Math 260: Python programming in math

Fall 2020

Intro to ODEs:
Euler's method, systems of ODEs

ODEs: introduction

A motivating example: population growth

- An amount $p(t)$ of bacteria live on a petri dish, starting with $p(0) = p_0$
- Unconstrained, the growth rate at time t is $rp(t)$
- But the petri dish can onld hold K bacteria...

A plausible model is the **ordinary differential equation** (ODE)

$$\frac{dp}{dt} = rp(1 - p/K),$$

With the 'initial condition' $p(0) = p_0$, we get an **initial value problem** (IVP)

$$\frac{dp}{dt} = rp(1 - p/K), \quad p(0) = p_0.$$

Simple case: In an infinitely large petri dish,

$$\frac{dp}{dt} = rp \implies p(t) = p_0 e^{rt} \text{ (exponential growth!)}$$

Notation (derivatives):

We will use $x'(t)$ and dx/dt for derivatives.
Second order, etc. are denoted by $x''(t)$ or d^2x/dt^2.
m-th order derivatives are $x^{(m)}(t)$ or $d^m x/dt^m$.

Aside: separable ODEs

Simple ODEs can be solved by 'separating variables'. For instance,

$$\frac{dy}{dt} = ry \implies \frac{1}{y}\,dy = r \implies \ln|y| = rt + C \implies |y| = Ce^{rt}$$

In general, a **separable equation** for $y(t)$ can be written the form

$$f(y)\frac{dy}{dt} = g(t)$$

which can be solved, informally, by integrating both sides:

$$f(y)\,dy = g(t)\,dt \implies \int f(y)\,dy = \int g(t)\,dt$$

- Not many ODEs of interest are separable
- We need other techniques or **numerics** (the point of this module!)

ODEs: numerics

We will see how to solve the 'standard' initial value problem for $y(t)$,

$$y' = f(t, y), \quad y(a) = c, \qquad a \le t \le b.$$

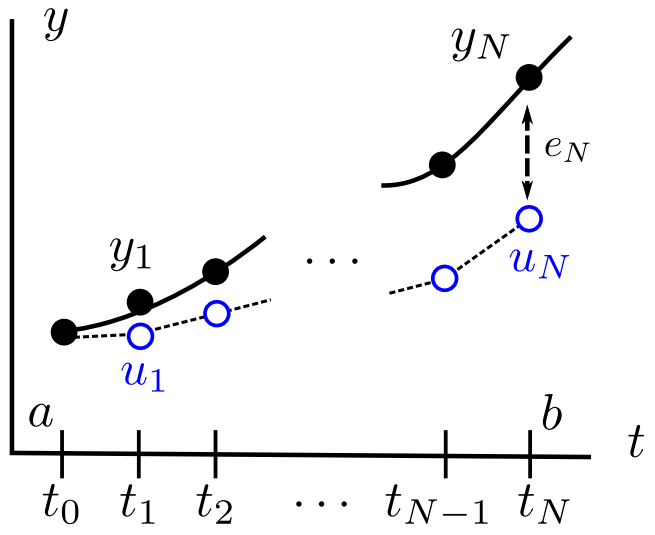
First, what is a numerical solution?

- We need to **dicretize** the item interval into discrete points, called a **mesh**:

$$a = t_0 < t_1 < \cdots < t_N = b.$$

- A numerical solution approximates $y(t)$ at the mesh points:

num. solution $u_n \approx y(t_n)$ for $n = 0, \cdots, N$.



Definition (errors)

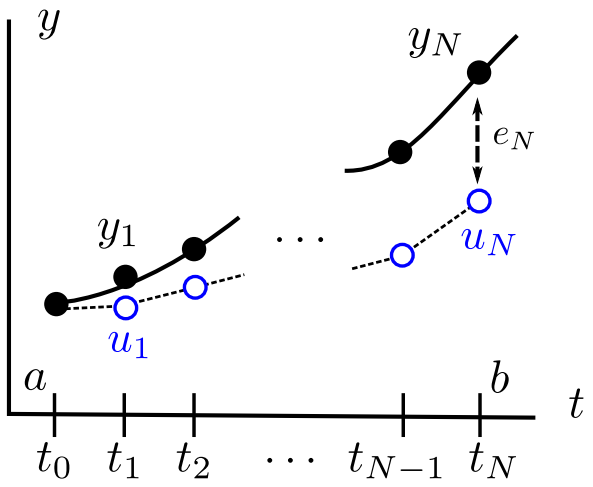
- Local error at a mesh point: $e_n = |y_n - u_n|$
- Global error in $[a, b]$: the largest of the errors at mesh points in the interval:

$$E = \max_{t_n \in [a,b]} |u_n - y(t_n)| = \max_{0 \le n \le N} e_n$$

ODEs: numerics

IVP: $y' = f(t, y), \quad y(a) = c$
Exact solution: $y(t)$

At mesh points: $y_n = y(t_n)$,
approximation: $u_n \approx y(t_n)$



To solve, we 'integrate forwards' from t_0 to t_1, then to t_2, etc.
For simplicity, let's assume that the spacing is h (constant).

