THE UNIVERSITY of TENNESSEE KNOXVILLE

Lecture 09: ODEs and PDEs

Sergei V. Kalinin

From derivatives to differential equations

1. **Newton's Laws**:
    1. **First Law (Inertia)**: An object remains in uniform motion unless acted upon by a force.
    2. **Second Law (Force and Acceleration)**: The force acting on an object is equal to the mass of that object times its acceleration ($F=ma$).
    3. **Third Law (Action and Reaction)**: For every action, there is an equal and opposite reaction.

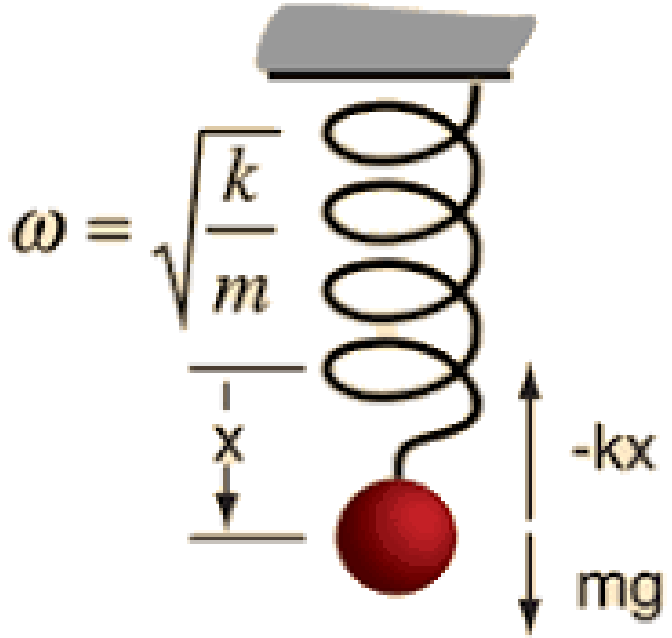
2. Acceleration (a) is the second derivative of position (x) with respect to time (t)

3. **Equation of motion:** $F= m\, dx^2/d^2t$.

4. **Example: Simple Harmonic Motion (SHM)**:
    1. Mass on a spring: $F = - k\, x$
    2. Differential equation for SHM:

    $dt^2/d^2x+\omega^2x=0$,

    where ω is the angular frequency.



Hooke's Law:

$$F_{spring} = -kx$$

http://hyperphysics.phy-astr.gsu.edu/hbase/shm2.html

Initial and boundary value problems

Find: $\mathbf{z}(t) = [x(t), y(t), \dot{x}(t), \dot{y}(t)]$

Boundry Conditions:

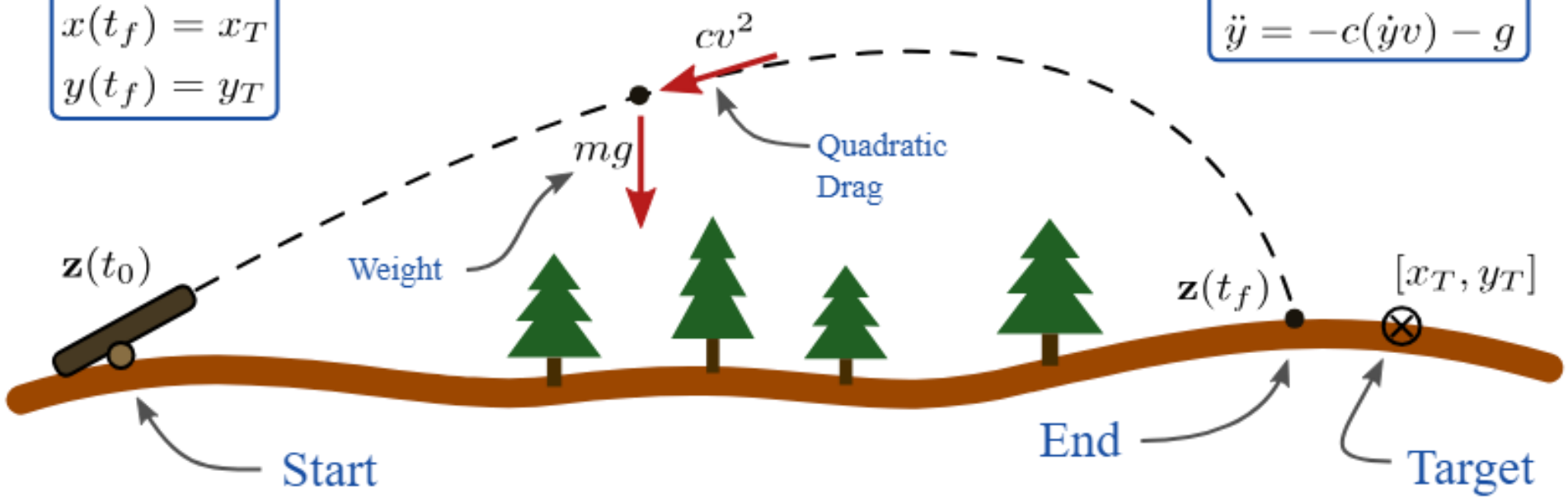
$$x(t_0) = 0$$
$$y(t_0) = 0$$
$$x(t_f) = x_T$$
$$y(t_f) = y_T$$

Cost Function:

$$J = \dot{x}(t_0)^2 + \dot{y}(t_0)^2$$

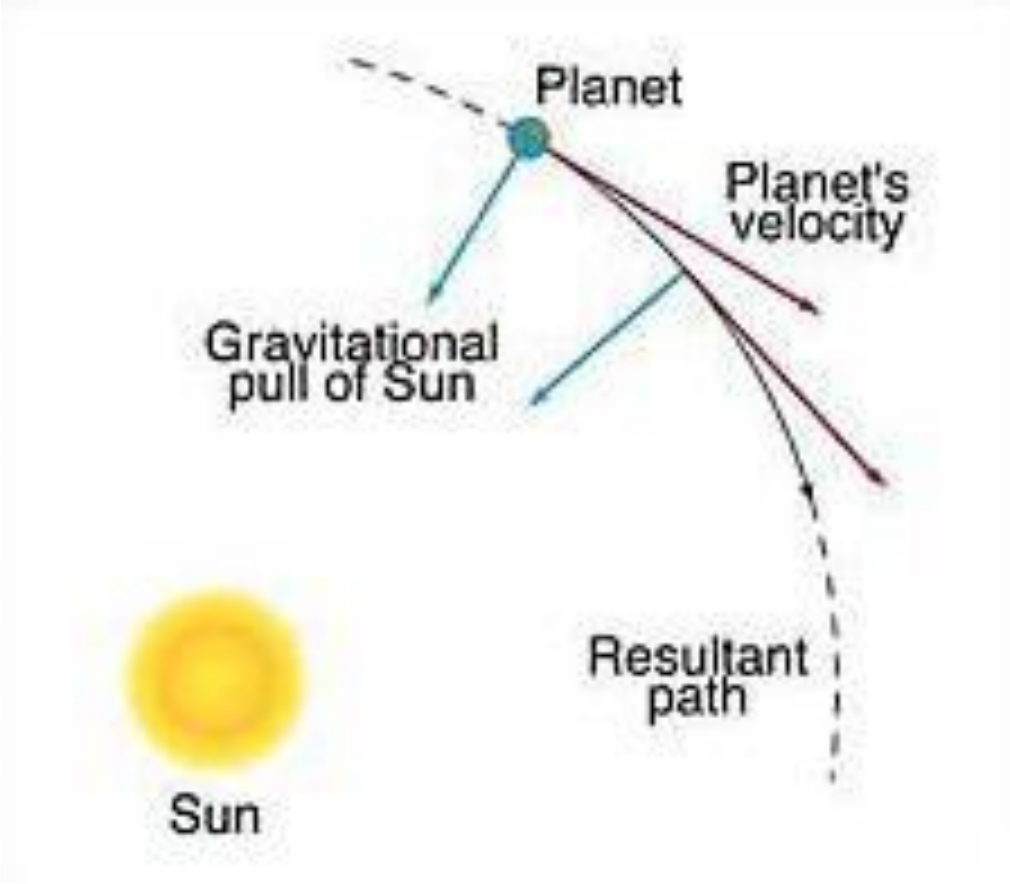
Dynamics:

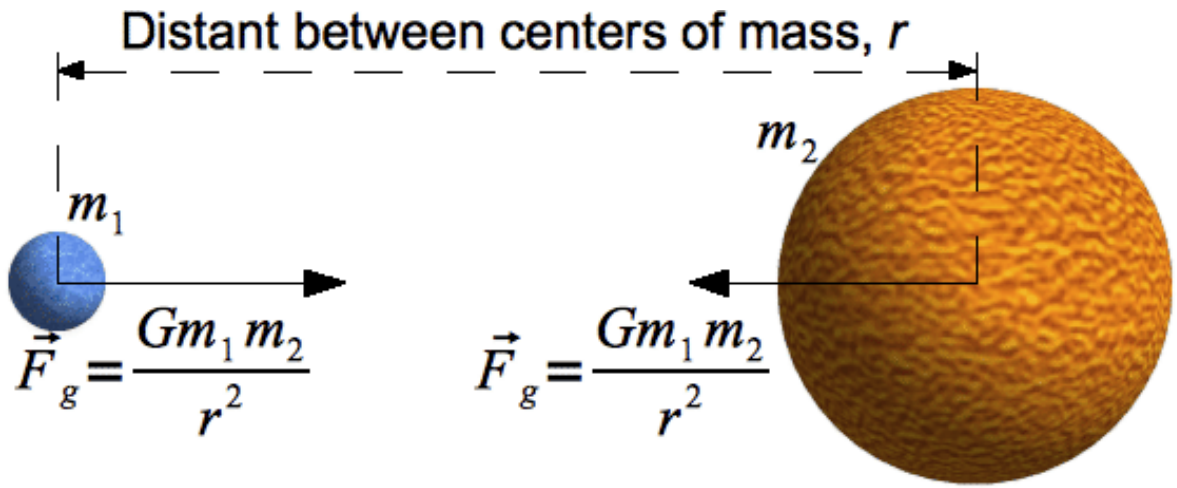
$$v = \sqrt{\dot{x}^2 + \dot{y}^2}$$
$$\ddot{x} = -c(\dot{x}v)$$
$$\ddot{y} = -c(\dot{y}v) - g$$



https://www.matthewpeterkelly.com/tutorials/trajectoryOptimization/canon.html

Stationary solutions





https://www.phy.olemiss.edu/~luca/astr/Topics-Introduction/Newton-N.html

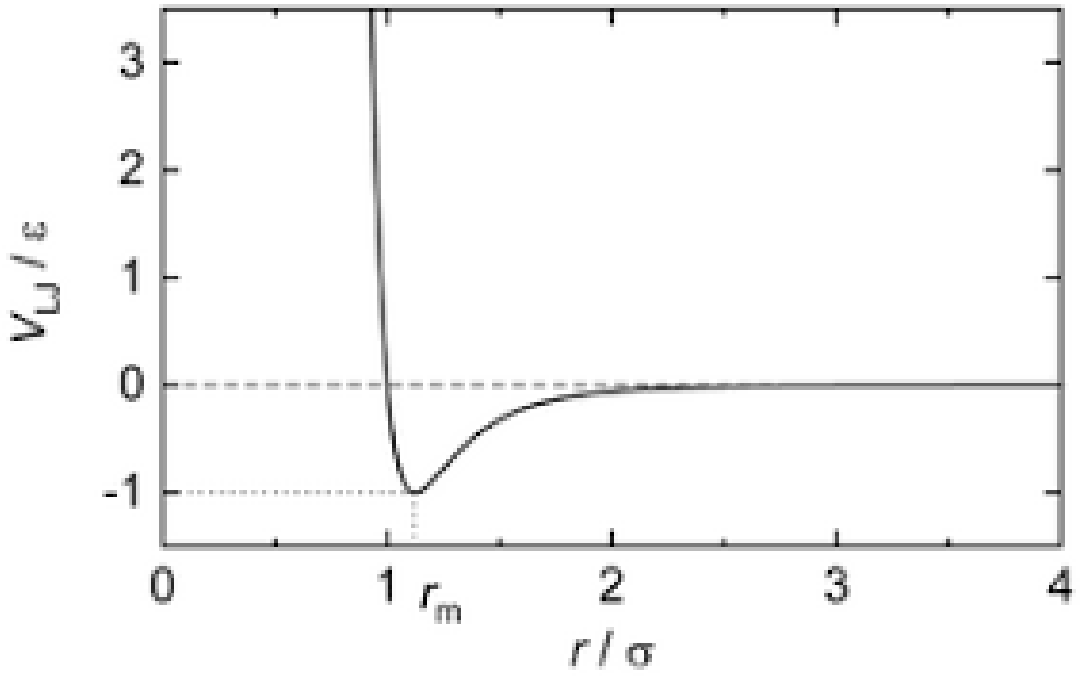
https://erikajanesite.wordpress.com/2017/09/24/225/

