Game Physics Notes 02

CSCI 321

WWU

October 12, 2017

Forces

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

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

> *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

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

- What we really want to know is: “How do things move?”

Forces and Motion

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

- What we really want to know is: “How do things move?”
- If we know the forces and masses, we know the acceleration.

Forces and Motion

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

- 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.

Forces and Motion

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

- 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.

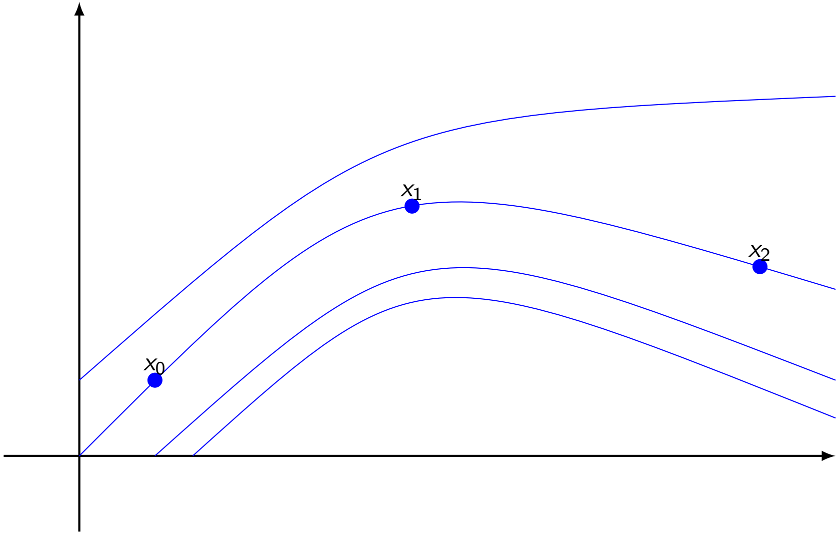
Forces and Motion

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

- 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.

The problem of Integration

There exists a vector field, and exact integration of this field would move a point along the flow lines. But exact integration is impossible in a discrete simulation.



Euler Integration

We need to find the position for a given moment in time. So we regard position as a function of time, $x(t)$. Assuming we know the position at a given time, and we can also somehow figure out the velocity at that time, $v(t)$, we find $x(t + \Delta t)$ by simply scaling the velocity and adding it to the position.

$$\begin{aligned} k(t) &= v(t)\Delta t \\ x(t + \Delta t) &= x(t) + k \end{aligned}$$

For convenience we often write the sequence of points

$$x(t), x(t + \Delta t), x(t + 2\Delta t), \ldots$$

as

$$x_0, x_1, x_2, \ldots$$

A similar operation can be used to update velocity, given acceleration.