Approach 1: Estimate y'. The simplest way uses t_n and t_{n+1}:

$$\frac{y_{n+1} - y_n}{h} \approx y'(t_n) = f(t_n, y_n).$$

This becomes a formula for the approx. u_n:

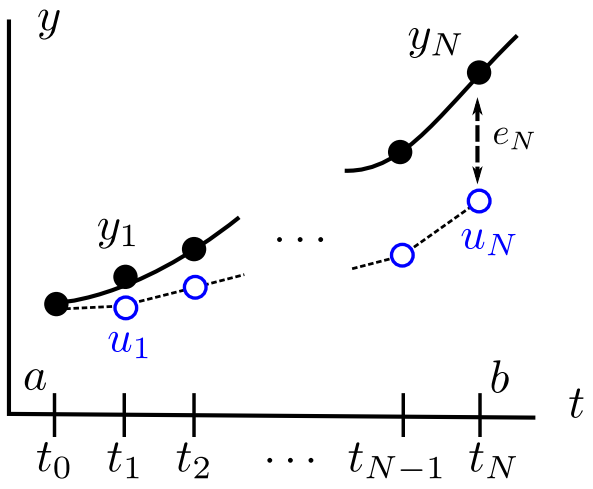
$$\frac{u_{n+1} - u_n}{h} = f(t_n, u_n).$$

which is (**Euler's method**).

ODEs: numerics

IVP: $y' = f(t, y), \quad y(a) = c$
Exact solution: $y(t)$

At mesh points: $y_n = y(t_n)$,
approximation: $u_n \approx y(t_n)$



Approach 2: Integrate from t_n to t_{n+1}, use the FTC:

$$y_{n+1} - y_n = \int_{t_n}^{t_{n+1}} y' \, dt = \int_{t_n}^{t_{n+1}} f(s, y(s)) \, ds.$$

Now we estimate the integral (e.g. trapezoidal rule...). Using the 'left hand rule', we get

$$y_{n+1} - y_n \approx h f(t_n, y_n)$$

which is Euler's method again.

Euler's method

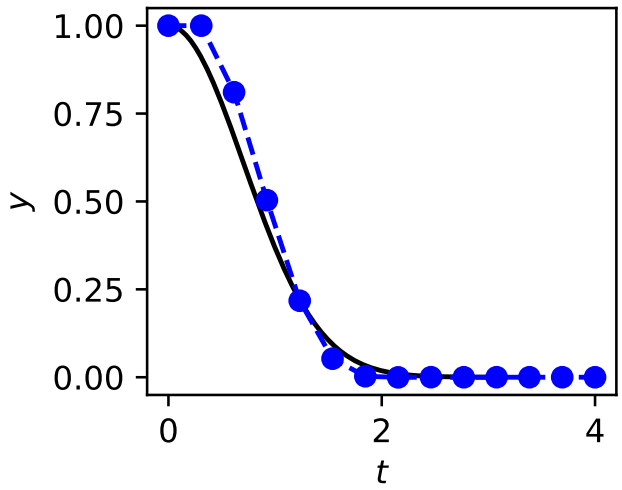
Thus, to solve the *differential* equation

$$y' = f(t, y), \qquad y(a) = c$$

we can use the '**difference** equation' given by **Euler's method**

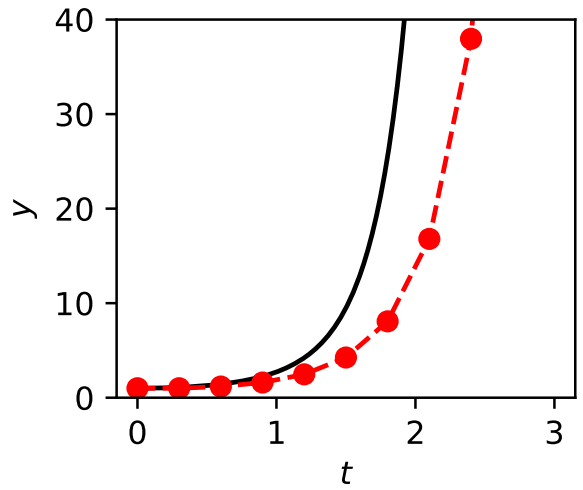
$$u_{n+1} = u_n + hf(t_n, u_n), \qquad u_0 = c.$$

Two typical examples (solved with $h = 0.3$):