Molecular dynamics

- System of N particles with a pair potential
- Newton's equations of motion (classical *N*-body problem)

$$m\ddot{\mathbf{r}}_\mathbf{i} = -\sum_j \nabla_i V_{\mathrm{LJ}}^{ij}(|\mathbf{r}_\mathbf{i} - \mathbf{r}_\mathbf{j}|)$$

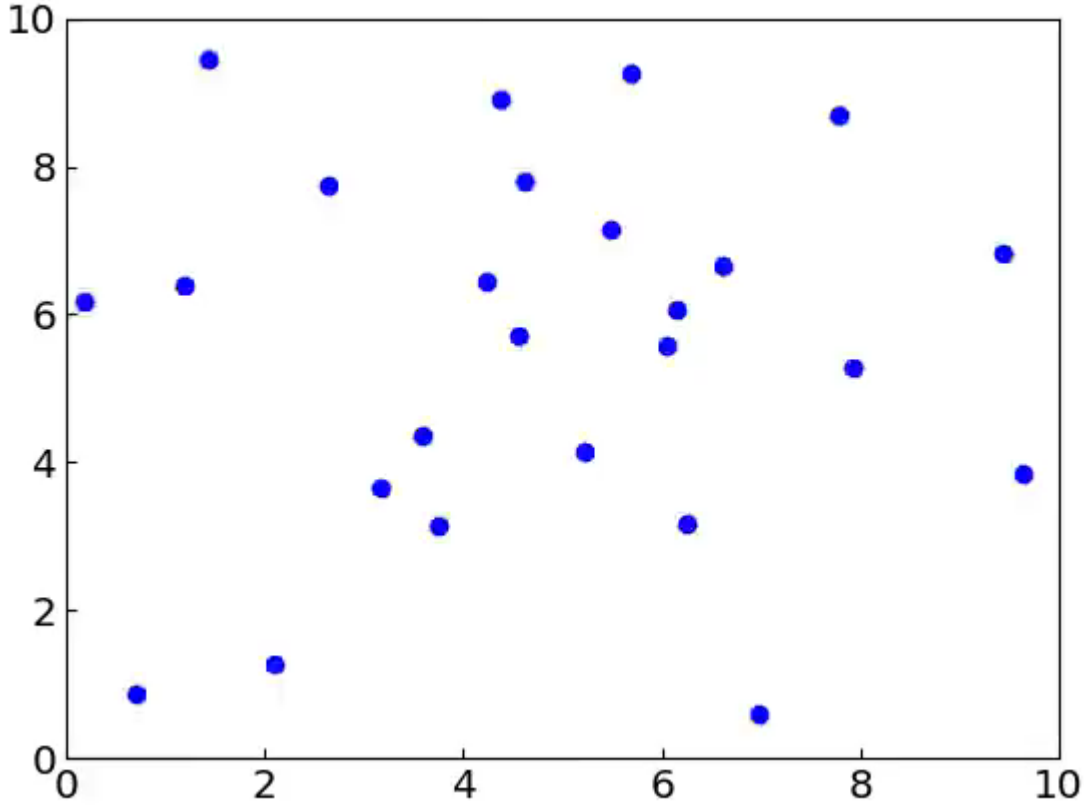
- Box simulation
  - Periodic boundary conditions
  - Minimum-image convention

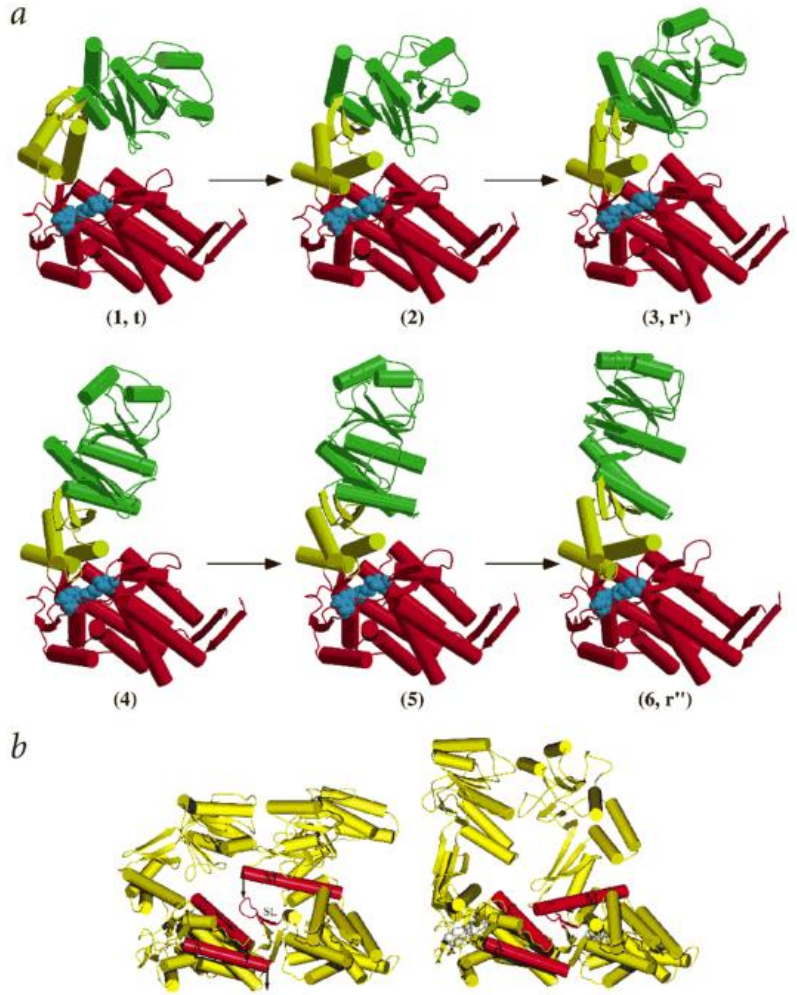


- If *N* is large enough, system can be characterized by macroscopic parameters
  - Energy-Volume-Number (UVN), microcanonical ensemble
  - Temperature-Volume-Number (TVN), canonical ensemble

- MD simulations give access to the **equation of state**

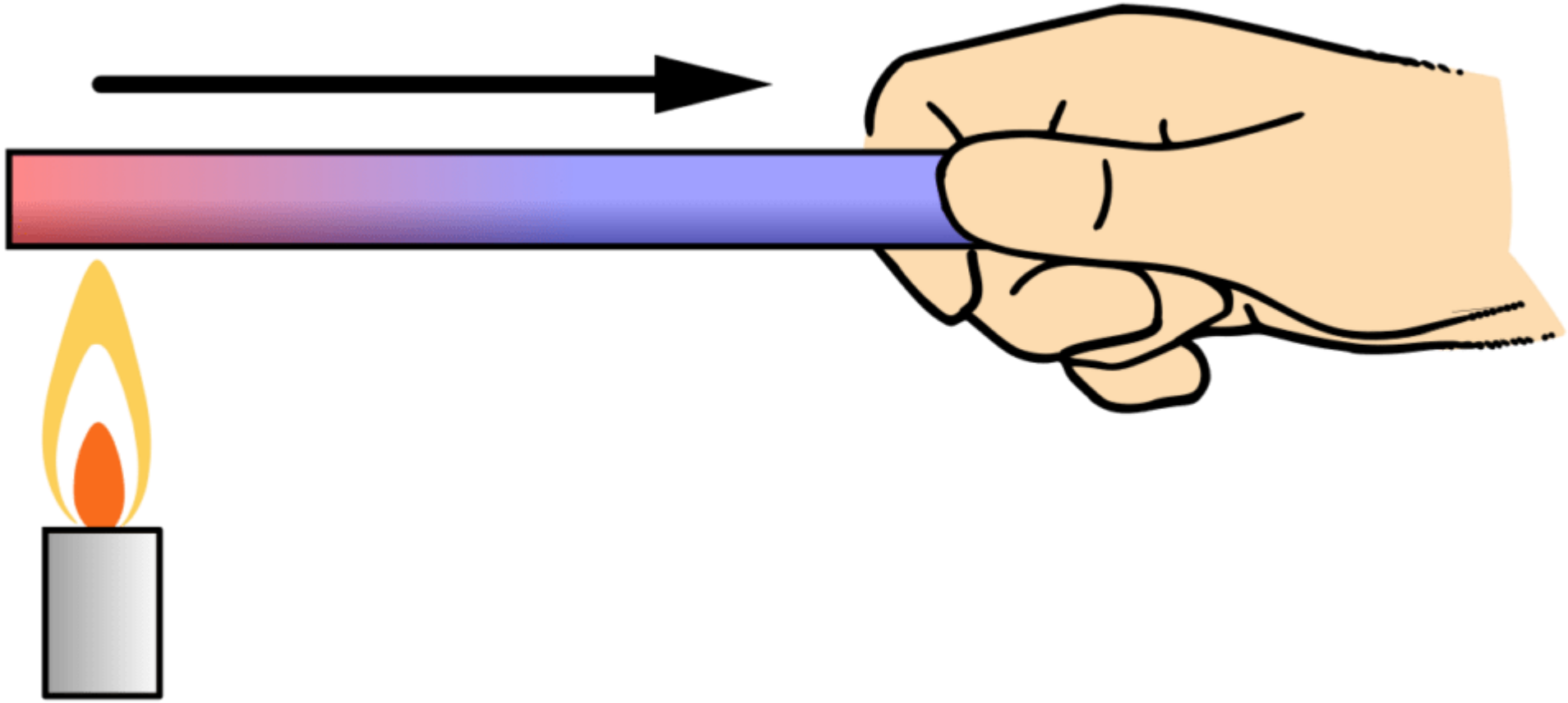
Molecular dynamics





https://www.nature.com/articles/nsb0902-646

Heat Transfer



https://chemicalengineeringworld.com/modes-of-heat-transfer/

Heat Transfer

1. Fourier's Law: The rate of heat transfer (q) through a material is proportional to the negative gradient of the temperature field (∇T) and the area (A) perpendicular to the direction of heat transfer, $q = -kA\nabla T$ where k is the thermal conductivity of the material

2. Conservation of Energy: For a given volume, the change in internal energy (U) over time (t) must equal the net heat flow into the volume minus the work done by the volume on its surroundings. In the absence of work and assuming constant density (ρ), this yields:

$$\partial U/\partial t = -\nabla \cdot q + \dot{q}$$

where $\dot{q}$ is the rate of heat generation per unit volume, and q is the heat flux vector

3. Relating Internal Energy to Temperature: Assuming the material's specific heat capacity (c_p) is constant, the internal energy change can be related to the temperature change:

$$U = \rho c_p T$$

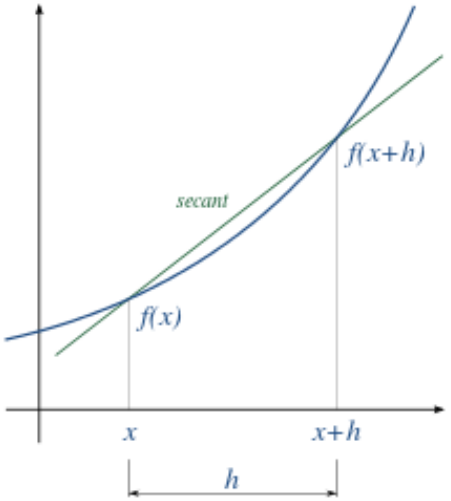
1.Substituting this into the conservation of energy equation and using Fourier's law, we get the heat equation for a homogeneous, isotropic material without internal heat generation as:

$$\rho c_p \partial T/\partial t = k\nabla^2 T$$

Numerical differentiation

Generic problem: evaluate

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$



We need numerical differentiation when

- Function f is known at a discrete set of points
- Too expensive/cumbersome to do directly
  - E.g. when f(x) itself is a solution to a complex web of non-linear equations, calculating f'(x) explicitly will require rewriting all the equations

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

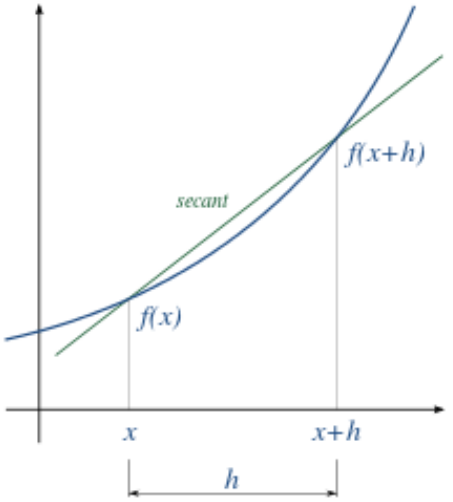
Forward difference

Simply approximate

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}$$

by

$$\frac{\mathrm{d}f}{\mathrm{d}x} \simeq \frac{f(x+h) - f(x)}{h}$$



where *h* is finite

Taylor theorem:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \ldots$$

gives the approximation error estimate of

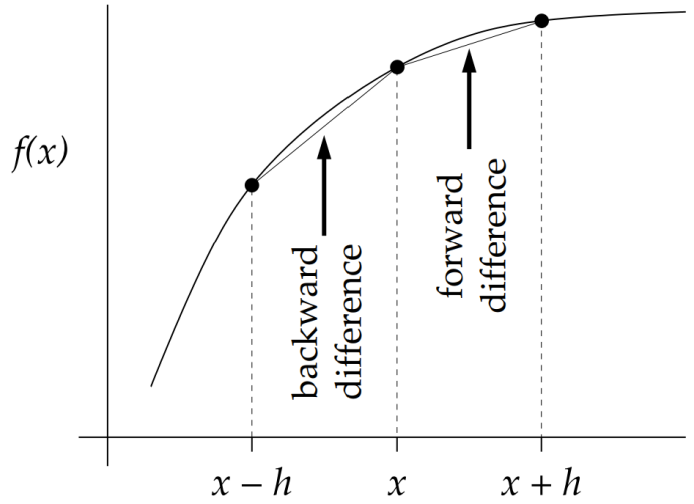
$$R_{\mathrm{forw}} = -\frac{1}{2}hf''(x) + \mathcal{O}(h^2)$$

