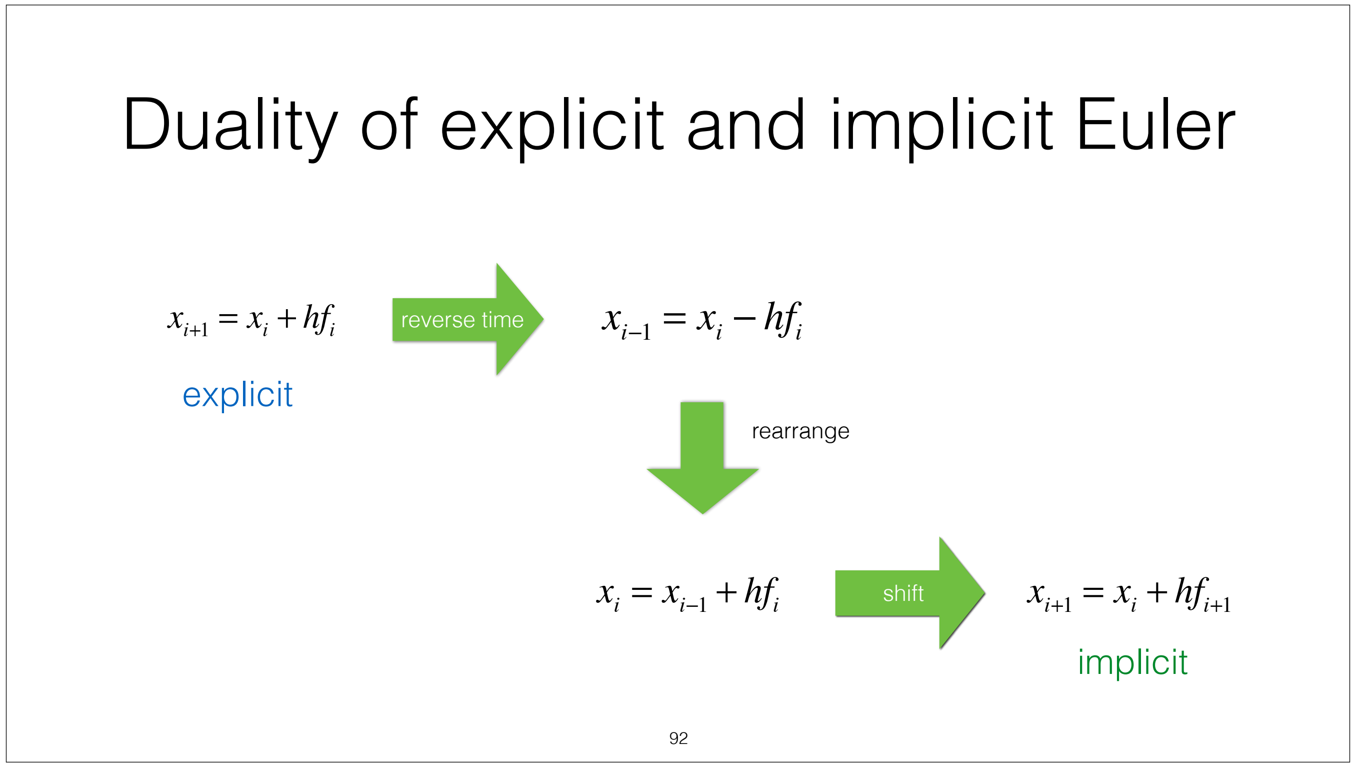
explicit



implicit



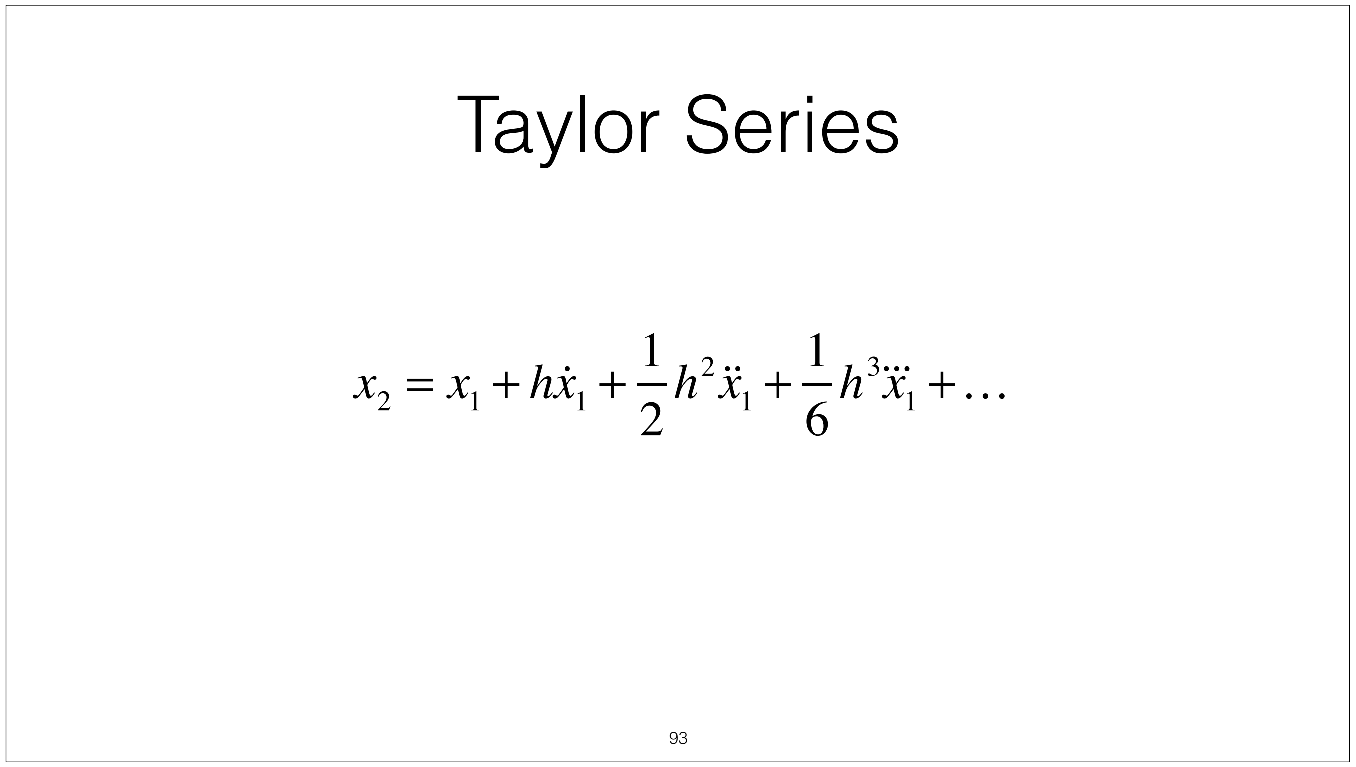
The phase portrait shows that implicit Euler is stable. Also, implicit Euler is damped. So even though the mass should oscillate forever, implicit Euler damps the motion and brings the mass to rest. I call this numerical damping.
If you compare this figure to explicit Euler, you should see that they look they look similar, except that explicit Euler is growing. This is because implicit Euler gives the same result as running explicit Euler in reverse.