$y' = -2ty$, $y(0) = 1$
Exact: $y(t) = \exp(-t^2)$

- Error stays small for all t



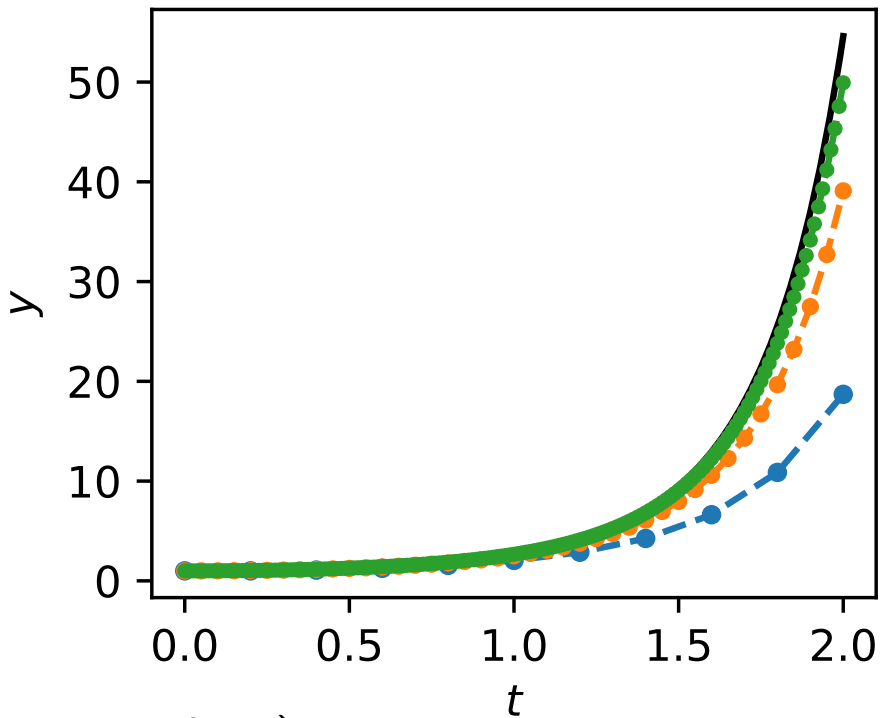
$y' = 2ty$, $y(0) = 1$
Exact: $y(t) = \exp(t^2)$

- error grows with t (by alot)

Euler's method

The approximation **converges** as $h \to 0$ (mesh spacing $\to 0$)

This is true on any fixed interval (even in bad cases):



($h = 0.2, 0.05, 0.00125, \cdots$ here)

Euler's method

Implementation is easy - just iterate the formula.

$$\text{IVP: } y' = f(t, y), \qquad y(a) = c$$

$$\text{Difference eq: } u_{n+1} = u_n + hf(t_n, u_n), \qquad u_0 = c.$$

Two structures: **for** or **while** loop. Roughly:

```
def fwd_euler(f, a, b, y0, h):
    n = round((b-a)/h)
    h = (b-a)/n  # fix if (b-a)/h
    t = [0]*(n+1) #  was not an int
    y = [0]*(n+1)
    t[0] = a
    y[0] = y0
    for k in range(0, n):
        y[k+1] = y[k] + h*f(t[k],y[k])
        t[k+1] = t[k] + h
    return t, y
```

```
def fwd_euler(f, a, b, y0, h):
    t = a
    y = y0
    tvals = [t]
    yvals = [y]
    while t < b - 1e-12:
        y += h*f(t, y)
        t += h
        tvals.append(t)
        yvals.append(y)
  return tvals, yvals
```

The 'while' structure is more versatile, e.g. for changing the step size h during the loop (so the number of steps is not known).

Euler's method: error

How do we determine how the error behaves?

- Consider the error due to approximation in going from t_n to t_{n+1}
- Plug the exact solution $y(t_n)$ into the **difference equation**:

$$u_{n+1} = u_n + hf(t_n, u_n)$$

$$y(t_{n+1}) = y(t_n) + hf(t_n, y(t_n)) + \tau_n$$

since $y(t)$ does not satisfy the difference equation exactly.

The 'leftover' τ_n is the **local truncation error**.

- Now we can use Taylor's theorem to find τ_n (let $y_n = y(t_n)$ etc.)

$$\begin{aligned}
\tau_n &= y_{n+1} - y_n - hf(t_n, y_n) \\
&= (y_n + hy_n' + \frac{h^2}{2}y_n'' + O(h^3)) - y_n - hf(t_n, y_n) \\
&= hy_n' - hf(t_n, y_n) + \frac{h^2}{2}y_n'' + O(h^3) \\
&= \frac{h^2}{2}y_n'' + O(h^3)
\end{aligned}$$

since the ODE says that $y_n' = f(t_n, y_n)$. In particular, $\boxed{\tau_n = O(h^2)}$.

Euler's method: error

Thus, for the IVP

$$y' = f(t, y), \quad y(a) = c,$$

Euler's method has a local truncation error $\tau_n = O(h^2)$.

What about the global error?

- The local error $e_n = |u_n - y_n|$ depends on two parts:
  - **truncation error** (the error from approximating $t_{n-1} \to t_n$)
  - **propagated error** (error building up from previous steps).
- After some work, we can show that in an interval $[a, b]$,

$$e_n \leq \frac{C}{h} \max |\tau_k| \text{ for all } n \text{ such that } t_n \in [a, b]$$

- $\max |\tau_k| =$ largest truncation error , $C =$ some constant

Euler's method: error

- Idea: $O(h^2)$ at each step, and $N = (b-a)/h$ steps

$$\text{global error} \sim \frac{1}{h} \cdot (\text{local error})$$

- This means that the global error is $O(h)$, that is

$$E(h) := \max_{t_n \in [a,b]} |u_n - y_n| = O(h) \text{ as } h \to 0.$$

  We say the Euler's method is **first order**.
- For typical ODE methods, $E(h) \sim Ch^p$ as $h \to 0$.

