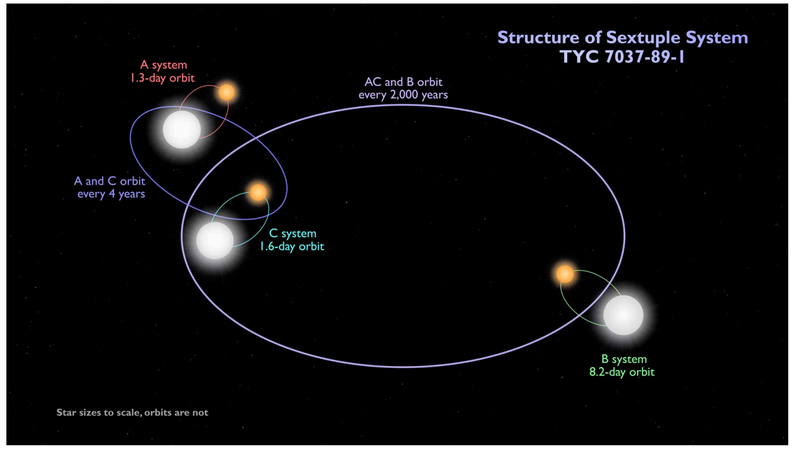
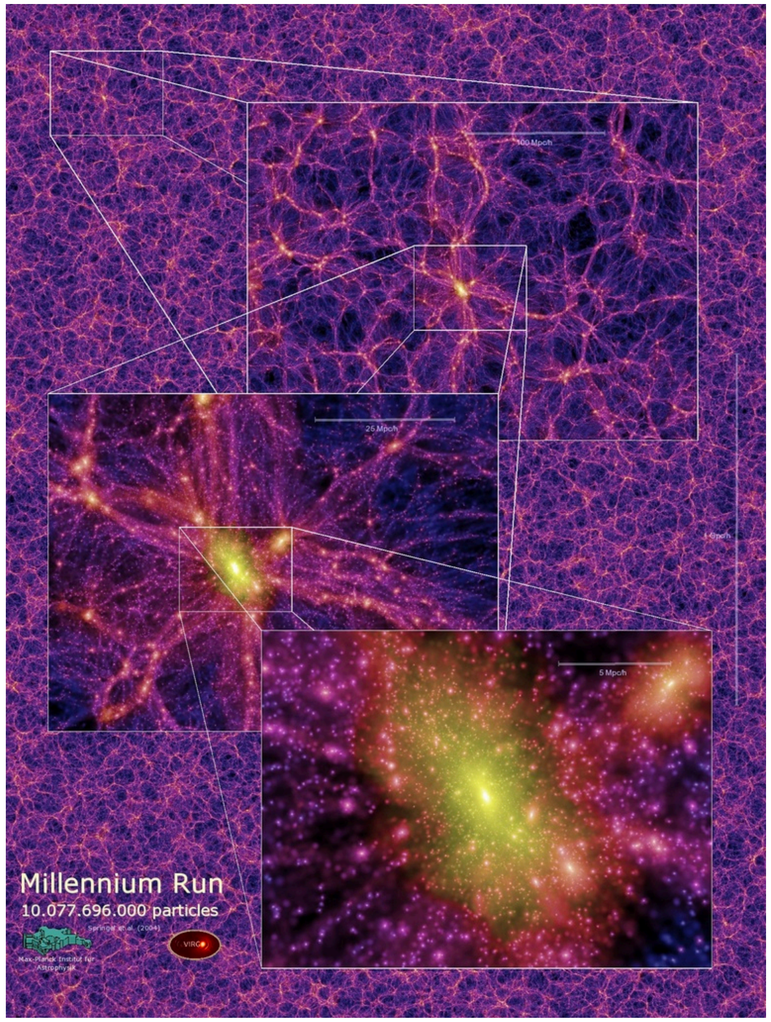
ASTR 660 / Class 3

