CS**5643**

03 Solving ordinary differential equations

Steve Marschner
Cornell University
Spring 2023

Ordinary differential equation

An equation involving an unknown function and its derivatives

- but with only one independent variable (typically time)
- general form $f(t, y(t), y'(t), y''(t), \ldots, y^{(k)}) = 0$ for all t

In an *initial value problem* we know what is happening now and want to know the future

- boundary conditions are all at $t = 0$: $y(0), y'(0), \ldots, y^{(k-1)}(0)$
- goal: find $y(t)$ for all $t > 0$
- (notice that we need starting values for the derivatives less than the highest one involved)

In this course usually $k = 2$ (but sometimes 1)

Systems of ODEs

Typically there are multiple unknown functions

- e.g. the x and y coordinates of a particle, or of many particles, …

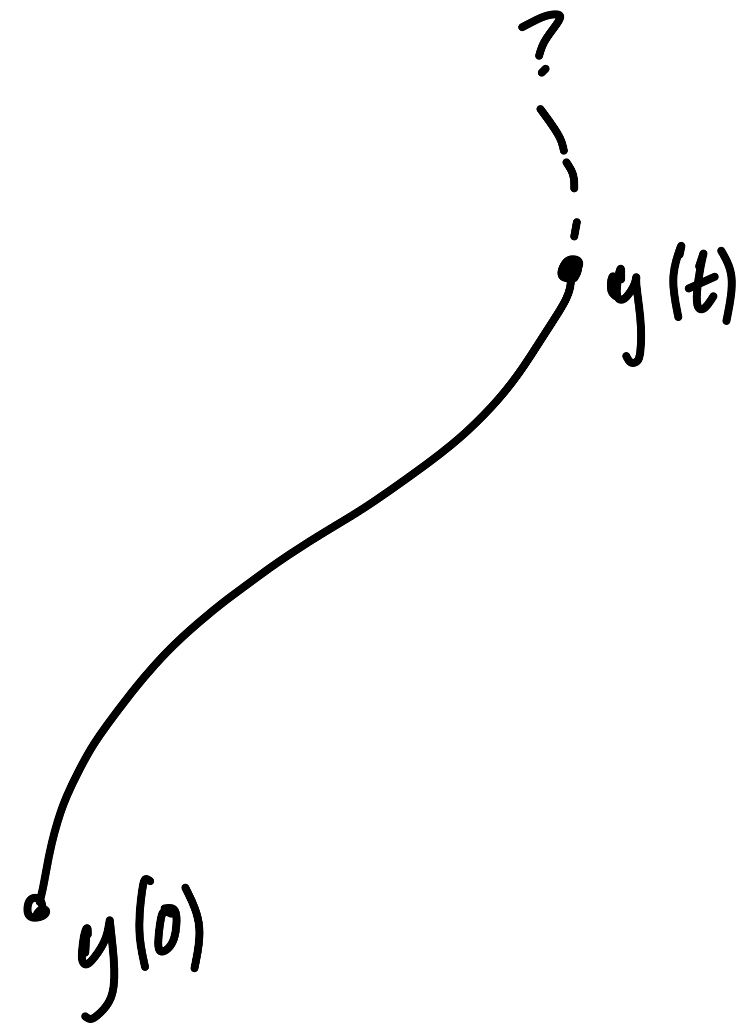
Can think of this as a system of interdependent ODEs…

…or simply as an ODE with a vector-valued unknown

- $\mathbf{f}(t, \mathbf{y}(t), \dot{\mathbf{y}}(t), \ddot{\mathbf{y}}(t)) = \mathbf{0}$ where $\mathbf{y} : \mathbb{R} \to \mathbb{R}^N$ and $\mathbf{f} : \ldots \to \mathbb{R}^N$

In this setting the solution is a path though $\mathbb{R}^N$

- an N-dimensional parameterized curve
- solving $\mathbf{f}$ tells you how to continue this curve by looking at the position, tangent, curvature, etc. at the end



Some simplifications

Most often we work with ODEs that are solved for the highest derivative:

- this is called an *explicit* ODE
- $\mathbf{y}^{(k)}(t) = \mathbf{f}(t, \mathbf{y}(t), \ldots, \mathbf{y}^{(k-1)})$
- or in the $k = 2$ case: $\ddot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t), \dot{\mathbf{y}}(t))$

Also we can choose to work only with:

- first-order systems ($k = 1$)
- autonomous systems ($\mathbf{f}$ independent of t)
- (next slides)

Reduction to first order

Someone gave me an ODE $\mathbf{y}^{(k)}(t) = \mathbf{f}(t, \mathbf{y}(t), \ldots, \mathbf{y}^{(k-1)})$ in N variables

I'll give back a first-order ODE

- unknown functions $\mathbf{y}(t), \mathbf{y}_1(t), \ldots, \mathbf{y}_{k-1}(t) : \mathbb{R} \to \mathbb{R}^N$

equation: $$\begin{bmatrix} \dot{\mathbf{y}} \\ \dot{\mathbf{y}}_1 \\ \vdots \\ \dot{\mathbf{y}}_{k-2} \\ \dot{\mathbf{y}}_{k-1} \end{bmatrix}(t) = \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_{k-1} \\ \mathbf{f}(t, \mathbf{y}, \mathbf{y}_1, \ldots, \mathbf{y}_{k-1}) \end{bmatrix}(t)$$

- this is a single first-order ODE in kN variables with the **same solution**

2nd order:

$$M\ddot{x}(t) = f(t, x(t), \dot{x}(t))$$

$$\Downarrow$$

$$\begin{bmatrix} x(t) \\ v(t) \end{bmatrix} = \begin{bmatrix} v(t) \\ M^{-1}f(t, x(t), v(t)) \end{bmatrix}$$

