



Computational Physics (PHYS6350)

Lecture 9: Ordinary Differential Equations

$$\frac{dx}{dt} = f(x, t),$$

February 16, 2023

Instructor: Volodymyr Vovchenko (vvovchenko@uh.edu)

Course materials: <https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Ordinary Differential Equations (ODE)

First-order ordinary differential equation (ODE) is an equation of the form

$$\frac{dx}{dt} = f(x, t),$$

with initial condition

$$x(t = 0) = x_0$$

This determines the $x(t)$ dependence at $t > 0$.

In many physical applications t plays the role of the time variable (classical mechanics problems), although this is not always the case.

References: Chapter 8 of *Computational Physics* by Mark Newman

When we need numerical methods for ODEs

The solution to an ODE

$$\frac{dx}{dt} = f(x, t), \quad x(t = 0) = x_0$$

can formally be written as

$$x(t) = x_0 + \int_0^t f[x(t'), t'] dt'$$

If f does not depend on x , the solution can be obtained through (numerical) integration

In some other cases the solution can be obtained through the separation of variables, e.g.

$$\frac{dx}{dt} = \frac{2x}{t}$$

In all other cases, the solution has to be obtained numerically.

Numerical methods for ODEs

Typically obtain the solution by taking small steps from $x(t)$ to $x(t+h)$

Characteristics:

- Explicit or implicit
 - **Explicit methods:** use $x(t)$ to calculate $x(t+h)$ directly
 - **Implicit methods:** have to solve a (non-linear) equation for $x(t+h)$
- Accuracy
 - Truncation error at each step is of order $O(h^n)$
 - Some schemes are explicitly time-reversal and/or conserve energy
 - Adaptive methods adjust the step size h to control the error to the desired accuracy
- Stability
 - Whether the accumulated error is bounded (that's where implicit methods shine)
- Consistency
 - Consistent methods reproduce the exact solution in the limit $h \rightarrow 0$

Euler's method

$$\frac{dx}{dt} = f(x, t),$$

Let us apply the Taylor expansion to express $x(t+h)$ in terms of $x(t)$:

$$x(t+h) = x(t) + h \frac{dx}{dt} + O(h^2) .$$

Given that $dx/dt = f(x,t)$ and neglecting the high-order terms in h we have

$$x(t+h) \approx x(t) + h f[x(t), t] \quad \textit{Euler method}$$

We can iteratively apply this relation starting from $t = 0$ to evaluate $x(t)$ at $t > 0$.

This is the essence of the **Euler method** -- the simplest method for solving ODEs numerically.

Error:

- Local (per time step): $O(h^2)$
- Global ($N=t_{end}/h$ time steps): $O(h)$

Euler's method

```
import numpy as np

def ode_euler_step(f, x, t, h):
    """Perform a single step h using Euler's scheme.

    Args:
        f: the function that defines the ODE.
        x: the value of the dependent variable at the present step.
        t: the present value of the time variable.
        h: the time step

    Returns:
        xnew: the value of the dependent variable at the step t+h
    """
    return x + h * f(x,t)

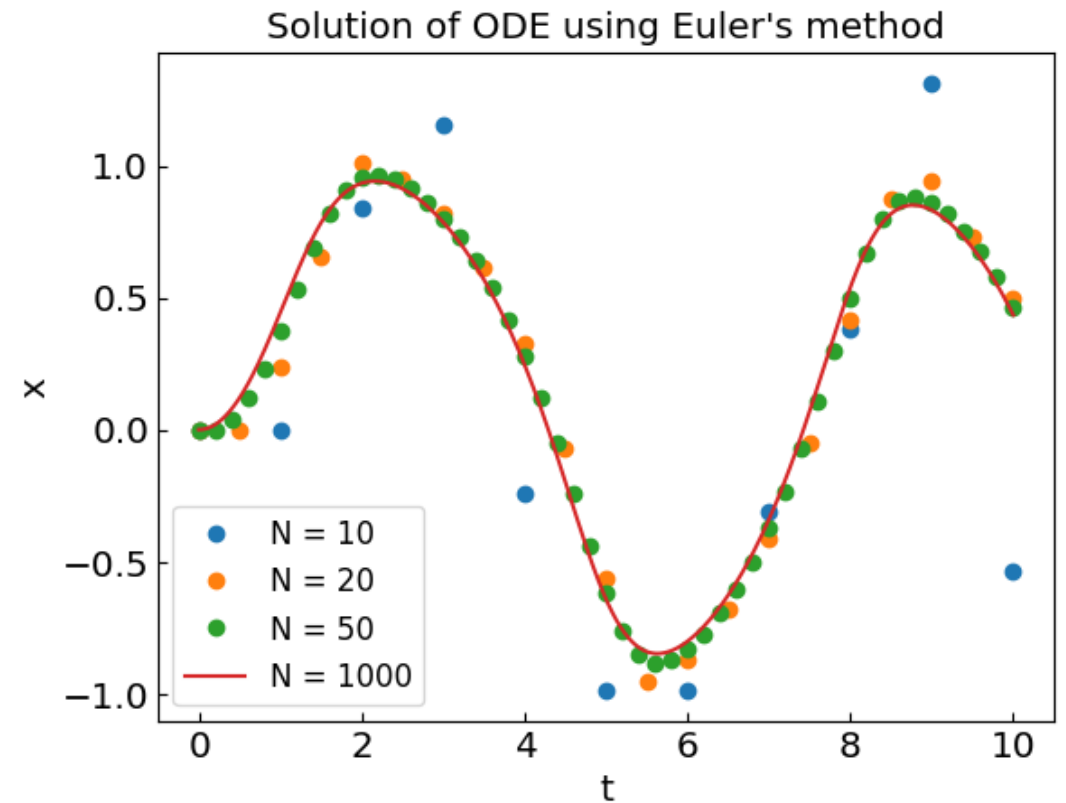
def ode_euler(f, x0, t0, h, nsteps):
    """Solve an ODE dx/dt = f(x,t) from t = t0 to t = t0 + h*nsteps using Euler's method.

    Args:
        f: the function that defines the ODE.
        x0: the initial value of the dependent variable.
        t0: the initial value of the time variable.
        h: the time step
        nsteps: the total number of Euler steps

    Returns:
        t,x: the pair of arrays corresponding to the time and dependent variables
    """

    t = np.zeros(nsteps + 1)
    x = np.zeros(nsteps + 1)
    x[0] = x0
    t[0] = t0
    for i in range(0, nsteps):
        t[i + 1] = t[i] + h
        x[i + 1] = ode_euler_step(f, x[i], t[i], h)
    return t,x
```

$$\frac{dx}{dt} = -x^3 + \sin t, \quad x(t=0) = 0.$$



Midpoint method (2nd order Runge-Kutta)