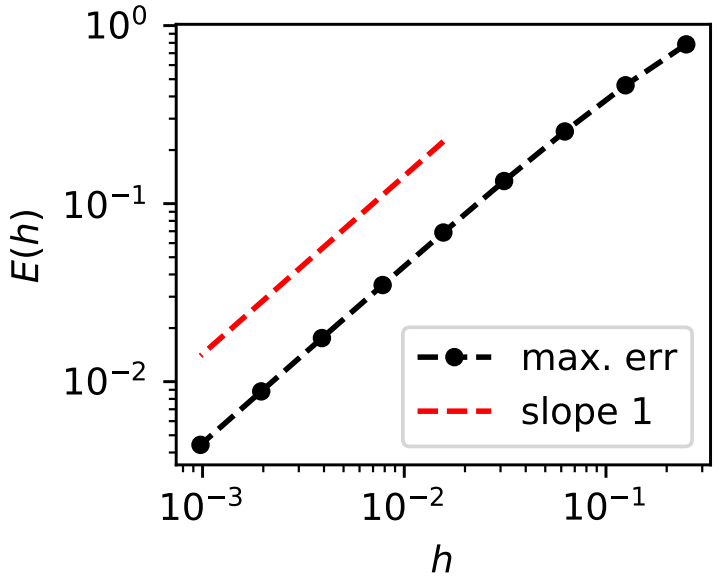
Euler's method: error

We can check the order in the usual two ways...

Approach 1: Use the global error

$$E(h) := \max_{t_n \in [a,b]} |u_n - y_n| = O(h) \text{ as } h \to 0.$$

and plot (with a log-log plot) vs. h or n.



Euler's method: error

Approach 2: Take the error at a single point (easiest: $t = b$)

- Note that changing h also changes the mesh points - except $t = b$
- The 'p-estimate' trick also works, since

$$u(b; h) \approx y(b) + Ch$$

where $u(b; h)$ is the approx. solution at $t = b$ with step size h.

N	u at $t = b$	p
4	$1.93e+00$	0.63
8	$2.26e+00$	0.79
16	$2.46e+00$	0.89
32	$2.58e+00$	0.94
64	$2.65e+00$	0.97
128	$2.68e+00$	0.98
256	$2.70e+00$	0.99

Example: $y' = 2ty$, at $t = 2$ (with $h = 2/N$), and

$$p \approx -\log_2 \left(\frac{u(4N) - u(2N)}{u(2N) - u(N)} \right)$$

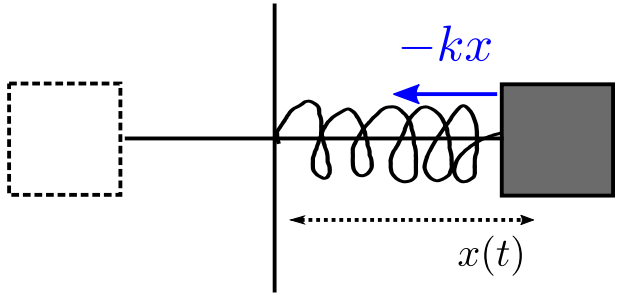
ODEs: some linear systems

ODEs: introduction

Another example: simple harmonic motion - oscillating systems!

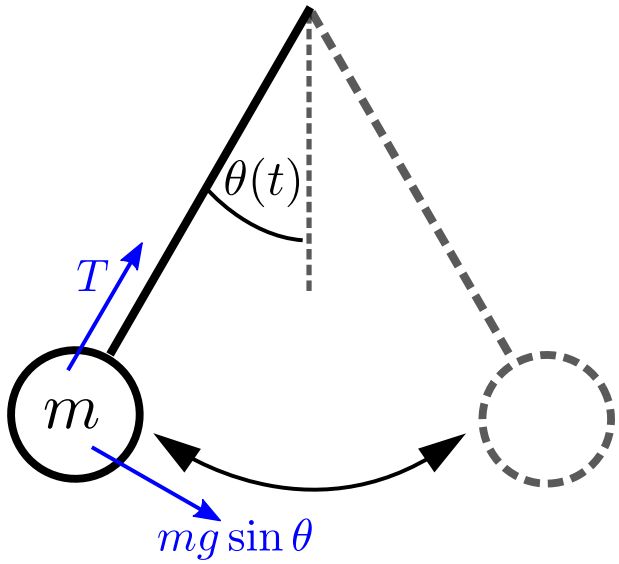
- Mass-spring system:
  - spring restoring force: $-kx(t)$
  - damping force $-cv(t)$
  - Newton's law $F = ma$

$$\implies m\frac{d^2x}{dt^2} = -kx - c\frac{dx}{dt}$$



- Simple pendulum:
  - Angular displacement $\theta(t)$
  - restoring force (gravity): $mg\sin\theta$
  - damping force (friction) $-c\,d\theta/dt$

$$\implies L\frac{d^2\theta}{dt^2} = -mg\sin\theta - c\frac{d\theta}{dt}$$



'Simple' case: displacement is small, so $\sin\theta \approx \theta + O(\theta^3)$:

$$L\frac{d^2\theta}{dt^2} = -mg\theta - c\frac{d\theta}{dt}.$$

Aside: linear systems

We can demystify the solution to

$$ay'' + by' + cy = 0 \tag{A}$$

by converting it to a linear system (also useful for numerics!).