So at the highest level of abstraction the order doesn't matter

- but sometimes can get better results by remembering it started as a higher-order system

Autonomous vs. non-autonomous

Sometimes you see time as an explicit parameter, sometimes not

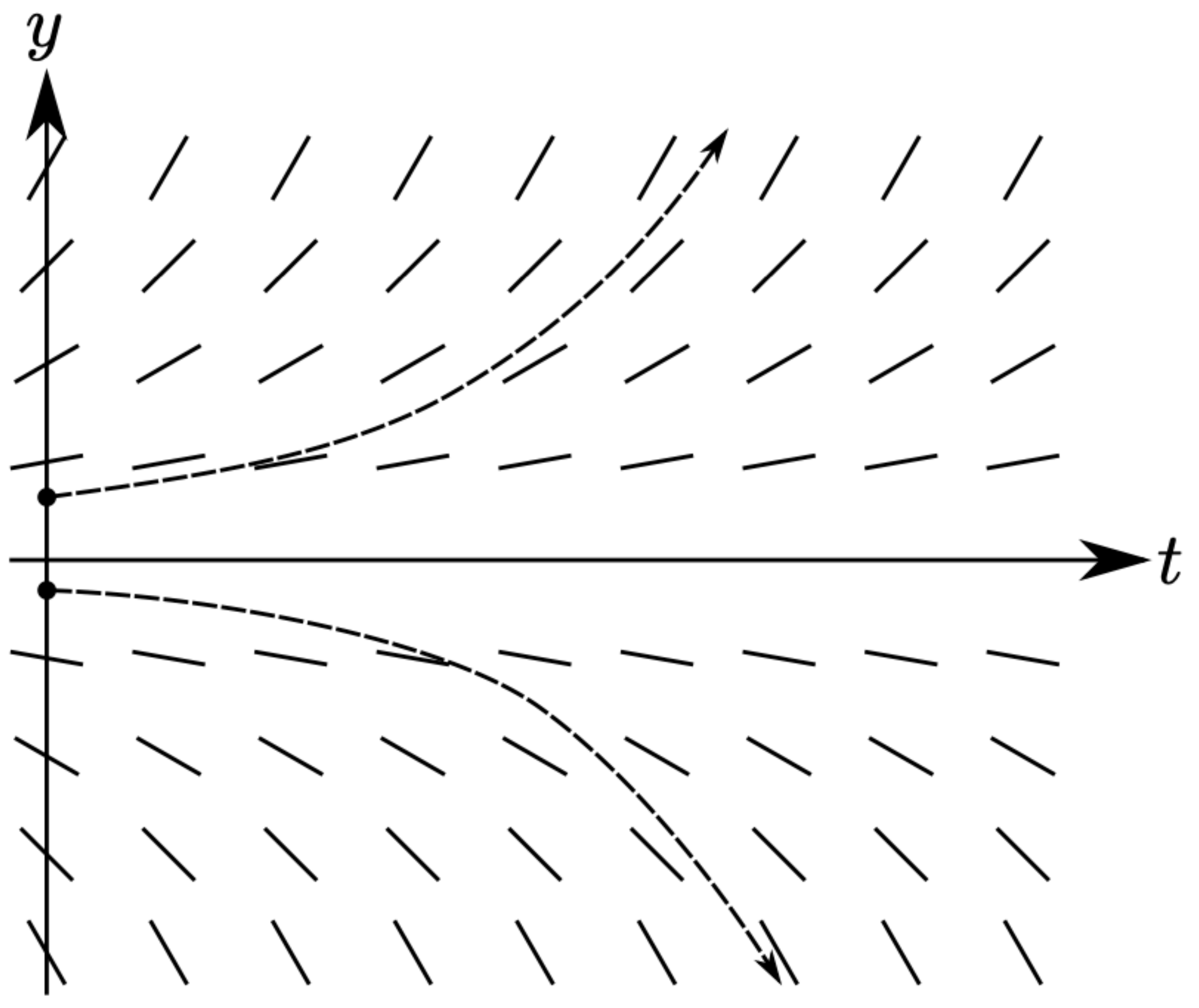
- $\dot{\mathbf{y}}(t) = \mathbf{f}(\mathbf{y}(t))$ is "autonomous"
- $\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t))$ is "non-autonomous"

If we want to do math or write code without the t, we can make a simple conversion:

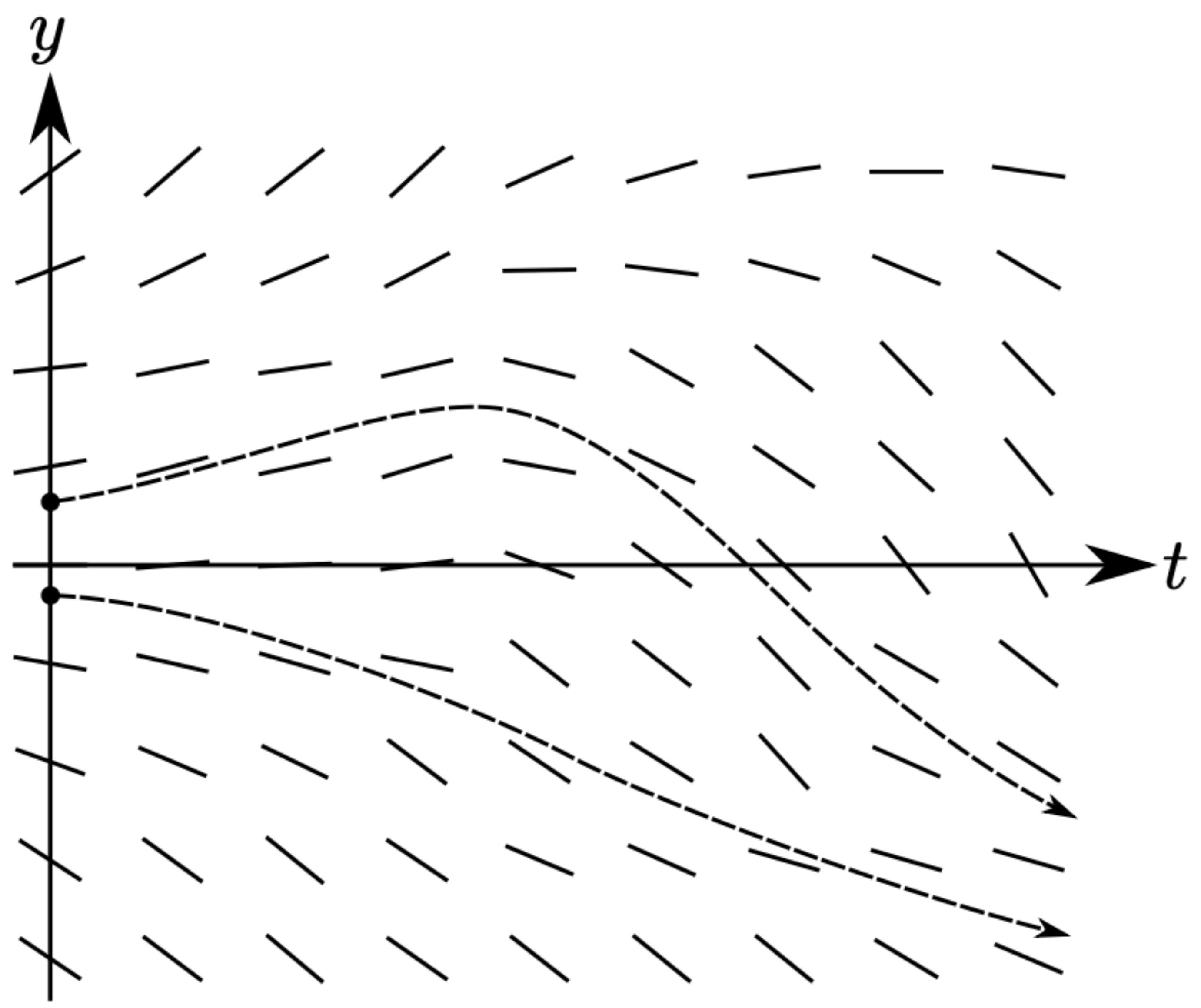
$$u(t) = \begin{bmatrix} y(t) \\ \tau \end{bmatrix}$$

$$\text{ODE: } \begin{bmatrix} \dot{y}(t) \\ \dot{\tau} \end{bmatrix} = \begin{bmatrix} f(\tau, y(t)) \\ 1 \end{bmatrix} \; ; \quad \tau(0) = 0$$

- and just relabel the axis to τ



Time-independent



Time-dependent

from Justin Solomon, *Numerical Algorithms*

Vector field picture

Now that we only have systems of the form $\dot{\mathbf{y}}(t) = \mathbf{f}(\mathbf{y}(t))$ there is a simple mental model:

- $\mathbf{y}(t)$ is the path of a point through the *state space* of the system
- remember $\mathbf{y}$ here is after a reduction to first order, so for instance in a Newtonian particle system $\mathbf{y}$ includes both the position $\mathbf{x}$ and the velocity $\mathbf{v}$
- $\mathbf{f}$ is a vector field in that state space that tells the particle which way to go
- so the process reduces to advection through a flow field (though in many dimensions)

canonical example: harmonic oscillator in 1D, $m\ddot{x} = -kx$

- $\begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v \\ -(k/m)x \end{bmatrix}$ or with appropriate choice of units $\begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v \\ -x \end{bmatrix}$
- aka $\dot{\mathbf{y}} = R\mathbf{y}$ where R is a rotation by –90 degrees

Canonical example: harmonic oscillator

**e.g. mass on a spring,
plucked rubber band,
tuning fork**

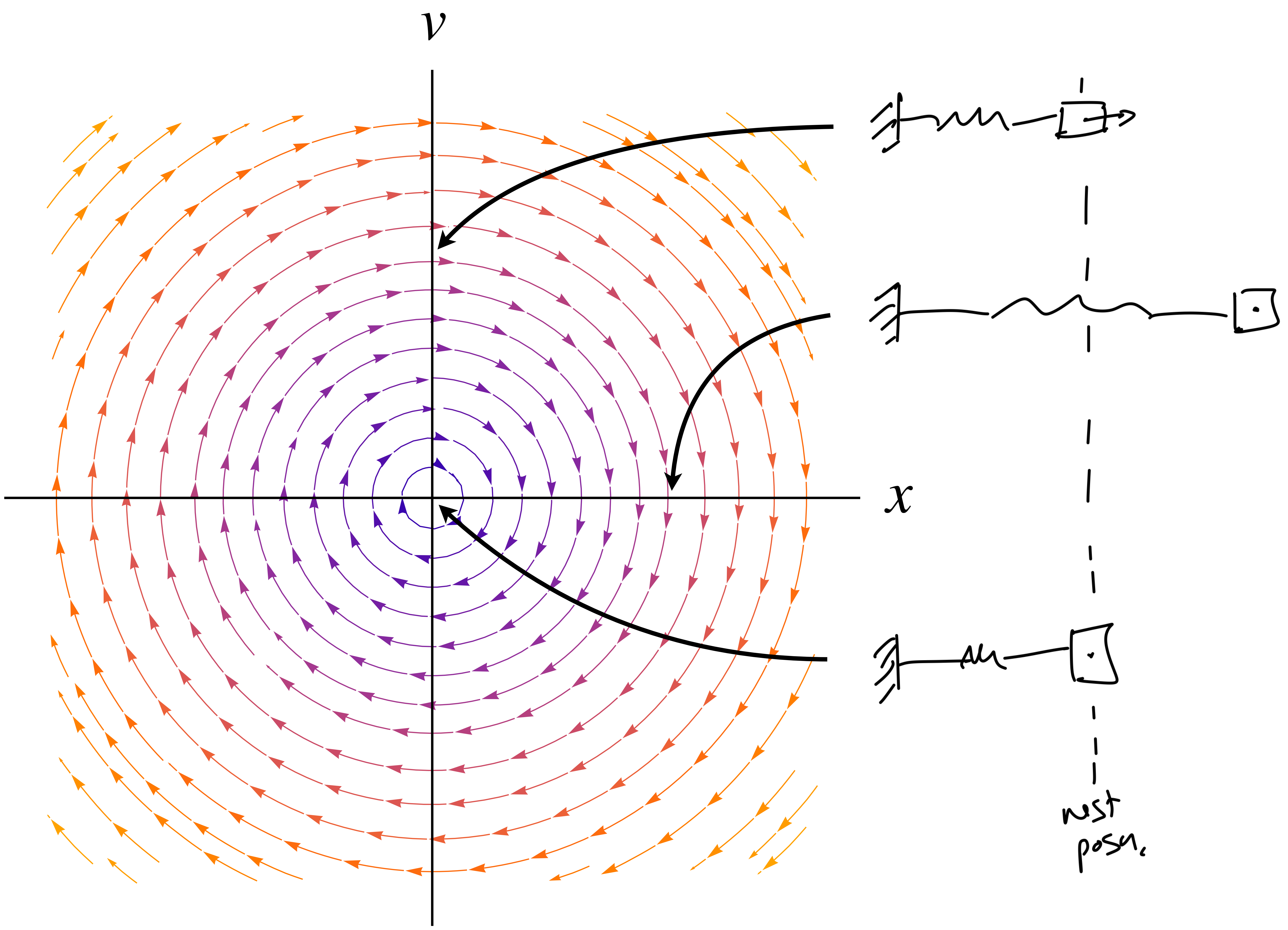
1D ODE $m\ddot{x} = -kx$

$$\begin{bmatrix}\dot{x}\\ \dot{v}\end{bmatrix} = \begin{bmatrix} v \\ -(k/m)x\end{bmatrix}$$

- or with appropriate units:

$$\begin{bmatrix}\dot{x}\\ \dot{v}\end{bmatrix} = \begin{bmatrix} v \\ -x\end{bmatrix}$$

- aka $\dot{\mathbf{y}} = R\mathbf{y}$ where R is a rotation by –90 degrees
- solutions are like $x(t) = \sin t, v(t) = \cos t$

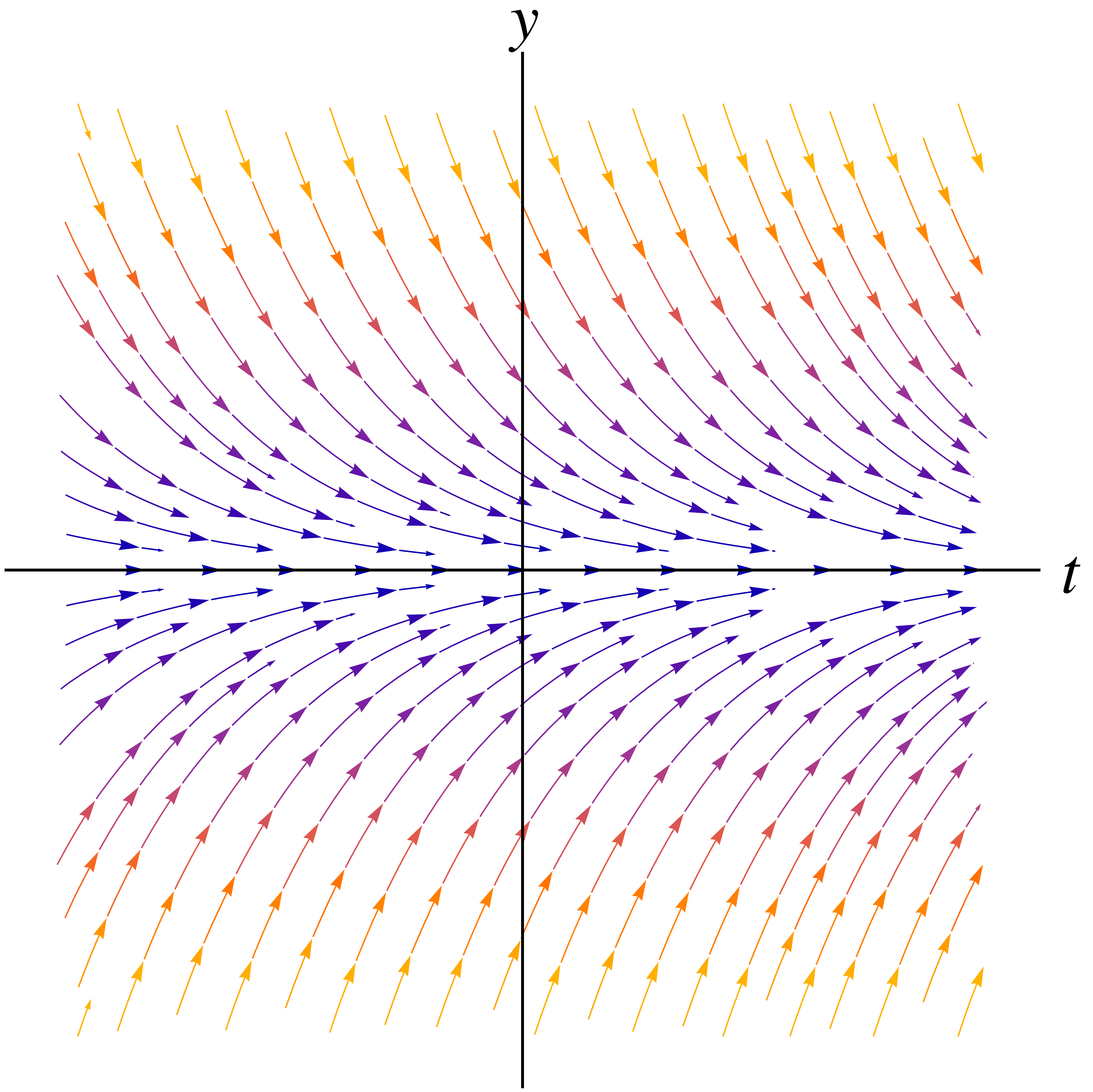


Canonical example: exponential decay

E.g. cup of tea cooling off
or particle slowing in fluid

1D ODE: $\dot{y} = -ky$

- solutions are like $y(t) = \exp(-t)$



Numerical solution methods

Most ODEs don't have closed form solutions so we resort to numerical approximation

- the only thing we know how to compute is the $\mathbf{f}$ in $\dot{\mathbf{y}}(t) = \mathbf{f}(\mathbf{y}(t))$

Want to compute approximate values of the unknown $\mathbf{y}(t)$ for desired values of t

- to do this we compute $\mathbf{y}(t_k)$ for a series of *time steps*
  - from where we know $\mathbf{y}$ (canonically at $t = 0$)
  - to where we want $\mathbf{y}$ (e.g. at the time of each animation frame)
- compute each step from the results of previous steps using a *local approximation* to $\mathbf{y}$
- different local approximations lead to different time stepping algorithms, known as numerical integration methods or ODE solvers or just "integrators" or "solvers."

Setup for simple integration methods

Start with a constant step size h

- time steps are equally spaced, $t_{k+1} = t_k + h$
- if we start at $t_0 = 0$ then $t_k = kh$ and the number of steps to reach time T is T/h

We want some equation we can solve to approximate $\mathbf{y}(t_{k+1})$ assuming we know $\mathbf{y}(t_k)$

- in practice we don't know $\mathbf{y}(t_k)$ exactly; we just have the approximation from the previous step
- I will use $\mathbf{y}_k$ for the approximation we computed at step k and $\mathbf{y}(t_k)$ for the actual value
- the goal of our method is to ensure $\mathbf{y}_k \approx \mathbf{y}(t_k)$ so that the points $(t_k, \mathbf{y}_k)$ are a good approximation to the solution function $\mathbf{y}(t)$
- an important question: how to quantify how accurately $\mathbf{y}_k$ approximates $\mathbf{y}(t_k)$

Euler's integrators

Most integrators can be derived from a Taylor expansion

- after all it's the first tool we reach for when we want a local approximation

E.g. let's expand $\mathbf{y}$ around $t = t_k$:

$$\mathbf{y}(t) = \mathbf{y}(t_k) + \dot{\mathbf{y}}(t_k)(t - t_k) + O((t - t_k)^2)$$

- evaluate at $t_{k+1} = t_k + h$ and substitute the ODE $\dot{\mathbf{y}}(t) = \mathbf{f}(\mathbf{y}(t))$

$$\mathbf{y}(t_{k+1}) = \mathbf{y}(t_k) + h\mathbf{f}(\mathbf{y}(t_k)) + O(h^2)$$

- leading to the *timestep equation* $\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}(\mathbf{y}_k)$ known as "Euler's method" or "forward Euler"

This is a *first order accurate*, *explicit* integration method

- "explicit" because the timestep equation is already solved for $\mathbf{y}_{k+1}$; it is an explicit formula
- "first order accurate" because the error is proportional to h^2

Euler's integrators

Alternatively we could expand $\mathbf{y}$ around $t = t_{k+1}$:

$$\mathbf{y}(t) = \mathbf{y}(t_{k+1}) + \dot{\mathbf{y}}(t_{k+1})(t - t_{k+1}) + O((t - t_{k+1})^2)$$

- evaluate at $t_k = t_{k+1} - h$ and substitute the ODE $\dot{\mathbf{y}}(t) = \mathbf{f}(\mathbf{y}(t))$

$$\mathbf{y}(t_k) = \mathbf{y}(t_{k+1}) - h\mathbf{f}(\mathbf{y}(t_{k+1})) + O(h^2)$$

- leading to the *timestep equation* $\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}(\mathbf{y}_{k+1})$ known as "backward Euler's method"

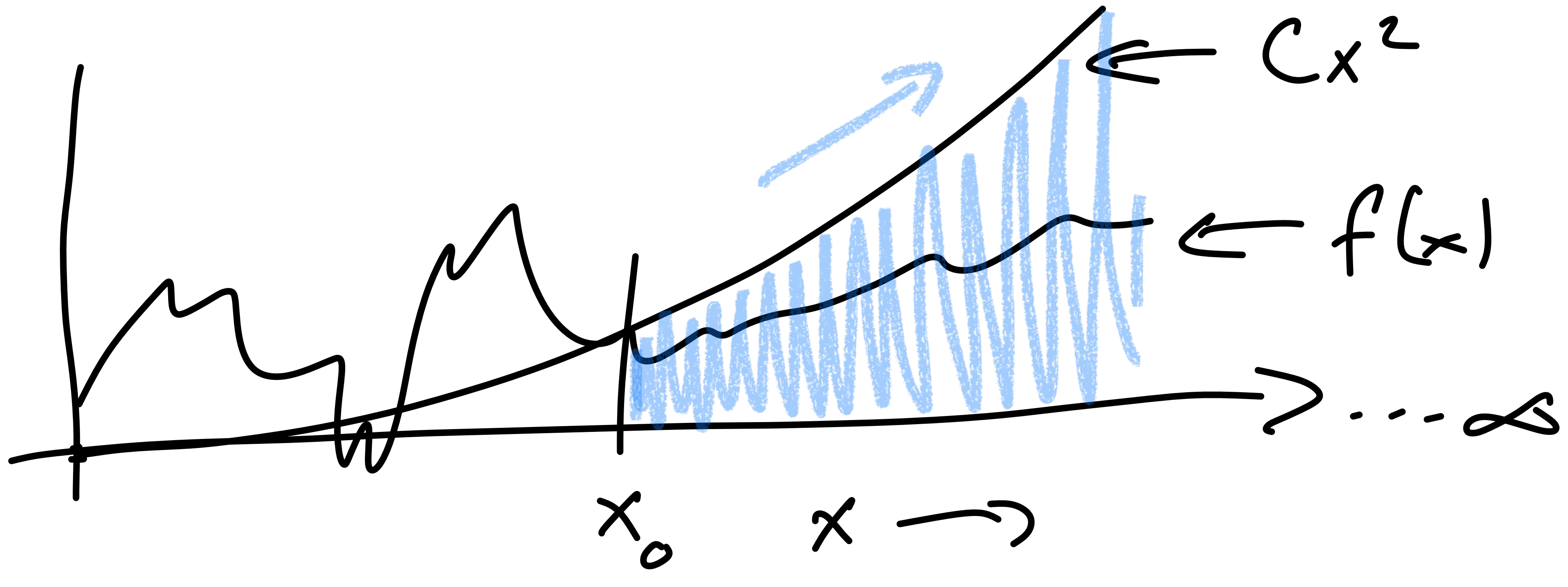
This is a first order accurate, *implicit* integration method

- "implicit" because the timestep equation needs to be solved to find $\mathbf{y}_{k+1}$
- "first order accurate" because the error is still proportional to h^2

How does your error shrink?

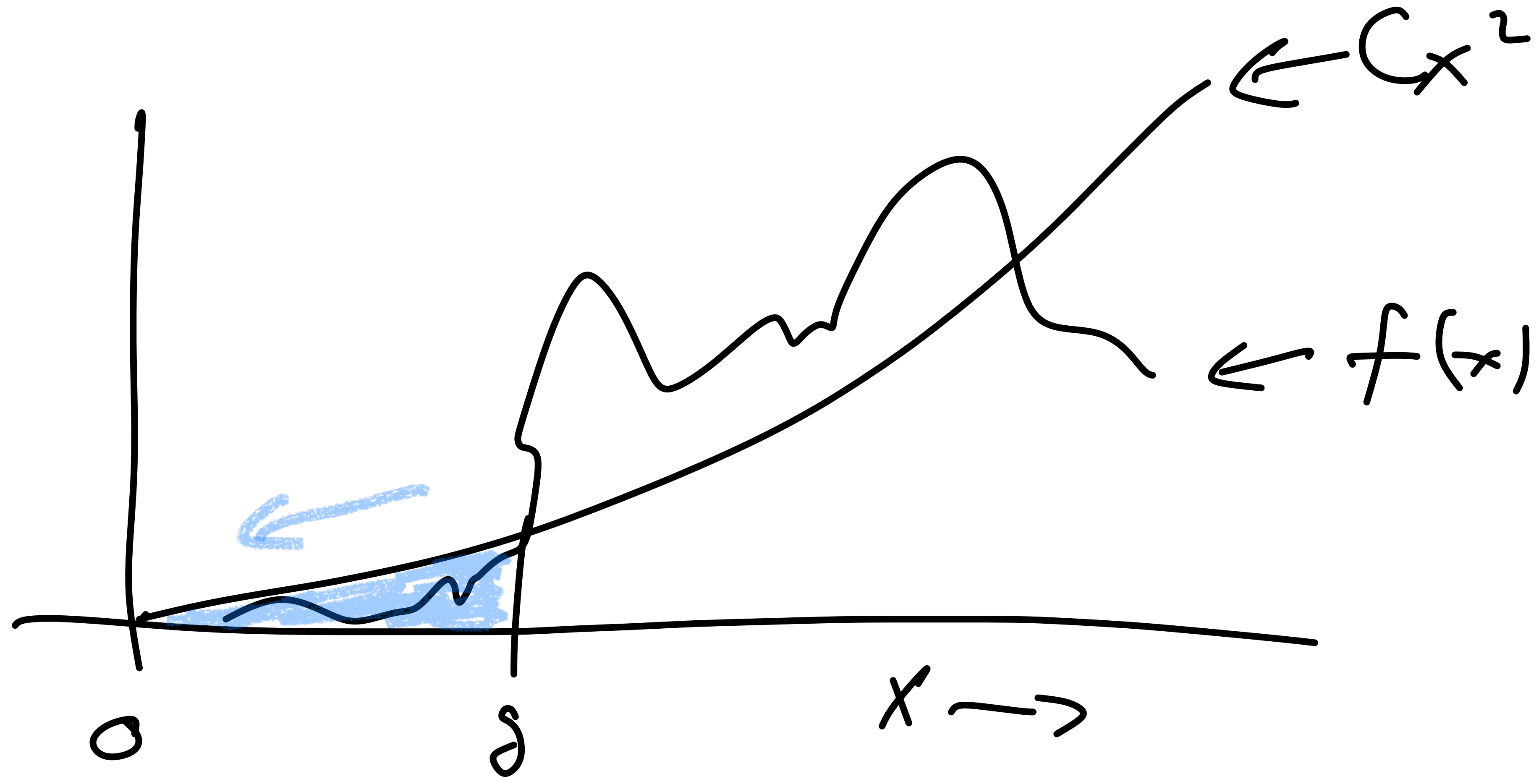
If things are working at all, we can get any accuracy we need by decreasing h

- that is, $\lim_{h\to 0}[\mathbf{y}_k - \mathbf{y}(t_k)] = 0$



we compare integrators' accuracy in terms of *asymptotic rate* of convergence

- recall big-O notation $f(x) \in O(x^2)$ as $x \to \infty$ means there are constants C and x_0 such that $x > x_0 \implies f(x) \le Cx^2$
- we can use the same idea for asymptotics as $x \to 0$: $f(x) \in O(x^2)$ as $x \to 0$ means there exist constants C and δ such that $x < \delta \implies f(x) \le Cx^2$



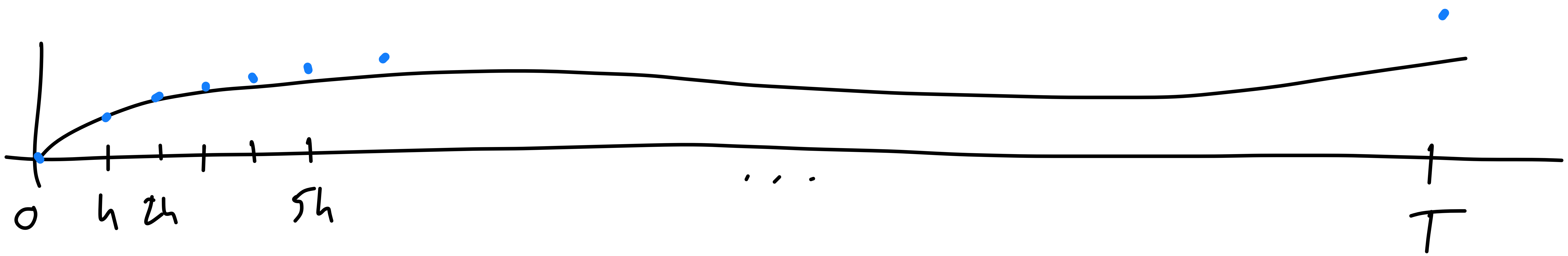
How does your error grow?

The error in a time-stepped approximation accumulates

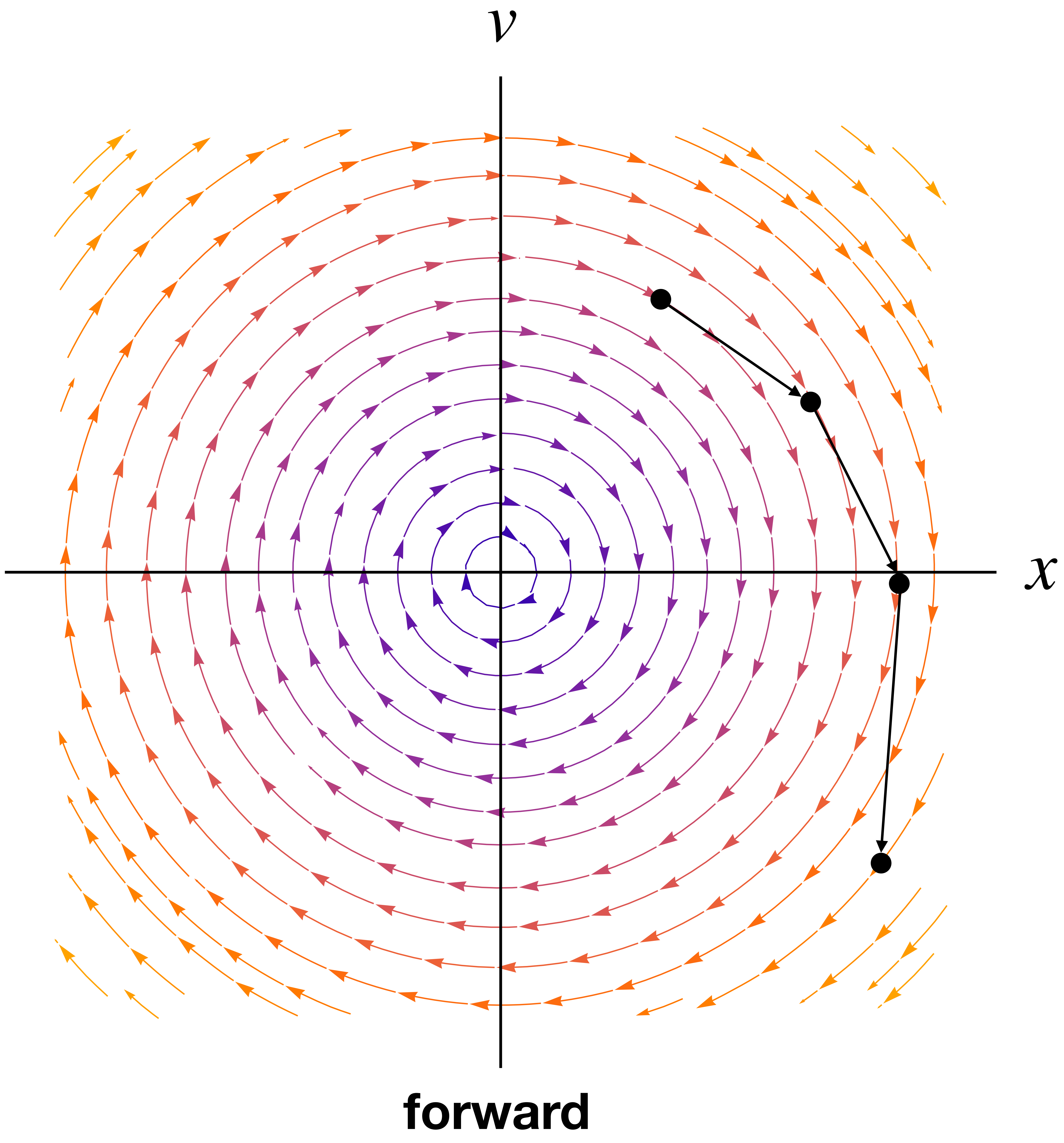
- in worst case (sadly not so uncommon) all the errors point the same way so the error after N steps is N times the error in one step
- to get to time T requires $N \approx T/h$ steps
- so if error in one step is $O(h^p)$ then error after N steps is $O(h^{p-1})$

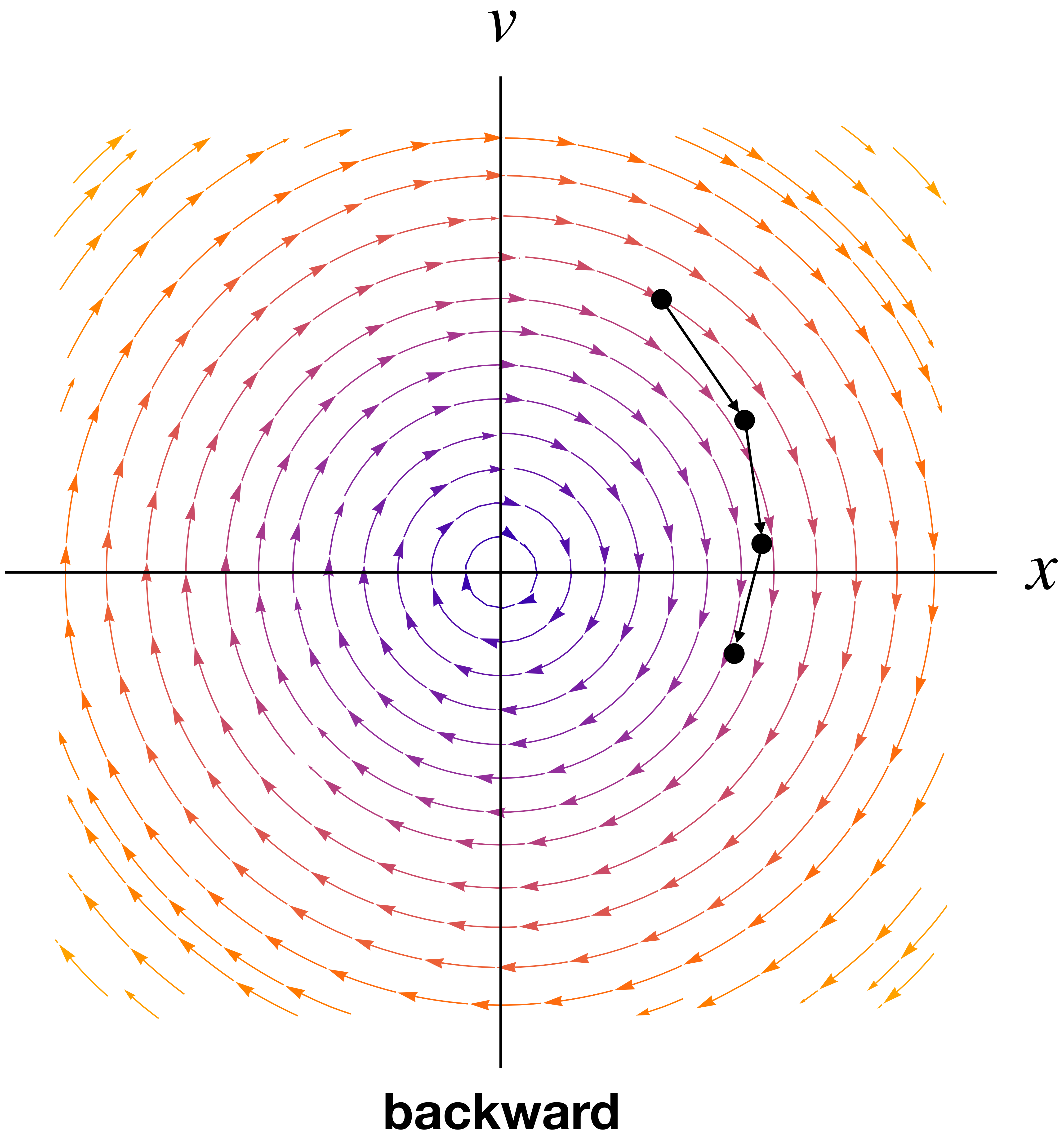
Nomenclature for integrators works two ways

- *n*th order integrator is "accurate to *n*th order" in one step, meaning the error is $O(h^{p+1})$
- *n*th order integrator has order-*n* error after a fixed time, meaning the error is $O(h^p)$



Behavior of Euler integrators





Behavior of Euler integrators