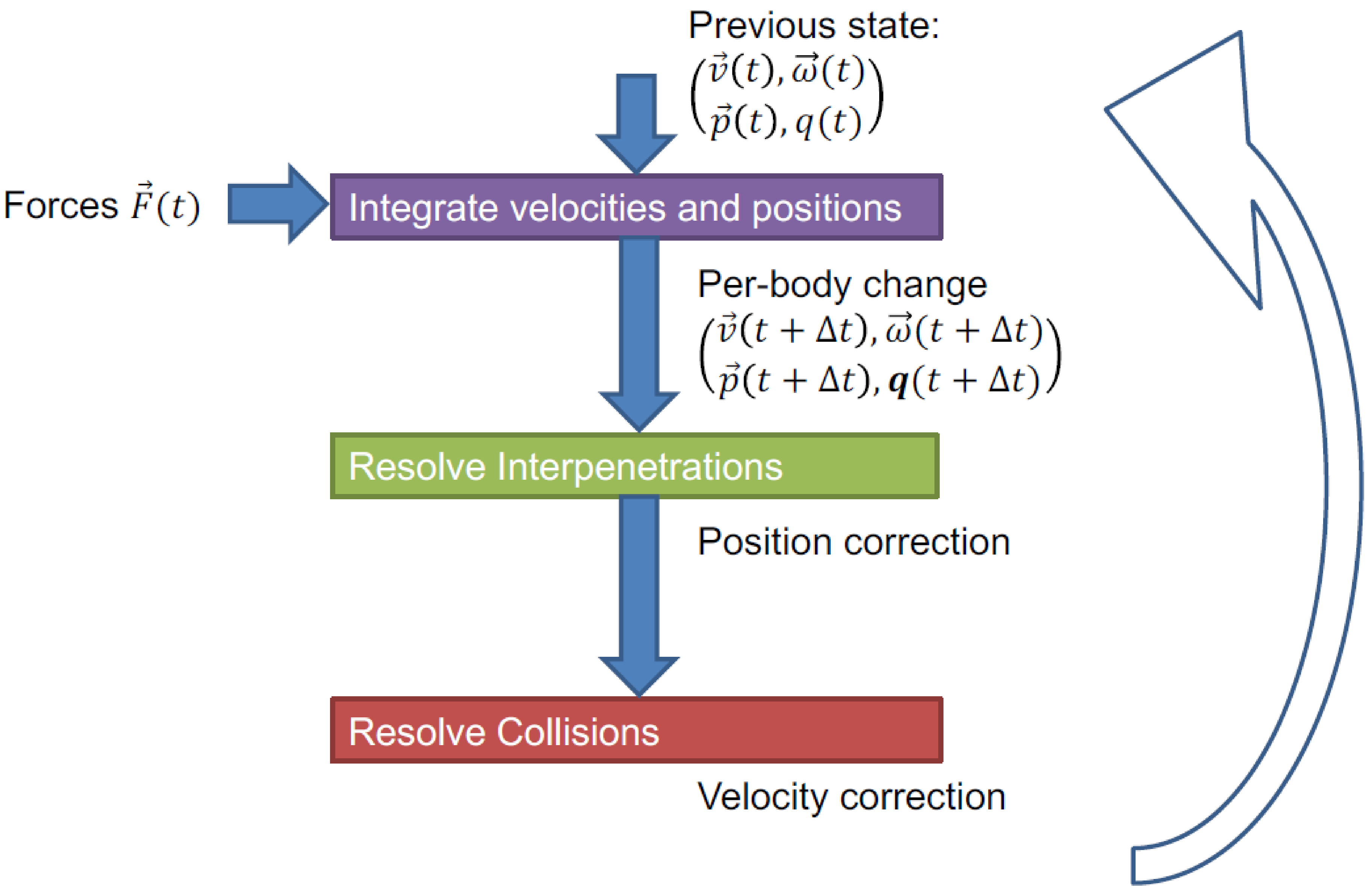
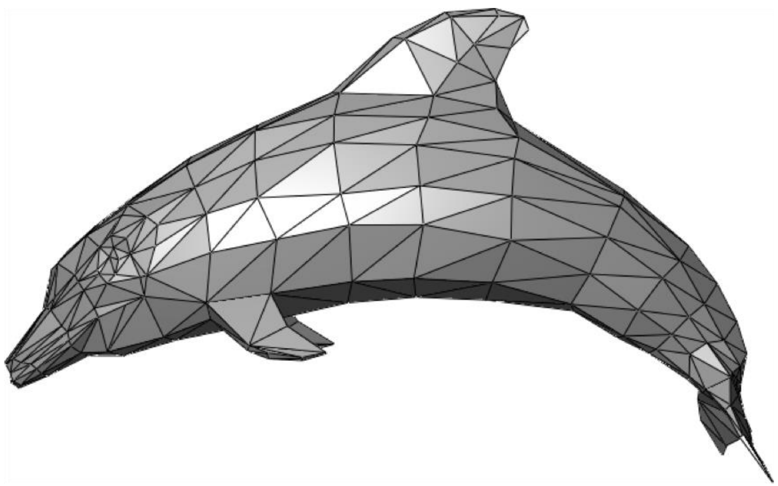
The game-engine loop & Time Integration