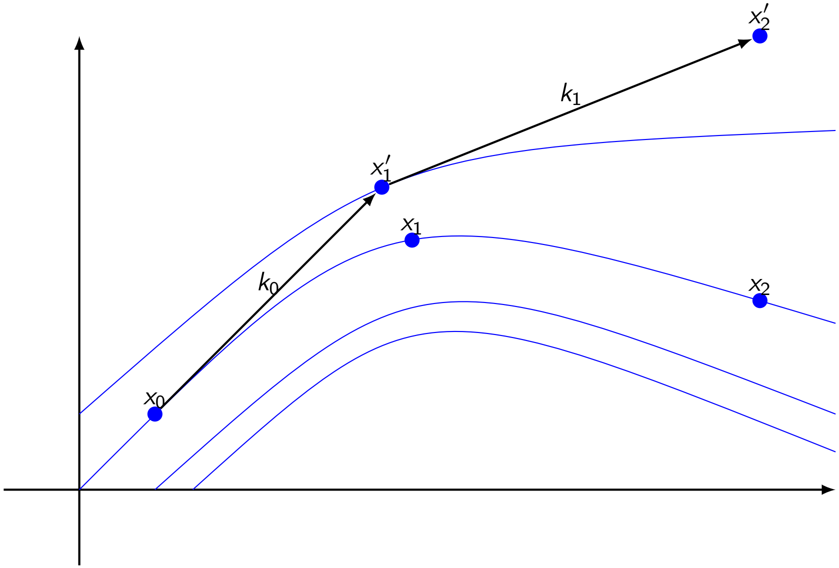
Euler Integration

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

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

- Why do we update the position before the velocity?
- Run `spring.py`

Euler Integration



Euler Calculations, $\Delta t = 1$

$$\begin{aligned}
m &= 10 \\
k &= 5 \\
f &= -kx \\
x' &= v \\
v' &= a \\
a &\leftarrow f/m = -kx/m = -x/2 \\
x &\leftarrow x + x'\Delta t = x + v\Delta t \\
v &\leftarrow v + v'\Delta t = v + a\Delta t
\end{aligned}$$

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

Euler Calculations, $\Delta t = 0.5$

$$\begin{aligned}
m &= 10 \\
k &= 5 \\
f &= -kx \\
x' &= v \\
v' &= a \\
a &\leftarrow f/m = -kx/m = -x/2 \\
x &\leftarrow x + x'\Delta t = x + v\Delta t \\
v &\leftarrow v + v' = v\Delta t + a\Delta t
\end{aligned}$$

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

Euler Calculations, $\Delta t = 0.25$

$$\begin{aligned}
m &= 10 \\
k &= 5 \\
f &= -kx \\
x' &= v \\
v' &= a \\
a &\leftarrow f/m = -kx/m = -x/2 \\
x &\leftarrow x + x'\Delta t = x + v\Delta t \\
v &\leftarrow v + v'\Delta t = v + a\Delta t
\end{aligned}$$

t	x	v	a
0.0	20.0	0.0	-10.0
0.3	20.0	-2.5	-10.0
0.5	19.4	-5.0	-9.7
0.8	18.1	-7.4	-9.1
1.0	16.3	-9.7	-8.1
1.3	13.8	-11.7	-6.9
1.5	10.9	-13.5	-5.5
1.8	7.6	-14.8	-3.8
2.0	3.9	-15.8	-1.9
2.3	-0.1	-16.2	0.0
2.5	-4.2	-16.2	2.1
2.8	-8.2	-15.7	4.1
3.0	-12.1	-14.7	6.1
3.3	-15.8	-13.2	7.9
3.5	-19.1	-11.2	9.6
3.8	-21.9	-8.8	10.9
4.0	-24.1	-6.1	12.1
4.3	-25.6	-3.1	12.8
4.5	-26.4	0.1	13.2
4.8	-26.3	3.4	13.2
5.0	-25.5	6.7	12.7

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

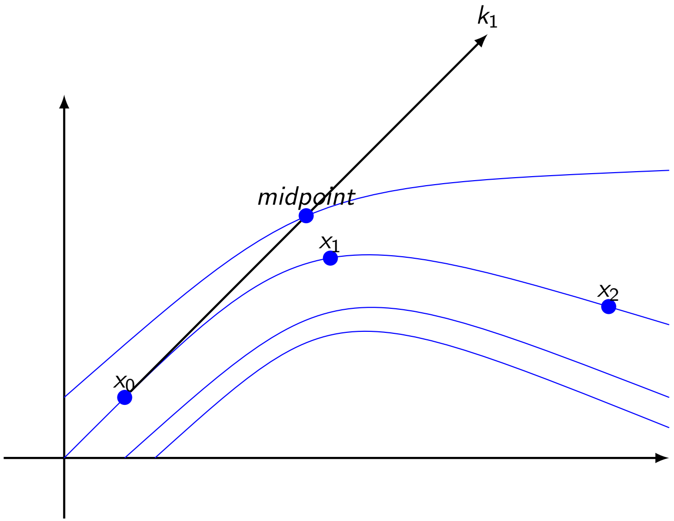
$$\begin{aligned} k_1 &= v\Delta t \\ x_{half} &= x + k_1/2 \\ k_2 &= v_{half}\Delta t \\ x &\leftarrow x + k_2 \end{aligned}$$

- 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:

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

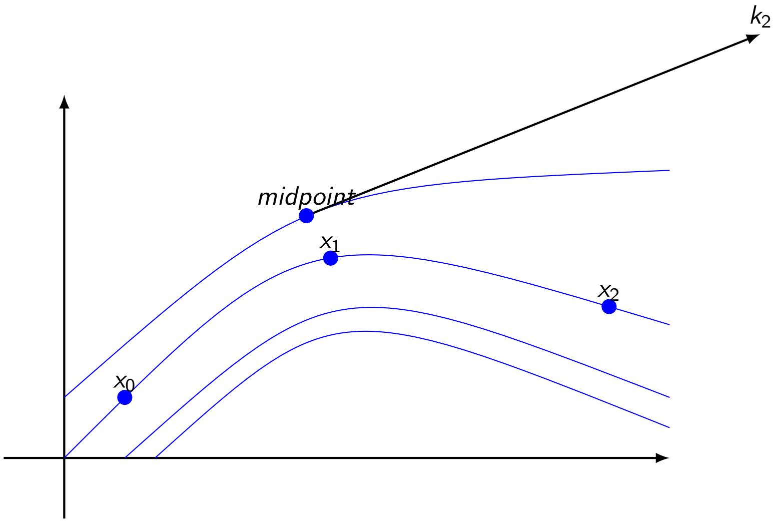
Midpoint Method

Move point halfway along vector.



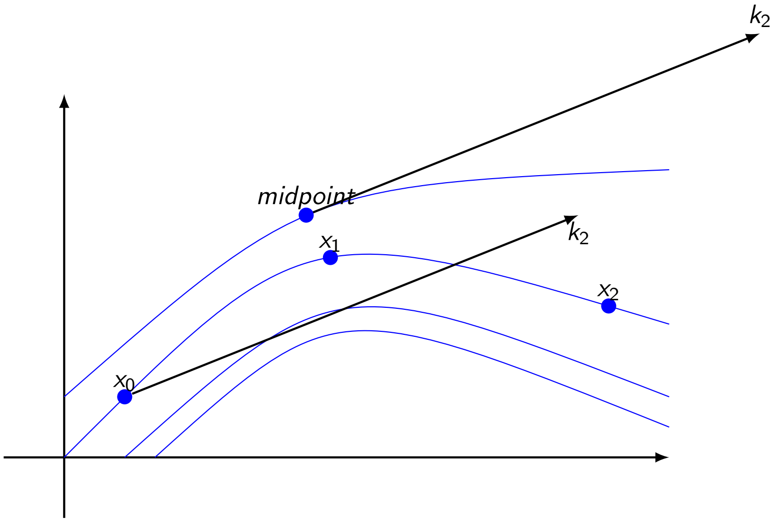
Midpoint Method

Find new derivative at halfway point.



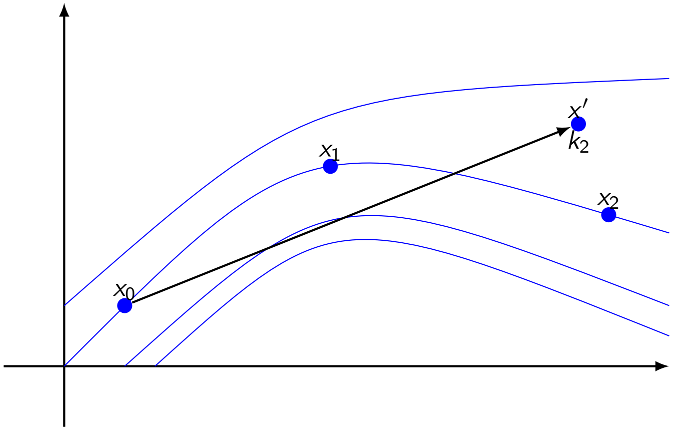
Midpoint Method

Translate new vector back to start.