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Backward difference

Backward difference

$$\frac{\mathrm{d}f}{\mathrm{d}x} \simeq \frac{f(x) - f(x-h)}{h}$$



Taylor theorem:

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) + \ldots$$

gives the approximation error estimate of

$$R_{\text{back}} = \frac{1}{2}hf''(x) + \mathcal{O}(h^2)$$

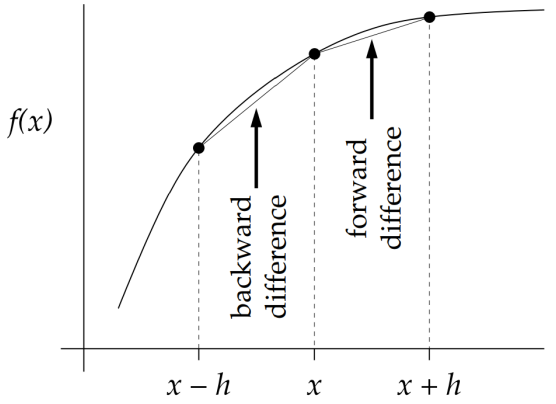
From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Central difference

Recall the forward and backward difference and their errors

$$\frac{\mathrm{d}f}{\mathrm{d}x} \simeq \frac{f(x+h)-f(x)}{h} \qquad \frac{\mathrm{d}f}{\mathrm{d}x} \simeq \frac{f(x)-f(x-h)}{h}$$

$$R_{\mathrm{forw}} = -\frac{1}{2}hf''(x) + \mathcal{O}(h^2) \qquad R_{\mathrm{back}} = \frac{1}{2}hf''(x) + \mathcal{O}(h^2)$$



Taking the average of the two cancels out the *O(h)* error term

central difference $$\frac{df}{dx} \simeq \frac{f(x+h)-f(x-h)}{2h}$$

Error estimate:

$$R_{\mathrm{cent}} = -\frac{f'''(x)}{6}h^2 + \mathcal{O}(h^3)$$

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

High-order central difference

To improve the approximation error use more than two function evaluations, e.g.

$$\frac{df}{dx} \simeq \frac{Af(x+2h) + Bf(x+h) + Cf(x) + Df(x-h) + Ef(x-2h)}{h} + O(h^4)$$

Determine *A,B,C,D,E* using Taylor expansion to cancel all terms up to h^4

$$\frac{df}{dx} \simeq \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h} + \frac{h^4}{30} f^{(5)}(x)$$

High-order terms:

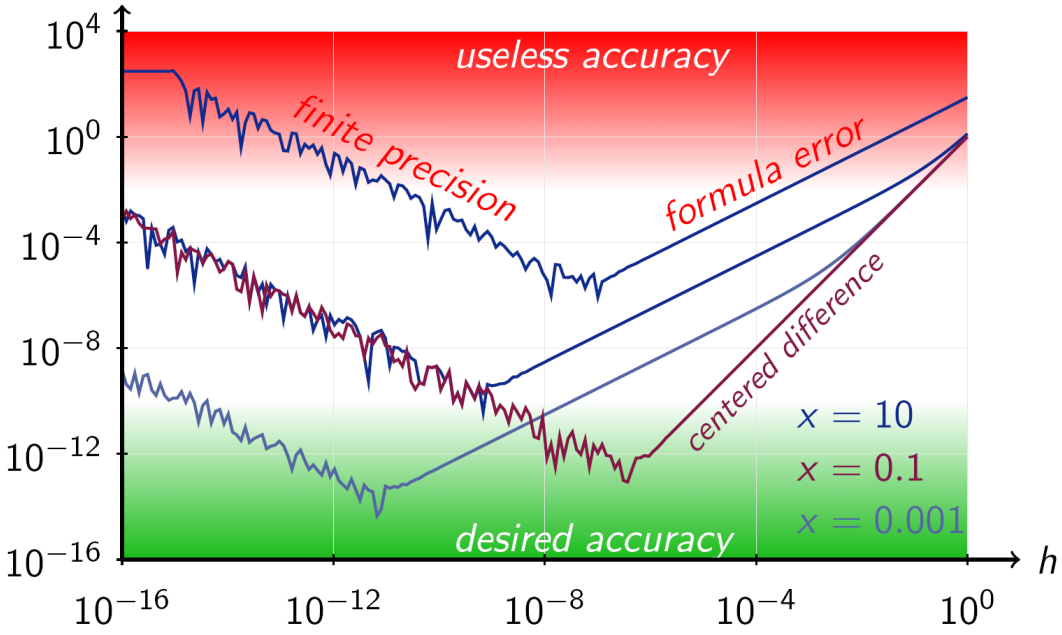
Derivative	Accuracy	−5	−4	−3	−2	−1	0	1	2	3	4	5
1	2					−1/2	0	1/2				
	4				1/12	−2/3	0	2/3	−1/12			
	6			−1/60	3/20	−3/4	0	3/4	−3/20	1/60		
	8		1/280	−4/105	1/5	−4/5	0	4/5	−1/5	4/105	−1/280	

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Balancing truncation and round-off errors

If *h* is too small, round-off errors become important

- cannot distinguish *x* and *x+h* and/or *f(x+h)* and *f(x)* with enough accuracy



Credit: Wikipedia

As a rule of thumb, if ε is machine precision and the truncation error is of order $O(h^n)$, then *h* should not be much smaller than $h \sim \sqrt[n+1]{\varepsilon}$

The higher the finite difference order is, the larger *h* should be

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Balancing truncation and round-off errors

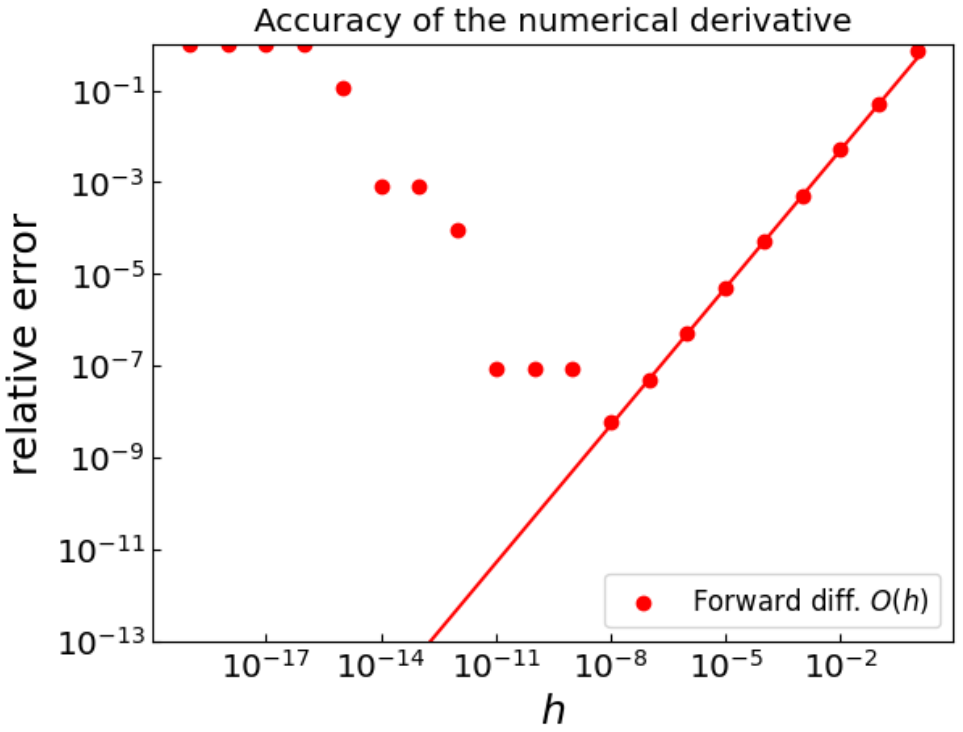
Let f(x) = exp(x)

Calculate the derivatives at x = 0

```
def f(x):
    return np.exp(x)

def df(x):
    return np.exp(x)
```

Forward difference O(h):

Optimal $h \sim \sqrt[2]{10^{-16}} \sim 10^{-8}$



From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Balancing truncation and round-off errors

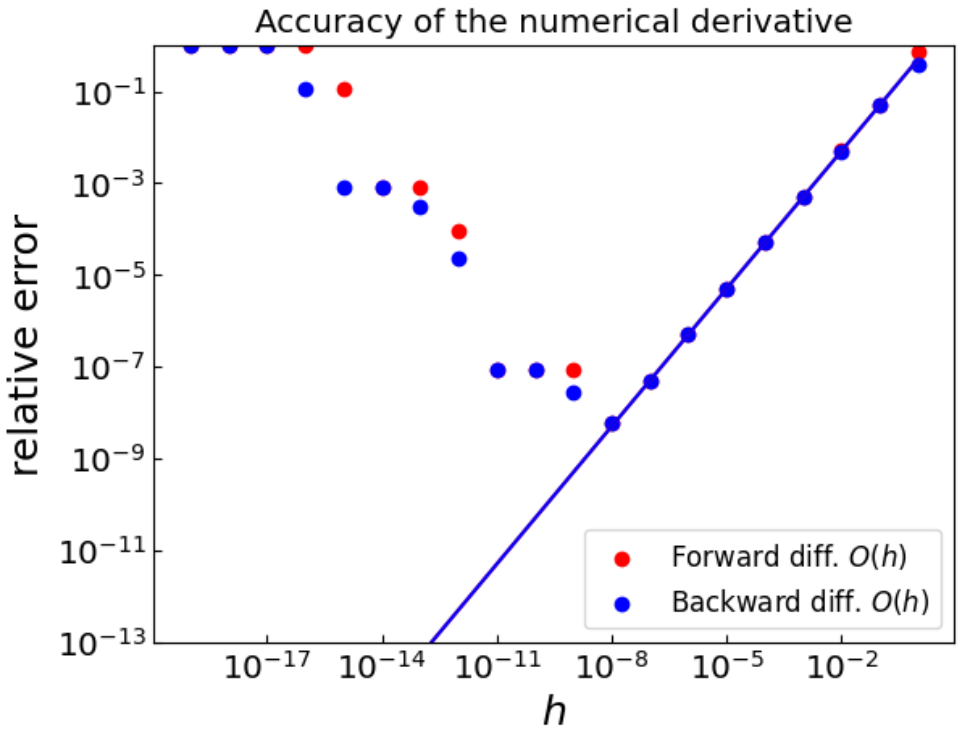
Let f(x) = exp(x)

Calculate the derivatives at x = 0

```
def f(x):
    return np.exp(x)

def df(x):
    return np.exp(x)
```

Backward difference O(h):

Optimal $h \sim \sqrt[2]{10^{-16}} \sim 10^{-8}$



From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Balancing truncation and round-off errors

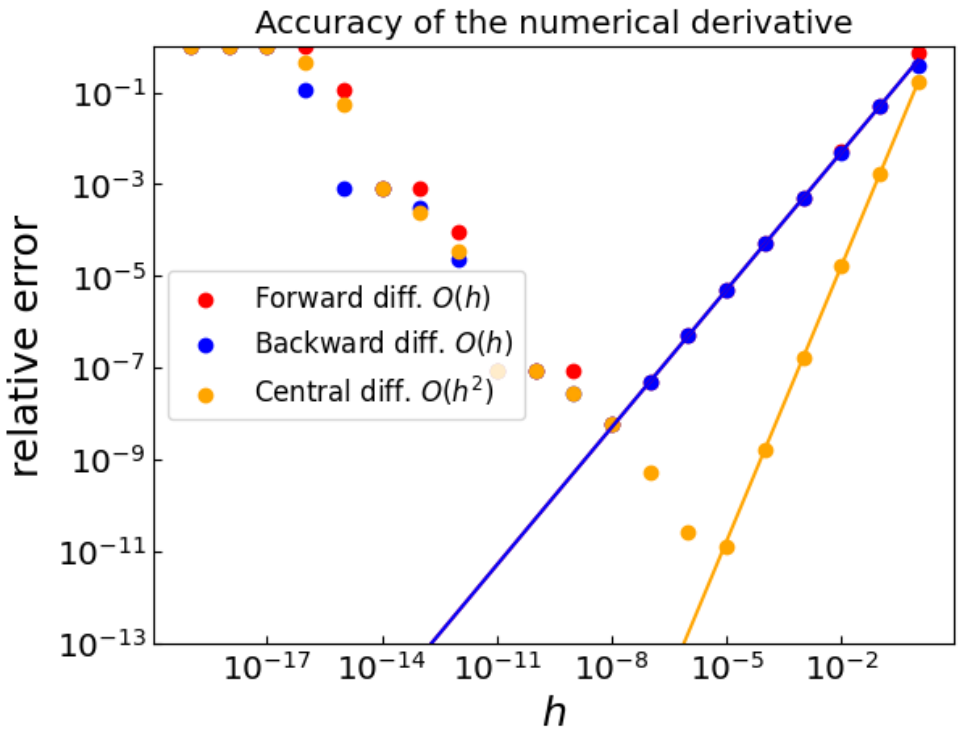
Let f(x) = exp(x)