ODE solvers / N-body problem

The N-Body Problem





$$\mathbf{F}_{ij} = -\frac{Gm_i m_j}{|r_{ij}|^2}\hat{\mathbf{r}}_{ij}$$

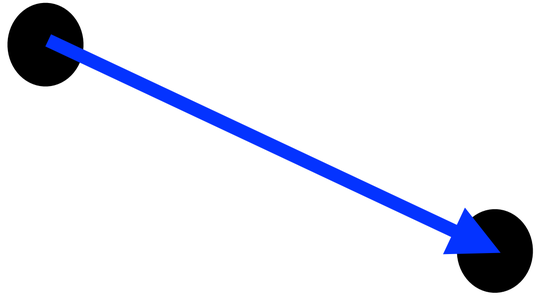
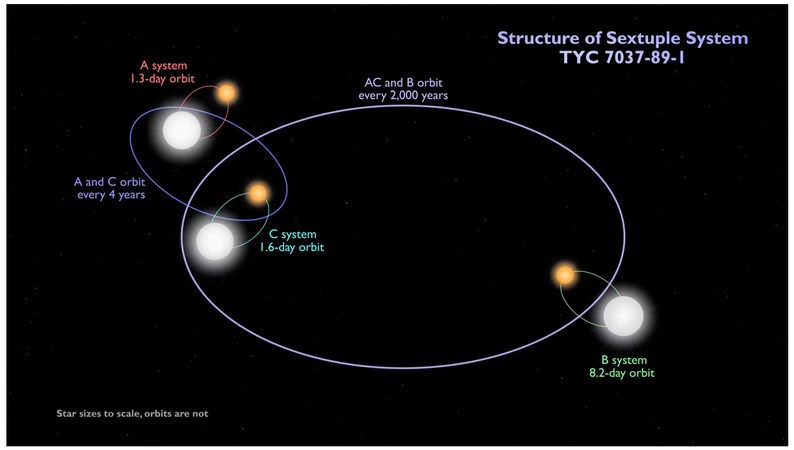
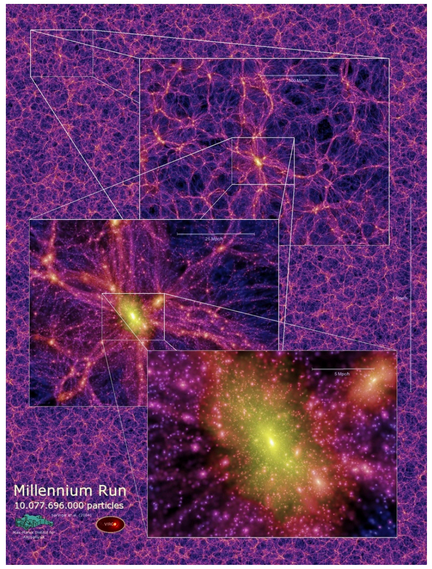


Image credits: NASA; Virgo Consortium

The N-Body Problem





Images: NASA; Virgo Consortium; B. Robertson, L. Hernquist

"**Collisional**": short-range, **two-body** interactions (not actual collisions!) are important.

Each particle typically represents an individual star / planets / asteroid.

"**Collisionless**": particles in the simulation represent elements of a continuous distribution of mass.

"Microscopic" two body interactions are unimportant; two-body interactions between the *simulated* "macroscopic" particles are unphysical.

The N-Body Problem

$\mathbf{F}_{ij} = -\dfrac{Gm_i m_j}{|r_{ij}|^2}\hat{\mathbf{r}}_{ij} \implies \mathbf{a}_{ij} = -\dfrac{Gm_i}{|r_{ij}|^2}\hat{\mathbf{r}}_{ij}$, acceleration of mass m_j due to the gravitational potential of m_i.

Since $\hat{\mathbf{r}}_{ij} = \dfrac{\mathbf{r}_{ij}}{|r_{ij}|}$,

$$\mathbf{a}_{ij} = -\frac{Gm_i}{|r_{ij}|^3}\mathbf{r}_{ij}$$

Equivalent to a system of 1st-order ODEs: $\dfrac{d\mathbf{r}}{dt} = \mathbf{v}(\mathbf{r}, t)$ and $\dfrac{d\mathbf{v}}{dt} = \mathbf{a}(\mathbf{r}, t)$.

Initial Value Problems

The solution of an ODE describes how the state of the system evolves as the independent variable (in physics, usually *time*) changes.

Analytic ODE systems can be solved through integration, so we often refer to the numerical methods for solving ODEs as "integrators". Although they have some mathematical connection, for practical purposes they are not the same as the "numerical integration methods" we will look at later!

ODEs only tell us about rates of change (slopes). The **initial conditions** need to be specified, hence the name "initial value problem".

Given some starting conditions, what happens next?

Initial Value Problems

In the N-body case, the ODE describes the path that a particle follows (i.e. its **position** as a function of time) as it moves along its **velocity** vector.