Duality of explicit and implicit Euler

$$\begin{tikzcd}
x_{i+1} = x_i + hf_i \arrow[r, "\text{reverse time}"] & x_{i-1} = x_i - hf_i \arrow[d, "\text{rearrange}"] & \\
\text{explicit} & x_i = x_{i-1} + hf_i \arrow[r, "\text{shift}"] & x_{i+1} = x_i + hf_{i+1} \\
 & & \text{implicit}
\end{tikzcd}$$

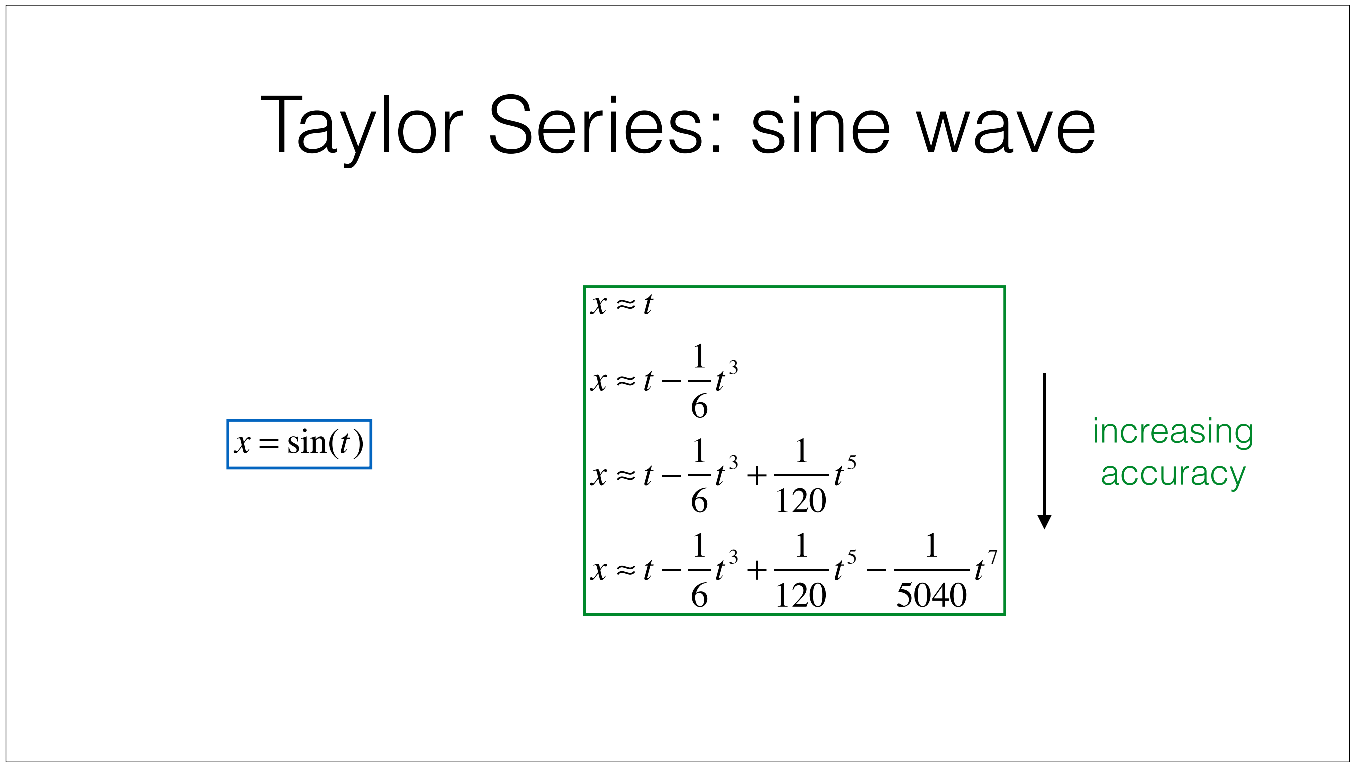
This shows that we can run time backwards on explicit Euler and get the same formula that is used for implicit Euler.



Taylor Series

$$x_2 = x_1 + h\dot{x}_1 + \frac{1}{2}h^2\ddot{x}_1 + \frac{1}{6}h^3\dddot{x}_1 + \dots$$

Finite difference formulas come from the Taylor Series, which states that any function can be extrapolated by using its current value and derivatives. The more terms we use, the more accurate the extrapolation. If we include the first derivative of x, we have first order accuracy. Including the second derivative gives second order accuracy, and so on. Notice that when h is small, high order derivatives contribute decreasing amounts to the approximation.