Euler's method essentially corresponds to approximating the derivative dx/dt with a *forward difference*

$$\frac{dx}{dt} = f(x, t) \approx \frac{x(t+h) - x(t)}{h} + \mathcal{O}(h).$$

Recall that central (midpoint) difference gives better accuracy

$$f(x, t + h/2) \approx \frac{x(t+h) - x(t)}{h} + \mathcal{O}(h^2).$$

therefore

$$x(t+h) = x(t) + hf[x(t+h/2), t+h/2] + \mathcal{O}(h^3)$$

How to calculate $x(t+h/2)$ in r.h.s? Use Euler's method $x(t+h/2) = x(t) + \frac{1}{2}hf(x, t) + \mathcal{O}(h^2)$

Therefore, $x(t+h) = x(t) + hf\left[x(t) + \frac{1}{2}hf(x, t), t + \frac{1}{2}h\right] + \mathcal{O}(h^3)$, which can be written in two steps

$$\begin{aligned} k_1 &= hf(x, t), & \text{trial step} \\ k_2 &= hf(x + k_1/2, t + h/2), & \text{real step} \\ x(t+h) &= x(t) + k_2. \end{aligned}$$

Midpoint method (2nd order Runge-Kutta)

```
def ode_rk2_step(f, x, t, h):
    """Perform a single step h using 2nd order Runge-Kutta scheme.

    Args:
        f: the function that defines the ODE.
        x: the value of the dependent variable at the present step.
        t: the present value of the time variable.
        h: the time step

    Returns:
        xnew: the value of the dependent variable at the step t+h
    """
    k1 = h * f(x,t)
    k2 = h * f(x + k1/2., t + h / 2.)
    return x + k2

def ode_rk2(f, x0, t0, h, nsteps):
    """Solve an ODE dx/dt = f(x,t) from t = t0 to t = t0 + h*nsteps using Euler's method.

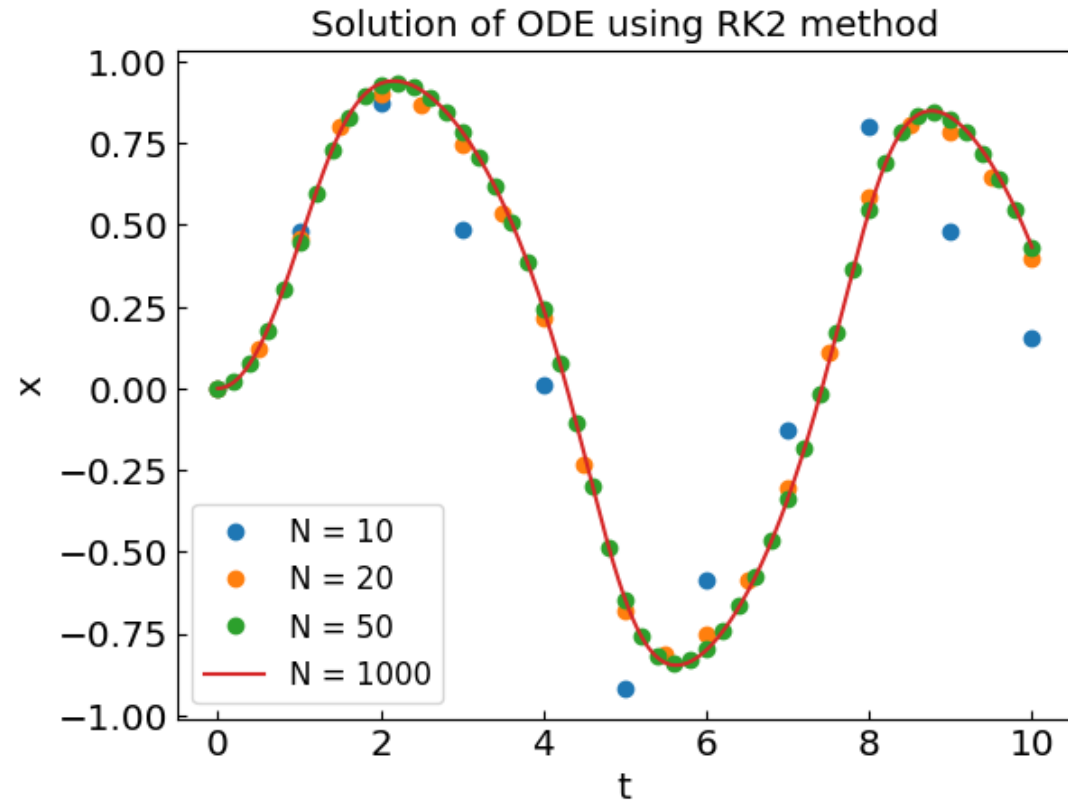
    Args:
        f: the function that defines the ODE.
        x0: the initial value of the dependent variable.
        t0: the initial value of the time variable.
        h: the time step
        nsteps: the total number of Euler steps

    Returns:
        t,x: the pair of arrays corresponding to the time and dependent variables
    """
    t = np.zeros(nsteps + 1)
    x = np.zeros(nsteps + 1)
    x[0] = x0
    t[0] = t0
    for i in range(0, nsteps):
        t[i + 1] = t[i] + h
        x[i + 1] = ode_rk2_step(f, x[i], t[i], h)
    return t,x
```

Error:

- Local (per time step): $O(h^3)$
- Global ($N=t_{end}/h$ time steps): $O(h^2)$

$$\frac{dx}{dt} = -x^3 + \sin t, \quad x(t=0) = 0.$$



Midpoint method (2nd order Runge-Kutta)

```
def ode_rk2_step(f, x, t, h):
    """Perform a single step h using 2nd order Runge-Kutta scheme.

    Args:
        f: the function that defines the ODE.
        x: the value of the dependent variable at the present step.
        t: the present value of the time variable.
        h: the time step

    Returns:
        xnew: the value of the dependent variable at the step t+h
    """
    k1 = h * f(x,t)
    k2 = h * f(x + k1/2., t + h /2.)
    return x + k2

def ode_rk2(f, x0, t0, h, nsteps):
    """Solve an ODE dx/dt = f(x,t) from t = t0 to t = t0 + h*nsteps using Euler's method.

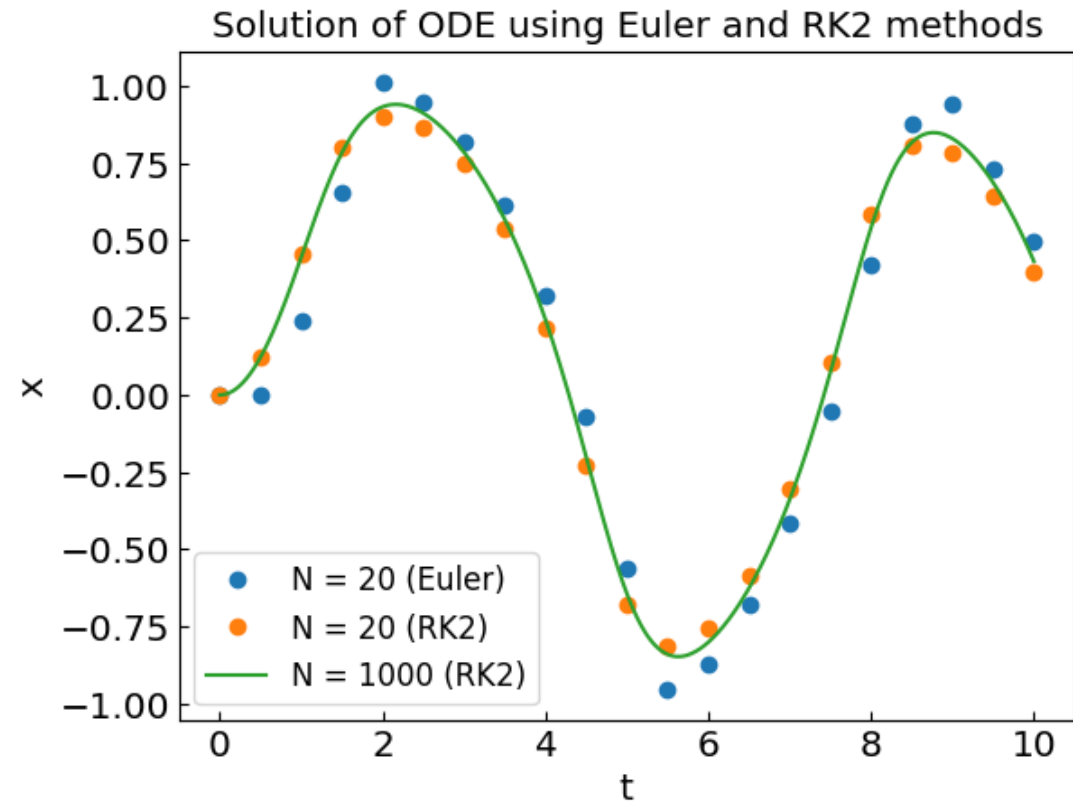
    Args:
        f: the function that defines the ODE.
        x0: the initial value of the dependent variable.
        t0: the initial value of the time variable.
        h: the time step
        nsteps: the total number of Euler steps

    Returns:
        t,x: the pair of arrays corresponding to the time and dependent variables
    """
    t = np.zeros(nsteps + 1)
    x = np.zeros(nsteps + 1)
    x[0] = x0
    t[0] = t0
    for i in range(0, nsteps):
        t[i + 1] = t[i] + h
        x[i + 1] = ode_rk2_step(f, x[i], t[i], h)
    return t,x
```

Error:

- Local (per time step): $O(h^3)$
- Global ($N=t_{end}/h$ time step): $O(h^2)$

$$\frac{dx}{dt} = -x^3 + \sin t, \quad x(t=0) = 0.$$



Classical 4th order Runge-Kutta method

The above logic can be generalized to cancel high-order error terms in various powers in h , requiring more and more evaluations of function $f(x,t)$ at intermediate steps.

The following classical 4th-order Runge-Kutta method is often considered a sweet spot.

It corresponds to the following scheme:

$$\begin{aligned}k_1 &= h f(x, t), \\k_2 &= h f(x + k_1/2, t + h/2), \\k_3 &= h f(x + k_2/2, t + h/2), \\k_4 &= h f(x + k_3, t + h), \\x(t + h) &= x(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) .\end{aligned}$$

Error:

- Local (per time step): $O(h^5)$
- Global ($N=t_{end}/h$ time steps): $O(h^4)$

The classical 4th-order Runge-Kutta method is a good first choice for solving physics ODEs.

