Game Physics Notes 02

CSCI 321

WWU

April 13, 2016

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.*

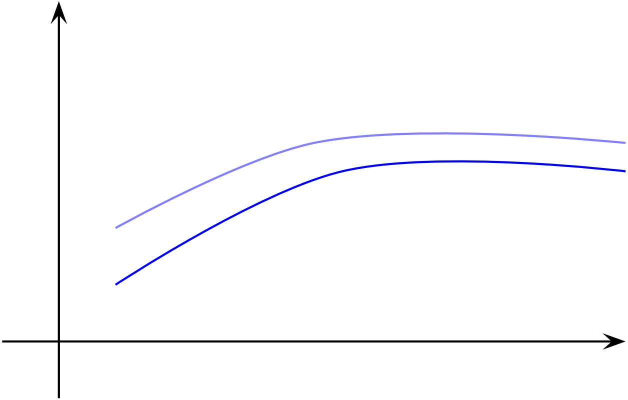
Forces and Motion

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

- 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.

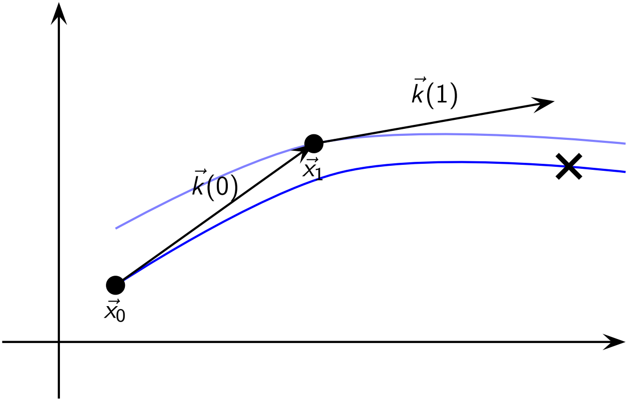
Euler Integration

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
0.0	20.0	0.0	-10.0
1.0	20.0	-10.0	-10.0
2.0	10.0	-20.0	-5.0
3.0	-10.0	-25.0	5.0
4.0	-35.0	-20.0	17.5
5.0	-55.0	-2.5	27.5

Sample Calculations with smaller timestep

$$
\begin{aligned}
dt &= 0.5 \\
m &= 10 \\
k &= 5 \\
f &= -kx \\
a &= f/m = -kx/m = -x/2 \\
x' &= v \\
v' &= a \\
x_{t+1} &= x_t + x'_t = x_t + v_t \\
v_{t+1} &= v_t + v'_t = v_t + a_t
\end{aligned}
$$

Euler:

t	x	v	a
0.0	20.0	0.0	-10.0
0.5	20.0	-5.0	-10.0
1.0	17.5	-10.0	-8.8
1.5	12.5	-14.4	-6.3
2.0	5.3	-17.5	-2.7
2.5	-3.4	-18.8	1.7
3.0	-12.9	-18.0	6.4
3.5	-21.8	-14.8	10.9
4.0	-29.2	-9.3	14.6
4.5	-33.9	-2.0	16.9
5.0	-34.9	6.5	17.4