Calculate the derivatives at x = 0

```
def f(x):
    return np.exp(x)

def df(x):
    return np.exp(x)
```

Central difference O(h²):

Optimal $h \sim \sqrt[3]{10^{-16}} \sim 10^{-5}$



From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Balancing truncation and round-off errors

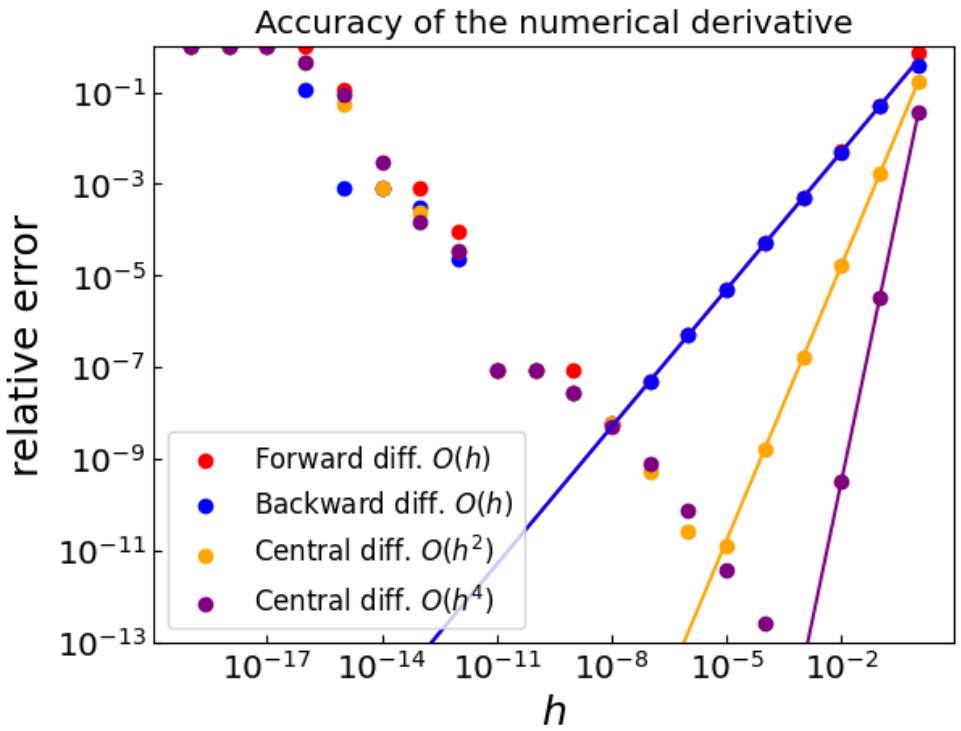
Let f(x) = exp(x)

Calculate the derivatives at x = 0

```
def f(x):
    return np.exp(x)

def df(x):
    return np.exp(x)
```

Central difference O(h⁴):

Optimal $h \sim \sqrt[5]{10^{-16}} \sim 10^{-3}$



From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

High-order derivatives

Central difference

$$\frac{df}{dx}(x) \simeq \frac{f(x+h/2) - f(x-h/2)}{h}$$

Now apply the central difference again to *f'(x+h/2)* and *f'(x-h/2)*

$$\begin{aligned} f''(x) &\simeq \frac{f'(x+h/2) - f'(x-h/2)}{h} \\ &= \frac{[f(x+h) - f(x)]/h - [f(x) - f(x-h)]/h}{h} \\ &= \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}. \end{aligned}$$

General formula (to order *h*)

$$f^{(n)}(x) = \frac{1}{h^n} \sum_{k=0}^{n} (-1)^k \binom{n}{k} f\left[x + (n/2 - k)h\right] + O(h^2)$$

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

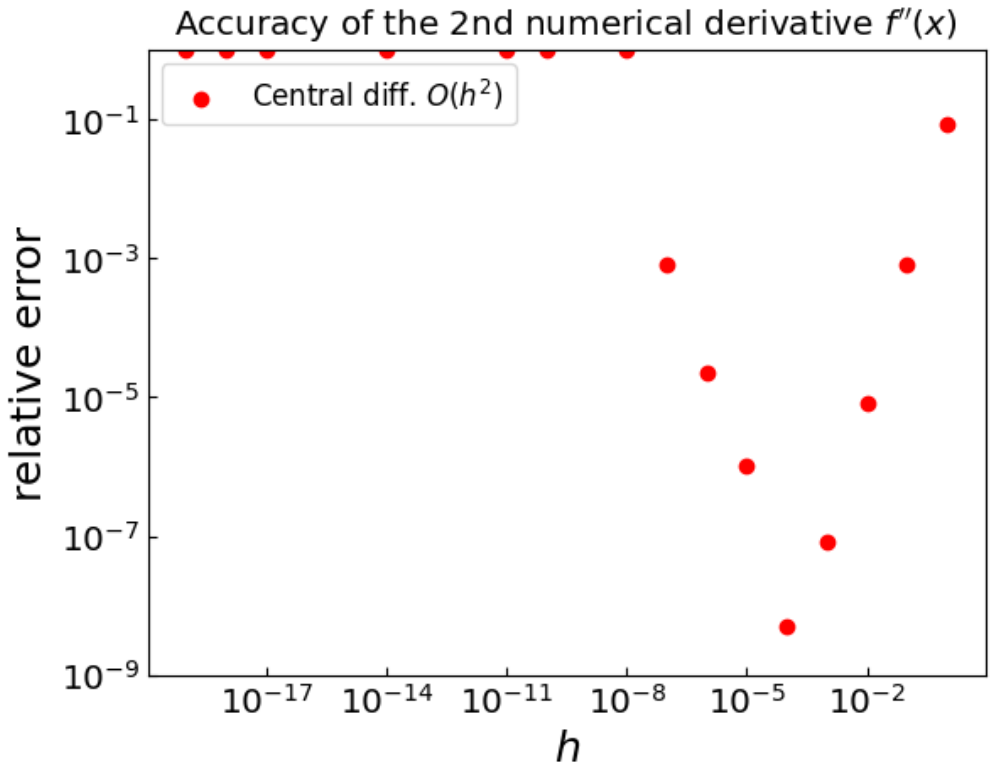
Second derivative

```python
def d2f_central(f,x,h):
    return (f(x+h) - 2*f(x) + f(x-h)) / (h**2)
```

f(x) = exp(x)

```python
def f(x):
    return np.exp(x)

def df(x):
    return np.exp(x)

def d2f(x):
    return np.exp(x)
```

Optimal $h \sim \sqrt[4]{10^{-16}} \sim 10^{-4}$



From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Partial derivatives

Let us have *f(x,y)*
Use central difference to calculate first-order derivatives

$$\frac{\partial f}{\partial x} = \frac{f(x+h/2,y) - f(x-h/2,y)}{h}$$
$$\frac{\partial f}{\partial y} = \frac{f(x,y+h/2) - f(x,y-h/2)}{h}$$

Reapply the central difference to calculate $\partial^2 f(x,y)/\partial x \partial y$

$$\frac{\partial^2 f}{\partial x \partial y} = \frac{f(x+h/2,y+h/2) - f(x-h/2,y+h/2) - f(x+h/2,y-h/2) + f(x-h/2,y-h/2)}{h^2}$$

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Summary: Numerical differentiation

- Forward/backward differences
  - Useful when we are given a grid of function values
  - Have limited accuracy (linear in *h*)

- Central difference
  - More precise than forward/backward differences (quadratic in *h*)
  - Gives f'(x) estimate at the midpoint of function evaluation points

- Higher-order formulas are obtained by using more than two function evaluations
  - Can be used when limited number of function evaluations available

- Straightforwardly extendable to high-order and partial derivatives

- Balance between truncation and round-off error must be respected
  - *h* should not be taken too small

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Numerical derivative and ordinary differential equations

Ordinary differential equation

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

with initial condition

$$f(x, t_0) = f_0$$

Use the forward difference to approximate dx/dt

$$\frac{dx}{dt} \approx \frac{x(t+h) - x(t)}{h}$$

gives the **Euler method** of solving the equation for x(t)

$$x(t+h) = x(t) + h\, f[x(t), t]$$

From the course by Volodymyr Vovchenko,
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.

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

When we need numerical methods for ODEs

The solution to an ODE

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

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.

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

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 $h \to 0$

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

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

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

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:

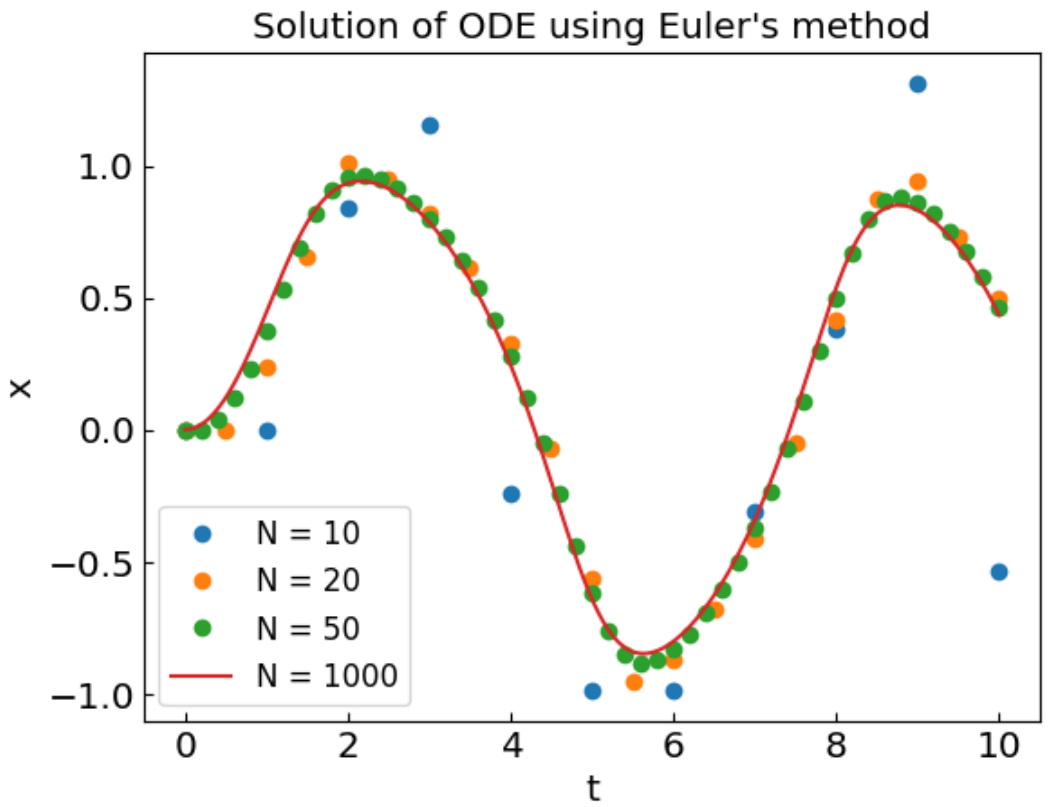
- Local (per time step): $O(h^2)$
- Global (N=t_{end}/h time steps): O(h)

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Euler's method

```python
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*steps 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
```