Taylor Series: sine wave

$\boxed{x = \sin(t)}$

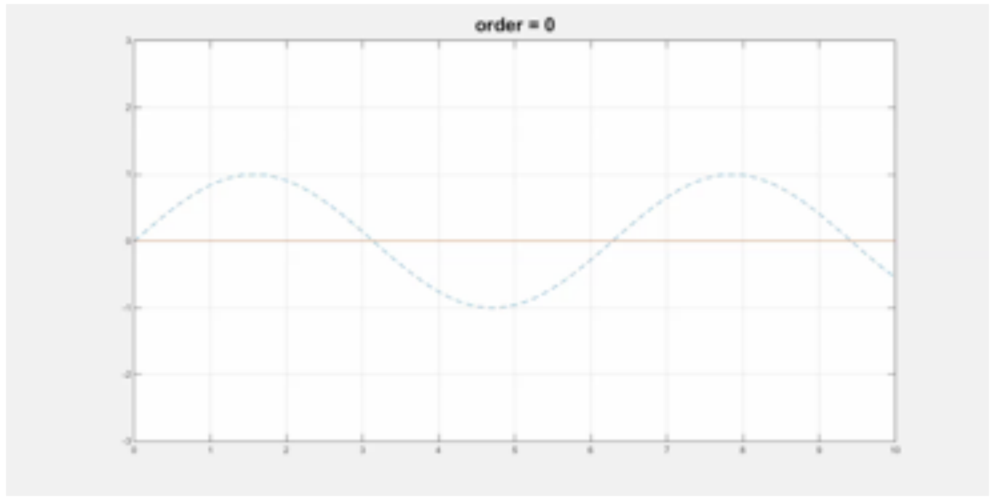
$$\begin{aligned}
&x \approx t \\
&x \approx t - \frac{1}{6}t^3 \\
&x \approx t - \frac{1}{6}t^3 + \frac{1}{120}t^5 \\
&x \approx t - \frac{1}{6}t^3 + \frac{1}{120}t^5 - \frac{1}{5040}t^7
\end{aligned}$$