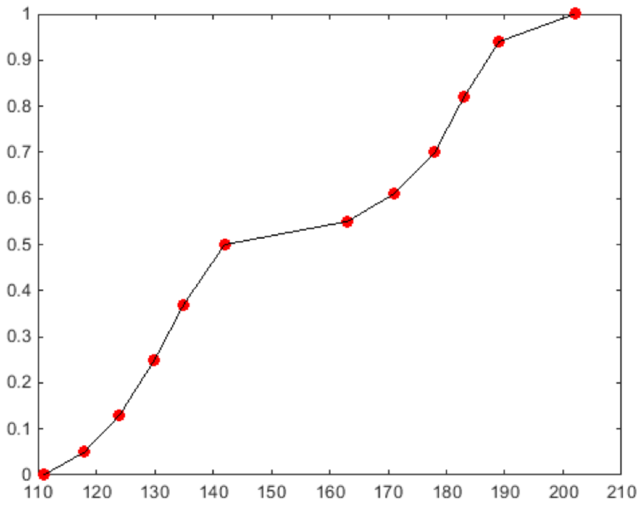
The Basic Game-Engine Loop



Challenges

- Kinematics: continuous motion in continuous time.
  - Events are local and always valid.
- Computer simulation:
  - Discrete time steps Δt.
  - Discrete Space (mesh, particles, grids)





Updating Position

- Force induces acceleration.
- When mass is constant:

$$F(p,t) = m \cdot a(p,t)$$

- Derivatives: $v'(t) = a(t)$ and $p'(t) = v(t)$
- Thus: $F(p,t) = m \cdot p''(t)$

- A differential equation.
  - Often impossible to solve analytically.
  - More often, $a = a(v,t)$ (velocity dependence).
- Discretization: stability and convergence issues.

Taylor Approximation

- A function at $t + \Delta t$ can be approximated by a polynomial centered at t with arbitrary precision:

$$p(t + \Delta t)$$
$$\approx p(t) + \Delta t \cdot p'(t) + \frac{\Delta t^2}{2} p''(t) + \cdots + \frac{\Delta t^n}{n!} p^{(n)}(t)$$

- We do not usually use (or have) more than 2nd derivatives.

First-Order Approximation

- If Δt is small enough, we approximate linearly:

$$p(t + \Delta t) \approx p(t) + \Delta t . p'(t)$$

- Euler's Method: approximating forward both velocity and position within the same step:

$$v(t + \Delta t) = v(t) + a(t)\Delta t = v(t) + \frac{F(t)}{m}\Delta t$$

$$p(t + \Delta t) = p(t) + v(t)\Delta t$$

Unknown for next time step

Assumed known for this time step

Euler’s Method

- **Note:** we approximate the velocity as constant between frames.
  - We compute the acceleration of the object from the net force applied on it:

$$a(t) = \frac{F(t)}{m}$$

  - We compute the velocity from the acceleration:

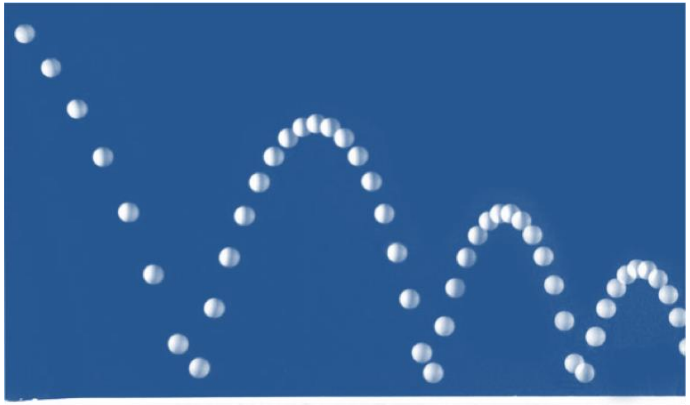
$$v(t + \Delta t) = v(t) + a(t)\Delta t$$

  - We compute the position from the velocity:

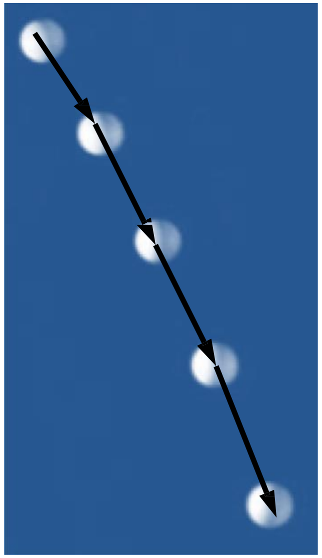
$$p(t + \Delta t) = p(t) + v(t)\Delta t$$

Issues with Linear Dynamics

- A mere sequence of instants.
  - Without the precise instant of bouncing.



- Trajectories are piecewise-linear.
  - Constant velocity and acceleration in-between frames.



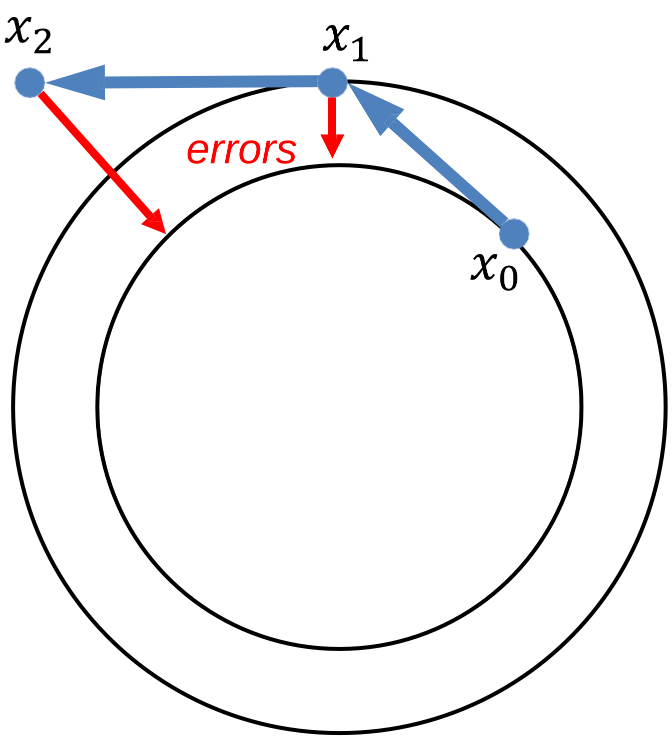
Time Step

When $\Delta t \rightarrow 0$, we converge to $p(t) = \int_0^t v(s)ds$

- (Im)possible solution: reducing Δt to convergence levels?
- First-order method, piecewise-constant velocity: not very stable.
- Our objective: make the most with every Δt you get.

Time Step

- First-order assumption: the slope at t as a good estimate for the slope over the entire interval Δt
- The approximation can drift off the function.
- Farther drifting ⇔ tangent approximation worse.



Time step

```
void takeStep(ParticleSystem* ps, float h)
{
        velocities = ps->getStateVelocities()
        positions = ps->getStatePositions()
        forces = ps->getForces(positions, velocities)
        masses = ps->getMasses()
        accelerations = forces / masses
        newPositions = positions + h*velocities
        newVelocities = velocities  + h*accelerations
        ps->setStatePositions(newPositions)
        ps->setStateVelocities(newVelocities)
}
```

Midpoint Method

- Estimating tangent in Half step:

$$v(t+\frac{\Delta t}{2}) = v(t) + a(t,v)\frac{\Delta t}{2}$$

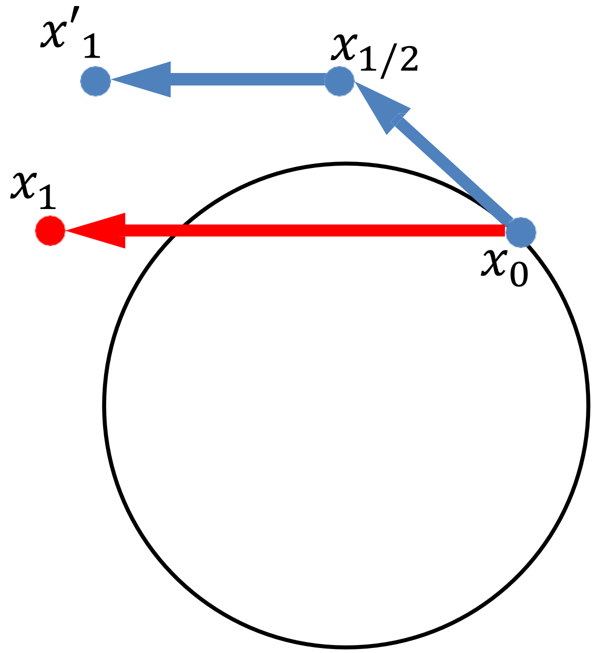
- Full step:

$$v(t+\Delta t) = v(t) + a(t+\frac{\Delta t}{2}, t+\frac{\Delta t}{2})\Delta t$$

- 2nd-order approximation.
- Compute position similarly with v.

Midpoint Method

- Approximating the tangent in mid-interval.
- Applying it to initial point across the entire interval.
- Error order: the square of the time step $O(\Delta t^2)$. Better than Euler's method ($O(\Delta t)$) when $\Delta t < 1$.
- Approximating with a quadratic curve instead of a line.
- …can still drift off the function.



Midpoint Method

```
void takeStep(ParticleSystem* ps, float h)
{
        velocities = ps->getStateVelocities()
        positions = ps->getStatePositions()
        forces = ps->getForces(positions, velocities)
        masses = ps->getMasses()
        accelerations = forces / masses
        midPositions = positions + 0.5*h*velocities
        midVelocities = velocities  + 0.5*h*accelerations
        midForces = ps->getForces(midPositions, midVelocities)
        midAccelerations = midForces / masses
        newPositions = positions + h*midVelocities
        newVelocities = velocities  + h*midAccelerations
        ps->setStatePositions(newPositions)
        ps->setStateVelocities(newVelocities)
}
```

Improved Euler's Method

- Considers the tangent lines to the solution curve at both ends of the interval.
- Velocity to the first point (Euler's prediction):

$$v_1 = v(t) + \Delta t. a(t, v)$$

- Velocity to the second point (correction point):

$$v_2 = v(t) + \Delta t. a(t + \Delta t, v_1)$$

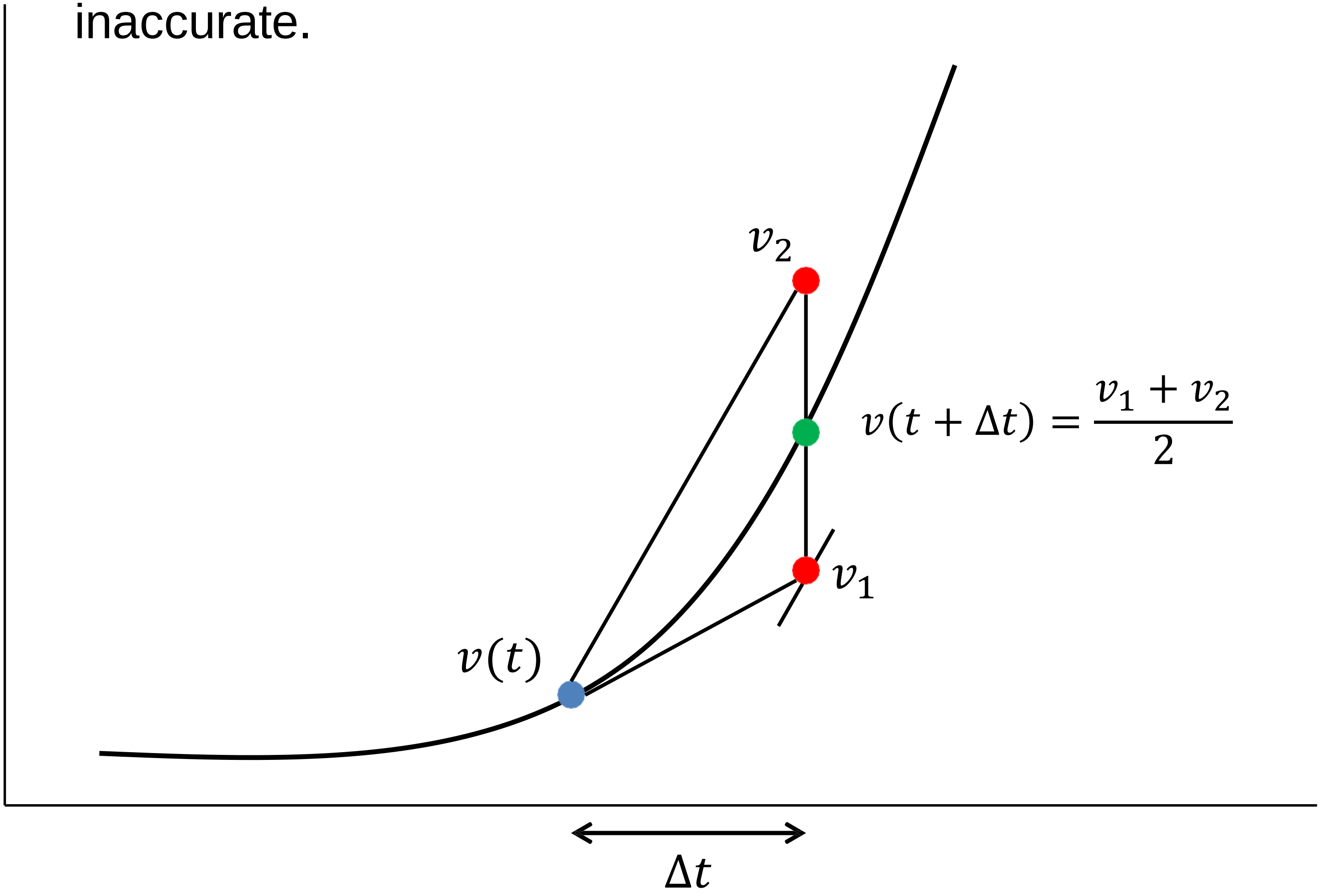
- Improved Euler's velocity

$$v(t + \Delta t) = \frac{v_1 + v_2}{2}$$

- Compute position similarly with $v_1, v_{2,}$ instead of a.

Improved Euler's Method

The order of the error is $O(\Delta t^2)$. The final derivative is still inaccurate.



Runge-Kutta Method

- There are methods that provide better than quadratic error.

- The Runge-Kutta order-four method (RK4) is $O(\Delta t^4)$.

- A combination of the midpoint and modified Euler's methods, with higher weights to the midpoint tangents than to the endpoints tangents.

RK4

- Computing the four following tangents (note dependence of acceleration on velocity):

$$v_1 = \Delta t \cdot a(t, v(t))$$

$$v_2 = \Delta t \cdot a\left(t + \frac{\Delta t}{2}, v(t) + \frac{1}{2} v_1\right)$$

$$v_3 = \Delta t \cdot a\left(t + \frac{\Delta t}{2}, v(t) + \frac{1}{2} v_2\right)$$

$$v_4 = \Delta t \cdot a\big(t + \Delta t, v(t) + v_3\big)$$

- Blend as follows:

$$v(t + \Delta t) = v(t) + \frac{v_1 + 2v_2 + 2v_3 + v_4}{6}$$

- Compute position similarly with v values.

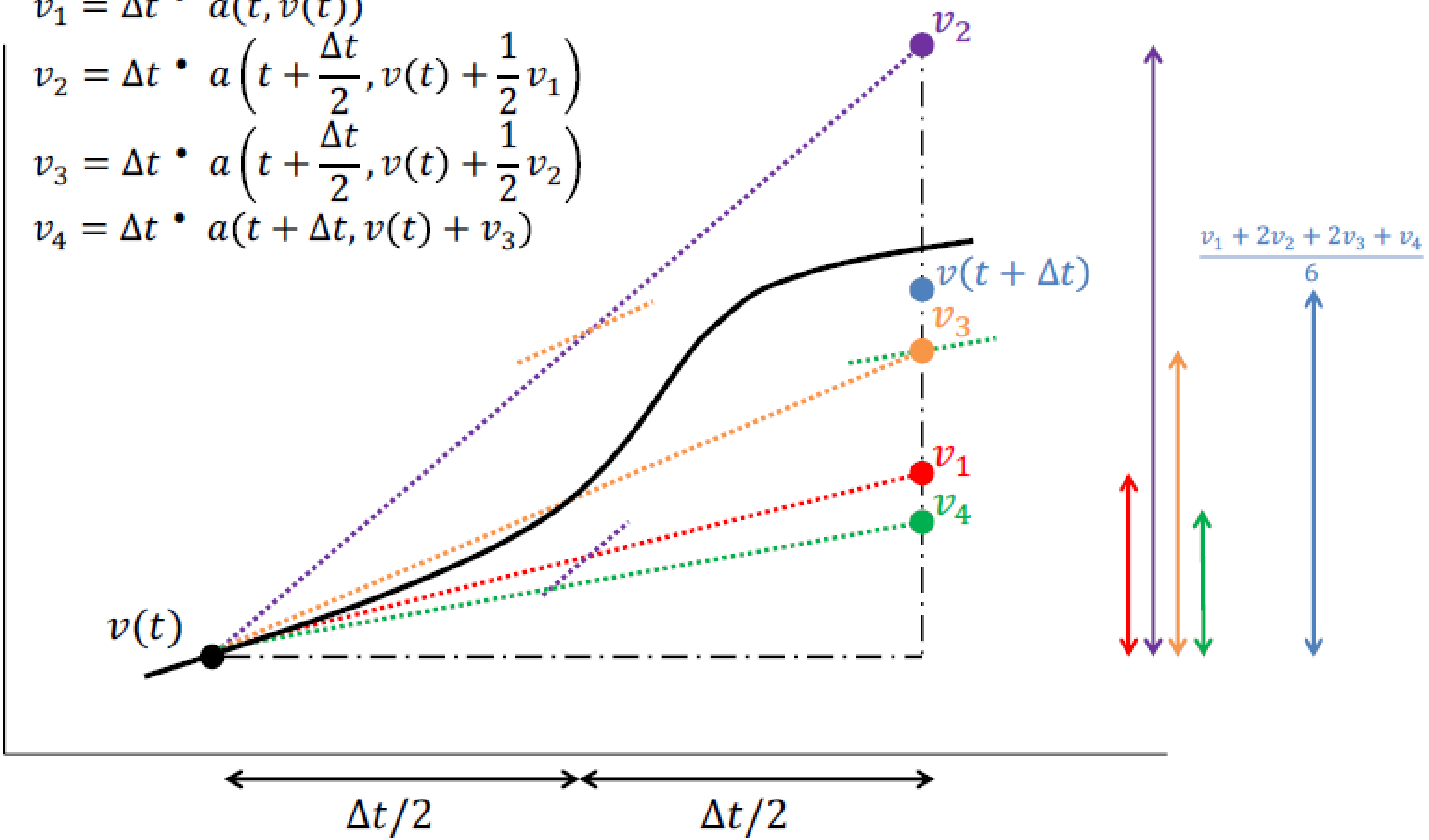
RK4

$$v_1 = \Delta t \cdot a(t, v(t))$$

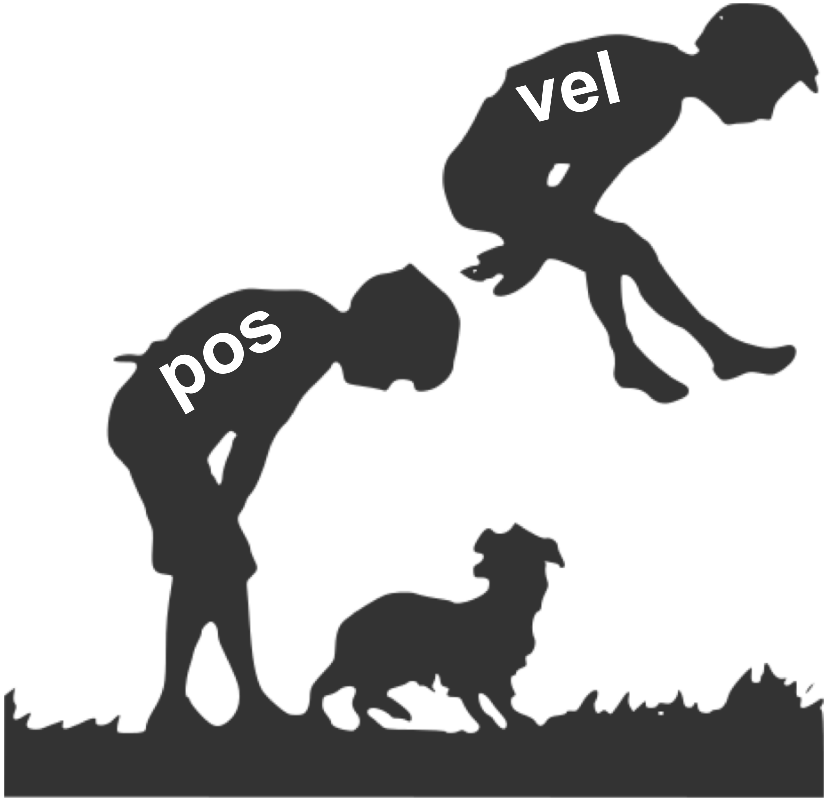
$$v_2 = \Delta t \cdot a\left(t + \frac{\Delta t}{2}, v(t) + \frac{1}{2} v_1\right)$$

$$v_3 = \Delta t \cdot a\left(t + \frac{\Delta t}{2}, v(t) + \frac{1}{2} v_2\right)$$

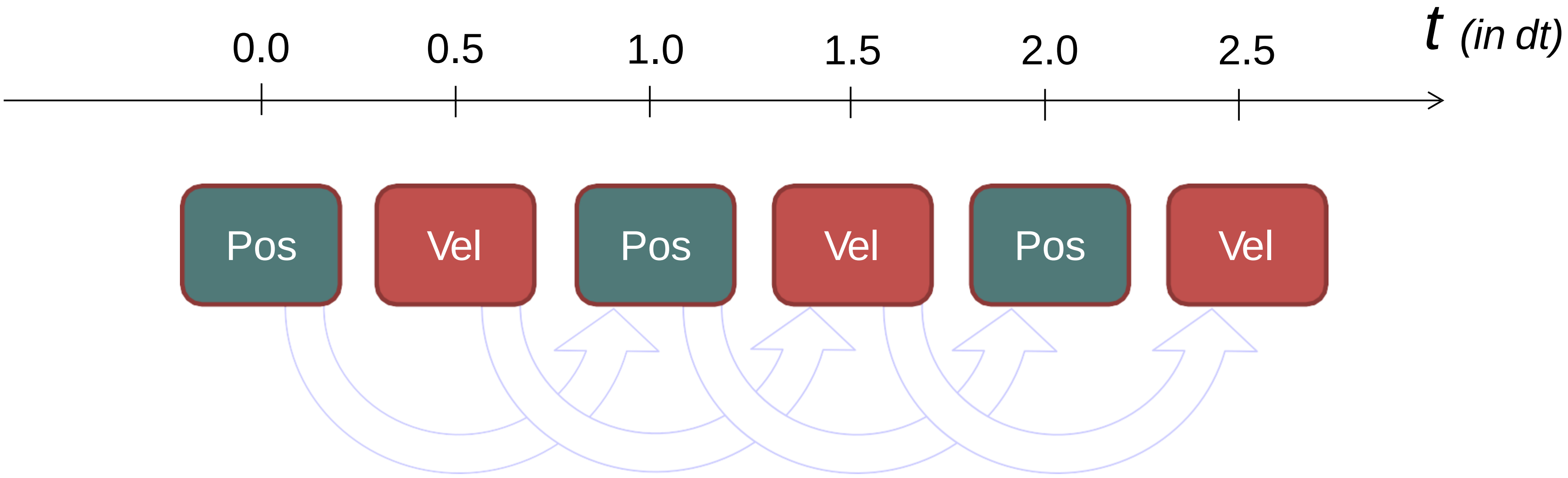
$$v_4 = \Delta t \cdot a(t + \Delta t, v(t) + v_3)$$



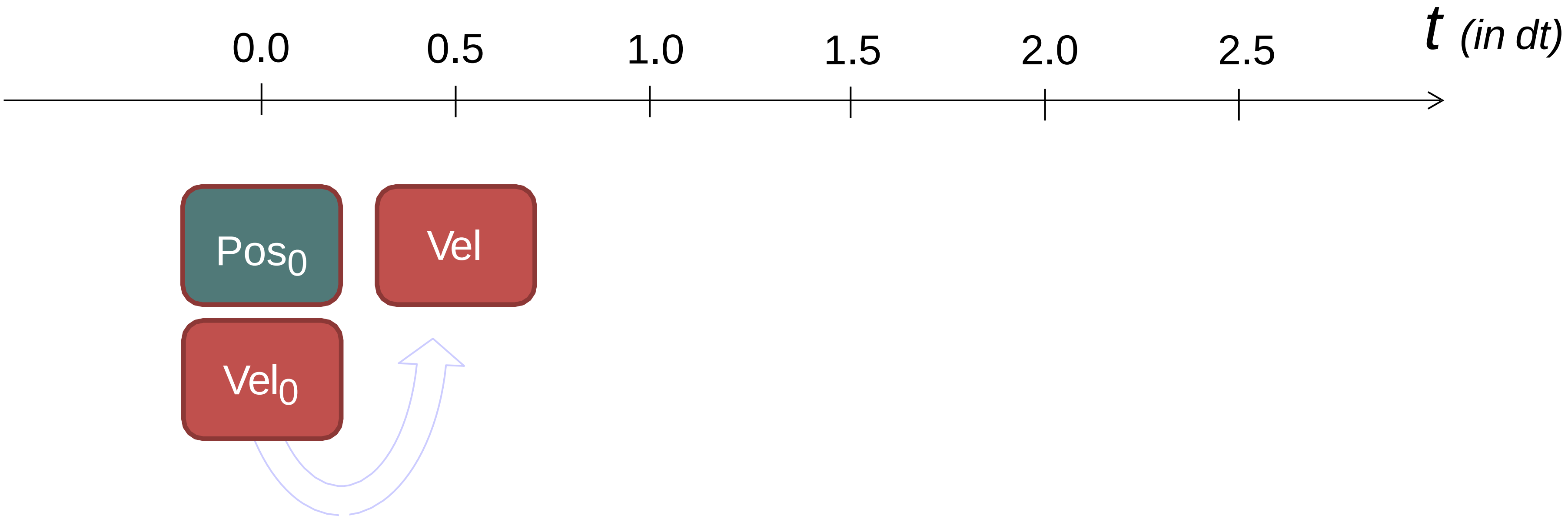
Leapfrog Integration ("a cavallina")



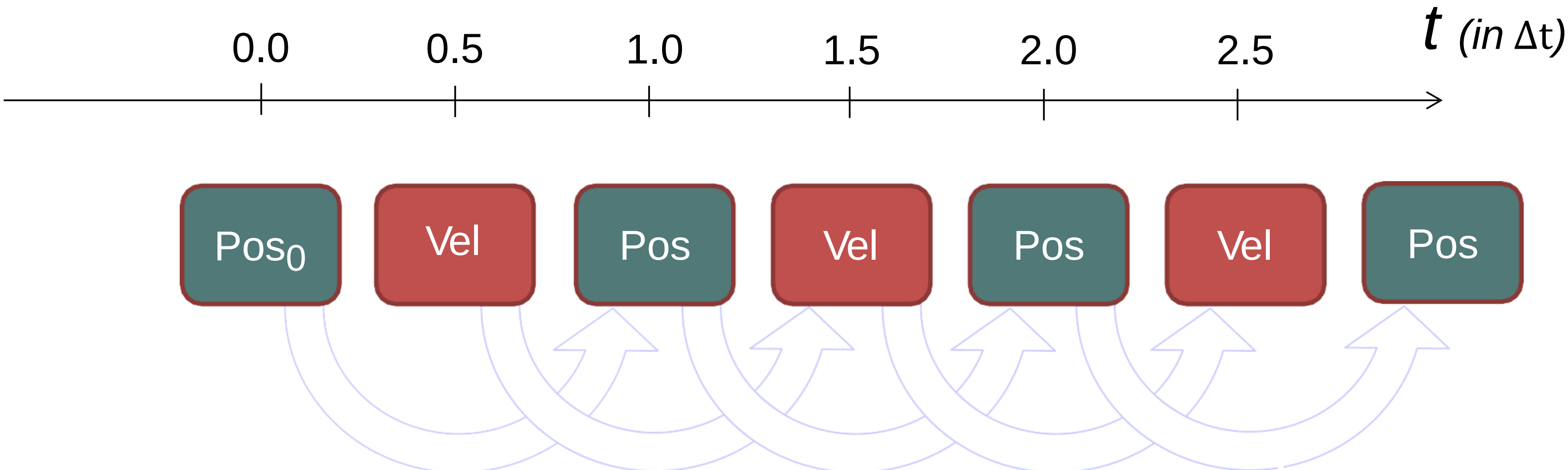
Leapfrog Integration



Leapfrog Integration
first step



Leapfrog Integration



$$\vec{p}(1) = \vec{p}(0) + \vec{v}(0.5)\Delta t$$
$$\vec{p}(2) = \vec{p}(1) + \vec{v}(1.5)\Delta t$$
$$\vec{p}(3) = \vec{p}(2) + \vec{v}(2.5)\Delta t$$

$$\vec{v}(1.5) = \vec{v}(0.5) + \vec{a}\Delta t$$
$$\vec{v}(2.5) = \vec{v}(1.5) + \vec{a}\Delta t$$

Leapfrog Method

- More accurate than Euler-based methods
  - Residue of $O(\Delta t^3)$
- But at the same cost as Euler's method!
- Major advantage: fully reversible!

Verlet integration

- Based on the Taylor expansion series of the previous time step and the next one:

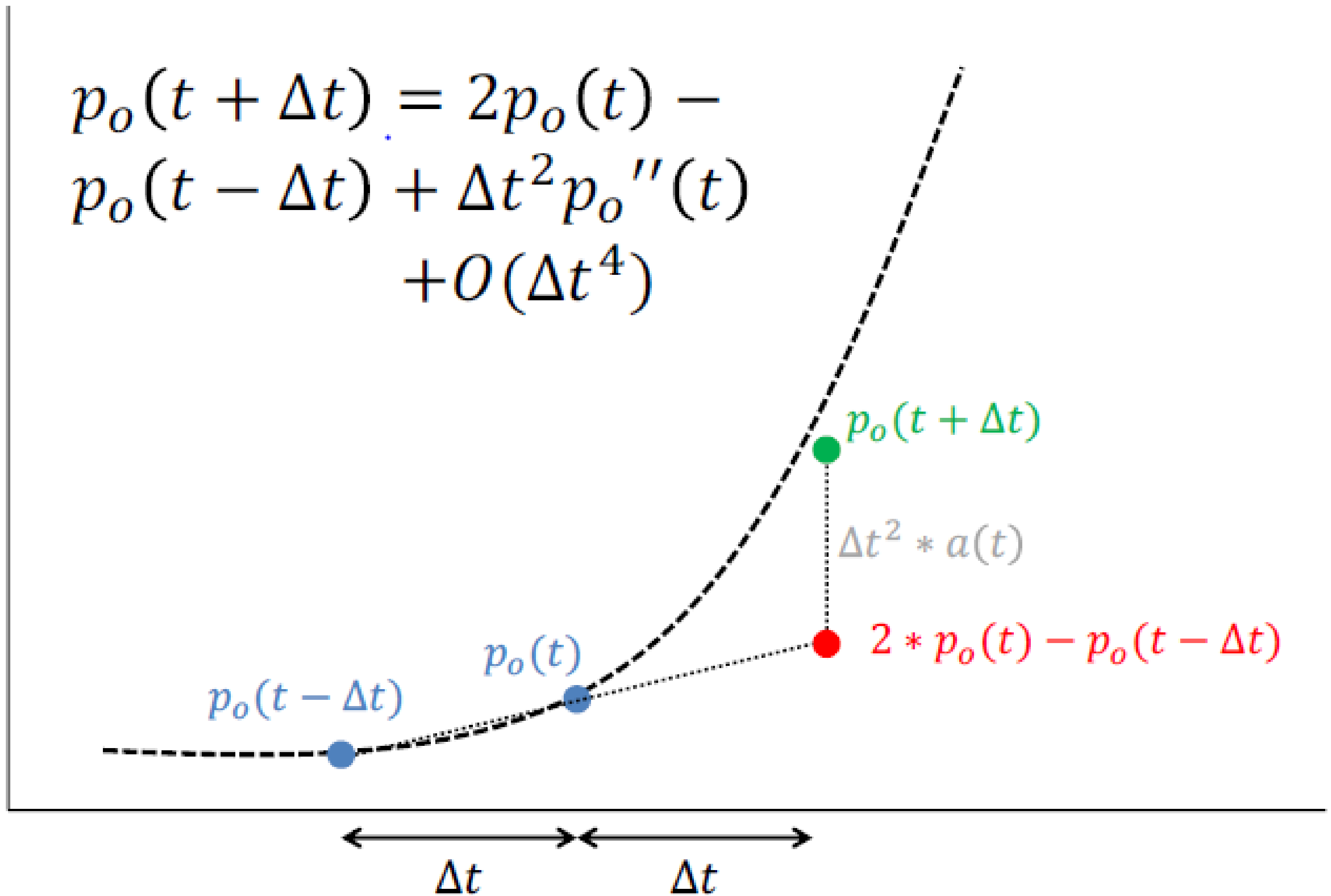
$$
\begin{aligned}
&p_o(t+\Delta t) + p_o(t-\Delta t) \\
&\approx p_o(t) \boxed{+\,\Delta t \cdot p_o{}'(t)} + \frac{\Delta t^2}{2} \cdot p_o{}''(t) + \cdots \\
&+ p_o(t) \boxed{-\,\Delta t \cdot p_o{}'(t)} + \frac{\Delta t^2}{2} * p_o{}''\,(t) - \cdots
\end{aligned}
$$

Cancels out!

Verlet integration

- Approximating without velocity:

$$p_o(t + \Delta t) = 2p_o(t) - p_o(t - \Delta t) + \Delta t^2 p_o''(t) + O(\Delta t^4)$$



Verlet integration

- An $O(\Delta t^2)$ order of error.
- Very stable and fast without the need to estimate velocities.
- We need an estimation of the first $p_0 = (t - \Delta t)$
  - Usually obtained from one step of Euler's or RK4 method.
- Difficult to manage velocity related forces such as drag or collision.

Implicit methods

- So far: computing current position $p(\mathrm{t})$ and velocity $v(t)$ for the next position (forward).
  - Those are denoted as explicit methods.
- In implicit methods, we make use of the quantities from the next time step!

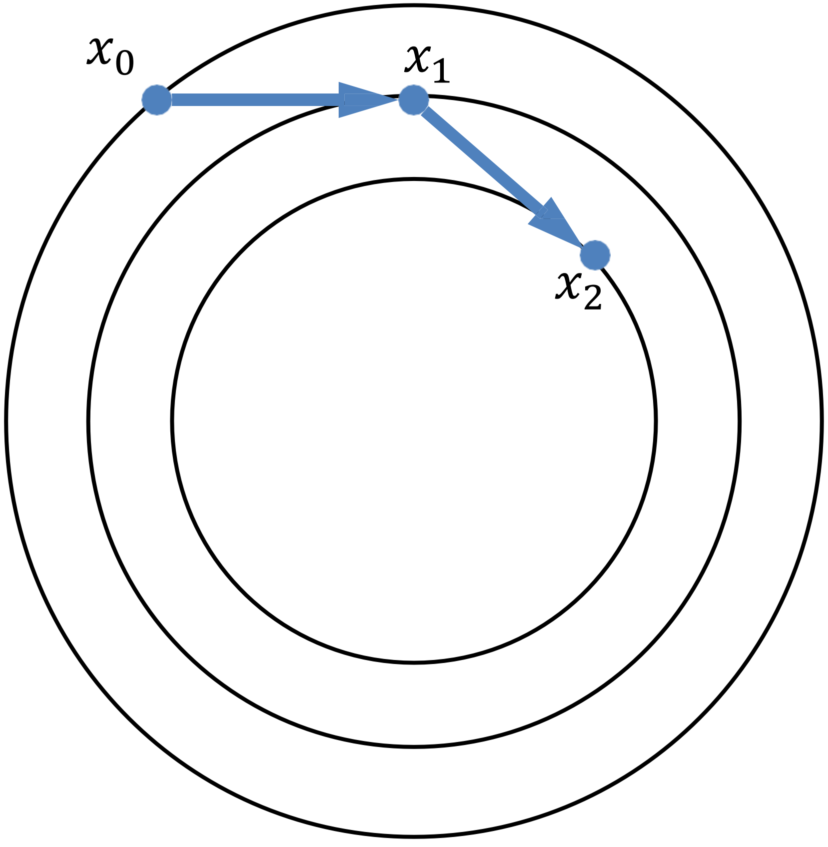
$$p(t) = p(t + \Delta t) - \Delta t . v(t + \Delta t)$$

  - This particular one: backward Euler.
- Computing in inverse:
- Finding position $p(t + \Delta t)$ which produces $p(t)$ if simulation is run backwards.

Implicit methods

- Not more accurate than explicit methods, but more stable.

- Especially for a damping of the position (*e.g.* drag force or kinetic friction).



Backward Euler

- How to compute the velocity from the future?

- Given the forces applied, extracting from the formula:
  - Example: a drag force $\mathrm{F_D} = -b.v$ is applied:

$$\frac{v(t+\Delta t) - v(t)}{\Delta t} = -b \cdot v(t+\Delta t)$$

  - And therefore

$$v(t+\Delta t) = \frac{v(t)}{1+\Delta t.b}$$

Backward Euler

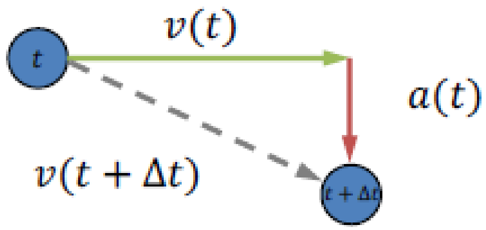
- Often not knowing the forces in advance (likely case in a game).
- Or that the backward equation is not easy to solve.
- We use a predictor-corrector method:
    - one step of explicit Euler's method
    - use the predicted position to calculate $v(t + \Delta t)$
- More accurate than explicit method but twice the amount of computation.

Semi-Implicit Method

- Combines the simplicity of explicit Euler and stability of implicit Euler.
- Runs an explicit Euler step for velocity and then an implicit Euler step for position:

$$v(t+\Delta t) = v(t) + \Delta t * a(t) = v(t) + \Delta t * F(t)/m$$

$$p(t+\Delta t) = p(t) + \Delta t * \cancel{v(t)} = p(t) + \Delta t * v(t+\Delta t)$$



Semi-Implicit Method

- The position update in the second step uses the next velocity in the implicit scheme.
  - Good for position-dependent forces.
  - Conserves energy over time, and thus stable.

- Not as accurate as RK4 (order of error is still $O(\Delta t)$), but cheaper and yet stable.

- Very popular choice for game physics engine.

Angular Integration

- Examples: Forward Euler

$$\frac{dq}{dt} = \frac{1}{2}(0, \vec{\omega})q$$

becomes:

$$\frac{q(t + \Delta t) - q(t)}{\Delta t} = \frac{1}{2}(0, \vec{\omega}(t))q(t) \Rightarrow$$
$$q(t + \Delta t) = q(t) + \frac{1}{2}\Delta t(0, \vec{\omega}(t))q(t)$$

- The rest of the formulations follow suit for angular displacement θ, velocity ω and acceleration α.

Summary

- First-order methods
  - Implicit and Explicit Euler method, Semi-implicit Euler, Exponential Euler
- Second-order methods
  - Verlet integration, Velocity Verlet, Trapezoidal rule, Beeman’s algorithm, Midpoint method, Improved Euler’s method, Heun’s method, Newmark-beta method, Leapfrog integration.
- Higher-order methods
  - Runge-Kutta family methods, Linear multistep method.
- Position-based methods
  - Leapfrog, Verlet.