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



From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

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

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

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

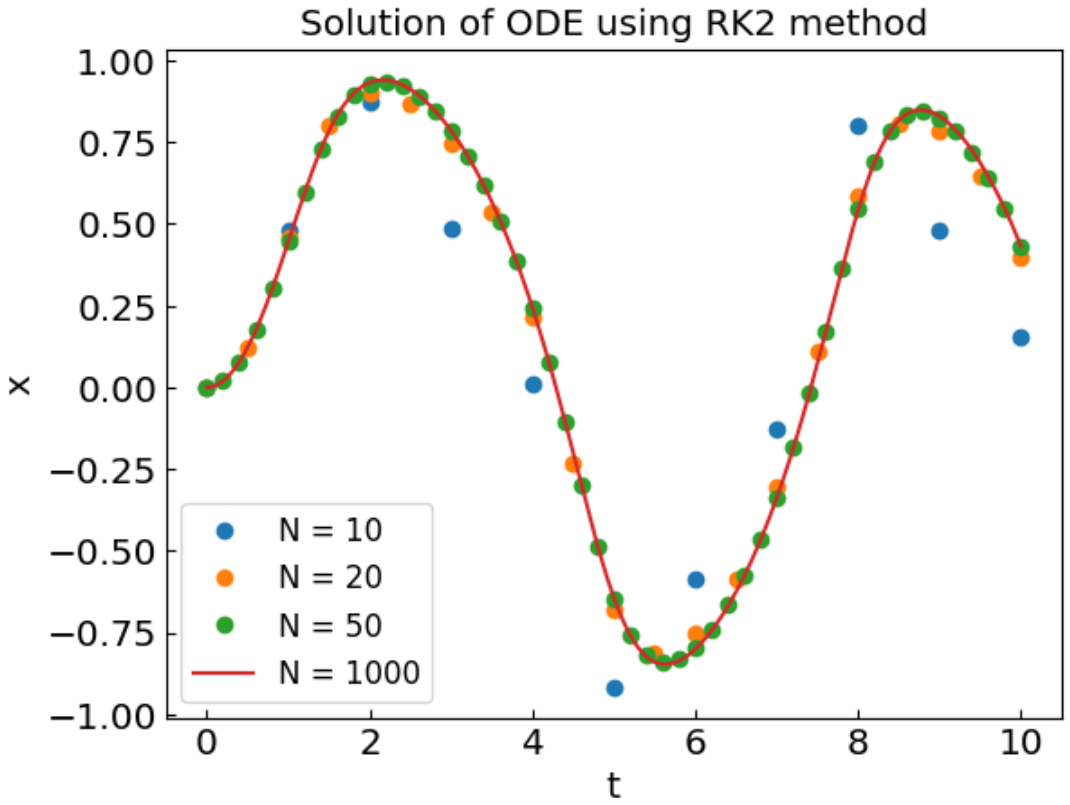
Midpoint method (2nd order Runge-Kutta)

```python
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*steps 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): $O(h^2)$

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



From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

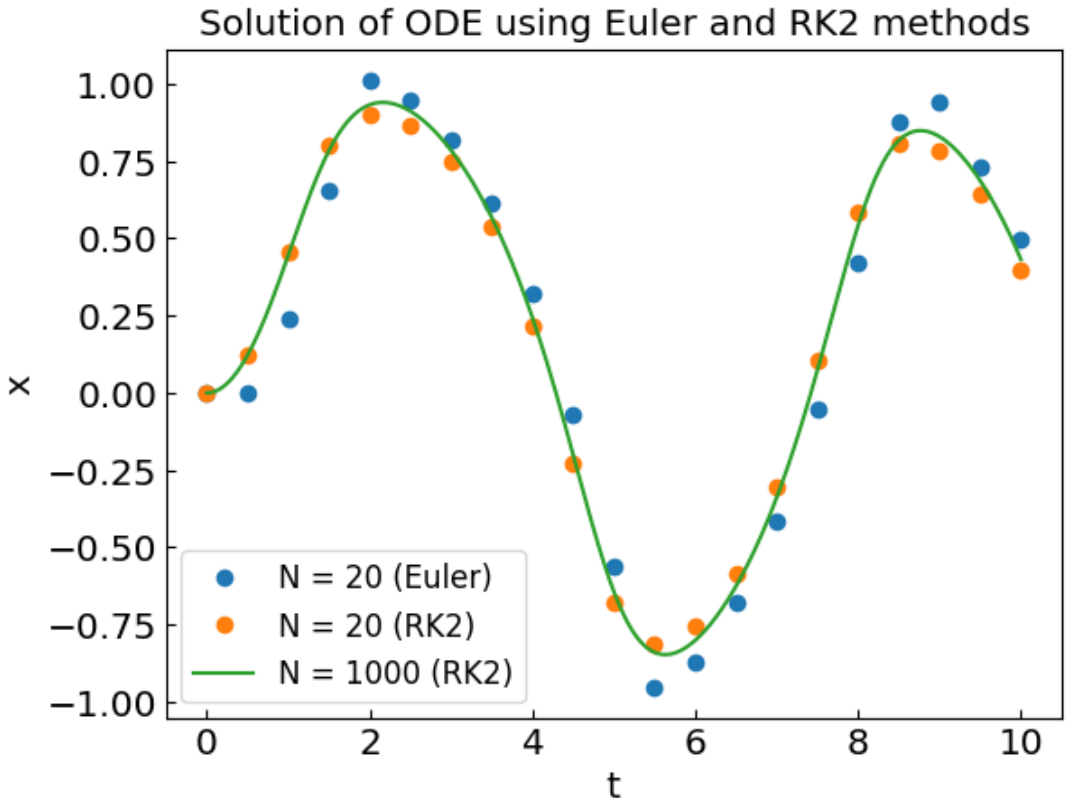
Midpoint method (2nd order Runge-Kutta)

```python
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*steps 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): $O(h^2)$

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



From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

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): $O(h^4)$

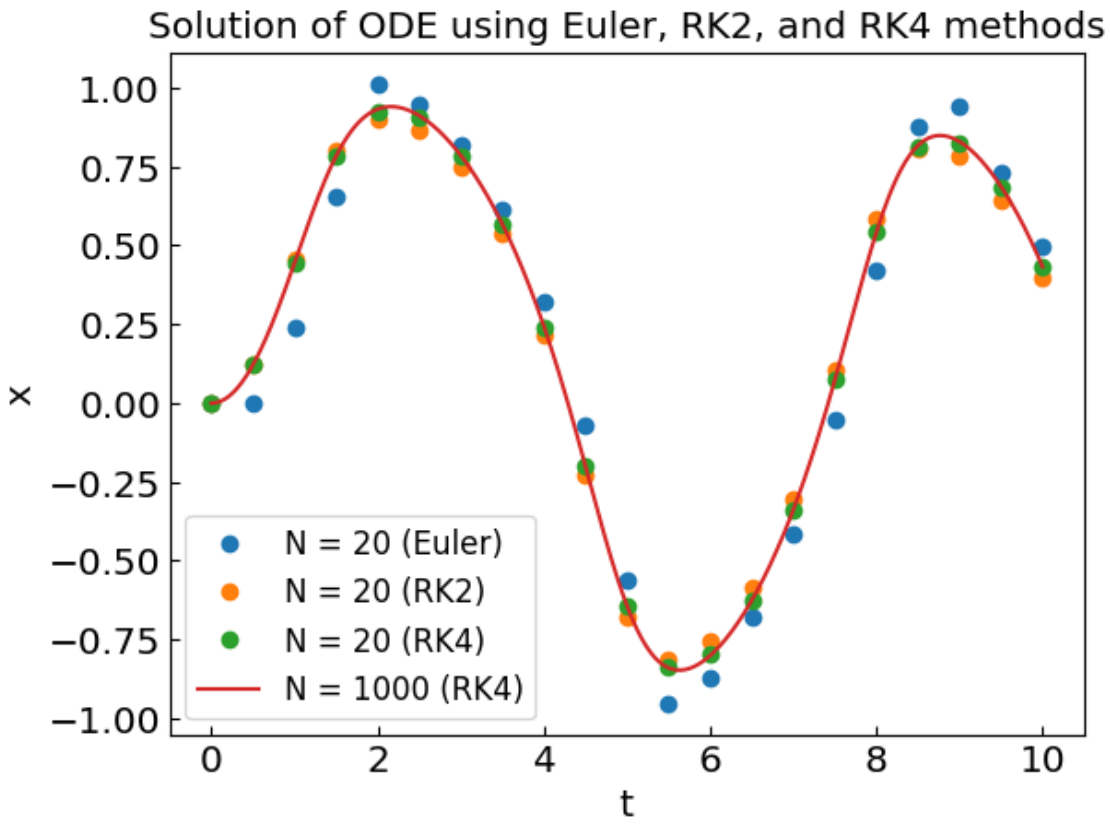
The classical 4th-order Runge-Kutta method is a good first choice for solving physics ODEs.

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Classical 4th order Runge-Kutta method

```python
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*steps 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
```

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



From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

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)

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

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

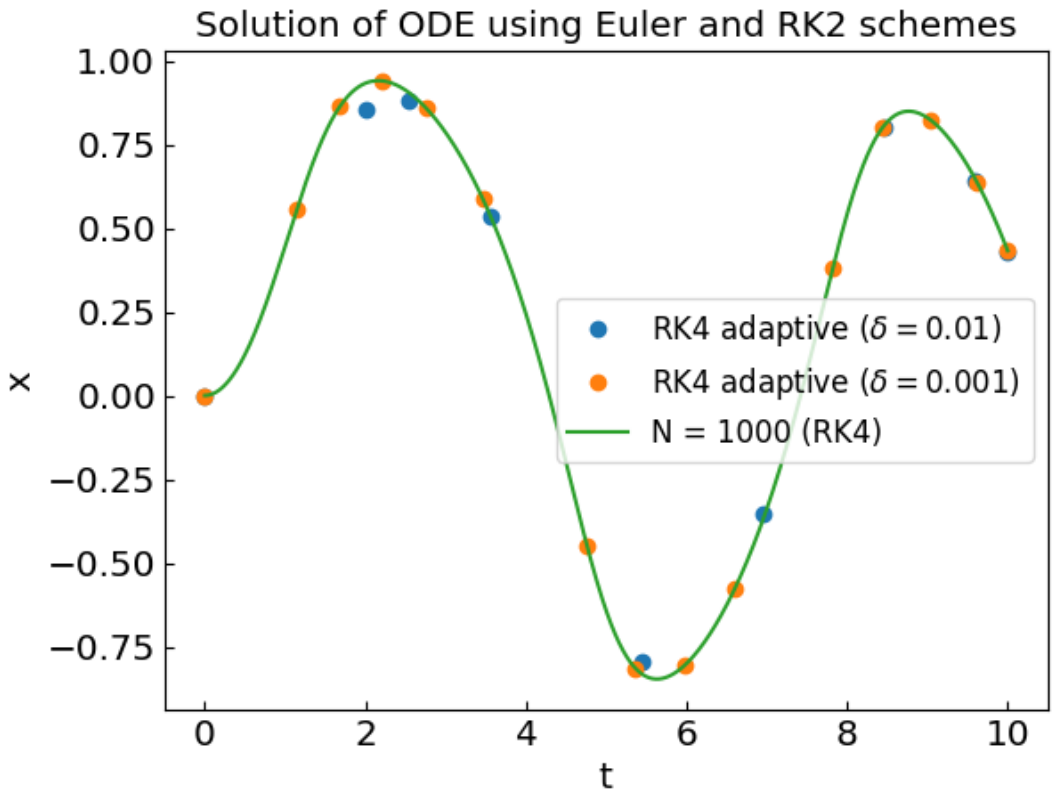
$$h' = h\left(\frac{30h\delta}{|x_1 - x_2|}\right)^{1/4}.$$

- $h'>h$: 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

```python
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
```

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



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

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Stability, stiff equations, and implicit methods

Consider the following ODE

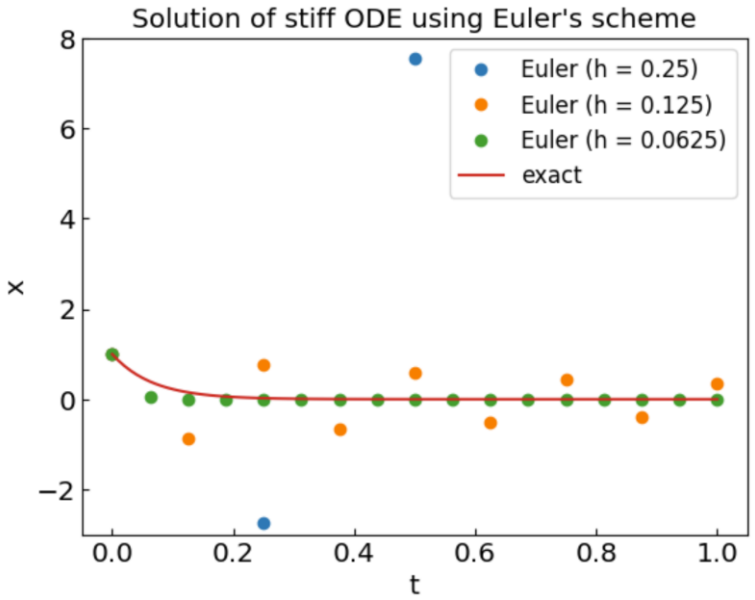
$$\frac{dx}{dt} = -15x, \qquad \textit{stiff equation}$$

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!