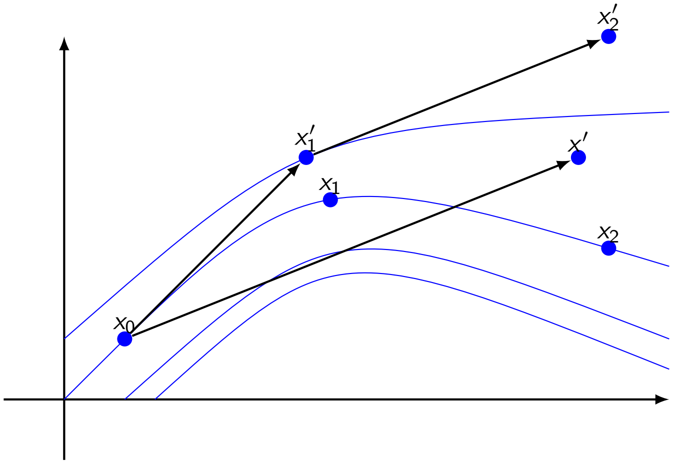
Midpoint Method

Add translated vector to start.



Midpoint Method

More accurate than Euler with same cost.



Midpoint Calculations, $\Delta t = 2$

$$\begin{aligned}
\Delta t &= 2 \\
m &= 10 \\
k &= 5 \\
f &= -kx \\
x' &= v \\
v' &= a \\
a &\leftarrow f/m = -kx/m = -x/2 \\
v_{half} &\leftarrow v + a\Delta t/2 \\
x_{half} &\leftarrow x + v\Delta t/2 \\
a_{half} &\leftarrow -x_{half}/2 \\
v &\leftarrow v + a_{half}\Delta t \\
x &\leftarrow x + v_{half}\Delta t
\end{aligned}$$

t	x	v	a
0.0	20.0	0.0	-10.0
1.0	20.0	-10.0	-10.0
2.0	0.0	-20.0	-0.0
3.0	-20.0	-20.0	10.0
4.0	-40.0	0.0	20.0
5.0	-40.0	20.0	20.0

At $t = 5$ Euler with $\Delta t = 1$ had $x = -55$.

Midpoint Calculations, $\Delta t = 1$

$$\begin{aligned}
m &= 10\\
k &= 5\\
f &= -kx\\
x' &= v\\
v' &= a\\
a &\leftarrow f/m = -kx/m = -x/2\\
v_{half} &\leftarrow v + a\Delta t/2\\
x_{half} &\leftarrow x + v\Delta t/2\\
a_{half} &\leftarrow -x_{half}/2\\
v &\leftarrow v + a_{half}\Delta t\\
x &\leftarrow x + v_{half}\Delta t
\end{aligned}$$

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
5.5	-13.8	14.5	6.9

At $t = 5$ Euler with $\Delta t = 0.5$ had $x = -34.9$.

Midpoint Calculations, $\Delta t = 0.5$

$$
\begin{aligned}
m &= 10 \\
k &= 5 \\
f &= -kx \\
x' &= v \\
v' &= a \\
a &\leftarrow f/m = -kx/m = -x/2 \\
v_{half} &\leftarrow v + a\Delta t/2 \\
x_{half} &\leftarrow x + v\Delta t/2 \\
a_{half} &\leftarrow -x_{half}/2 \\
v &\leftarrow v + a_{half}\Delta t \\
x &\leftarrow x + v_{half}\Delta t
\end{aligned}
$$

t	x	v	a
0.0	20.0	0.0	-10.0
0.3	20.0	-2.5	-10.0
0.5	18.8	-5.0	-9.4
0.8	17.5	-7.3	-8.8
1.0	15.1	-9.4	-7.5
1.3	12.7	-11.3	-6.4
1.5	9.4	-12.6	-4.7
1.8	6.3	-13.7	-3.2
2.0	2.6	-14.1	-1.3
2.3	-1.0	-14.5	0.5
2.5	-4.7	-13.9	2.3
2.8	-8.1	-13.3	4.1
3.0	-11.3	-11.9	5.7
3.3	-14.3	-10.5	7.1
3.5	-16.5	-8.3	8.3
3.8	-18.6	-6.2	9.3
4.0	-19.6	-3.6	9.8
4.3	-20.6	-1.2	10.3
4.5	-20.2	1.5	10.1
4.8	-19.9	4.0	9.9
5.0	-18.2	6.5	9.1
5.3	-16.6	8.7	8.3

At $t = 5$ Euler with $\Delta t = 0.25$ had $x = -25.5$.