- Run `spring.py`

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

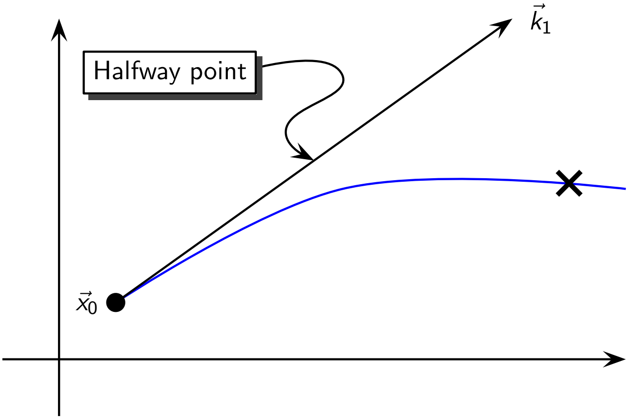
Midpoint Method

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

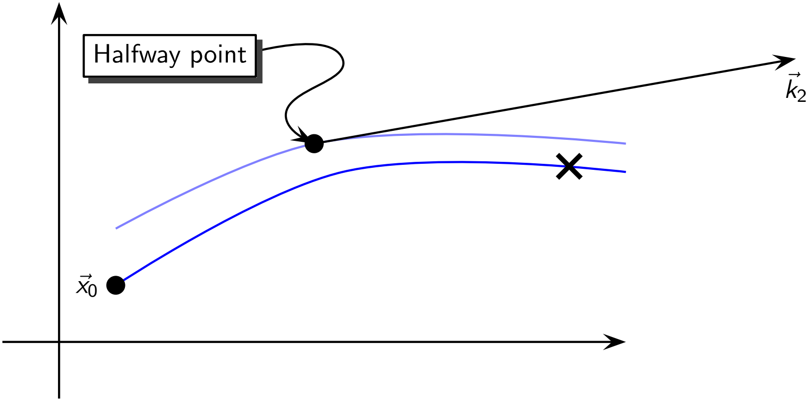
- Euler method has errors $O(\Delta t^2)$
- Midpoint method has errors $O(\Delta t^3)$
- Can take steps twice as big and get smaller errors:

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

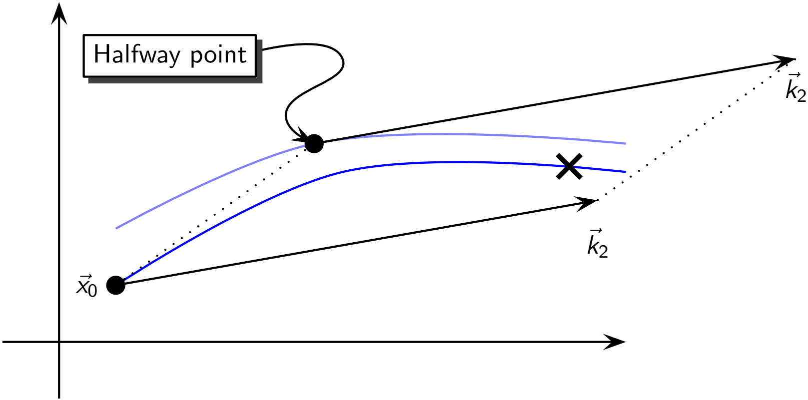
Midpoint Method



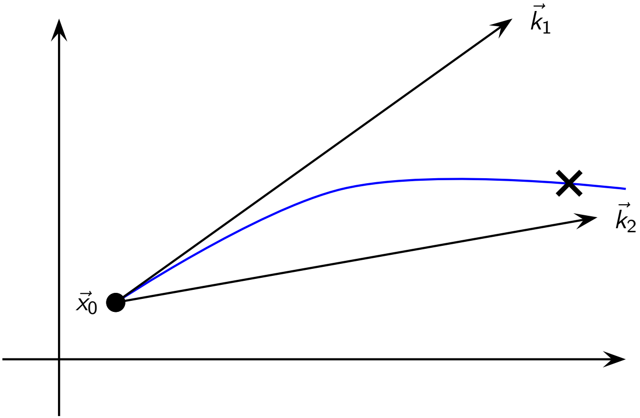
Midpoint Method



Midpoint Method

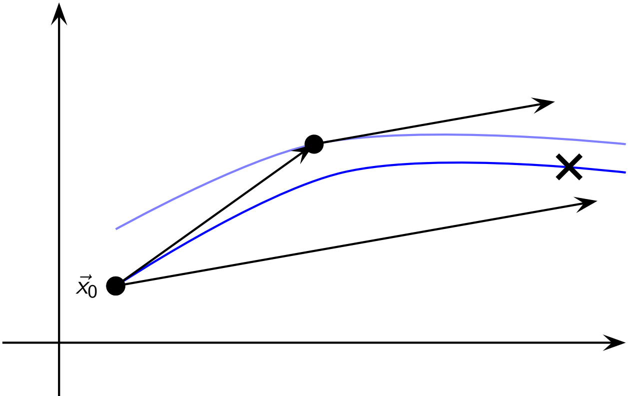


Midpoint Method



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	0.0	-10.0
0.5	20.0	-5.0	-10.0
1.0	15.0	-10.0	-7.5
1.5	10.0	-13.8	-5.0
2.0	1.3	-15.0	-0.6
2.5	-6.3	-15.3	3.1
3.0	-14.1	-11.9	7.0
3.5	-20.0	-8.4	10.0
4.0	-22.4	-1.9	11.2
4.5	-23.4	3.7	11.7
5.0	-18.7	9.8	9.3