Classical 4th order Runge-Kutta method

```
def ode_rk4_step(f, x, t, h):
    """Perform a single step h using 4th order Runge-Kutta method.

    Args:
        f: the function that defines the ODE.
        x: the value of the dependent variable at the present step.
        t: the present value of the time variable.
        h: the time step

    Returns:
        xnew: the value of the dependent variable at the step t+h
    """
    k1 = h * f(x,t)
    k2 = h * f(x + k1/2., t + h /2.)
    k3 = h * f(x + k2/2., t + h /2.)
    k4 = h * f(x + k3, t + h)
    return x + (k1 + 2. * k2 + 2. * k3 + k4) / 6.

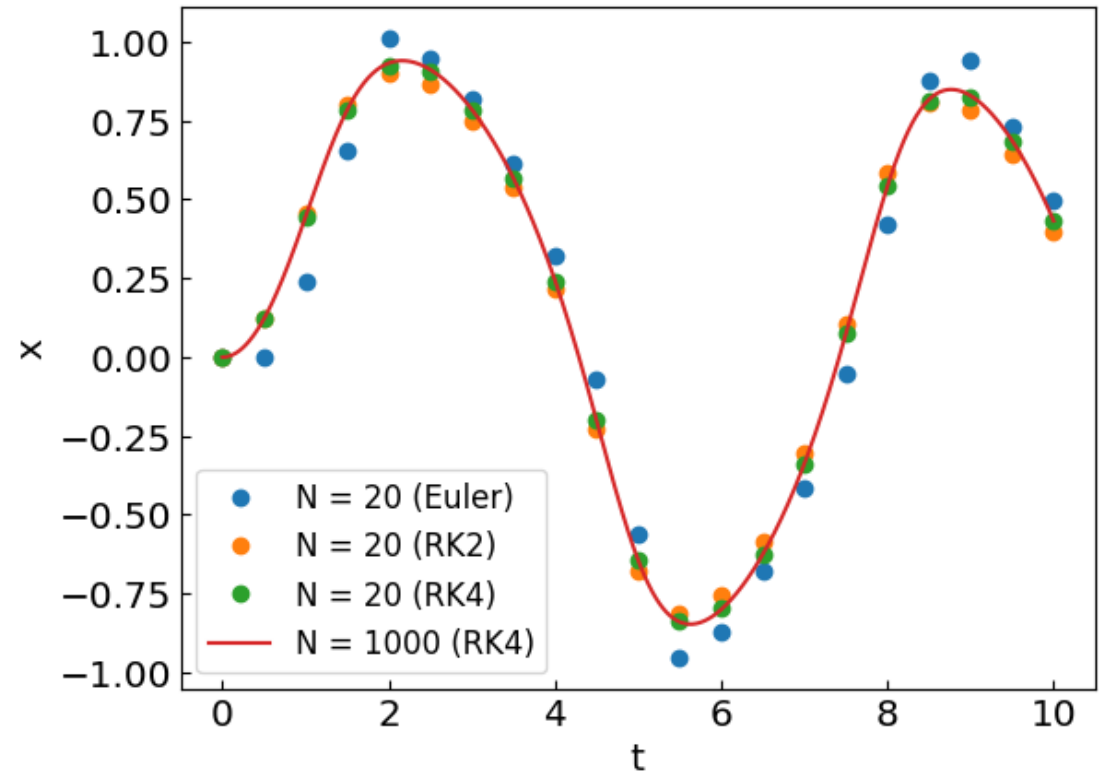
def ode_rk4(f, x0, t0, h, nsteps):
    """Solve an ODE dx/dt = f(x,t) from t = t0 to t = t0 + h*nsteps using 4th order Runge-Kutta method.

    Args:
        f: the function that defines the ODE.
        x0: the initial value of the dependent variable.
        t0: the initial value of the time variable.
        h: the time step
        nsteps: the total number of Euler steps

    Returns:
        t,x: the pair of arrays corresponding to the time and dependent variables
    """
    t = np.zeros(nsteps + 1)
    x = np.zeros(nsteps + 1)
    x[0] = x0
    t[0] = t0
    for i in range(0, nsteps):
        t[i + 1] = t[i] + h
        x[i + 1] = ode_rk4_step(f, x[i], t[i], h)
    return t,x
```

$$\frac{dx}{dt} = -x^3 + \sin t, \quad x(t=0) = 0.$$

Solution of ODE using Euler, RK2, and RK4 methods



Adaptive time step

$$\frac{dx}{dt} = f(x, t)$$

The choice of the time step is important to reach the desired accuracy/performance.

- h too large: the desired accuracy not reached
- h too small: we waste computing resources on unnecessary iterations
- Local truncation error itself is a function of time depending on the behavior of $f(x, t)$

Adaptive time step: make a local error estimate and adjust h to correspond to the desired accuracy

Ways to estimate the error:

- Make two small steps (h) and compare $x(t+2h)$ to the one from a single double step $2h$
- Use two methods of a different order and compare their results (e.g. [Runge-Kutta-Fehlberg method](#) RKF45)

Adaptive time step in RK4 using double step

Recall that the error for one RK4 time step h is of order ch^5 .

Let us take two RK4 steps h to approximate $x(t + 2h) \approx x_1$. Then,

$$x(t + 2h) \approx x_1 + 2ch^5$$

Now take single RK4 step $x(t + 2h) \approx x_2$ of length $2h$

$$x(t + 2h) \approx x_2 + 32ch^5$$

The local error estimate for a single RK4 time step h is then

$$\epsilon_{\text{RK4}} = |ch^5| = \frac{|x_1 - x_2|}{30}.$$

If the desired accuracy per unit time is δ , the desired accuracy per time step h' is

$$h'\delta = ch'^5$$

so the time step should be adjusted from h to h' as

$$h' = h \left(\frac{30h\delta}{|x_1 - x_2|} \right)^{1/4}.$$

- $h' > h$: our step size is too small, move on to $x(t+2h)$ and increase the step size to h'
- $h' < h$: our step size is too large, decrease step size to h' and try the current step again