Define $x_1 = y$ and $x_2 = y'$ and the vector $\mathbf{x} = (y, y')$. Then $y'' = x_2'$ so

$$x_1' = x_2, \qquad x_2' = -(c/a)x_1 - (b/a)x_2.$$

In matrix form, this is the **linear system**

$$\mathbf{x}' = A\mathbf{x}, \qquad A := \begin{bmatrix} 0 & 1 \\ -c/a & -b/a \end{bmatrix}$$

To solve, look for exponential solutions

$$\mathbf{x}(t) = e^{\lambda t}\mathbf{v}$$

and plug in to find that this is a solution if and only if

$$A\mathbf{v} = \lambda\mathbf{v}.$$

Aside: linear systems

Thus, we have found that for the LCC system

$$\mathbf{x}' = Ax,$$

the eigenvalues λ and eigenvectors $\mathbf{v}$ yield solutions

$$\mathbf{x}(t) = e^{\lambda t}\mathbf{v}$$

and these solutions are linearly independent for distinct λ (from linear algebra).

- For the second-order converted system

$$\mathbf{x}' = A\mathbf{x}, \qquad A := \begin{bmatrix} 0 & 1 \\ -c/a & -b/a \end{bmatrix}$$

  The eigenvalues satisfy $\det(A - \lambda I) = 0$, or

$$a\lambda^2 + b\lambda + c = 0$$

  which is exactly the characteristic equation from before!

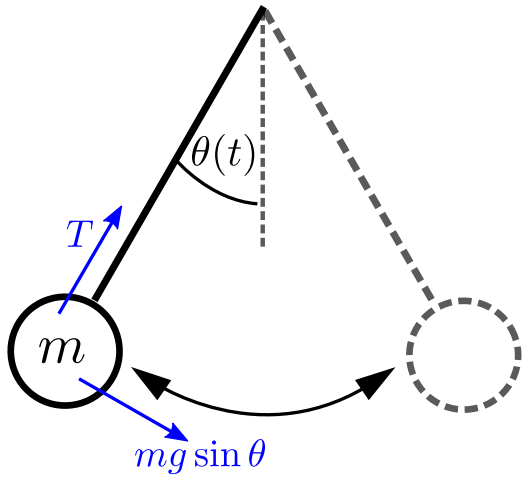
- The full solution is a linear combination of these exponentials, e.g.

$$\mathbf{x}(t) = c_1 e^{\lambda_1 t}\mathbf{v}_1 + c_2 e^{\lambda_2 t}\mathbf{v}_2.$$

- (Note: for repeated eigenvalues, more work is required)

The pendulum (linear case)

- Simple pendulum:
  - Angular displacement $\theta(t)$
  - restoring force (gravity): $mg \sin\theta$
  - damping force (friction) $-c\, d\theta/dt$



$$\implies L\frac{d^2\theta}{dt^2} = -mg\sin\theta - c\frac{d\theta}{dt}$$

After rescaling, we get a (nonlinear) ODE

$$\theta'' = -\sin\theta - 2\beta\theta'$$

with some initial displacement $\theta(0)$ and angular velocity $\theta'(0)$.

In the 'small displacement' case, we get

$$\theta'' = -\theta - 2\beta\theta'.$$

The exact solution tells us about the behavior...

The pendulum (linear case)

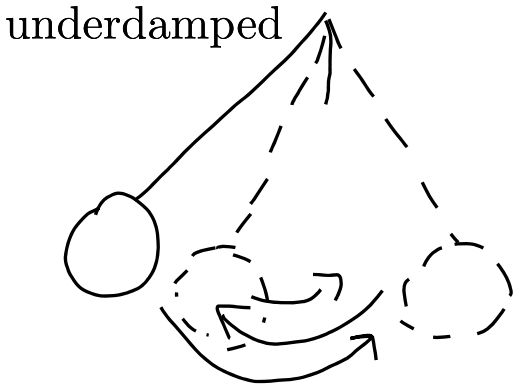
$$\text{ODE: } \theta'' = -\theta - 2\beta\theta'.$$

Looking for solutions $\theta = e^{rt}$ we get

$$r^2 + 2\beta r + 1 = 0.$$

This has roots

$$r = -\beta \pm \sqrt{\beta^2 - 1}.$$

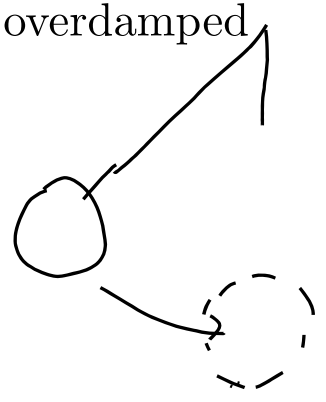


There are two important cases:

- **Overdamped:** If $\beta > 1$, both $r's$ are real and negative - decaying (non-oscillating) solutions
- **Underdamped:** But if $0 < \beta < 1$, then

$$r = -\beta \pm i\sqrt{1 - \beta^2} = -\beta \pm \omega i$$

which gives solutions $e^{-\beta t}(\cos \omega t + i \sin \omega t)$ (decaying oscillations)



In either case, solutions **will decay to** $\theta = 0$ (the pendulum slows down)

Euler's method: systems

To solve such ODEs, we must extend Euler's method to the **first order system**

$$\mathbf{y}' = F(t, \mathbf{y}), \quad \mathbf{y}(a) = \vec{c}$$

where $\mathbf{y}(t)$ is a vector in $\mathbb{R}^m$ for each t.

- This is easy! Simply replace scalars with vectors:

$$\mathbf{u}_{n+1} = \mathbf{u}_n + hF(t_n, \mathbf{u}_n), \quad \mathbf{u}_0 = \vec{c}.$$

- The 'error' e_n is then the max of the errors for each component.

Conversion: any n-th order ODE

$$y^{(m)} = f(t, y, \cdots, y^{(n-1)})$$

can be converted to this standard form by setting

$$x_1 = y,\ x_2 = y', \cdots x_m = y^{(m-1)}$$

and $\mathbf{x} = (x_1, \cdots, x_m)$ so that $x_1' = x_2$ and so on, giving

$$\begin{bmatrix} x_1 \\ \vdots \\ x_{m-1} \\ x_m \end{bmatrix}' = \begin{bmatrix} x_2 \\ \vdots \\ x_m \\ f(t, x_1, \cdots, x_m) \end{bmatrix} \implies \mathbf{x}' = F(t, \mathbf{x})$$

Euler's method: systems

We can use operator overloading for the code (`numpy` arrays are good here!)...

For scalar ODEs:

```
def fwd_euler(f, t, b, y0, h):
    y = y0
    tvals = [t]
    yvals = [y]
    while t < b:
        y += h*f(t, y)
        t += h
        tvals.append(t)
        yvals.append(y)
   return tvals, yvals
```

A quick version for systems:

```
def fwd_euler(f, t, b, y0, h):
    y = np.array(y0)  # copy
    tvals = [t]
    yvals = [[v] for v in y]
    while t < b:
        y += h*f(t, y)
        t += h
        for k in range(len(y)):
            yvals[k].append(y[k])
        tvals.append(t)
    return tvals, yvals

# example f (must return a numpy array)!
def f(t, y):
    return np.array((y[0]*y[1], y[1]**2))
```

Euler's method: systems

```
def fwd_euler(f, t, b, y0, h):
    y = np.array(y0)  # copy
    tvals = [t]
    yvals = [[v] for v in y]
    while t < b:
        y += h*f(t, y)
        t += h
        for k in range(len(y)):
            yvals[k].append(y[k])
        tvals.append(t)
    return tvals, yvals
```

- You have to make a choice on the 'shape' of the return...
- Two options (e.g. for $\mathbf{y} = (x, y)$ in 2d)

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} [x_0, y_0] \\ [x_1, y_1] \\ \vdots \\ [x_n, y_n] \end{bmatrix}$$

- (You could use a numpy array, but it has no append...)

The pendulum: computation

Now back to the pendulum...

$$\theta'' = -\theta - 2\beta\theta'$$

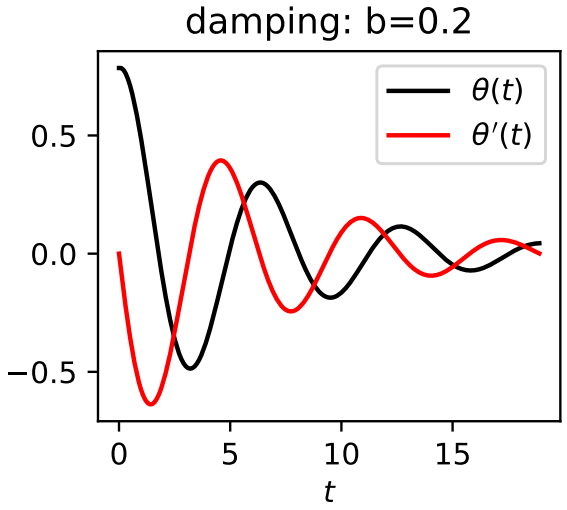
To compute, convert to a first order system.
Let $x_1 = \theta$ and $x_2 = \theta'$, Then

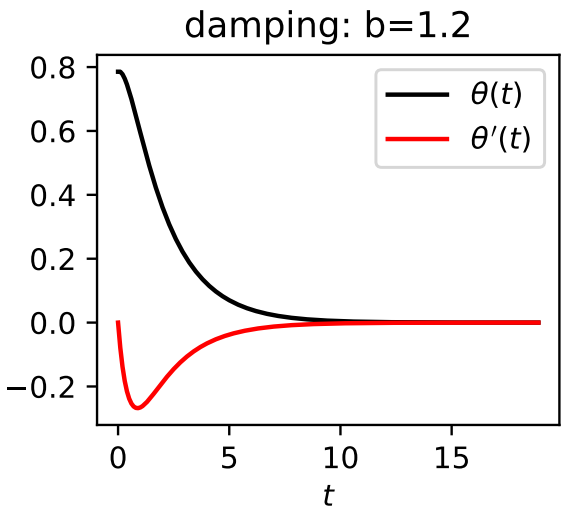
$$x_1' = x_2$$
$$x_2' = -x_1 - 2\beta x_2.$$

with initial position/velocity $(x_1(0), x_2(0))$

```
b = 0.1
def pend(t, x):
    return np.array((x[1], -x[0] - 2*b*x[1]))

# typical call
t, x = fwd_euler(pend, 0, 20, [1.0,0], 0.1)
plt.plot(t, x[0], '-k', t, x[1], '-b')
```





- Note that the output shape depends on implementation (here `x[0]` is $x_1(t)$).

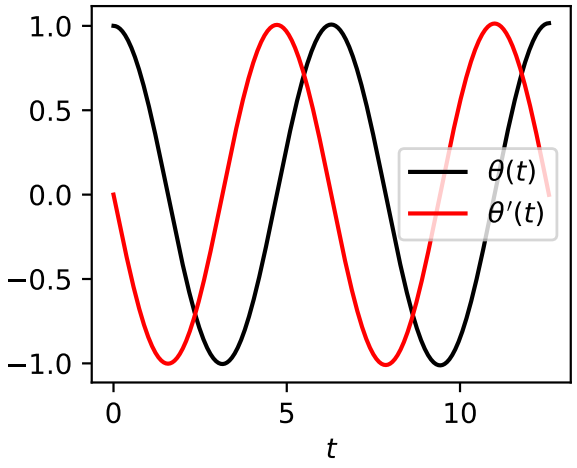
The pendulum: computation

It's also useful to plot a system in **phase space**.

- Plot in the (θ, θ') plane (plot $\theta'(t)$ vs. $\theta(t)$)
- Quick example: simple harmonic motion... ($\beta = 0$ case)

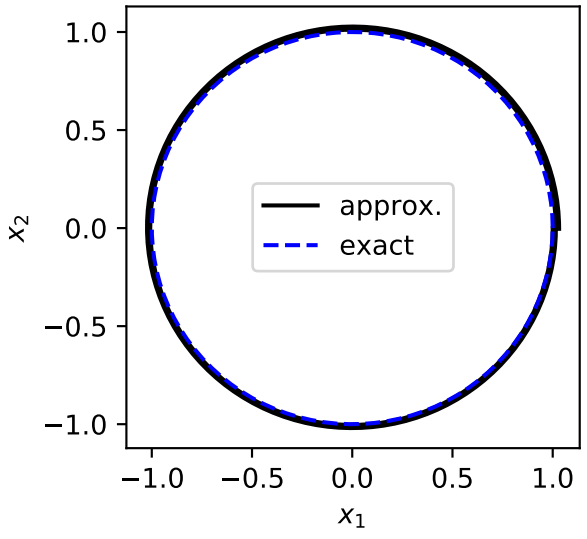
$$\theta'' = -\theta \implies x_1' = x_2,\ x_2' = -x_1$$

- Plug in $e^{rt} \implies r = \pm i$
- Solutions oscillate - $(theta, \theta')$ is a circle!



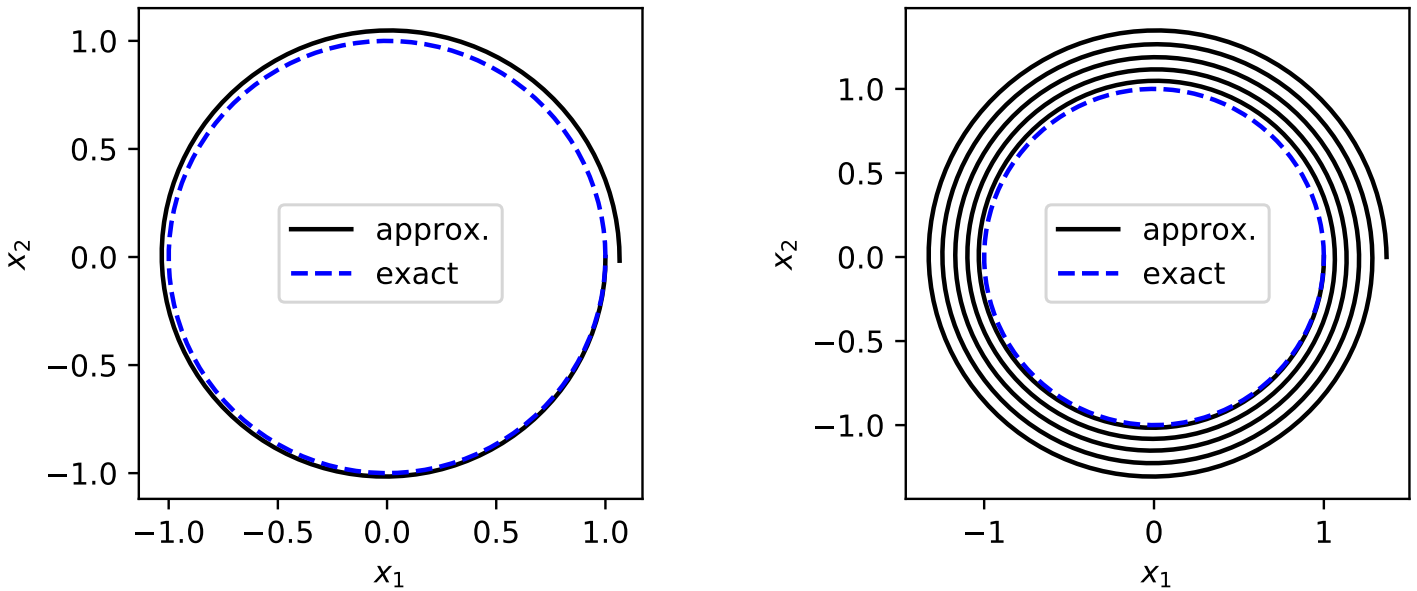

```
def f(t, x):
    return np.array((x[1], -x[0]))

# typical call
pos = 1.0  # initial displacement
vel = 0  # initial velocity
t, x = fwd_euler(f, 0, 10, [pos, vel], 0.1)
plt.plot(x[0], x[1], '-k')
```



The pendulum: computation

Be careful with the choice of h...
Euler's method may not behave the same way as the true solution!



(Left: $h = 0.01$ up to $t = 2\pi$... right: $h = 0.01$ up to $t = 10\pi$)

The forced pendulum

Now consider the non-linear pendulum equation

$$\theta'' = -\sin\theta - \beta\theta'.$$

When θ is not small, this term is quite different!

- The pendulum may swing 'over' the top (θ is really periodic)
- More complex behavior - but not more complex numerics!
- We can add a forcing to this as well:

$$\theta'' = -\sin\theta - \beta\theta + A\sin\omega_0 t$$

- A more accurate method is needed here to see the finer details without an unacceptably small h (see next slides).
- For certain parameters, the system is extremely sensitive
  (a big problem for not-so-accurate solvers like Euler's method!)

(see python code)

ODEs: more numerical methods

Let's go back to the integral derivation of Euler's method

$$y' = f(t, y)$$

integrated from t_n to t_{n+1} gives

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(s, y(s))\, ds$$

We got Euler's method from the (not very accurate) left hand rule

$$\int_a^b g(x)\, dx \approx (b-a)g(a).$$

Instead, let's use the trapezoidal rule

$$\int_a^b g(x)\, dx = \frac{b-a}{2}(g(a) + g(b))$$

to obtain

$$y_{n+1} = y_n + \frac{h}{2}(f(t_n, y_n) + f(t_{n+1}, y_{n+1}) + O(h^3)$$

using the result we derived for the error ($O(h^3)$ for an h-sized interval).

$$y_{n+1} = y_n + \frac{h}{2}(f(t_n, y_n) + f(t_{n+1}, y_{n+1}) + O(h^3)$$

- This yields the **trapezoidal method**

$$u_{n+1} = u_n + \frac{h}{2}(f(t_n, u_n) + f(t_{n+1}, u_{n+1}))$$

- Unlike Euler, the method is **implicit**: the RHS depends on the unknown u_{n+1}.
- $O(h^3)$ trunc. error $\implies$ $O(h^3) \cdot (1/h) = O(h^2)$ global error (second order!)
- We must use a zero-finder to solve for u_{n+1} at each step.
- (Why bother? The method has nice properties on some nasty ODEs...)

Can we avoid the implicit part? Idea: use an approximation.

- The order is preserved as long as the approximation is within $O(h^2)$
- We know how to do this: use Euler's method!

This idea yields the **explicit trapezoidal rule**

$$\tilde{u}_{n+1} = u_n + hf(t_n, u_n)$$

$$u_{n+1} = u_n + \frac{h}{2}(f(t_n, u_n) + f(t_{n+1}, \tilde{u}_{n+1}))$$

Runge-Kutta methods

More generally, we can construct 'one step methods' that

- Start at u_n
- compute some sub-steps $\tilde{u}$ from u_n and multiples of f at these sub-steps
- Add them up in the right way to get u_{n+1}

which are called **Runge-Kutta** methods.

For example, we can write the explicit rule from the previous slide as,

$$\begin{aligned} f_1 &= f(t_n, u_n) \\ f_2 &+ f(t_n + h, u_n + hf_1) \\ u_{n+1} &= u_n + \frac{h}{2} f_1 + \frac{h}{2} f_2. \end{aligned}$$

More sub-steps $\implies$ higher order methods, like 'classical' **RK4 method**:

$$\begin{aligned} f_1 &= f(t_n, u_n) \\ f_2 &= f(t_n + \frac{h}{2}, u_n + \frac{h}{2} f_1) \\ f_3 &= f(t_n + \frac{h}{2}, u_n + \frac{h}{2} f_2) \\ f_4 &= f(t_n + h, y_n + hf_3) \\ u_{n+1} &= u_n + \frac{h}{6}(f_1 + 2f_2 + 2f_3 + f_4) \end{aligned}$$

Runge-Kutta methods

Classical RK4:

$$
\begin{aligned}
f_1 &= f(t_n, u_n) \\
f_2 &= f(t_n + \frac{h}{2}, u_n + \frac{h}{2} f_1) \\
f_3 &= f(t_n + \frac{h}{2}, u_n + \frac{h}{2} f_2) \\
f_4 &= f(t_n + h, y_n + h f_3) \\
u_{n+1} &= u_n + \frac{h}{6}(f_1 + 2f_2 + 2f_3 + f_4)
\end{aligned}
$$

- RK4 is fourth order (!) and a good method to use with a fixed step size h.
- (More or less) strictly better than Euler's method
- Similar methods are better for changing h (e.g. the **Runge-Kutta-Fehlberg** method, which has a similar form and is used in `scipy.integrate`).
- RK methods are great general purpose solvers - good accuracy, easy to implement, easy to implement for systems...
- The catch: for some ODEs, there are estrictions on h that can be bad...