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	0.0	-10.0
0.5	20.0	-5.0	-10.0
1.0	15.0	-10.0	-7.5
1.5	10.0	-13.8	-5.0
2.0	1.3	-15.0	-0.6
2.5	-6.3	-15.3	3.1
3.0	-14.1	-11.9	7.0
3.5	-20.0	-8.4	10.0
4.0	-22.4	-1.9	11.2
4.5	-23.4	3.7	11.7
5.0	-18.7	9.8	9.3

Then add all the “half-derivative.”

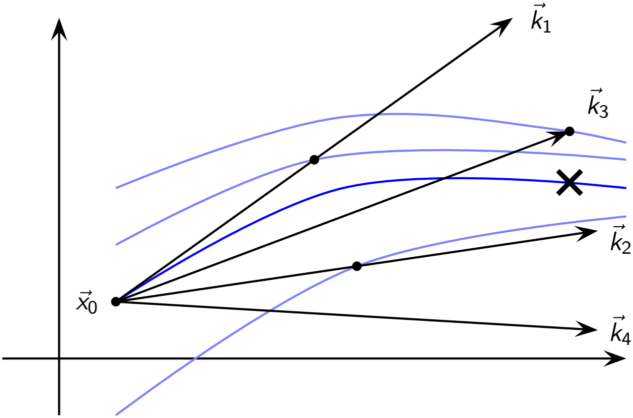
Note that this is far more accurate than Euler with half step size.

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 $O(\Delta t^5)$
-

$$\begin{aligned} 0.05^2 &= 0.00250 \\ 0.10^3 &= 0.00100 \\ 0.20^5 &= 0.00032 \end{aligned}$$

Stepsize Matching Refresh Rate

- The simplest approach to stepsize is to use the framerate:

```
framerate = 30.0
t = 0.0
dt = 1.0/framerate
while !quitting:
  clock.tick(framerate)
  handle.input()
  integrate(state, t, dt)
  t += dt
  display()
```

- This may be OK for simple games, but if more accuracy is needed the physics should use as small a timestep as possible.
- Also, the game refresh rate may not keep up with the nominal clock rate.

Stepsize Matching Refresh Rate $\times n$

- Can also match n steps to each frame:

```
framerate = 30.0
t = 0.0
dt = 1.0/framerate
while !quitting:
  clock.tick(framerate)
  handle.input()
  for i in range(n):
     integrate(state, t, dt/n)
  t += dt
  display()
```

Use actual timestep

- `clock.tick` returns milliseconds since last call.

```
framerate = 30.0
t = 0.0
while !quitting:
  dt = clock.tick(framerate) * 0.001
  handle.input()
  integrate(state, t, dt)
  t += dt
  display()
```

- Physics will be “same” regardless of computer’s speed.
- But again physics update should be as fast as possible for most realism.
- We could increase the framerate, but then we’d be doing unnecessary rendering.

Use smaller time step

```
framerate = 30.0
t = 0.0
dt = 0.01
while !quitting:
  timespan = clock.tick(framerate) * 0.001
  handle.input()
  while (timespan > 0):
    integrate(state, t, dt)
    timespan -= dt
  display()
```

- Problem with the fractional part of `dt`?
  - Can interpolate for fractional `dt`.
- What if the physics gets behind? Spiral of death!
  - Make sure your physics can keep up with `dt`.

Use separate time for display and physics

```
framerate = 30.0
rendertime, physicstime = 0.0, 0.0
dt = 0.01
while !quitting:
  rendertime += clock.tick(framerate) * 0.001
  handle.input()
  while (physicstime < rendertime):
    integrate(state, physicstime, dt)
    physicstime += dt
  display()
```

- Spiral of death still possible.
  - Note: matching stepsize to framerate$\times n$ avoids death spiral.
- Leftover fraction of `dt` is carried forward to next render.
- Can interpolate again for fractional `dt`.
- Note that `dt` can be *longer* than time for a frame and it still works.

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`