From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Stability, stiff equations, and implicit methods

Consider the following ODE

$$\frac{dx}{dt} = -15x, \qquad \textit{stiff equation}$$

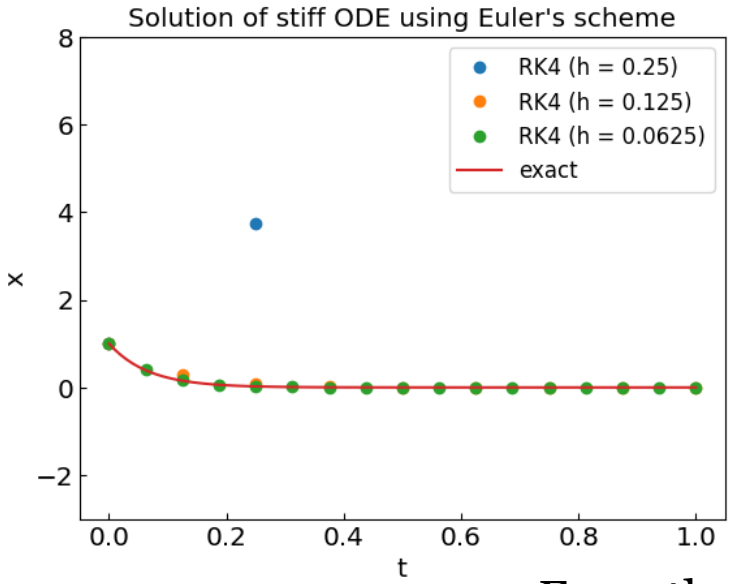
with x(t=0)=1.

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



From the course by Volodymyr Vovchenko, https://github.com/vlvovch/PHYS6350-ComputationalPhysics

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 $x_{n+1} = x_n - 15hx_n = (1-15h)x_n = (1-15h)^n x_0, \qquad x_n \equiv x(t+nh)$

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

Solution: *implicit methods*

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

Our stiff equation: $x_{n+1} = x_n - 15hx_{n+1}$ thus $x_{n+1} = \frac{x_n}{1+15h} = \frac{x_0}{(1+15h)^n} \overset{n\to\infty}{\to} 0$ 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

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

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

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

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

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

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Systems of Ordinary Differential Equations

```python
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
```

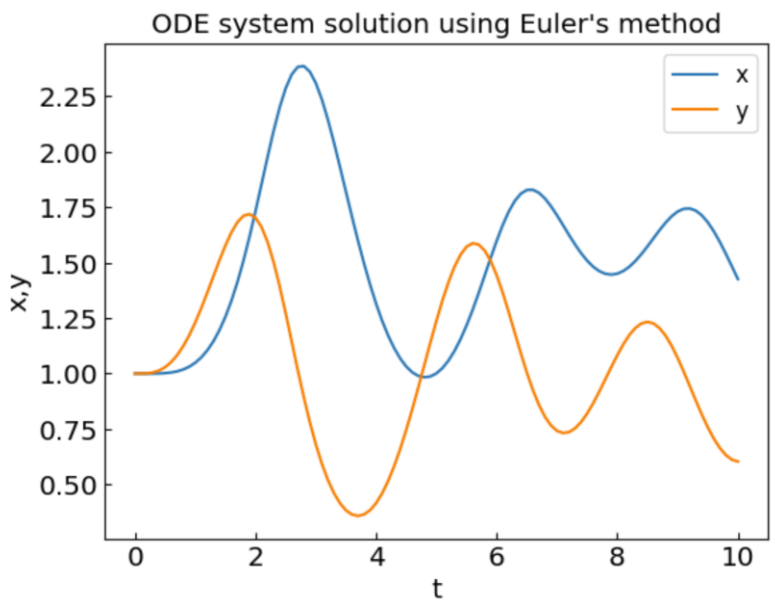
```python
def ode_rk2_multi(f, x0, t0, h, nsteps):
    """Multi-dimensional version of the RK2 method.
    """

    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
```

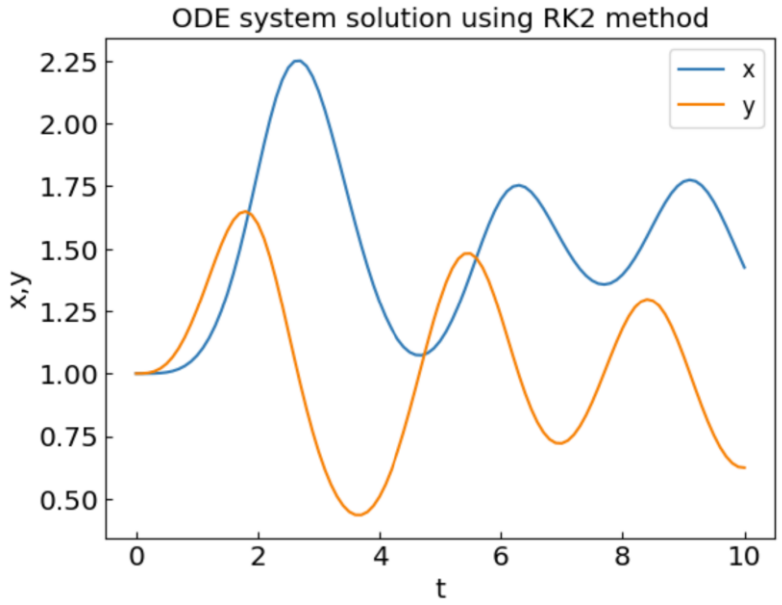
```python
def ode_rk4_multi(f, x0, t0, h, nsteps):
    """Multi-dimensional version of the RK4 method.
    """

    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
```

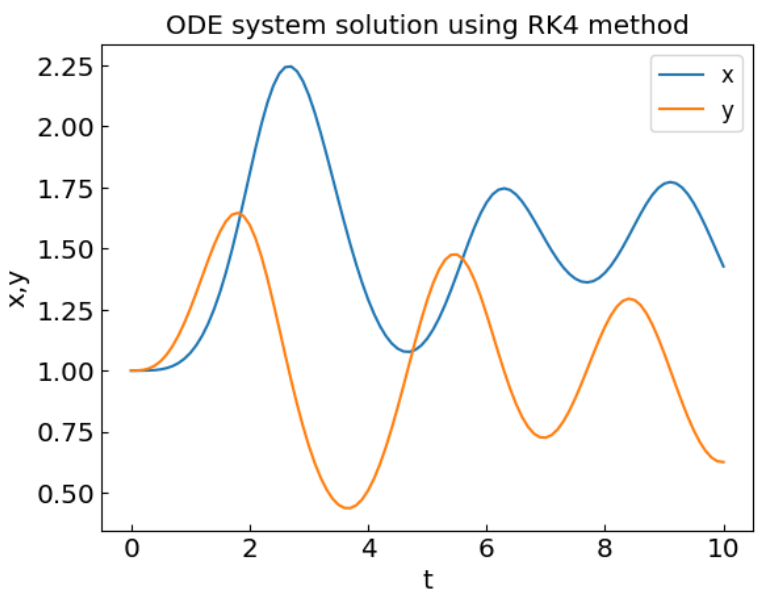
From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Systems of Ordinary Differential Equations: Example

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







From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

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 $\frac{d\mathbf{x}}{dt} = \mathbf{v}$

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

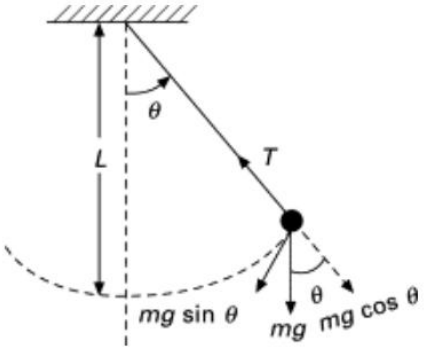
and can be solved for **x**(t) and **v**(t) using standard methods

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

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

$$\frac{d\theta}{dt} = \omega,$$
$$\frac{d\omega}{dt} = -\frac{g}{L}\sin\theta\,,$$

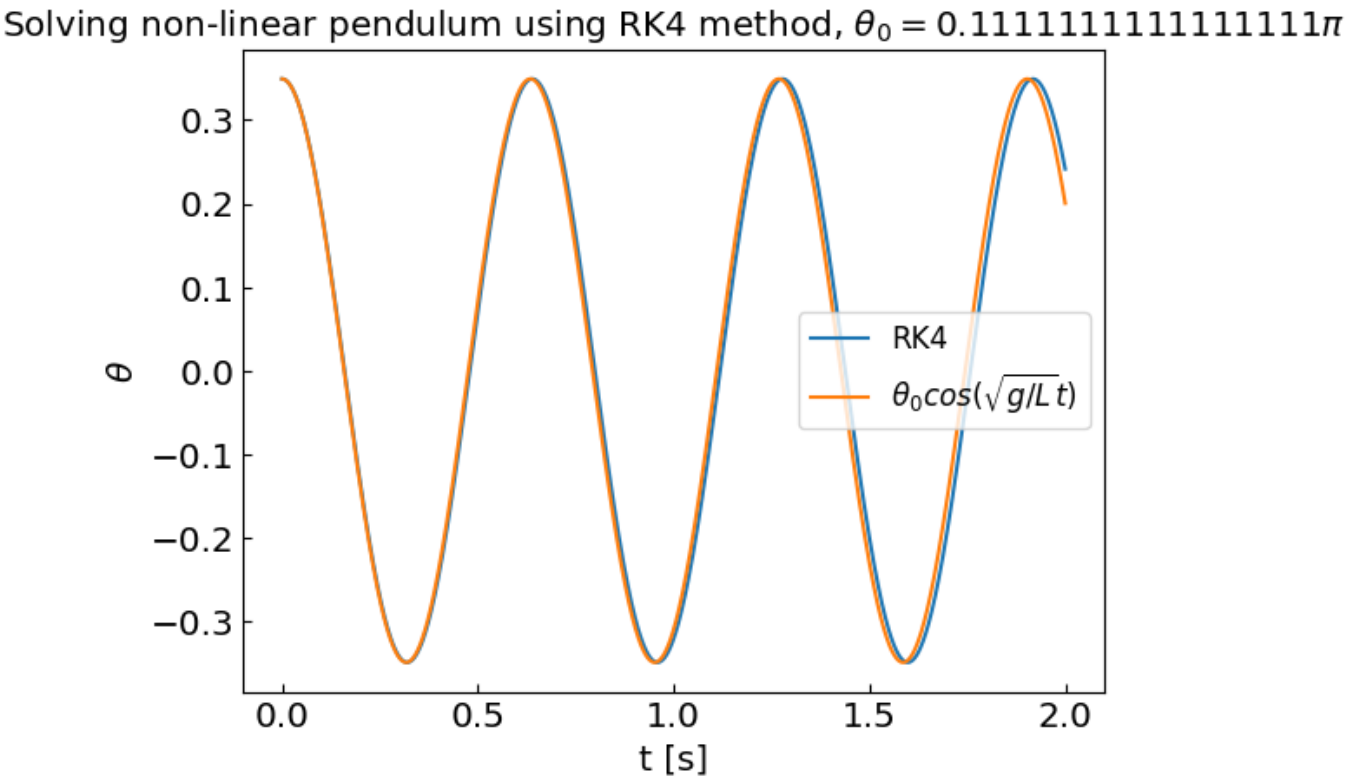
For small angles $\sin\theta \approx \theta$, an analytic solution exists

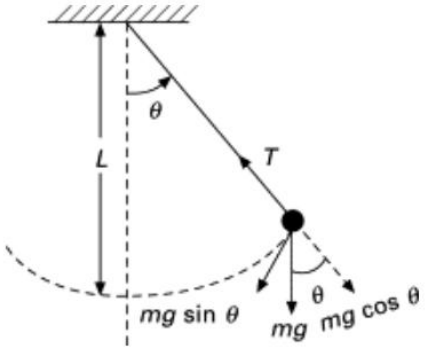
$$\theta(t) \approx \theta_0 \cos\left(\sqrt{\frac{g}{L}}t + \phi\right)$$

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Example: Simple pendulum

Initially at rest at angle $\theta_0 = 20° \approx 0.111\pi$ L=0.1 m, g=9.81 m/s^2