RK4 method with adaptive step size

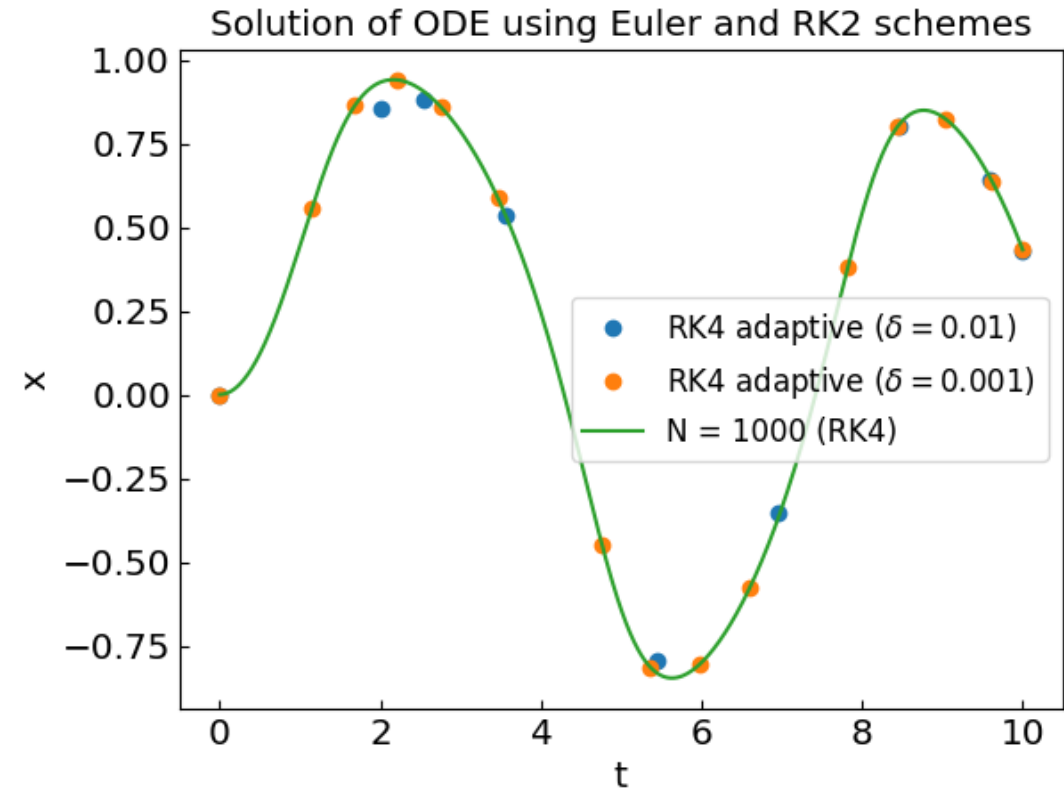
```
def ode_rk4_adaptive(f, x0, t0, h0, tmax, delta = 1.e-6):
    ts = [t0]
    xs = [x0]
    h = h0
    t = t0
    i = 0
    while (t < tmax):
        if (t + h >= tmax):
            ts.append(tmax)
            h = tmax - t
            xs.append(ode_rk4_step(f, xs[i], ts[i], h))
            t = tmax
            break

        x1 = ode_rk4_step(f, xs[i], ts[i], h)
        x1 = ode_rk4_step(f, x1, ts[i] + h, h)
        x2 = ode_rk4_step(f, xs[i], ts[i], 2*h)

        rho = 30. * h * delta / np.abs(x1 - x2)
        if rho < 1.:
            h *= rho**(1/4.)
        else:
            if (t + 2.*h) < tmax:
                xs.append(x1)
                ts.append(t + 2*h)
                t += 2*h
            else:
                xs.append(ode_rk4_step(f, xs[i], ts[i], h))
                ts.append(t + h)
                t += h
            i += 1
            h = min(2.*h, h * rho**(1/4.))

    return ts, xs
```

$$\frac{dx}{dt} = -x^3 + \sin t, \quad x(t=0) = 0.$$



Step size tends to decrease when dx/dt (the r.h.s) is large

Stability, stiff equations, and implicit methods

Consider the following ODE

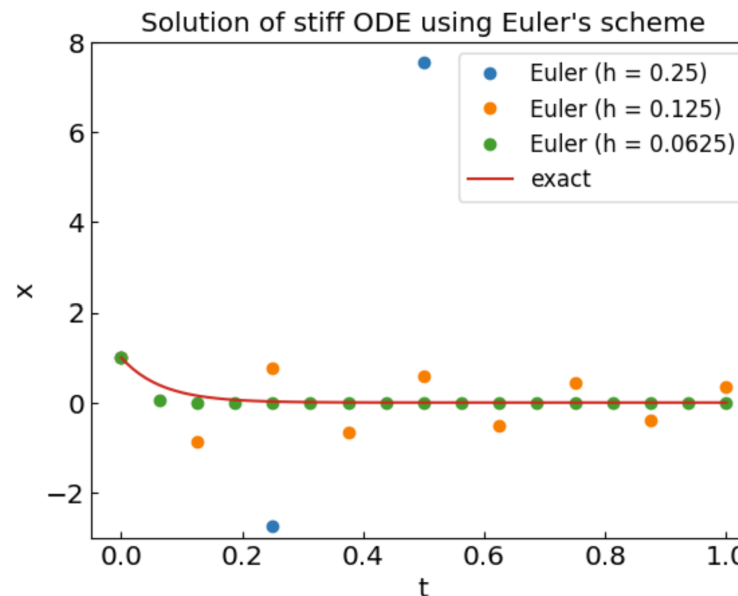
$$\frac{dx}{dt} = -15x, \quad \text{stiff equation}$$

with the initial condition $x(t=0)=1$.

The exact solution is of course $x(t) = e^{-15t}$ and goes to zero at large times.

Let us apply Euler's method
with $h=1/4, 1/8, 1/16$

Divergence for $h=1/4$!



Stability, stiff equations, and implicit methods

Consider the following ODE

$$\frac{dx}{dt} = -15x, \quad \text{stiff equation}$$

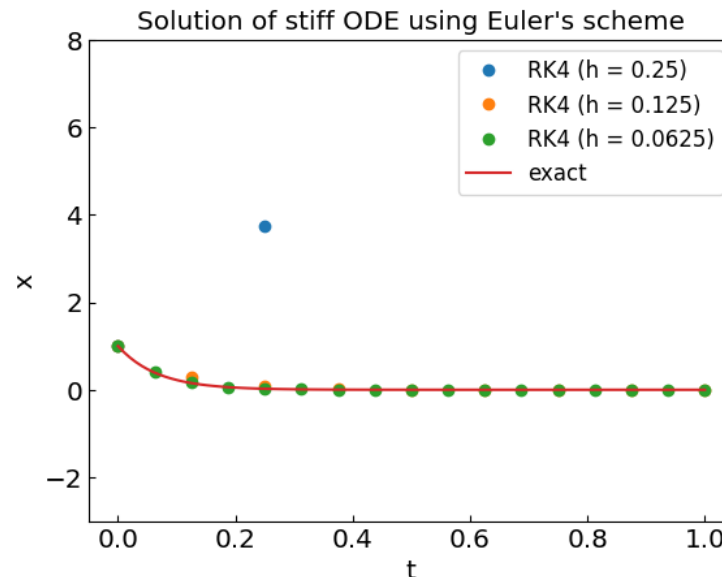
with $x(t=0)=1$.