The velocity vector also changes with time, in response to an **acceleration** that depends on the instantaneous positions of all the other particles.

The system therefore consists of $2N$ ODEs. The initial conditions that need to be specified are ($\mathbf{x}$,$\mathbf{v}$) for every particle.

N-Body Problem

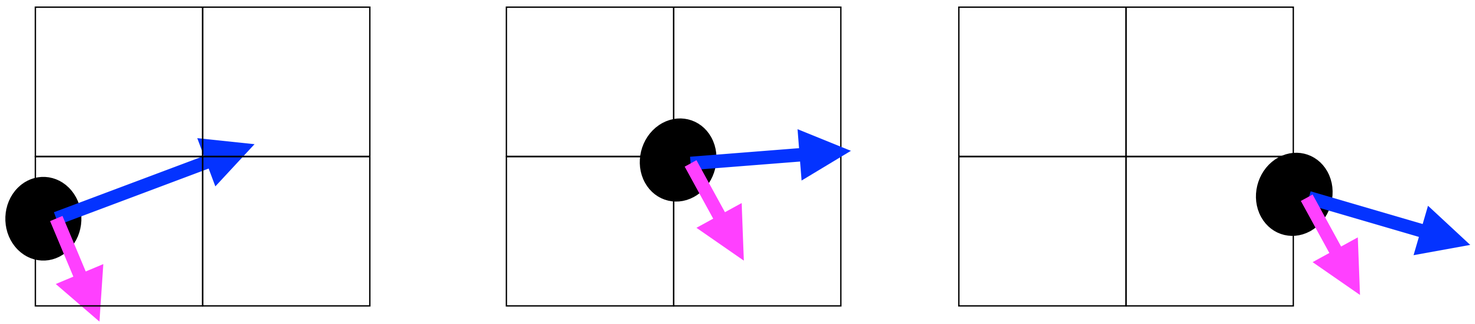
In practice, a numerical solution to the N-body problem involves taking many small steps in time, Δt. At each step, we:

1. *Calculate instantaneous accelerations for each particle at the start of the step;*

2. *Compute the velocity of the particle at the end of the step, updating the initial velocity to account for the change (acceleration) across the step;*

3. *Compute the position of the particle at the end of the step, updating the initial position to account for the change (velocity) across the step.*

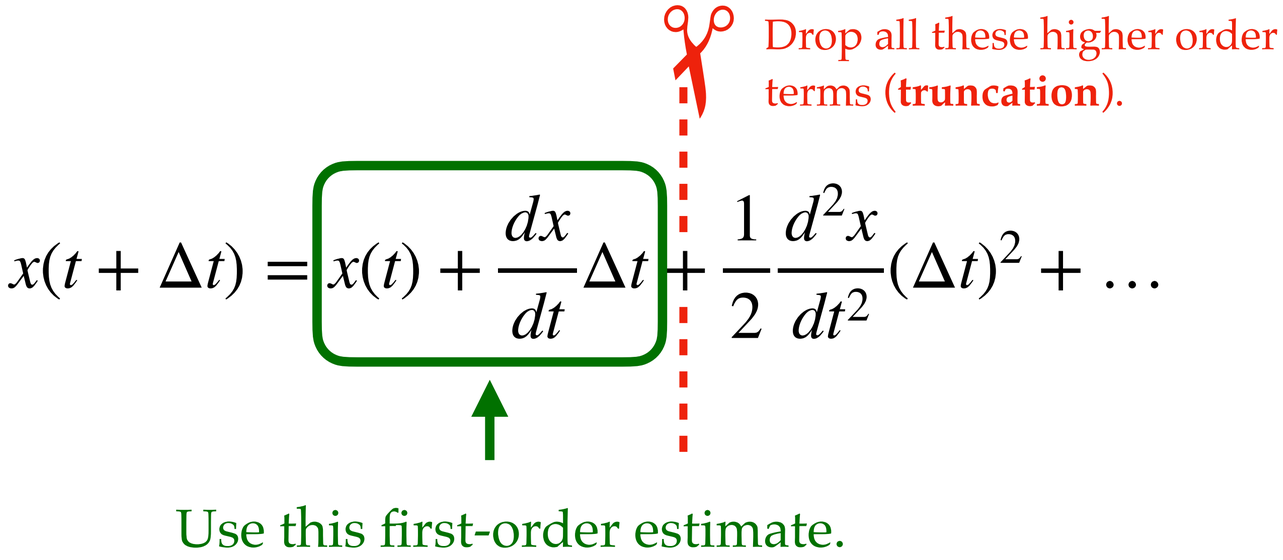
The acceleration, velocity and position of each particle are supposed to be changing smoothly in response to the motion of all the other particles, but this 'algorithm' only considers a finite number of discrete steps.