$\downarrow$ increasing accuracy

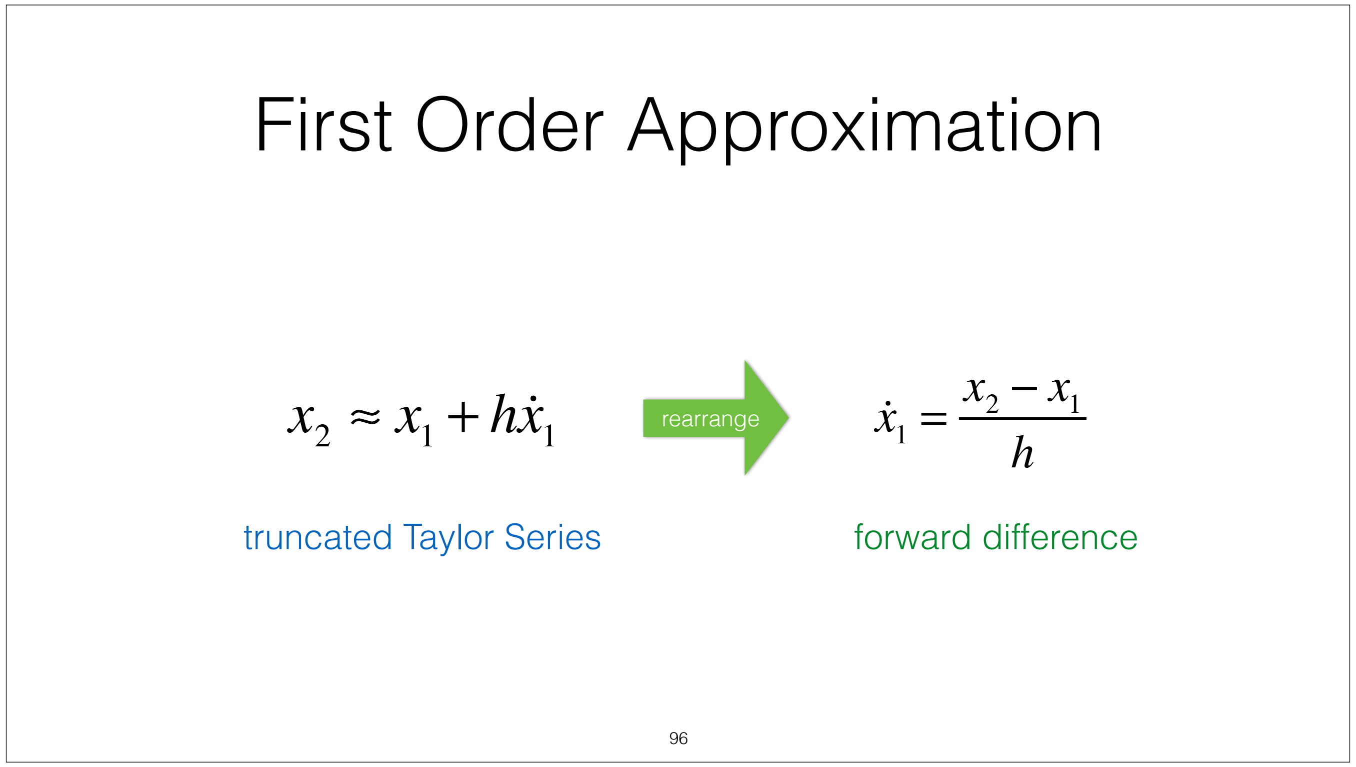
Here is the Taylor Series approximation of the sine function with increasing accuracy as the number of terms increases. Note that the approximations are polynomials in time.

The time could be just our time step h because we can start the Taylor Series extrapolation at any point of time.

Taylor Series: sine wave



Here is how the Taylor Series can approximate a sine wave with increasing order.



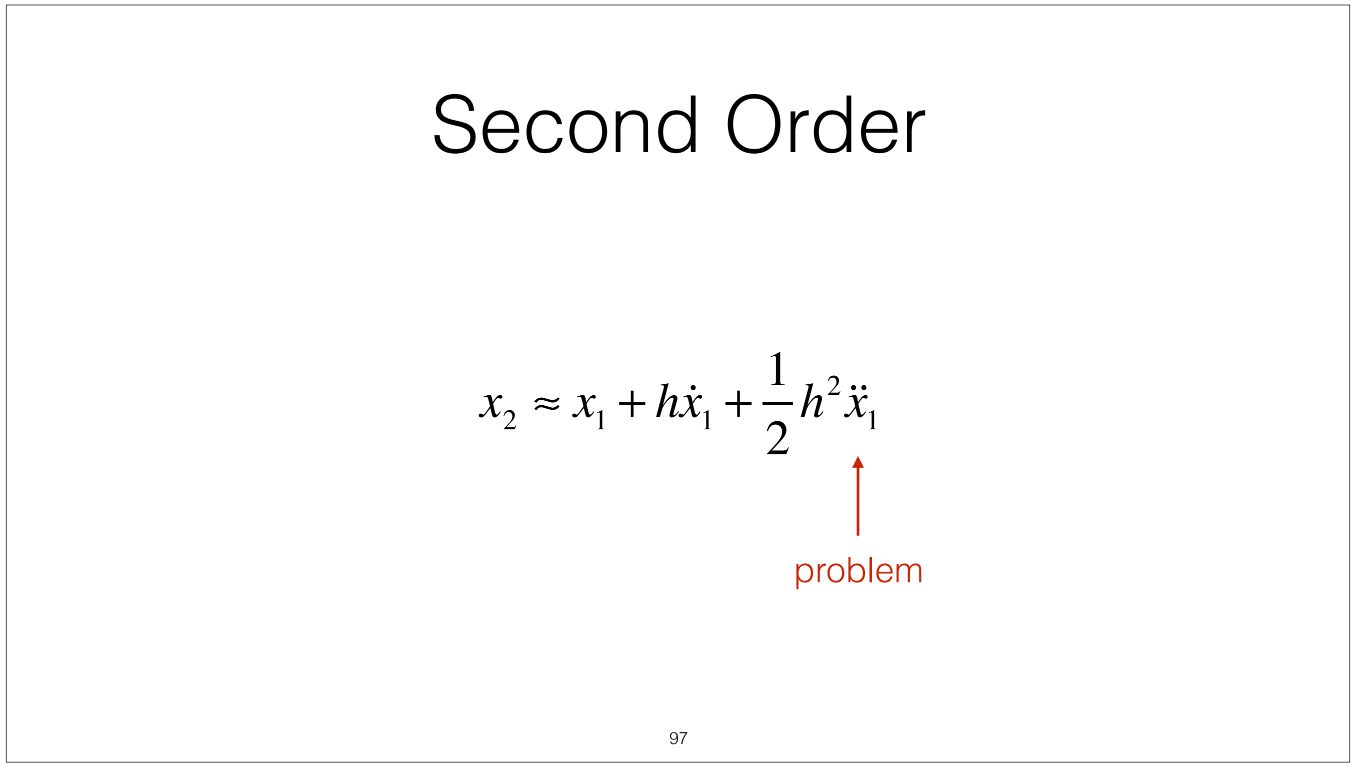
First Order Approximation

$$x_2 \approx x_1 + h\dot{x}_1 \xrightarrow{\text{rearrange}} \dot{x}_1 = \frac{x_2 - x_1}{h}$$

truncated Taylor Series

forward difference

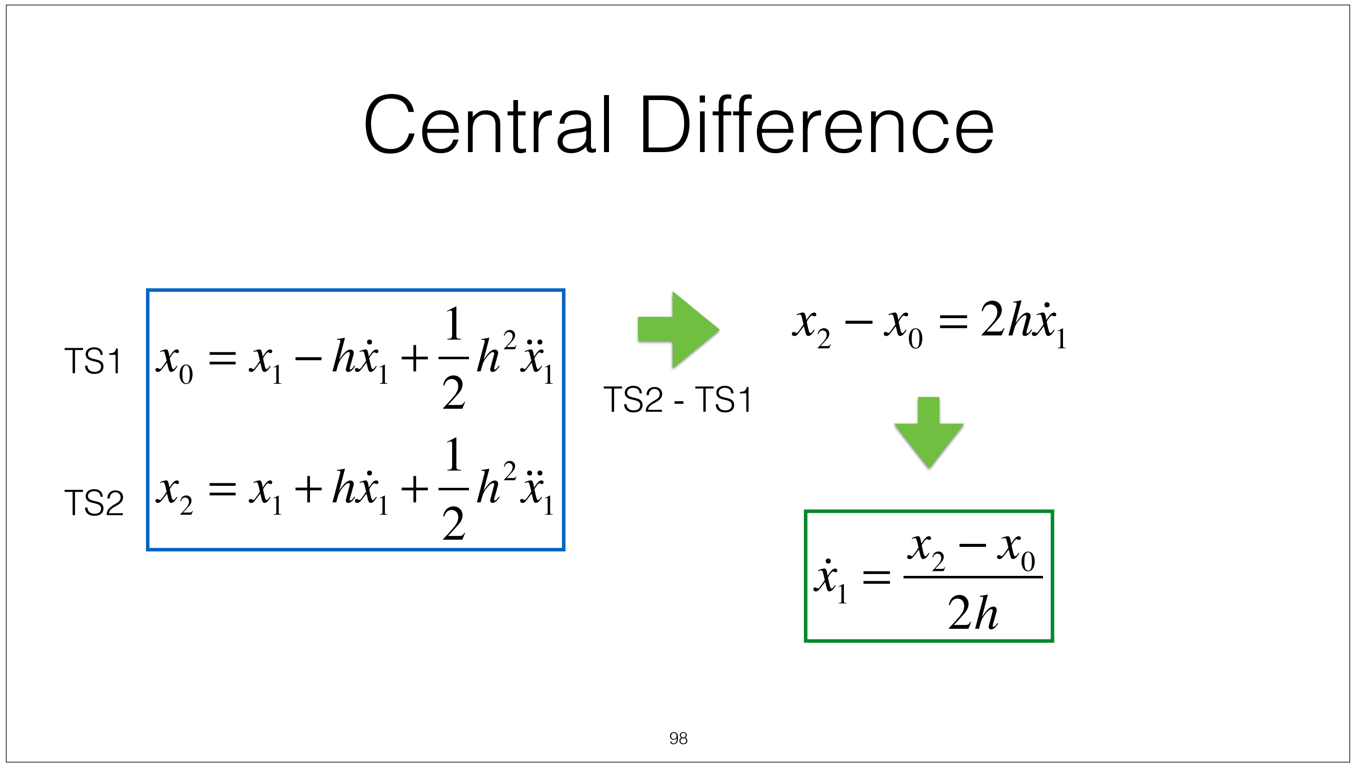
The first order approximation of the Taylor Series only keeps the first derivative. This is what I used already for the forward difference.



Second Order

$$x_2 \approx x_1 + h\dot{x}_1 + \frac{1}{2}h^2 \underset{\substack{\uparrow \\ \text{problem}}}{\ddot{x}_1}$$

Keeping the second derivative results in a second order approximation. I can use this to derive a second order finite difference approximation. However, it may not be easy to compute the second derivative of x when x defined by a differential equation. Fortunately, there is a trick for eliminating the second derivative.



Central Difference

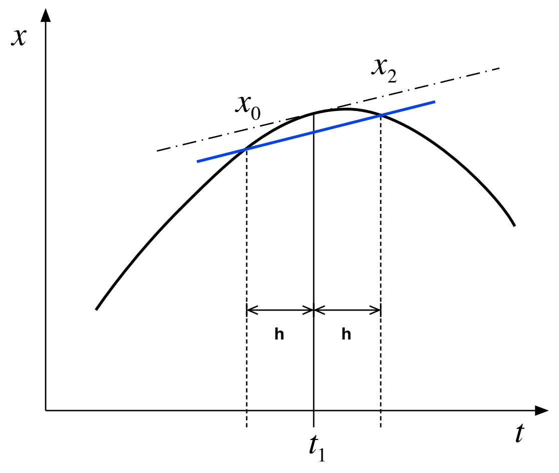
$$\begin{array}{l} \text{TS1} \\[1.5em] \text{TS2} \end{array} \boxed{\begin{array}{l} x_0 = x_1 - h\dot{x}_1 + \frac{1}{2}h^2\ddot{x}_1 \\ x_2 = x_1 + h\dot{x}_1 + \frac{1}{2}h^2\ddot{x}_1 \end{array}} \xrightarrow[\text{TS2 - TS1}]{} x_2 - x_0 = 2h\dot{x}_1$$

$$\Downarrow$$

$$\boxed{\dot{x}_1 = \frac{x_2 - x_0}{2h}}$$

Here I have two second order Taylor Series approximations. By subtracting the first equation from the second equation, I can eliminate the second derivative. This results in the second order finite difference formula called the central difference. This shows the power of the Taylor Series. You can combine more Taylor Series instances to generate ever more accurate finite difference formulas. Although this is exciting, it is not really necessary for games. We can get by on first order accuracy most of the time.

Central Difference



$$\dot{x}_1 = \frac{x_2 - x_0}{2h}$$

Here’s how the central difference looks graphically. It really does look like a better approximation of the slope at t1.