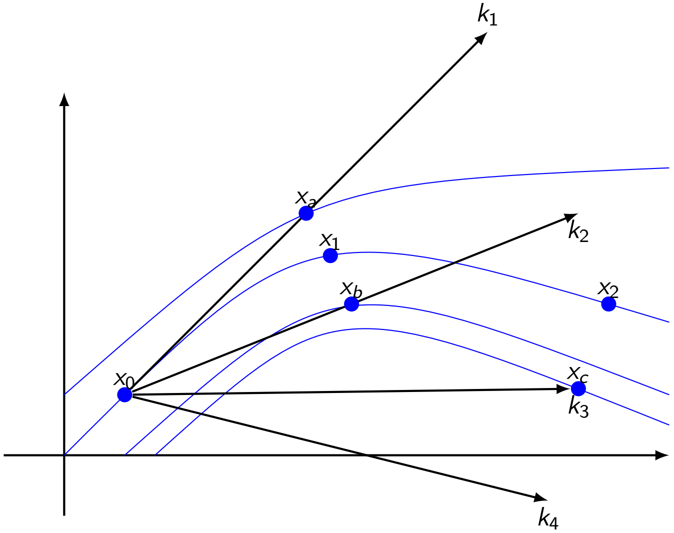
Fourth Order Runge-Kutta

$$\begin{aligned}
k_1 &= v\Delta t \\
x_a &= x + k_1/2 \\
k_2 &= v_a\Delta t \\
x_b &= x + k_2/2 \\
k_3 &= v_b\Delta t \\
x_c &= x + k_3 \\
k_4 &= v_c\Delta t
\end{aligned}$$

$$x \leftarrow x + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}$$

Fourth order Runge Kutta

Final vector used: $\frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}$



Fourth Order Runge Kutta Calculations, $\Delta t = 4$

$$\begin{aligned} m &= 10 \\ k &= 5 \\ f &= -kx \\ x' &= v \\ v' &= a \\ a &\leftarrow f/m = -kx/m = -x/2 \end{aligned}$$

t	x	v	a
0.0	20.0	0.0	-10.0
2.0	20.0	-20.0	-10.0
2.0	-20.0	-20.0	10.0
4.0	-60.0	40.0	30.0
4.0	2.2	13.3	-1.1
6.0	28.9	11.1	-14.4
6.0	24.4	-15.6	-12.2
8.0	-60.0	-35.6	30.0
8.0	-29.4	-3.0	14.7
10.0	-35.3	26.4	17.7
10.0	23.5	32.3	-11.7
12.0	100.0	-49.9	-50.0
12.0	3.3	-18.6	-1.7

Fourth Order Runge Kutta Calculations, $\Delta t = 2$

$$
\begin{aligned}
m &= 10 \\
k &= 5 \\
f &= -kx \\
x' &= v \\
v' &= a \\
a &\leftarrow f/m = -kx/m = -x/2
\end{aligned}
$$

t	x	v	a
0.0	20.0	0.0	-10.0
1.0	20.0	-10.0	-10.0
1.0	10.0	-10.0	-5.0
2.0	0.0	-10.0	-0.0
2.0	-1.1	-13.3	0.6
3.0	-14.4	-12.8	7.2
3.0	-13.9	-6.1	6.9
4.0	-13.3	0.6	6.7
4.0	-14.0	-1.5	7.0
5.0	-15.5	5.5	7.7
5.0	-8.5	6.3	4.2
6.0	-1.5	7.0	0.7
6.0	-0.8	9.1	0.4
7.0	8.3	9.5	-4.2
7.0	8.7	4.9	-4.4
8.0	9.1	0.4	-4.5
8.0	9.6	2.0	-4.8

Fourth Order Runge Kutta Calculations, $\Delta t = 1$

$$\begin{aligned} m &= 10 \\ k &= 5 \\ f &= -kx \\ x' &= v \\ v' &= a \\ a &\leftarrow f/m = -kx/m = -x/2 \end{aligned}$$

t	x	v	a
0.0	20.0	0.0	-10.0
0.5	20.0	-5.0	-10.0
0.5	17.5	-5.0	-8.8
1.0	15.0	-8.8	-7.5
1.0	13.7	-9.2	-6.8
1.5	9.1	-12.6	-4.5
1.5	7.4	-11.4	-3.7
2.0	2.2	-12.9	-1.1
2.0	1.3	-13.2	-0.7
2.5	-5.3	-13.6	2.6
2.5	-5.5	-11.9	2.7
3.0	-10.6	-10.5	5.3
3.0	-10.7	-10.7	5.4
3.5	-16.0	-8.0	8.0
3.5	-14.7	-6.7	7.4
4.0	-17.4	-3.3	8.7
4.0	-16.7	-3.2	8.3
4.5	-18.3	1.0	9.1
4.5	-16.2	1.4	8.1
5.0	-15.3	4.9	7.7
5.0	-14.2	5.2	7.1
5.5	-11.6	8.8	5.8
5.5	-9.8	8.1	4.9
6.0	-6.1	10.1	3.1
6.0	-5.2	10.5	2.6

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-nearing-collapse`

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.

Comparing Euler and Symplectic Euler, $\Delta t = 1$

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
6.0	-57.5	25.0	28.8
7.0	-32.5	53.8	16.3
8.0	21.3	70.0	-10.6
9.0	91.3	59.4	-45.6
10.0	150.6	13.8	-75.3
11.0	164.4	-61.6	-82.2
12.0	102.8	-143.8	-51.4
13.0	-40.9	-195.2	20.5
14.0	-236.1	-174.7	118.0
15.0	-410.8	-56.6	205.4
16.0	-467.4	148.8	233.7
17.0	-318.7	382.5	159.3

Symplectic Euler

t	x	v	a
0.0	20.0	0.0	-10.0
1.0	10.0	-10.0	-5.0
2.0	-5.0	-15.0	2.5
3.0	-17.5	-12.5	8.8
4.0	-21.3	-3.8	10.6
5.0	-14.4	6.9	7.2
6.0	-0.3	14.1	0.2
7.0	13.9	14.2	-7.0
8.0	21.2	7.3	-10.6
9.0	17.9	-3.3	-8.9
10.0	5.6	-12.2	-2.8
11.0	-9.4	-15.0	4.7
12.0	-19.8	-10.3	9.9
13.0	-20.2	-0.4	10.1
14.0	-10.5	9.7	5.3
15.0	4.4	14.9	-2.2
16.0	17.1	12.7	-8.6
17.0	21.3	4.2	-10.7

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.

Comparing Symplectic Euler and Verlet, $\Delta t = 1$

Symplectic Euler

t	x	v	a
0.0	20.0	0.0	-10.0
1.0	10.0	-10.0	-5.0
2.0	-5.0	-15.0	2.5
3.0	-17.5	-12.5	8.8
4.0	-21.3	-3.8	10.6
5.0	-14.4	6.9	7.2
6.0	-0.3	14.1	0.2
7.0	13.9	14.2	-7.0
8.0	21.2	7.3	-10.6
9.0	17.9	-3.3	-8.9
10.0	5.6	-12.2	-2.8
11.0	-9.4	-15.0	4.7
12.0	-19.8	-10.3	9.9
13.0	-20.2	-0.4	10.1
14.0	-10.5	9.7	5.3
15.0	4.4	14.9	-2.2
16.0	17.1	12.7	-8.6
17.0	21.3	4.2	-10.7

Velocityless Verlet

t	x	v	a
0.0	20.0	0.0	-10.0
1.0	10.0	-10.0	-5.0
2.0	-5.0	-15.0	2.5
3.0	-17.5	-12.5	8.8
4.0	-21.3	-3.8	10.6
5.0	-14.4	6.9	7.2
6.0	-0.3	14.1	0.2
7.0	13.9	14.2	-7.0
8.0	21.2	7.3	-10.6
9.0	17.9	-3.3	-8.9
10.0	5.6	-12.2	-2.8
11.0	-9.4	-15.0	4.7
12.0	-19.8	-10.3	9.9
13.0	-20.2	-0.4	10.1
14.0	-10.5	9.7	5.3
15.0	4.4	14.9	-2.2
16.0	17.1	12.7	-8.6
17.0	21.3	4.2	-10.7

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-game-physics-r2714`

- A nice web demo:

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

- 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`