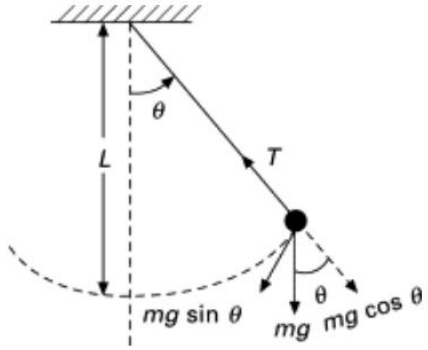


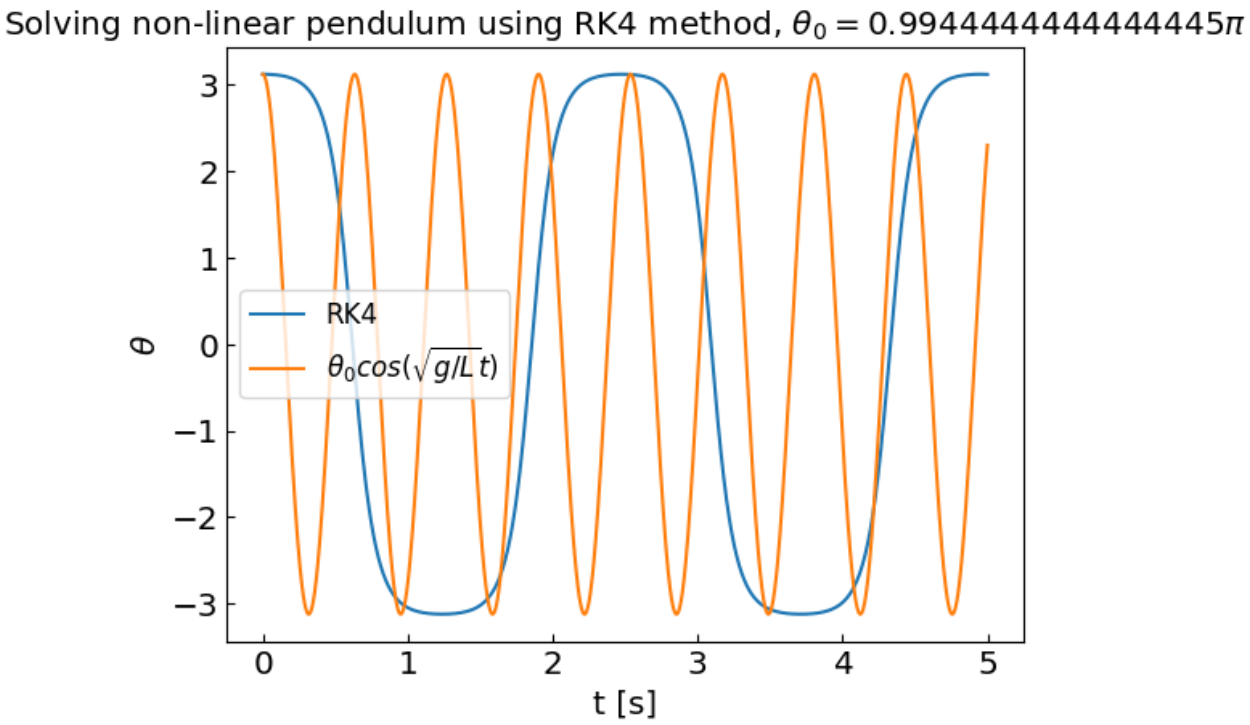
Linear regime at small angles

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Example: Simple pendulum

Initially at rest at angle $\theta_0 = 179° \approx 0.994\pi$ L=0.1 m, g=9.81 m/s^2



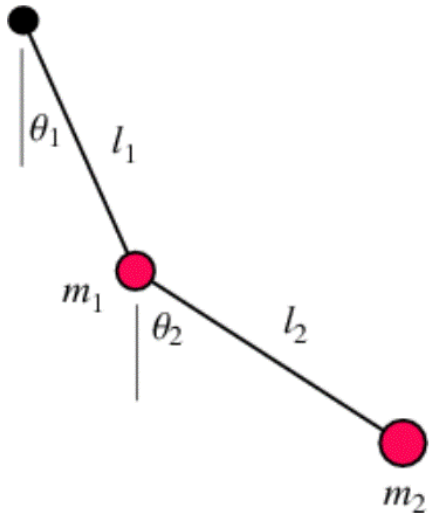


Non-linear regime at large angles, approximate analytic solution fails

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Double pendulum

Double pendulum is the simplest system exhibiting chaotic motion – *deterministic chaos*



Two degrees of freedom: the displacement angles θ_1 and θ_2.

$$
\begin{aligned}
x_1 &= l_1 \sin(\theta_1), \\
y_1 &= -l_1 \cos(\theta_1), \\
x_2 &= l_1 \sin(\theta_1) + l_2 \sin(\theta_2), \\
y_2 &= -l_1 \cos(\theta_1) - l_2 \cos(\theta_2).
\end{aligned}
$$

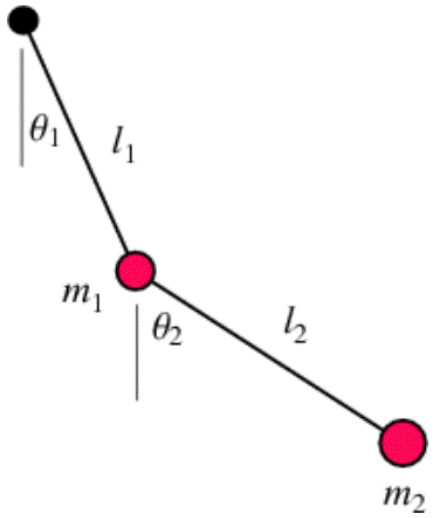
$$T = \frac{m_1 \dot{x}_1^2}{2} + \frac{m_2 \dot{x}_2^2}{2} = \frac{1}{2} m_1 l_1^2 \dot{\theta}_1 + \frac{1}{2} m_2 \left[l_1^2 \dot{\theta}_1^2 + l_2^2 \dot{\theta}_2^2 + 2 l_1 l_2 \dot{\theta}_1 \dot{\theta}_2 \cos(\theta_1 - \theta_2), \right]$$ kinetic energy

$$V = m_1 g y_1 + m_2 g y_2 = -(m_1 + m_2) g l_1 \cos(\theta_1) - m_2 g l_2 \cos(\theta_2).$$ potential energy

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Double pendulum

Double pendulum is the simplest system exhibiting chaotic motion – *deterministic chaos*



Two degrees of freedom: the displacement angles θ_1 and θ_2.

$$
\begin{aligned}
x_1 &= l_1 \sin(\theta_1), \\
y_1 &= -l_1 \cos(\theta_1), \\
x_2 &= l_1 \sin(\theta_1) + l_2 \sin(\theta_2), \\
y_2 &= -l_1 \cos(\theta_1) - l_2 \cos(\theta_2).
\end{aligned}
$$

$$
T = \frac{m_1 \dot{x}_1^2}{2} + \frac{m_2 \dot{x}_2^2}{2} = \frac{1}{2} m_1 l_1^2 \dot{\theta}_1 + \frac{1}{2} m_2 \left[l_1^2 \dot{\theta}_1^2 + l_2^2 \dot{\theta}_2^2 + 2 l_1 l_2 \dot{\theta}_1 \dot{\theta}_2 \cos(\theta_1 - \theta_2), \right]
$$

kinetic energy

$$
V = m_1 g y_1 + m_2 g y_2 = -(m_1 + m_2) g l_1 \cos(\theta_1) - m_2 g l_2 \cos(\theta_2).
$$

potential energy

The Lagrange equations of motion read

$$
\begin{aligned}
(m_1 + m_2) l_1 \ddot{\theta}_1 + m_2 l_2 \cos(\theta_1 - \theta_2) \ddot{\theta}_2 &= -m_2 l_1 \dot{\theta}_2^2 \sin(\theta_1 - \theta_2) - (m_1 + m_2) g \sin(\theta_1), \\
m_2 l_1 \cos(\theta_1 - \theta_2) \ddot{\theta}_1 + m_2 l_2 \ddot{\theta}_2 &= m_2 l_1 \dot{\theta}_1^2 \sin(\theta_1 - \theta_2) - m_2 g \sin(\theta_2).
\end{aligned}
$$

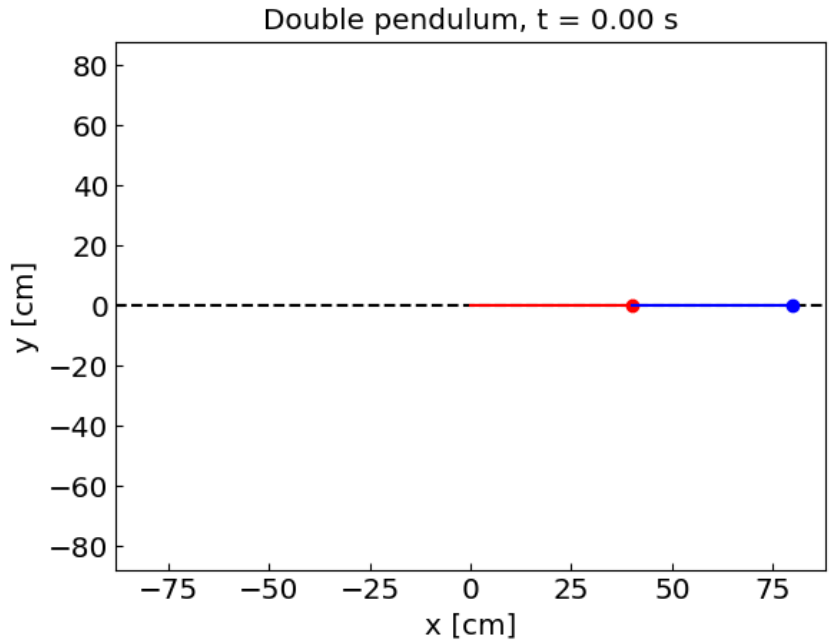
This is a system of two linear equation for $\ddot{\theta}_{1,2}$ that can be solved easily.

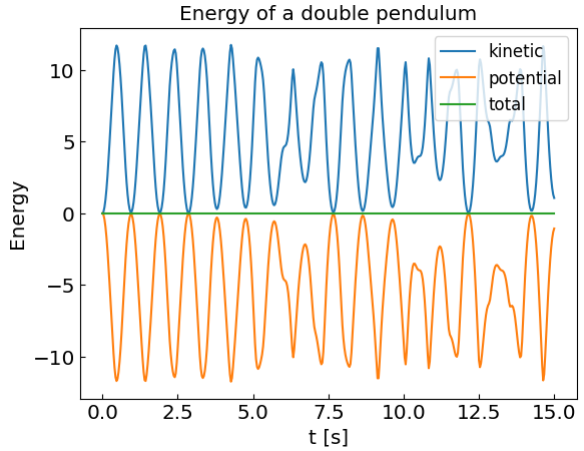
From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Double pendulum

```
g = 9.81
l1 = 0.4
l2 = 0.4
m1 = 1.0
m2 = 1.0

def fdoublependulum(xin, t):
    global f_evaluations
    f_evaluations += 1

    theta1 = xin[0]
    theta2 = xin[1]
    omega1 = xin[2]
    omega2 = xin[3]
    a1 = (m1 + m2)*l1
    b1 = m2*l2*np.cos(theta1 - theta2)
    c1 = m2*l2*omega2*omega2*np.sin(theta1 - theta2) + (m1 + m2)*g*np.sin(theta1)
    a2 = m2*l1*np.cos(theta1 - theta2)
    b2 = m2*l2;
    c2 = -m2*l1*omega1*omega1*np.sin(theta1 - theta2) + m2*g*np.sin(theta2) # + k
    domega1 = - ( c2/b2 - c1/b1 ) / ( a2/b2 - a1/b1 )
    domega2 = - ( c2/a2 - c1/a1 ) / ( b2/a2 - b1/a1 )
    return np.array([omega1,
                     omega2,
                     domega1,
                     domega2
                    ])
```

```
sol = bulirsch_stoer(fpendulum, x0, t0, 1, tend, eps, error_definition_pendulum, 10)
```