The exact solution is of course $x(t) = e^{-15t}$ and goes to zero at large times.

Let us apply Euler's method
with $h=1/4, 1/8, 1/16$

Divergence for $h=1/4$!

RK4: better but still diverges for
 $h=1/4$



Euler methods and stiff equations

Recall that in Euler's method $x(t+h) = x(t) + h f(x,t)$

For $\frac{dx}{dt} = -15x$, we have $x_{n+1} = x_n - 15hx_n = (1 - 15h)x_n = (1 - 15h)^n x_0$, $x_n \equiv x(t + nh)$

If $|1-15h| > 1$, i.e. $h > 2/15$, the Euler method diverges!

Solution: *implicit methods*

Implicit Euler method: $x(t+h) = x(t) + hf[x(t+h), t+h]$

Our stiff equation: $x_{n+1} = x_n - 15hx_{n+1}$ thus $x_{n+1} = \frac{x_n}{1 + 15h} = \frac{x_0}{(1 + 15h)^n} \xrightarrow{n \rightarrow \infty} 0$ for all $h > 0$.

- Implicit methods are *more stable* than explicit methods
- But require solving non-linear equation for $x(t+h)$ at each step
- *Semi-implicit methods*: use one iteration of Newton's method to solve for $x(t+h)$

Other implicit methods: trapezoidal rule, family of implicit Runge-Kutta methods

Systems of Ordinary Differential Equations

System of N first-order ODE

$$\begin{aligned}\frac{dx_1}{dt} &= f_1(x_1, \dots, x_N, t), \\ \frac{dx_2}{dt} &= f_2(x_1, \dots, x_N, t), \\ &\dots \\ \frac{dx_N}{dt} &= f_N(x_1, \dots, x_N, t).\end{aligned}$$

Vector notation:

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}, t).$$

- all the methods we covered have the same structure when applied for systems of ODEs
- apply component by component

Systems of Ordinary Differential Equations

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}, t).$$

- Euler method

$$\mathbf{x}(t + h) = \mathbf{x}(t) + h \mathbf{f}[\mathbf{x}(t), t] .$$

- RK2

$$\begin{aligned} \mathbf{k}_1 &= h \mathbf{f}(\mathbf{x}, t), \\ \mathbf{k}_2 &= h \mathbf{f}(\mathbf{x} + \mathbf{k}_1/2, t + h/2), \\ \mathbf{x}(t + h) &= \mathbf{x}(t) + \mathbf{k}_2 . \end{aligned}$$

- RK4

$$\begin{aligned} \mathbf{k}_1 &= h \mathbf{f}(\mathbf{x}, t), \\ \mathbf{k}_2 &= h \mathbf{f}(\mathbf{x} + \mathbf{k}_1/2, t + h/2), \\ \mathbf{k}_3 &= h \mathbf{f}(\mathbf{x} + \mathbf{k}_2/2, t + h/2), \\ \mathbf{k}_4 &= h \mathbf{f}(\mathbf{x} + \mathbf{k}_3, t + h), \\ \mathbf{x}(t + h) &= \mathbf{x}(t) + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) . \end{aligned}$$