The Forward Euler Method

The most basic ODE solver: easy to understand, but **horribly inaccurate**. "Forward" means the method uses only information about the state of the system at the *start* of each timestep.

The method can be thought of as a **truncated** Taylor expansion of $x(t)$, although there are many other ways to justify it (see textbooks):



The Forward Euler Method

For example, let's solve $\frac{dx}{dt} = x$. The analytic solution is $x(t) = Ce^t$.

The forward Euler step function is $x(t + \Delta t) = x_0 + x_0 \Delta t$.

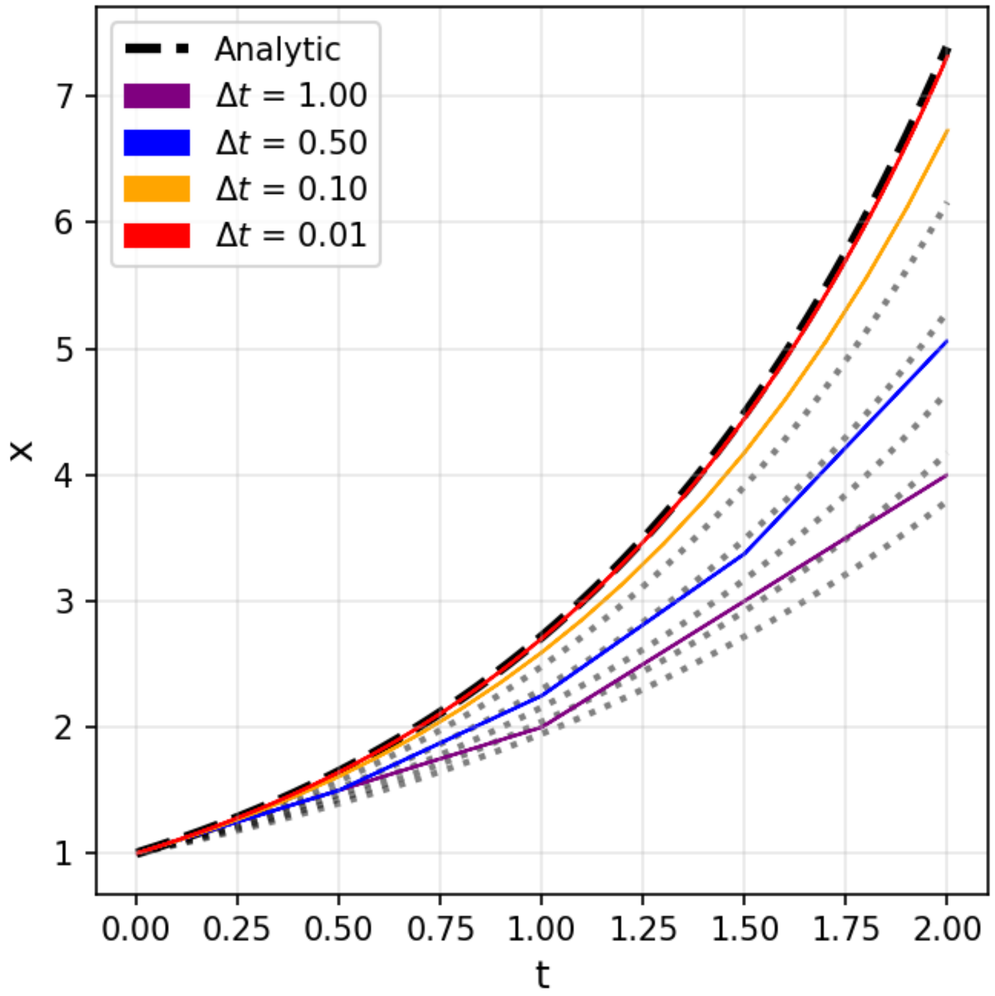
$$\frac{dx}{dt}\Big|_{x=x_0}$$

```
def euler_step(x0,dt):
    """
    """
    x = x0 + x0*dt
    return x
```

```
def solve_euler(x0,dt,nsteps      ):
    """
    """
    x = x0

    snapshots = [x]
    for i in range(0,nsteps):
        x = euler_step(x,dt)
        snapshots.append(x)

    return np.array(snapshots)
```



Error Analysis

The solution on the previous slide diverges due to **numerical error**. For each step, there are two sources of error:

- **Round-off** (the answer is only computed to a finite number of decimal places; depends on the numerical representation used).
- **Truncation** (we ignored the higher-order terms in the Taylor series). Each step ‘lands’ on a different member of the family of solutions, drifting further and further away from the correct one.

In most practical cases, unless the number of steps is very large, the truncation error dominates. This suggests two obvious ways to improve the accuracy of the calculation:

- Use smaller timesteps (brute force);
- Use higher-order terms in the expansion.

Aside: Stability

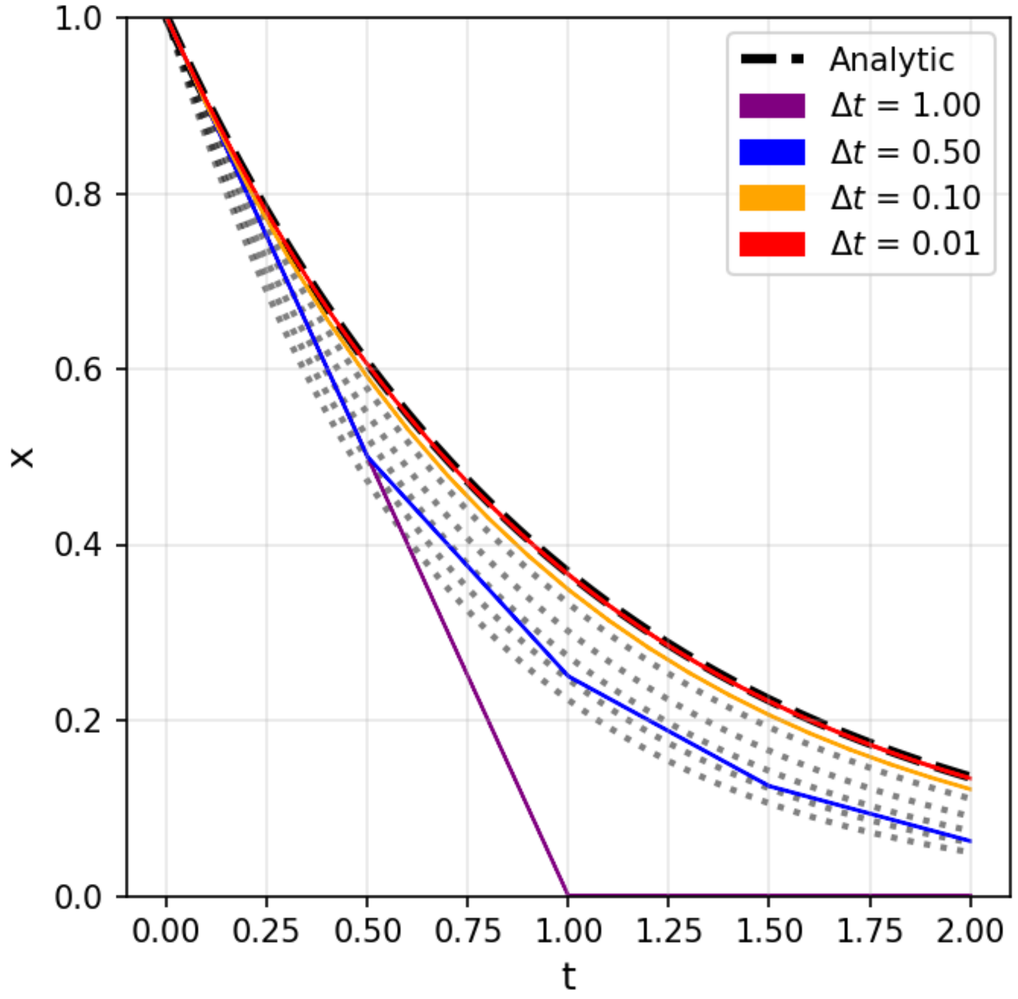
Not all ODE solutions diverge due to numerical error. This depends on the ODE system, the integrator and the initial conditions!

For example, let's solve $\frac{dx}{dt} = -x$. The analytic solution is $x(t) = Ce^{-t}$.