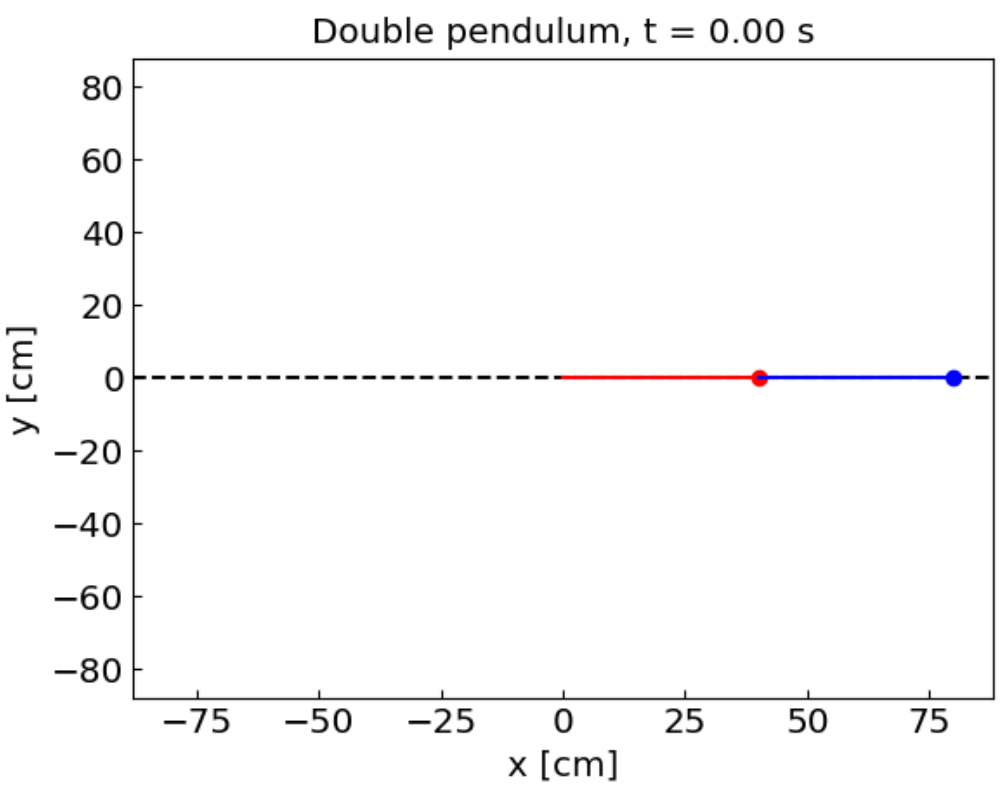




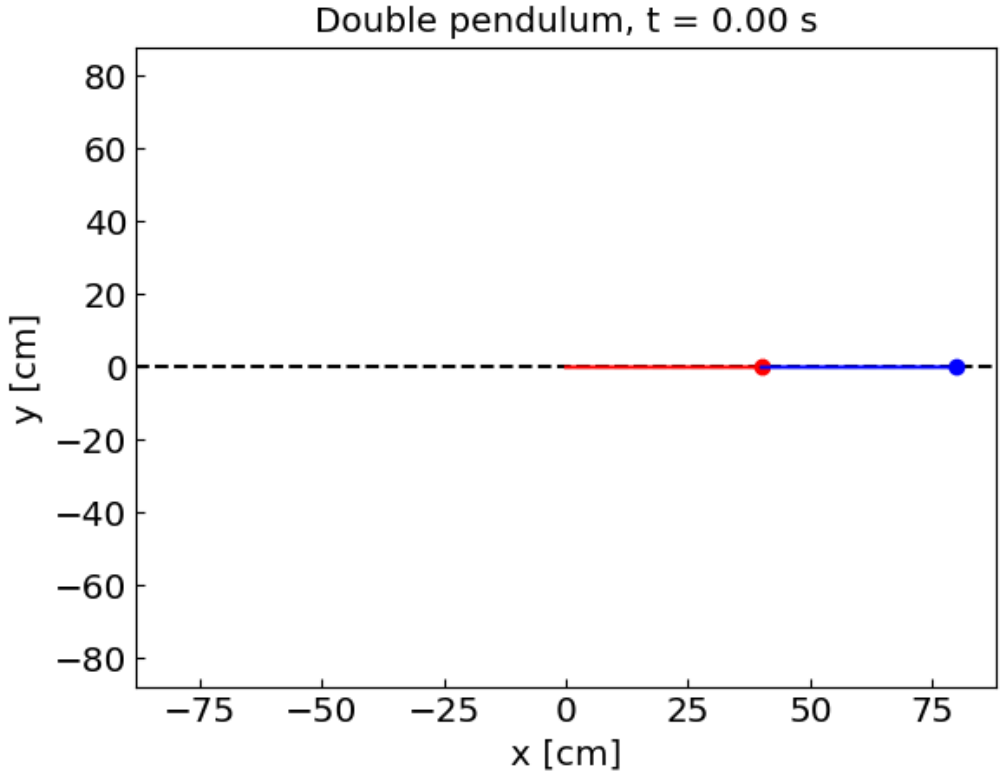
From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Double pendulum: chaotic behavior

$$\theta_1^0 = \theta_2^0 = \pi/2$$



$$\theta_1^0 = \theta_2^0 = \pi/2 + 10^{-4}$$

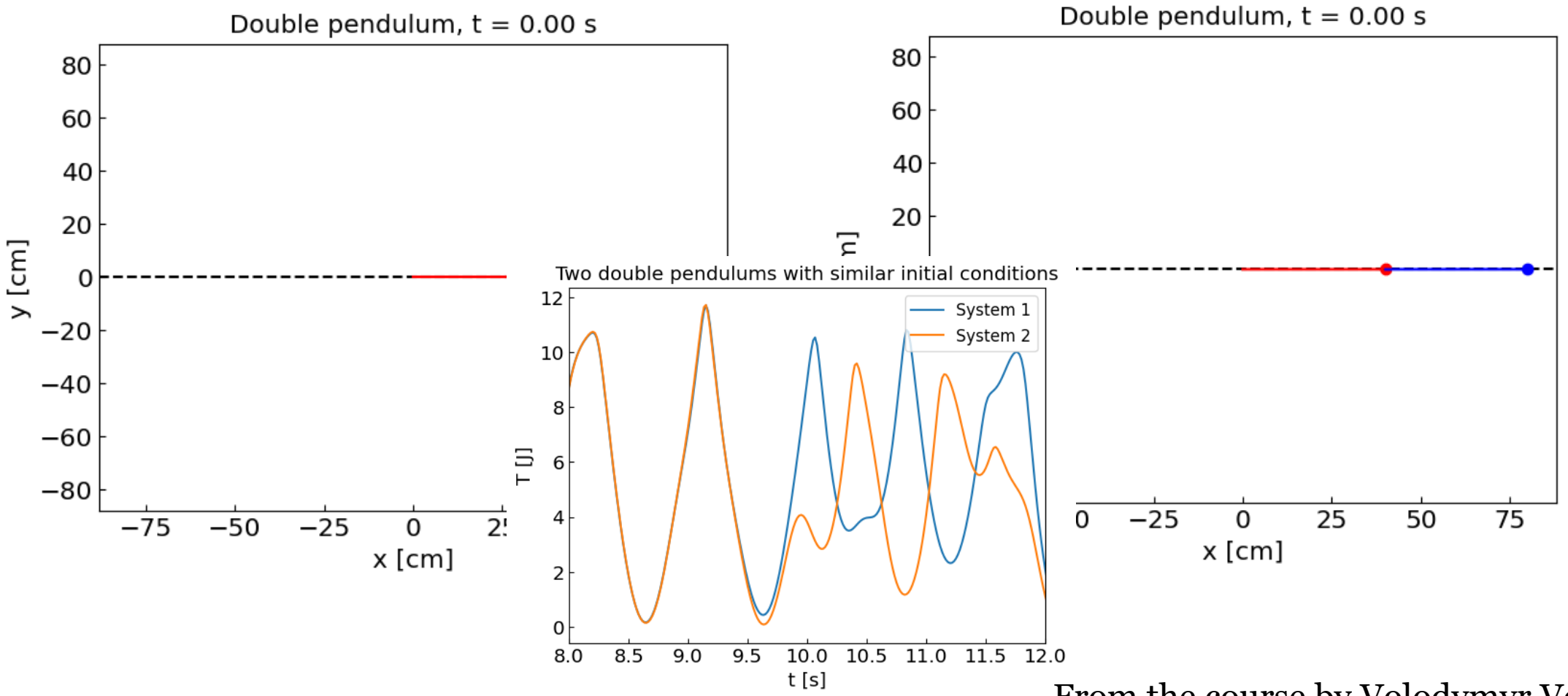


From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Double pendulum: chaotic behavior

$$\theta_1^0 = \theta_2^0 = \pi/2$$

$$\theta_1^0 = \theta_2^0 = \pi/2 + 10^{-4}$$



From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Three-body problem

Three stars interacting through gravitational force

The equations of motion are

$$\frac{d^2\mathbf{r}_1}{dt} = Gm_2\frac{\mathbf{r}_2-\mathbf{r}_1}{|\mathbf{r}_2-\mathbf{r}_1|^3} + Gm_3\frac{\mathbf{r}_3-\mathbf{r}_1}{|\mathbf{r}_3-\mathbf{r}_1|^3},$$
$$\frac{d^2\mathbf{r}_2}{dt} = Gm_1\frac{\mathbf{r}_1-\mathbf{r}_2}{|\mathbf{r}_1-\mathbf{r}_2|^3} + Gm_3\frac{\mathbf{r}_3-\mathbf{r}_2}{|\mathbf{r}_3-\mathbf{r}_2|^3},$$
$$\frac{d^2\mathbf{r}_3}{dt} = Gm_1\frac{\mathbf{r}_1-\mathbf{r}_3}{|\mathbf{r}_1-\mathbf{r}_3|^3} + Gm_2\frac{\mathbf{r}_2-\mathbf{r}_3}{|\mathbf{r}_2-\mathbf{r}_3|^3}.$$

Name	Mass	x	y
Star 1	150.	3	1
Star 2	200.	-1	-2
Star 3	250.	-1	1

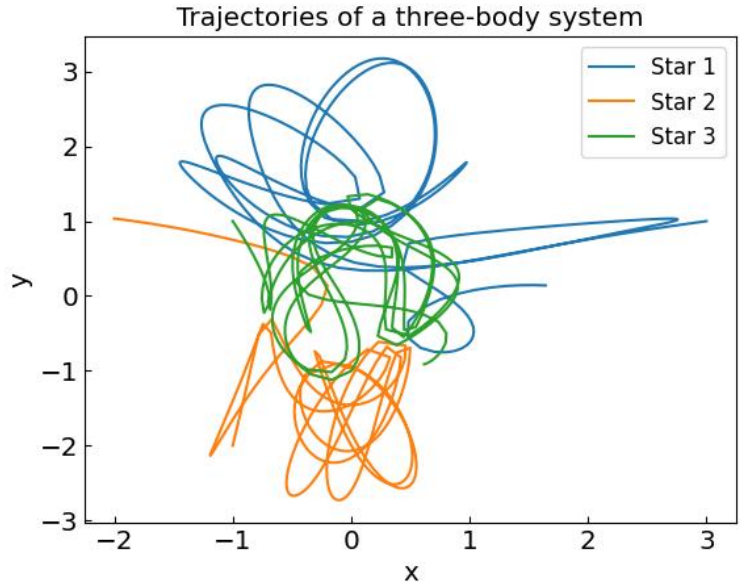
Cannot be solved analytically!

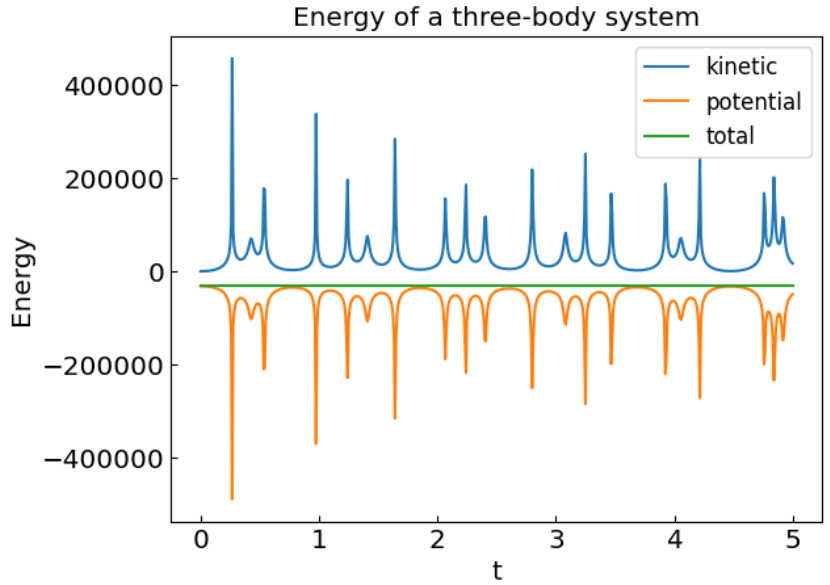
Take *G = 1* (dimensionless)

Initially at rest and move in plane z = 0

Initial coordinates: $\mathbf{r}_1 = (3,1)$, $\mathbf{r}_2 = (-1,-2)$, $\mathbf{r}_3 = (-1,1)$

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics



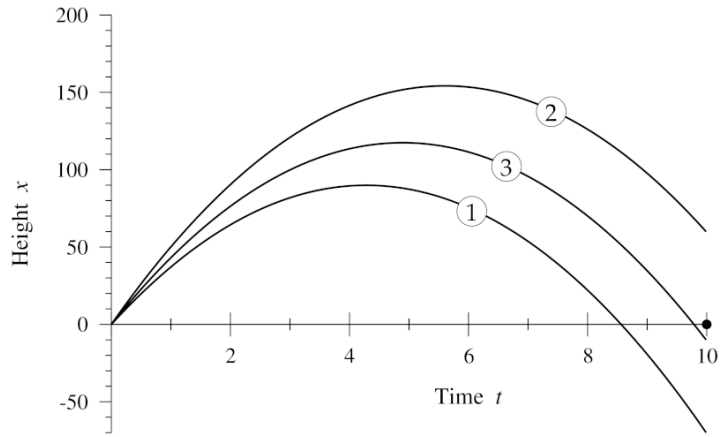


Boundary value problems and the shooting method

Sometimes we have equations, such as vertically thrown object

$$\frac{dx}{dt} = v,$$
$$\frac{dv}{dt} = -g,$$

and boundary conditions, e.g. x(0) = 0 and x(10) = 0 instead of initial conditions v(0) = v_0.



How to solve this problem?

In the shooting method one takes trial values of v_0 until finding the one where the solution satisfies the boundary condition x(10) = 0.

To find v_0 efficiently one combines numerical ODE method (e.g. RK4) with non-linear equation solver (e.g. bisection method).

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics