Game Physics Notes 02

CSCI 321

WWU

April 14, 2015

Forces

Newton's second law of motion: $F = ma$

$$\begin{aligned} a &= F/m \\ v' &= a \\ p' &= v \end{aligned}$$

> *Corpus omne perseverare in statu suo quiescendi vel movendi uniformiter in directum, nisi quatenus a viribus impressis cogitur statum illum mutare.*

Or, in English:

> *Every body perseveres in its state of being at rest or of moving uniformly straight forward, except insofar as it is compelled to change its state by force impressed.*

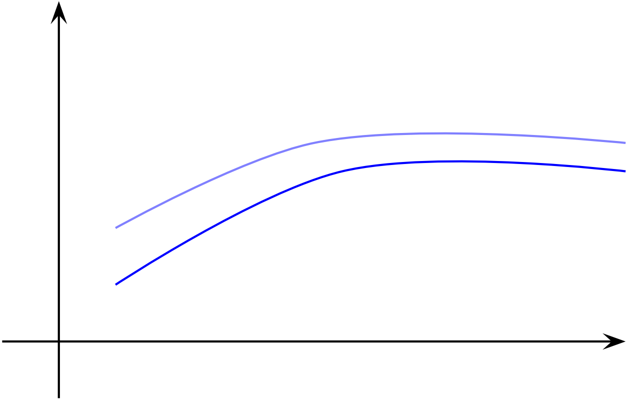
Forces and Motion

$$\begin{aligned} F &= ma \\ a &= F/m \\ v' &= a \\ p' &= v \end{aligned}$$

- What we really want to know is: "How do things move?"
- If we know the forces and masses, we know the acceleration.
- If we can integrate the acceleration we can get the velocity.
- If we can integrate the velocity we can get the position.
- The problem is integration—generally unsolvable.
- So we use approximate integration.

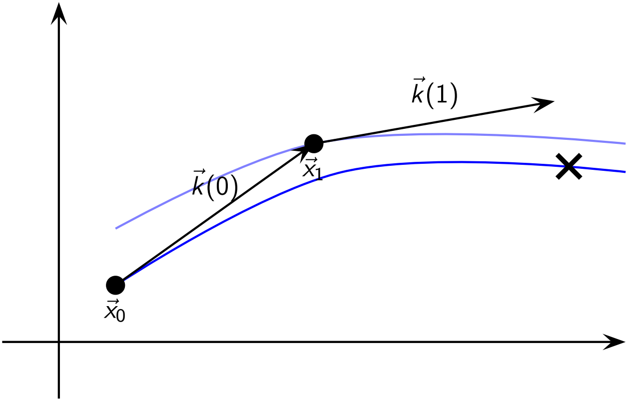
Euler Integration

Exact integration would move the point along the blue lines.



Euler Integration

$$
\begin{aligned}
\vec{k}(n) &= v(t, \vec{x}_n)\Delta t \\
\vec{x}_{n+1} &= \vec{x}_n + \vec{k}(n)
\end{aligned}
$$



Euler Integration

$$
\begin{aligned}
a &= F/m \\
v' &= a \\
p' &= v
\end{aligned}
$$

```
def update(F, m, dt):
  a = F / m
  v += a * dt
  p += v * dt
```

Sample Calculations

$$
\begin{aligned}
dt &= 1 \\
m &= 10 \\
k &= 5 \\
f &= -kx \\
a &= f/m = -kx/m = -x/2 \\
x' &= v \\
v' &= a \\
x_{t+1} &= x_t + x'_t = x_t + v_t \\
v_{t+1} &= v_t + v'_t = v_t + a_t
\end{aligned}
$$

- Run `spring.py`

Euler:

t	x	v	a
0	20	0	-10
1	20	-10	-10
2	10	-20	-5
3	-10	-25	5
4	-35	-20	17
5	-55	-3	27

Online discussions of Midpoint and Runge Kutta

Readings:

- `http://www.pixar.com/companyinfo/research/pbm2001/`, Differential equation basics, and Particle dynamics
- `http://www.nrbook.com/c/`, 16.0, 16.1

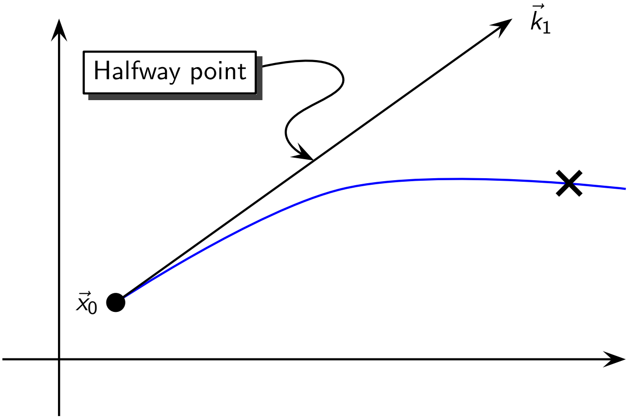
Midpoint Method

$$
\begin{aligned}
\vec{k}_1 &= d(\vec{x}_n)\Delta t \\
\vec{k}_2 &= d(\vec{x}_n + \frac{1}{2}\vec{k}_1)\Delta t \\
\vec{x}_{n+1} &= \vec{x}_n + \vec{k}_2
\end{aligned}
$$

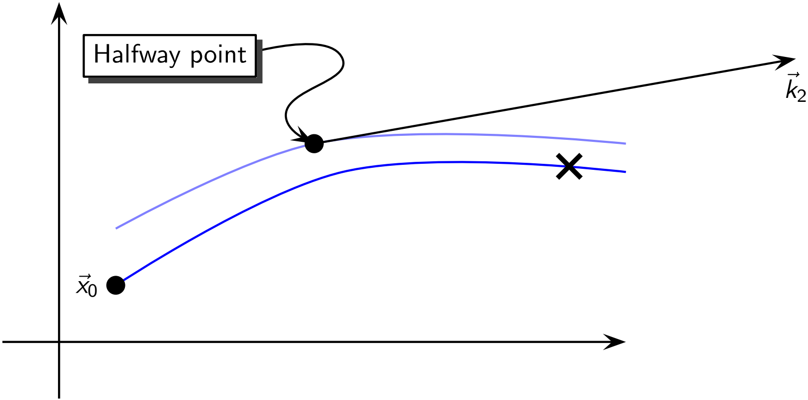
- Euler method has errors $O(\Delta t^2)$
- Midpoint method has errors $O(\Delta t^3)$
- Can take steps twice as big and get smaller errors:

$$
\begin{aligned}
0.05^2 &= 0.0025 \\
0.10^3 &= 0.001
\end{aligned}
$$

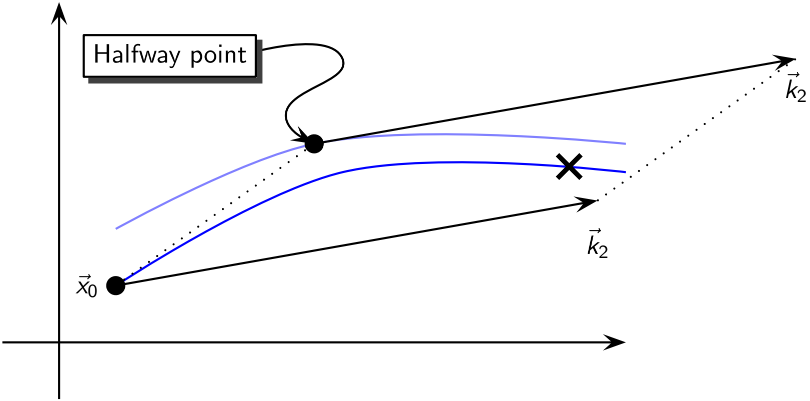
Midpoint Method



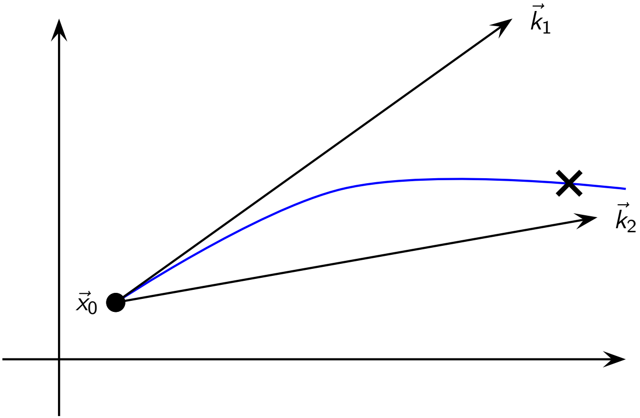
Midpoint Method



Midpoint Method

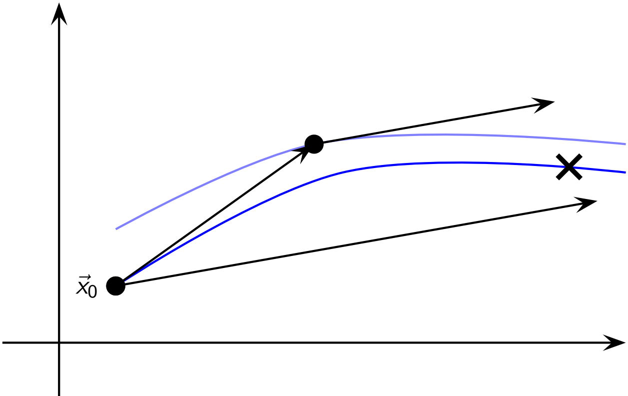


Midpoint Method



Midpoint Method

One midpoint method step of size Δt is more accurate than two Euler steps of size $\Delta t/2$.



Sample Calculations

$$
\begin{aligned}
dt &= 1 \\
m &= 10 \\
k &= 5 \\
f &= -kx \\
a &= f/m = -kx/m = -x/2 \\
x' &= v \\
v' &= a \\
x_{t+1} &= x_t + x'_t = x_t + v_t \\
v_{t+1} &= v_t + v'_t = v_t + a_t
\end{aligned}
$$

Midpoint:

t	x	v	a
0.0	20	0	-10
0.5	20	-5	-10
1.0	15	-10	-7
1.5	10	-13	-5
2.0	2	-15	-1
2.5	-5	-15	2
3.0	-13	-13	6
3.5	-19	-10	9
4.0	-23	-4	11
4.5	-25	1	12
5.0	-24	8	12

First add half of the derivative.

Sample Calculations

$$
\begin{aligned}
dt &= 1 \\
m &= 10 \\
k &= 5 \\
f &= -kx \\
a &= f/m = -kx/m = -x/2 \\
x' &= v \\
v' &= a \\
x_{t+1} &= x_t + x'_t = x_t + v_t \\
v_{t+1} &= v_t + v'_t = v_t + a_t
\end{aligned}
$$

Midpoint:

t	x	v	a
0.0	20	0	-10
0.5	20	-5	-10
1.0	15	-10	-7
1.5	10	-13	-5
2.0	2	-15	-1
2.5	-5	-15	2
3.0	-13	-13	6
3.5	-19	-10	9
4.0	-23	-4	11
4.5	-25	1	12
5.0	-24	8	12

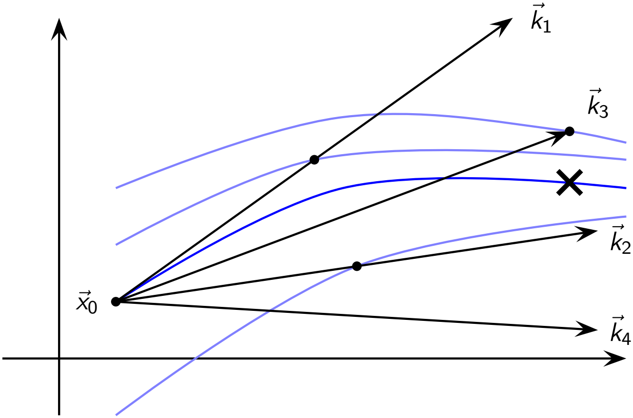
Then add all the "half-derivative."

Fourth Order Runge-Kutta

$$\begin{aligned}
\vec{k}_1 &= d(\vec{x}_n)\Delta t \\
\vec{k}_2 &= d(\vec{x}_n + \frac{1}{2}\vec{k}_1)\Delta t \\
\vec{k}_3 &= d(\vec{x}_n + \frac{1}{2}\vec{k}_2)\Delta t \\
\vec{k}_4 &= d(\vec{x}_n + \vec{k}_3)\Delta t \\
\vec{x}_{n+1} &= \vec{x}_n + \frac{\vec{k}_1}{6} + \frac{\vec{k}_2}{3} + \frac{\vec{k}_3}{3} + \frac{\vec{k}_4}{6}
\end{aligned}$$

Fourth order Runge Kutta

Tangents calculated at the dots: $\frac{\vec{k}_1}{6} + \frac{\vec{k}_2}{3} + \frac{\vec{k}_3}{3} + \frac{\vec{k}_4}{6}$



Fourth Order Runge-Kutta

- Euler method has errors $O(\Delta t^2)$
- Midpoint method has errors $O(\Delta t^3)$
- Fourth order Runge Kutta has errors $O(\Delta t^5)$
- $$\begin{aligned} 0.05^2 &= 0.00250 \\ 0.10^3 &= 0.00100 \\ 0.20^5 &= 0.00032 \end{aligned}$$

Adaptive stepsize

- Change Δt as you go along...
- ...depending on how much things are changing.

Differential Equations

Reading:

- Strange attractors http://en.wikipedia.org/wiki/Attractor
- Run: strange??.py
- The Limits to Growth

http://www.manicore.com/fichiers/Turner_Meadows_vs_historical_data.pdf

http://www.theguardian.com/commentisfree/2014/sep/02/limits-to-growth-was-right-new-research-shows-were-nea

Symplectic Euler/Semi-implicit Euler

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

$$\begin{aligned} v_{n+1} &= v_n + a_n \Delta t \\ p_{n+1} &= p_n + v_{n+1} \Delta t \end{aligned}$$

and

$$\begin{aligned} p_{n+1} &= p_n + v_n \Delta t \\ v_{n+1} &= v_n + a_{n+1} \Delta t \end{aligned}$$

- Can use either one by itself, or alternate between them.
- Not accurate, but almost conserves energy.
- Easy to program when updates are by assignment.

Verlet Integration

- Begin with symplectic Euler

$$\begin{aligned} v_{n+1} &= v_n + a_n \Delta t \\ p_{n+1} &= p_n + v_{n+1} \Delta t \end{aligned}$$

- Substitute for v_{n+1}

$$\begin{aligned} v_{n+1} &= v_n + a_n \Delta t \\ p_{n+1} &= p_n + (v_n + a_n \Delta t) \Delta t \\ &= p_n + v_n \Delta t + a_n \Delta t^2 \end{aligned}$$

- Use old positions to approximate $v_n \Delta t \approx p_n - p_{n-1}$

$$\begin{aligned} p_{n+1} &= p_n + v_n \Delta t + a_n \Delta t^2 \\ &= p_n + (p_n - p_{n-1}) + a_n \Delta t^2 \\ &= 2p_n - p_{n-1} + a_n \Delta t^2 \end{aligned}$$

- This is *velocityless Verlet*. There are other versions.

Verlet Integration

- A Verlet based approach for 2D game physics (`www.gamedev.net`)

  `http://www.gamedev.net/page/resources/_/technical/math-and-physics/a-verlet-based-approach-for-2d-`

- A nice web demo:

  `http://gamedev.tutsplus.com/tutorials/implementation/simulate-fabric-and-ragdolls-with-simple-verl`

- Can be used as the basis of a collision response system.
- Run `VerletPhysicsDemo.py`

True elastic collisions

- `http://en.wikipedia.org/wiki/Elastic_collision`
- Run `BouncingBalls.py`