Systems of Ordinary Differential Equations

```
def ode_euler_multi(f, x0, t0, h, nsteps):
    """Multi-dimensional version of the Euler method.
    """

    t = np.zeros(nsteps + 1)
    x = np.zeros((len(t), len(x0)))
    t[0] = t0
    x[0,:] = x0
    for i in range(0, nsteps):
        t[i + 1] = t[i] + h
        x[i + 1,:] = ode_euler_step(f, x[i], t[i], h)
    return t,x
```

```
def ode_rk2_multi(f, x0, t0, h, nsteps):
    """Multi-dimensional version of the RK2 method.
    """

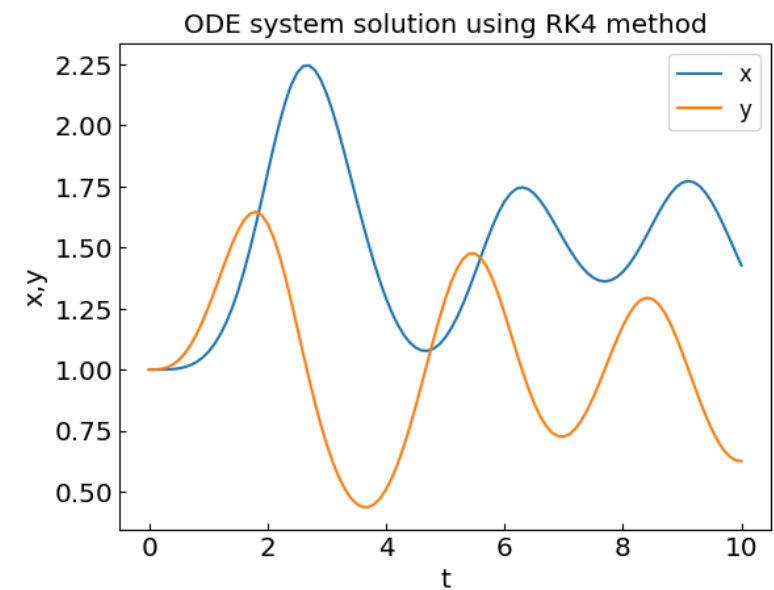
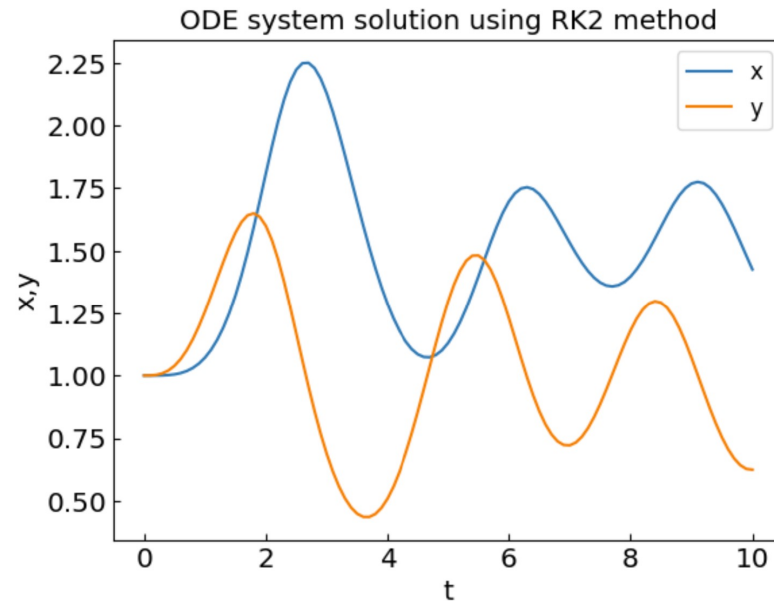
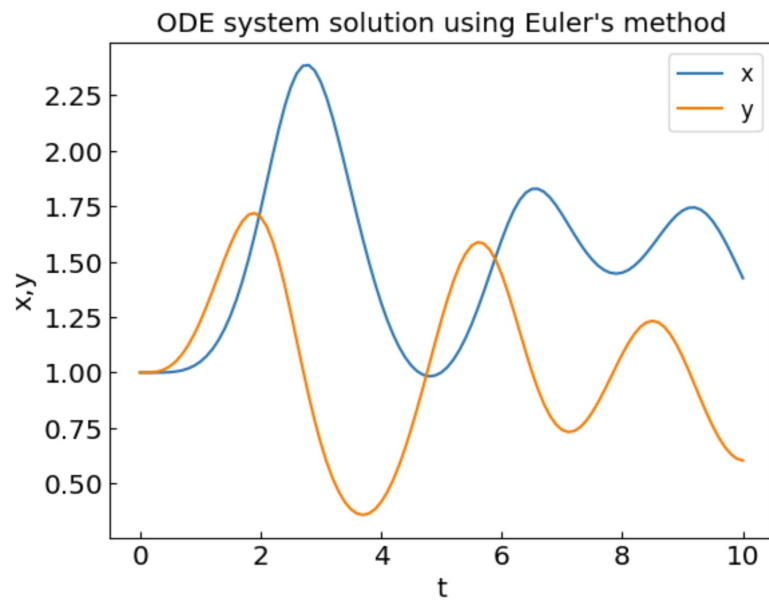
    t = np.zeros(nsteps + 1)
    x = np.zeros((len(t), len(x0)))
    t[0] = t0
    x[0,:] = x0
    for i in range(0, nsteps):
        t[i + 1] = t[i] + h
        x[i + 1] = ode_rk2_step(f, x[i], t[i], h)
    return t,x
```

```
def ode_rk4_multi(f, x0, t0, h, nsteps):
    """Multi-dimensional version of the RK4 method.
    """

    t = np.zeros(nsteps + 1)
    x = np.zeros((len(t), len(x0)))
    t[0] = t0
    x[0,:] = x0
    for i in range(0, nsteps):
        t[i + 1] = t[i] + h
        x[i + 1] = ode_rk4_step(f, x[i], t[i], h)
    return t,x
```

Systems of Ordinary Differential Equations: Example

$$\begin{aligned}\frac{dx}{dt} &= xy - x, \\ \frac{dy}{dt} &= y - xy + (\sin t)^2\end{aligned}$$



Systems of 2nd-order ODEs

Newton/Lagrange equations of motion are 2nd order systems of ODE

$$m_i \frac{d^2 x_i}{dt^2} = F_i(\{x_j\}, \{dx_j/dt\}, t)$$

A system of N second-order ODEs

$$\frac{d^2 \mathbf{x}}{dt^2} = \mathbf{f}(\mathbf{x}, d\mathbf{x}/dt, t),$$

can be written as a system of $2N$ first-order ODEs by denoting $\frac{d\mathbf{x}}{dt} = \mathbf{v}$

$$\begin{aligned} \frac{d\mathbf{x}}{dt} &= \mathbf{v}, \\ \frac{d\mathbf{v}}{dt} &= \mathbf{f}(\mathbf{x}, \mathbf{v}, t), \end{aligned}$$

and can be solved for $\mathbf{x}(t)$ and $\mathbf{v}(t)$ using standard methods

Example: Simple pendulum

The equation of motion for a simple pendulum reads

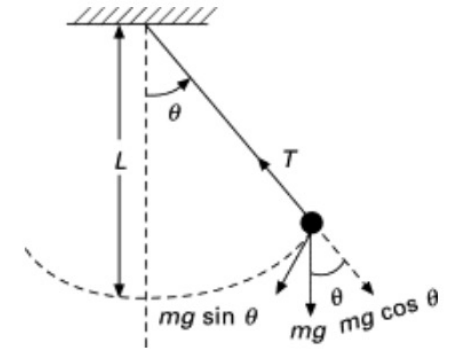
$$mL \frac{d^2\theta}{dt^2} = -mg \sin \theta.$$

denote $\frac{d\theta}{dt} = \omega$ and write a system of two first-order ODE

$$\begin{aligned} \frac{d\theta}{dt} &= \omega, \\ \frac{d\omega}{dt} &= -\frac{g}{L} \sin \theta, \end{aligned}$$

For small angles $\sin \theta \approx \theta$, an analytic solution exists

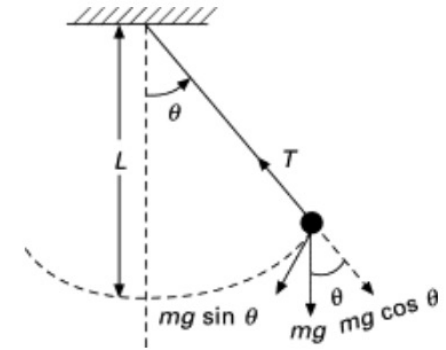
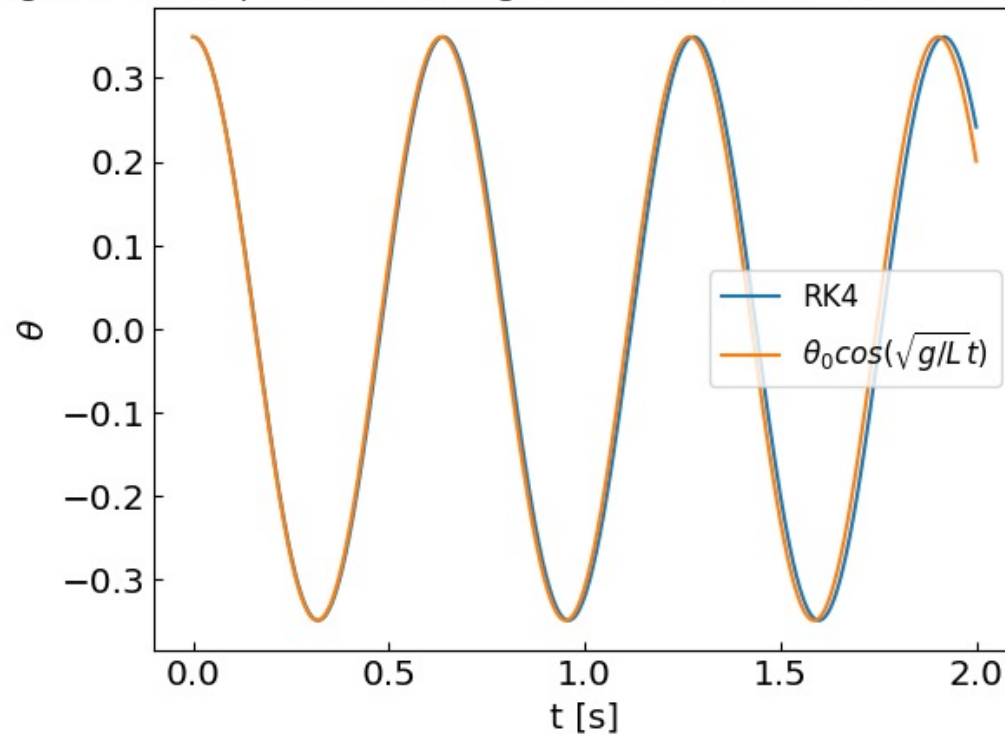
$$\theta(t) \approx \theta_0 \cos\left(\sqrt{\frac{g}{L}}t + \phi\right)$$



Example: Simple pendulum

Initially at rest at angle $\theta_0 = 20^\circ \approx 0.111\pi$ $L=0.1$ m, $g=9.81$ m/s²

Solving non-linear pendulum using RK4 method, $\theta_0 = 0.1111111111111111\pi$

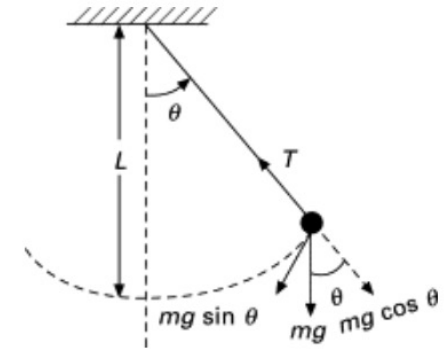
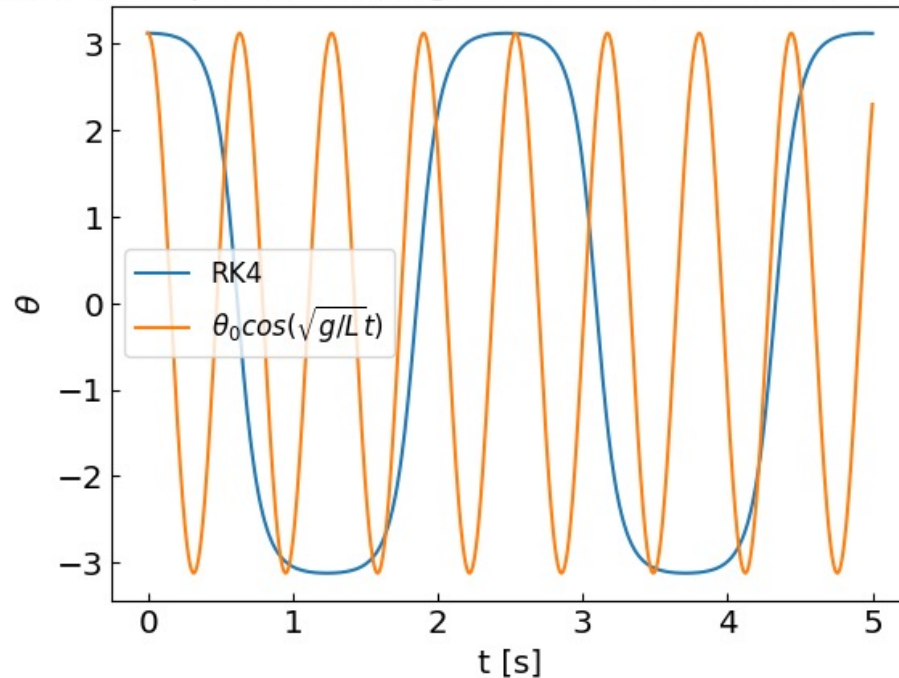


Linear regime at small angles

Example: Simple pendulum

Initially at rest at angle $\theta_0 = 179^\circ \approx 0.994\pi$ $L=0.1$ m, $g=9.81$ m/s²

Solving non-linear pendulum using RK4 method, $\theta_0 = 0.9944444444444445\pi$



Non-linear regime at large angles, approximate analytic solution fails