The forward Euler step function is $x(t + \Delta t) = x_0 - x_0 \Delta t$.

```
def euler_step(x0,dt):
    """
    """
    x = x0 - x0*dt
    return x
```

The solution converges for large t (i.e. asymptotically), even though the individual steps are as inaccurate as those the previous example.



Error Analysis

The error made on each step (i.e. “local” or “truncation” error) can be estimated. Compare the Euler estimate with the full Taylor expansion:

$$E_{\text{step}} \equiv x_{\text{true}} - x_{\text{est}} = \left[x_0 + \Delta t x_0' + \frac{\Delta t^2}{2} x_0'' + O(\Delta t^3) \right] - \left[x_0 + \Delta t x_0' \right] = \frac{\Delta t^2}{2} + O(\Delta t^3)$$

Assuming nothing crazy happens with the higher derivatives, the error per step scales $\propto \Delta t^2$.

If we advance the system up to time T, we need $T/\Delta t$ steps, so (intuitively?) we expect the accumulated total error to scale as $\frac{T}{\Delta t} \Delta t^2 \propto \Delta t$.

The forward Euler method is said to be **first order**, because the accumulated error scales linearly with the step size.

See textbooks for a more rigorous approach to this. The actual accumulated error is only bounded by this argument, and even then under assumptions about the higher derivatives. If the steps are too large, the accumulated error can “blow up”, diverging from the true solution.

Alternative Methods

There is a zoo of alternative integration methods, designed to optimise for some combination of

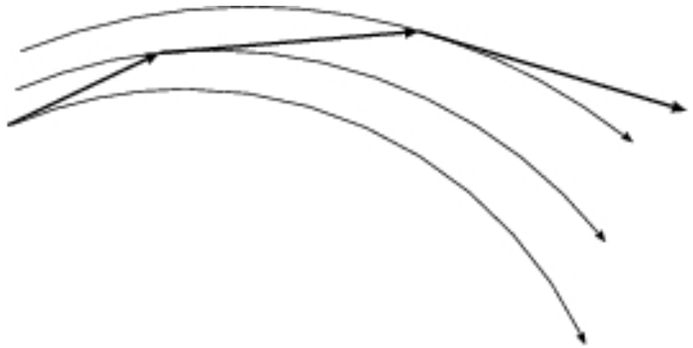
- High accuracy;
- High stability (larger step sizes can be taken without the solutions diverging);
- Low computational work.

The last point is obviously important: even the Euler method can be accurate enough if the step size can be made arbitrarily small, but such calculations will take forever.

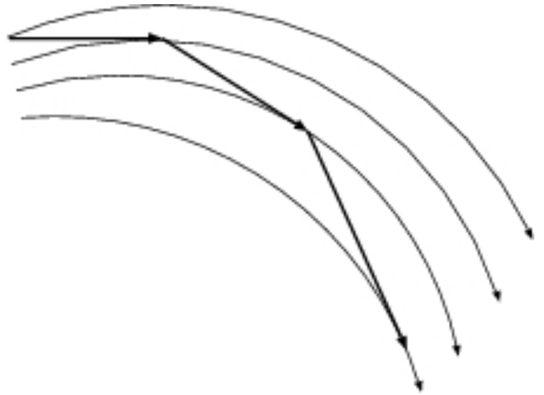
Even if a method requires more calculations per step, it can “win” over another if the result is that many fewer steps can be taken to achieve the same accuracy.

Backwards Euler

One alternative to forwards Euler is backwards Euler!



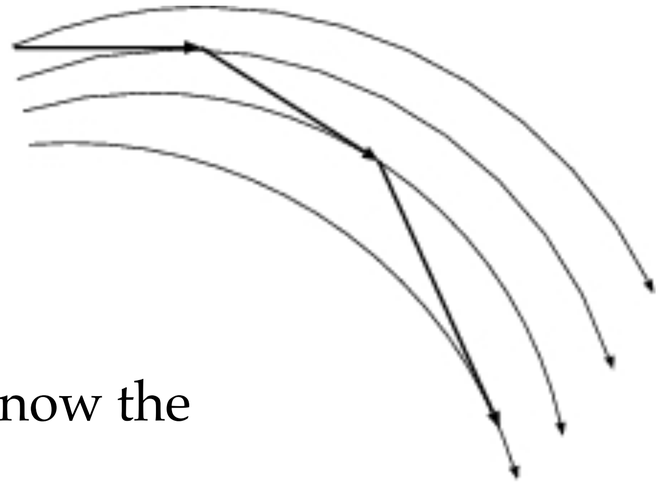
Forwards: right slope at the start, wrong slope at the end.



Backwards: right slope at the end, wrong slope at the start.

Graphics: Hut & Makino

Backwards Euler



Hut & Makino

$$x_1 = x_0 + \Delta t\, f(t_1, x_1)$$

Unlike the forwards method, this is **implicit**: we don't immediately know the value of $f(t_1, x_1)$ because x_1 appears on both sides of the equation.

We can solve for it by **iteration**, but that is lot more computational work. Also backwards Euler is still first-order accurate! Why bother?

The positive feature of backwards Euler is that it is more **stable**. The solution does not diverge for larger step sizes (for a broader range of ODEs).

Semi-Implict Euler (N-body)

In the N-body problem (a Hamiltonian system), we have the Euler step functions:

$$v_1 = v_0 + \Delta t\, a_0$$
$$x_1 = x_0 + \Delta t\, v_0$$

The **semi-implicit** Euler method instead uses:

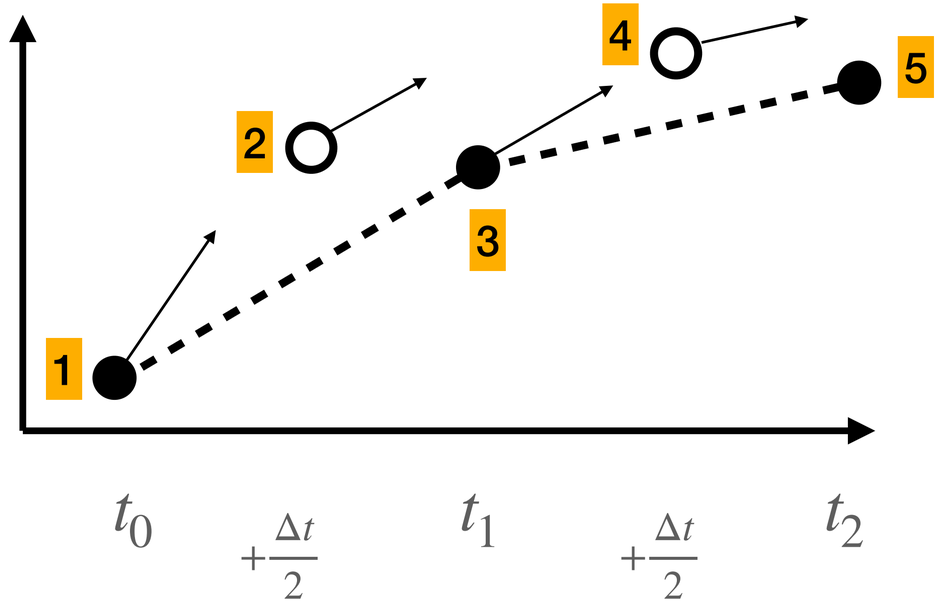
$$v_1 = v_0 + \Delta t\, a_0$$
$$x_1 = x_0 + \Delta t\, v_1$$

This (not at all intuitive!) version turns out to be first order accurate, but it has a special property: it is **symplectic**: it conserves **phase space area** and hence **energy** in systems where energy should be conserved (common in physics!).

The constraint of conserving energy leads to much better stability, allowing larger step sizes for a given accuracy.

Higher-order methods

One route to better ODE solvers is to find ways of using **multiple evaluations** at intermediate points to approximate the effect of keeping higher-order terms in the Taylor expansion.



The **midpoint** method uses the starting slope to find a point half-way across the interval, finds the slope at that point, then goes back to the start and uses the midpoint slope over the whole step.

Higher-order methods

One route to better ODE solvers is to find ways of using **multiple evaluations** at intermediate points to approximate the effect of keeping higher-order terms in the Taylor expansion.

Important examples for our purposes are:

- 4th-order Runge-Kutta (everyone's favourite, not symplectic)
- The Leapfrog method (an N-body favourite, 2nd-order, symplectic).

In the end, these schemes come down to evaluating different sequences of Euler-like steps. You can look these up and try to implement them in your code.

Other considerations that are relevant to the N-body problem are **time reversibility** and whether or not the method can be used with a **variable step size**.