Computational Physics (PHYS6350)

Lecture 12: Ordinary Differential Equations Part II

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

Instructor: Volodymyr Vovchenko (vvovchenko@uh.edu)

Course materials: https://github.com/vlvovch/PHYS6350-ComputationalPhysics
Online textbook: https://vovchenko.net/computational-physics/

Adaptive time step for systems of ODEs

For a single ODE we devised an adaptive RK4 scheme

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

Time step is adjusted as

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

Here $x_1 = RK4(RK4(x, t, h), t + h, h)$ and $x_2 = RK4(x, t + 2h, 2h)$

How to generalize $\varepsilon = |x_1 - x_2|$ to a system of ODEs where we have a state vector $\mathbf{x}$?

The answer depends on the physical problem at hand. One could take for example

$$\varepsilon = |\mathbf{x_1} - \mathbf{x_2}|$$

Alternatively, if the accuracy of only one variable matters (e.g. the position but not the velocity), one can use just this one coordinate to define ε

The implementation of the adaptive step in systems of ODEs should thus allow for flexibility to define the accuracy

Multi-dimensional RK4 with adaptive time step

```python
def ode_rk4_adaptive_multi(f, x0, t0, h0, tmax, delta = 1.e-6, distance_definition = distance_definition_default):
    """Solve an ODE dx/dt = f(x,t) from t = t0 to t = t0 + h*steps
    using 4th order Runge-Kutta method with adaptive time step.

    Args:
          f: the function that defines the ODE.
         x0: the initial value of the dependent variable.
         t0: the initial value of the time variable.
         h0: the initial time step
       tmax: the maximum time
      delta: the desired accuracy per unit time

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

    ts = [t0]
    xs = [x0]

    h = h0
    t = t0
    i = 0
```

```python
    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)

        diffnorm = distance_definition(x1, x2)
        if diffnorm == 0.: # To avoid the division by zero
            rho = 2.**4
        else:
            rho = 30. * h * delta / diffnorm
        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
```
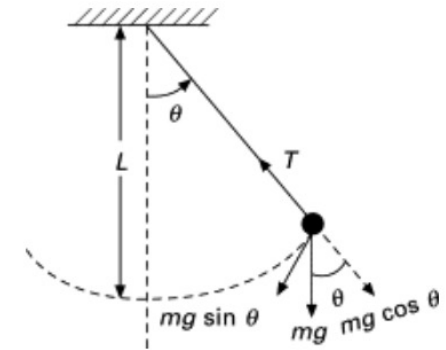
Adaptive time step RK4 for non-linear pendulum

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

Accuracy: only the angle θ matters

```python
def error_definition_pendulum(x1, x2):
    return np.abs(x1[0] - x2[0])
```

```python
a = 0.
b = 3.0
N = 500
h0 = (b-a)/N
eps = 1.e-3 # accuracy in theta
sol = ode_rk4_adaptive_multi(fpendulum, x0, a, h0, b, eps, error_definition_pendulum)
```


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

Time step lenghts
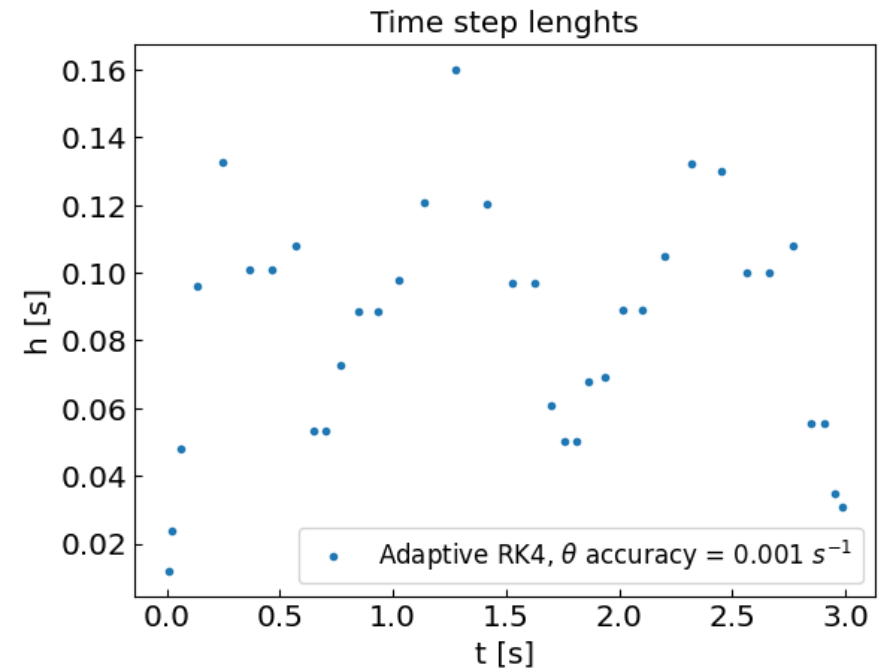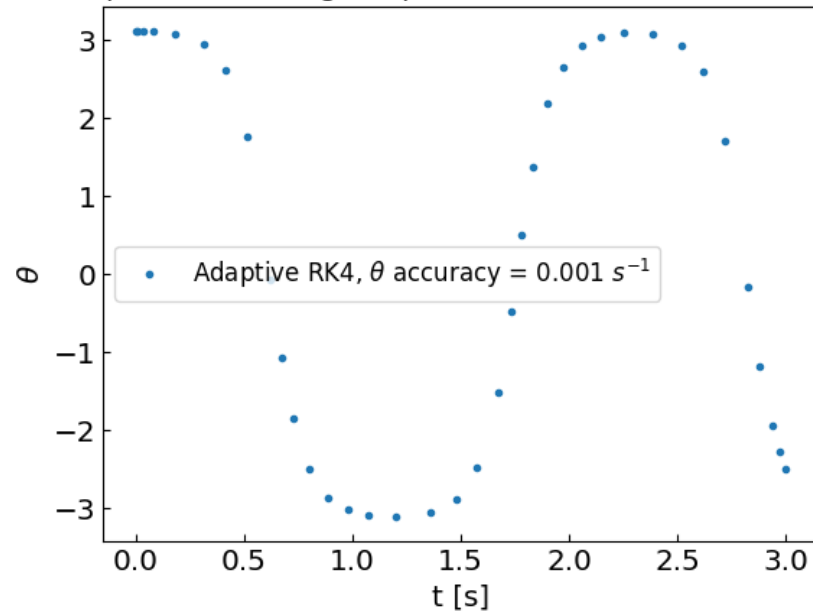
Comet motion
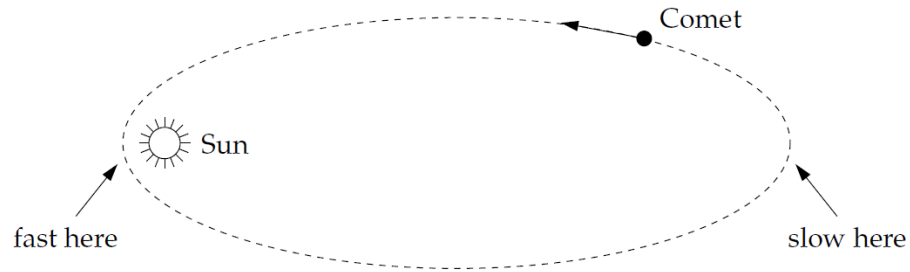
$$m\frac{d^2\vec{r}}{dt^2} = -\left(\frac{GMm}{r^2}\right)\frac{\vec{r}}{r}$$

Angular momentum is conserved, the motion is in the plane (z=0), thus only two equations needed

$$\frac{d^2x}{dt^2} = -GM\frac{x}{r^3},$$
$$\frac{d^2y}{dt^2} = -GM\frac{y}{r^3},$$

where $r = \sqrt{x^2 + y^2}$.

Initial conditions:
$x(0) = 4 \cdot 10^{12}$ m, $y(0) = 0$
$v_x(0) = 0$, $v_y(0) = 500$ m/s

```python
G = 6.67430e-11   # m^3 / kg / s^2
Msun = 1.9885e30 # kg

def fcomet(xin, t):
    x = xin[0]
    y = xin[1]
    vx = xin[2]
    vy = xin[3]
    r = np.sqrt(x*x+y*y)
    return np.array([vx,vy,-G*Msun*x/r**3,-G*Msun*y/r**3])

x0 = [4.e12,0.,0.,500.]
```
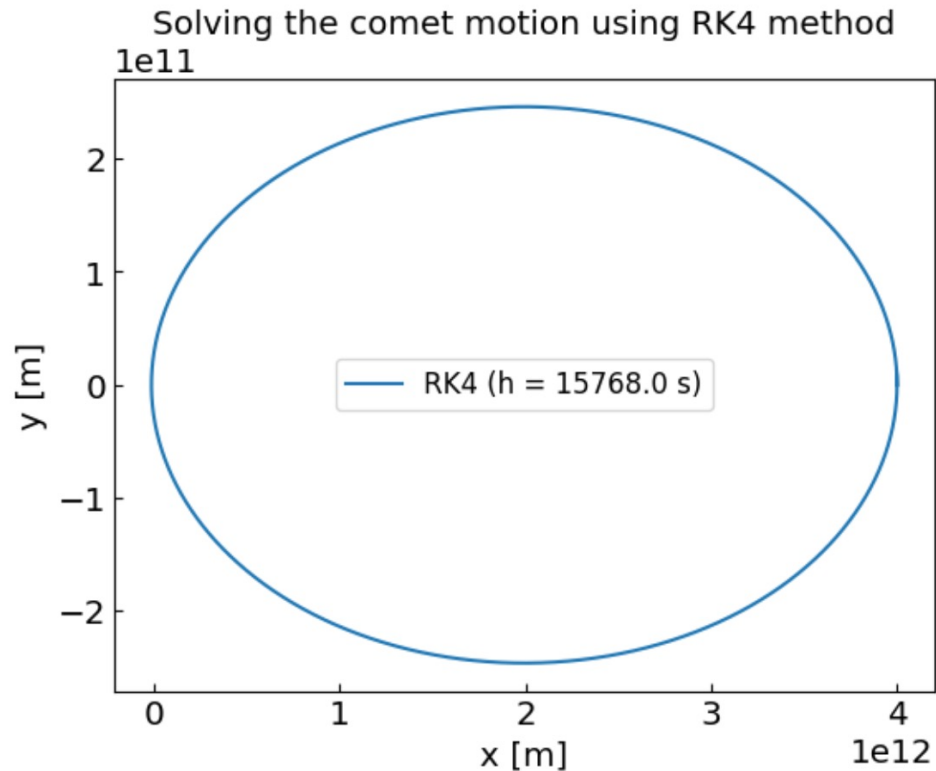
Comet motion: RK4 with fixed time step
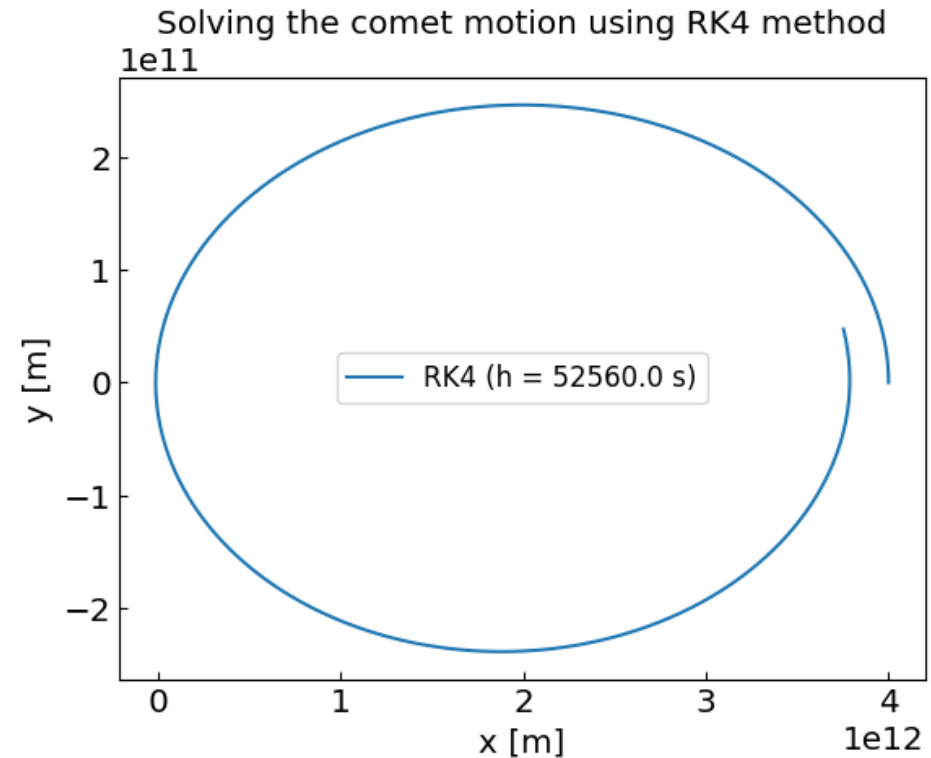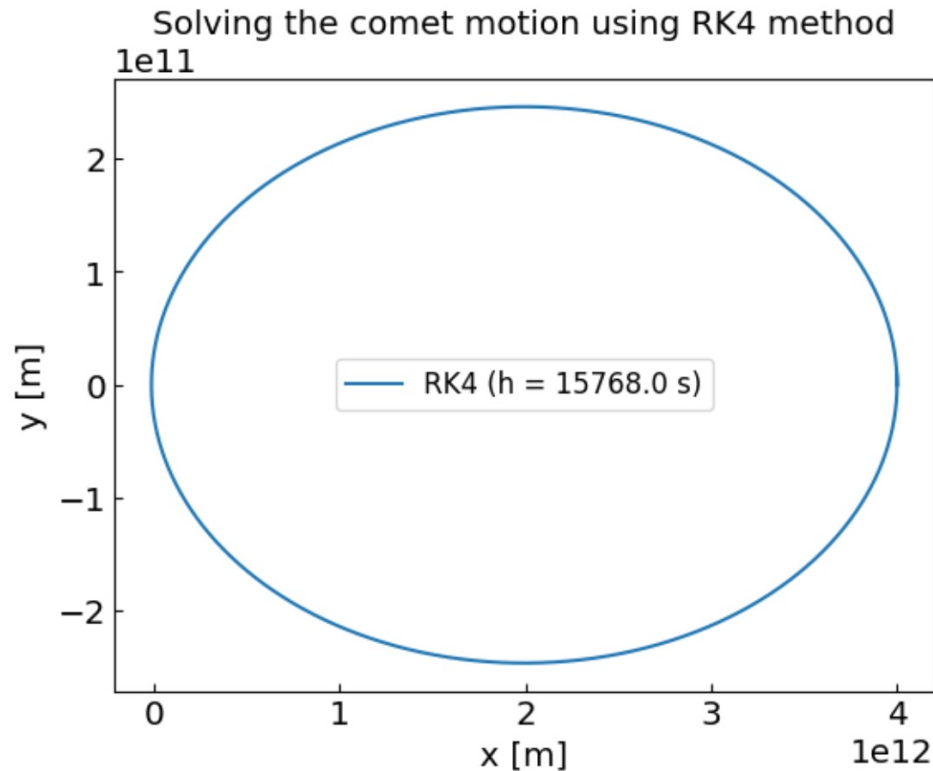
```
a = 0.
b = 50. * 365. * 24. * 60. * 60.  # 50 years
N = 100000                        # 100 thousand RK4 steps
h = (b - a) / N                   # Time step: around 1/5th of a day
sol = ode_rk4_multi(fcomet, x0, a, h, N)
```


Nice elliptic shape but are we wasting computational resources (time step very small)

Comet motion: RK4 with fixed time step

```
a = 0.
b = 50. * 365. * 24. * 60. * 60.  # 50 years
N = 100000                        # 100 thousand RK4 steps
h = (b - a) / N                   # Time step: around 1/5th of a day
sol = ode_rk4_multi(fcomet, x0, a, h, N)
```
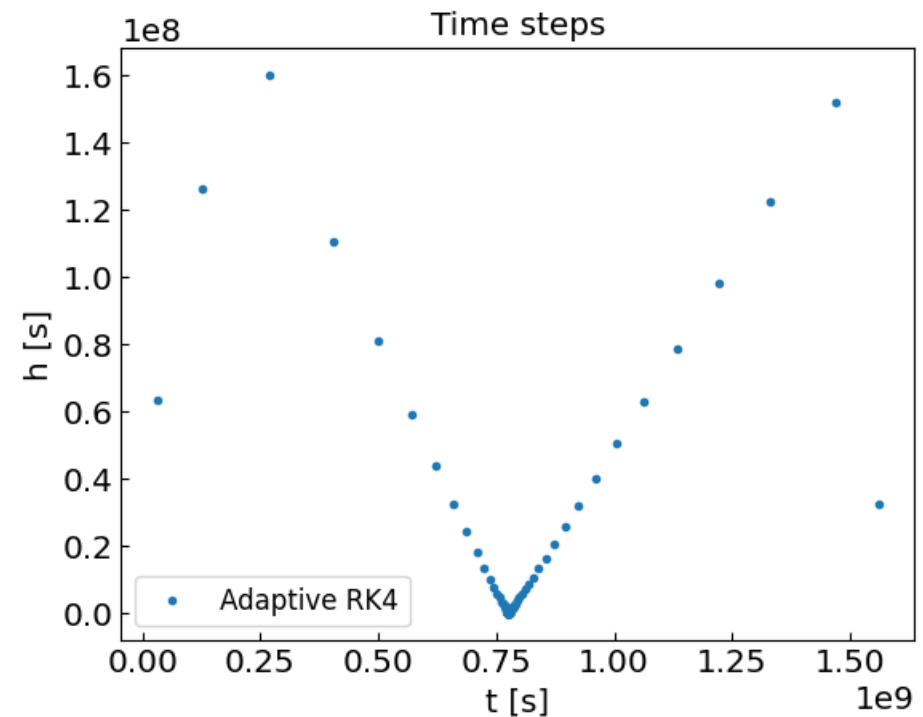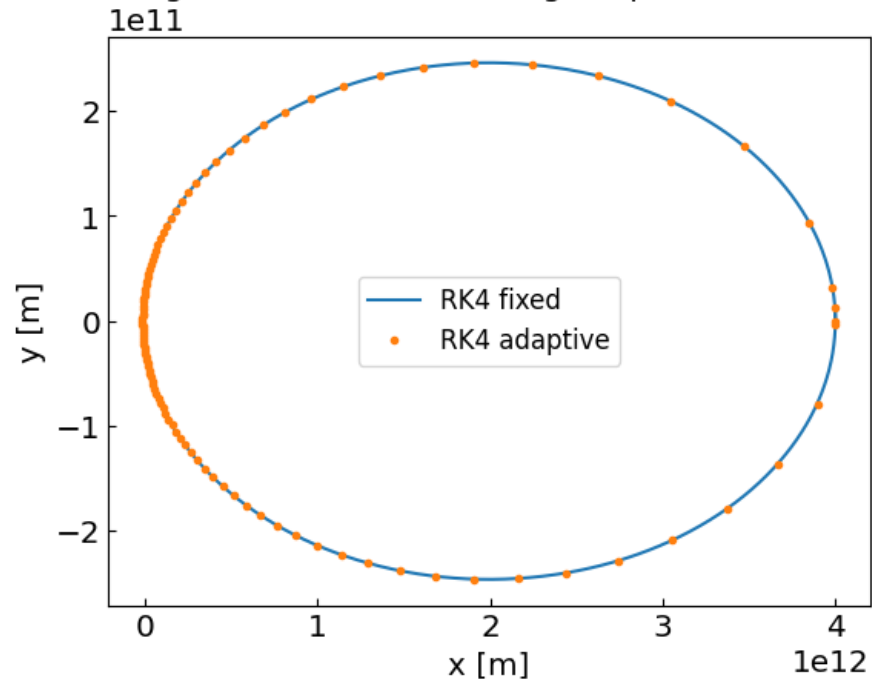

Nice elliptic shape but are we wasting computational resources (time step very small)
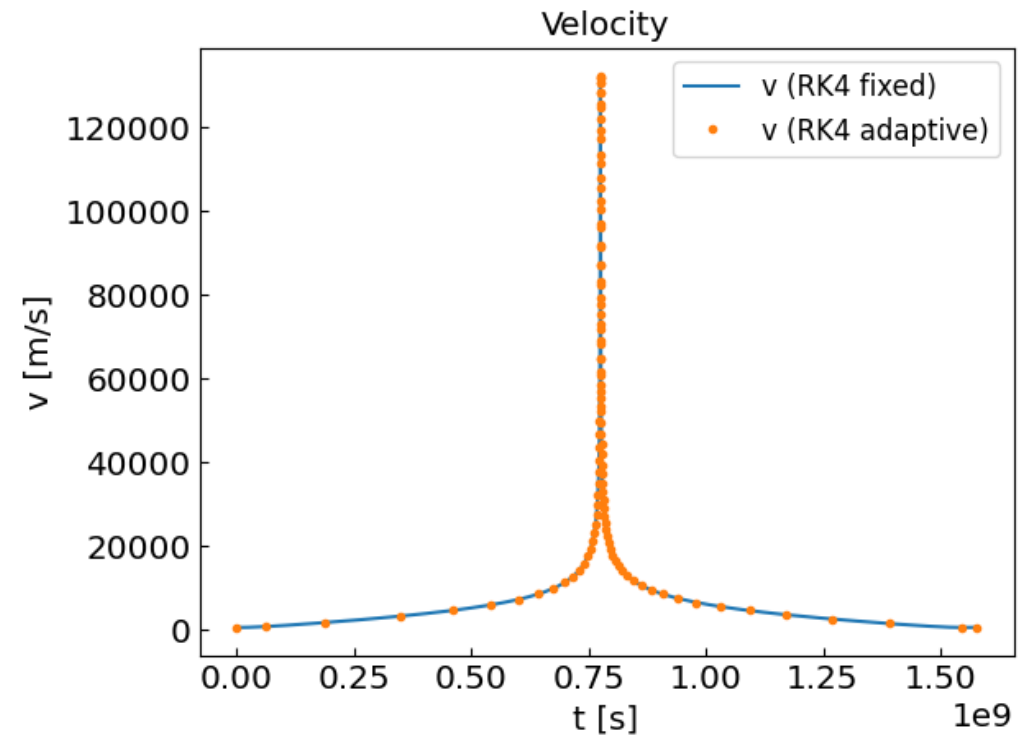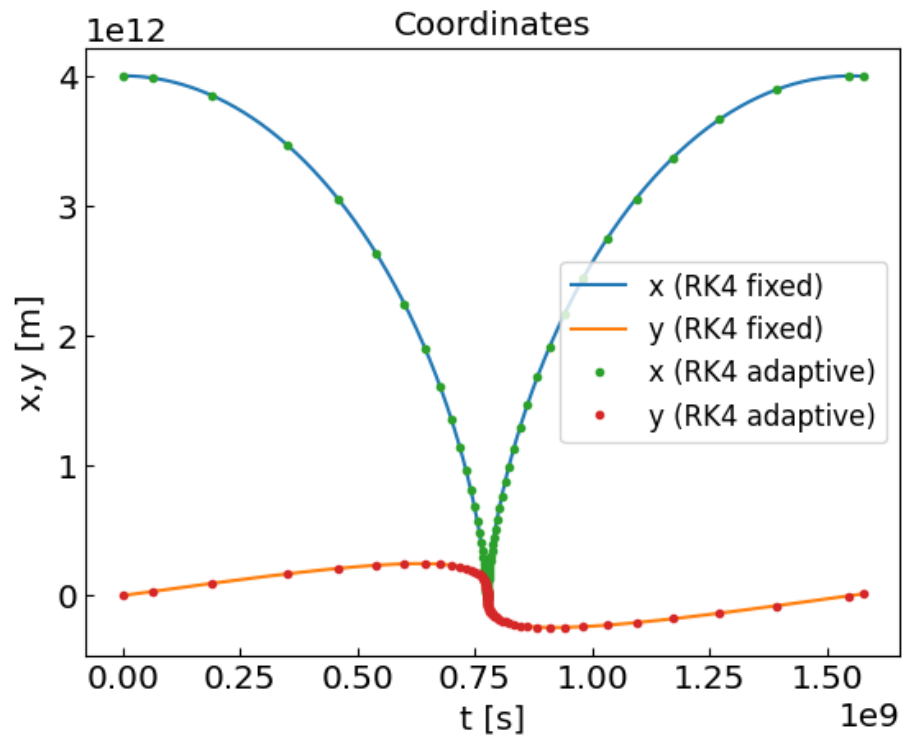
Comet motion: RK4 with adaptive time step

```python
def error_definition_comet(x1, x2):
    return np.sqrt((x1[0]-x2[0])**2 + (x1[1]-x2[1])**2)


x0 = [4.e12,0.,0.,500.]

a = 0.
b = 50. * 365. * 24. * 60. * 60. # 50 years
h0 = 1. * 365. * 24. * 60. * 60. # Initial time step: 1 year
delta = 1000. * 1.e3 / (365. * 24. * 60. * 60.)
sol = ode_rk4_adaptive_multi(fcomet, x0, a, h0, b, delta, error_definition_comet)
```

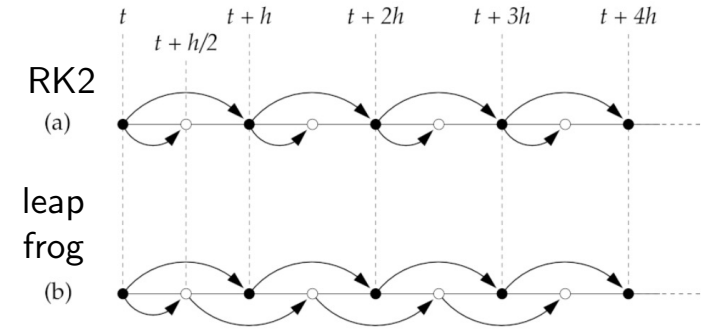# Comet motion: RK4 with adaptive time step

Leapfrog method

Recall the RK2 (midpoint) method

$$x(t + h) = x(t) + hf[x(t + h/2), t + h/2],$$

$$x(t + h/2) = x(t) + \frac{1}{2}hf(x, t).$$

RK2

(a)

leap frog

(b)

Leapfrog method: given x(t) and x(t+h/2), estimate x(t+h) and x(t+3h/2) using first equation only

$$x(t + h) = x(t) + hf[x(t + h/2), t + h/2],$$

$$x(t + 3h/2) = x(t + h/2) + hf[x(t + h), t + h].$$

Leapfrog method

Euler's half-step is used in the first iteration only.

The method is **time reversible**:
By changing $h \to -h$ one recovers *x(t)* and *x(t+h/2)* from previous iteration.

Error:

- Local (per time step): $O(h^3) + O(h^5) + O(h^7) + \ldots$
- Global (N=t_{end}/h time steps): $O(h^2) + O(h^3) + \ldots$ Odd powers in the global error propagated from the Euler's half-step at the 1st iteration

Leapfrog method implementation

```python
def ode_leapfrog_step(f, x, x2, t, h):
    """Perform a single step h using the leapfrog method.

    Args:
     f: the function that defines the ODE.
     x: the value of x(t)
    x2: the value of x(t+h/2)
     t: the present value of the time variable.
     h: the time step

    Returns:
    xnew, xnew2: the value of the dependent variable at the steps t+h, t+3h/2
    """

    xnew = x + h * f(x2,t+h/2.)
    xnew2 = x2 + h * f(xnew, t + h)
    return xnew, xnew2
```

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

    t = np.zeros(nsteps + 1)
    x = np.zeros((len(t), len(x0)))
    x2 = np.zeros(len(x0))
    t[0] = t0
    x[0,:] = x0
    x2[:] = ode_euler_step(f, x0, t0, h/2.)
    for i in range(0, nsteps):
        t[i + 1] = t[i] + h
        x[i + 1], x2 = ode_leapfrog_step(f, x[i], x2, t[i], h)
    return t,x
```
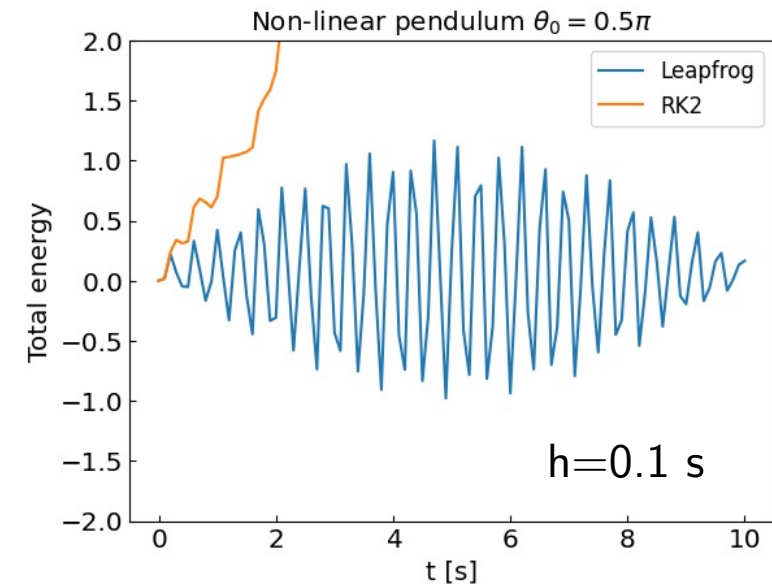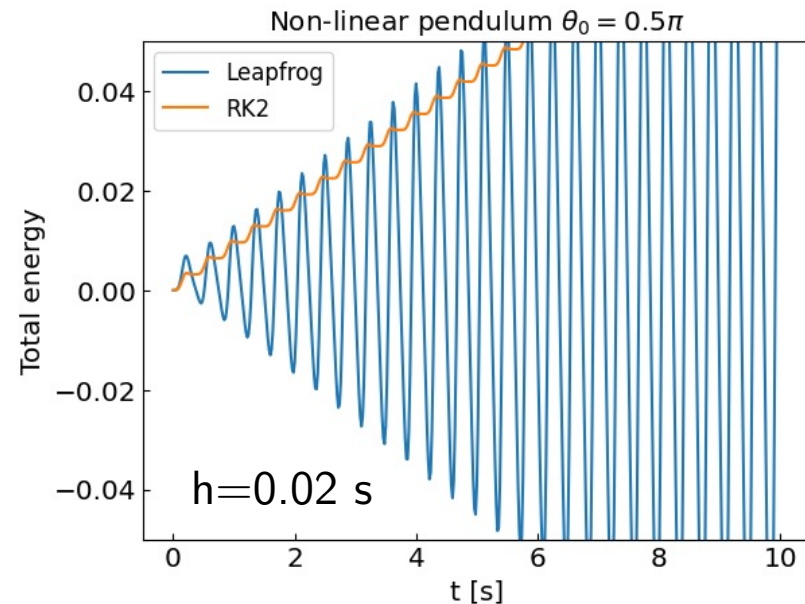
Leapfrog method and non-linear pendulum

Time-reversal symmetry implies average energy conservation

The pendulum energy is

$$E = mL^2\dot{\theta}^2/2 - mgL\cos(\theta)$$

Let us solve it with the leapfrog and RK2 methods and see how energy evolves with time

Non-linear pendulum $\theta_0 = 0.5\pi$

h=0.02 s

Non-linear pendulum $\theta_0 = 0.5\pi$

h=0.1 s
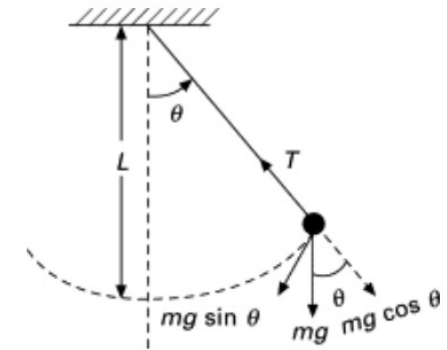
Energy is drifting in RK2 but conserved (on average) in leapfrog method

Modified midpoint method

Recall the error in the leapfrog method when integrating from t to t+H in steps of h = H/N

- Local (per time step): $O(h^3) + O(h^5) + O(h^7) + ...$
- Global (N=H/h time steps): $O(h^2) + O(h^3) + ...$

Odd powers in the global error are propagated from Euler's half-step at 1st iteration $\quad x(t + h/2) = x(t) + \frac{1}{2}hf(x,t).$

They can be canceled out with an additional Euler half-step at the end

Let $y_n = x(t + H - h/2)$ and $x_n = x(t + H)$ be the solution estimates resulting from the leapfrog method.

$$x(t + H) = \frac{1}{2}[x_n + y_n + \frac{h}{2}f(x_n, t + H)]. \qquad \textit{modified midpoint method}$$

Global error: $O(h^2) + O(h^4) + O(h^6) + ...$
(even powers only)

```python
def ode_MMM_multi(f, x0, t0, H, nsteps):
    """Multi-dimensional version of the modified midpoint method.
    """

    h = H / nsteps
    t = np.zeros(nsteps + 1)
    x = np.zeros((len(t), len(x0)))
    x2 = np.zeros(len(x0))
    t = t0
    x = x0
    y = ode_euler_step(f, x0, t0, h/2.)
    for i in range(0, nsteps):
        yprev = y
        x, y = ode_leapfrog_step(f, x, y, t, h)
        t = t + h

    return 0.5 * (x + yprev + 0.5 * h * f(x,t))
```

Bulirsch-Stoer method

The error in the modified midpoint method when integrating from t to $t+H$ in steps of $h_n = H/n$ is $O(h^2) + O(h^4) + O(h^6) + ...$ (even powers only)

Bulirsch-Stoer method: Use the modified midpoint method with various steps n to cancel error terms of higher and higher order (Richardson extrapolation, similar to Romberg method for numerical integration)

Let $R_{n,1}$ be an estimate of $x(t+H)$ from the n-step modified midpoint method ($h_n = H/n$)

$$x(t + H) = R_{n,1} + O(h_n^2).$$

One constructs high-order approximations $R_{n,m}$ such that

$$x(t + H) = R_{n,m} + O(h_n^{2m}),$$

Similar to Romberg integration one can derive

$$R_{n,m+1} = R_{n,m} + \frac{R_{n,m} - R_{n-1,m}}{[n/(n-1)]^{2m} - 1}. \qquad \textit{Bulirsch-Stoer method}$$

The method stops when the desired accuracy is achieved, $|R_{n,n}\text{-}R_{n,n\text{-}1}| < \varepsilon$

If n grows too large, it is better to split the $(t,t+H)$ interval into two subintervals $(t,t+H/2)$ & $(t+H/2,t+H)$ and apply the method recursively to each of them

Bulirsch-Stoer method implementation

```python
def bulirsch_stoer_step(f, x0, t0, H, delta = 1.e-6, distance_definition = distance_definition_default, maxsteps = 10):
    """Use Bulirsch-Stoer method to integrate for t to t+H.
    """
    n = 1
    R1 = np.empty([1,len(x0)],float)
    R1[0] = ode_MMM_multi(f, x0, t0, H, 1)
    error = 2. * H * delta
    while error > H*delta and n < maxsteps:
        n += 1
        R2 = R1
        R1 = np.empty([n,len(x0)],float)
        R1[0] = ode_MMM_multi(f, x0, t0, H, n)
        for m in range(1,n):
            epsilon = (R1[m-1]-R2[m-1])/((n/(n-1))**(2*m)-1)
            R1[m] = R1[m-1] + epsilon
        error = distance_definition(R1[n-2],R1[n-1])

    if n == maxsteps:
        # Reached maximum number of substeps in Bulirsch-Stoer method
        # reducing the time step and applying the method recursively
        sol1 = bulirsch_stoer_step(f, x0, t0, H/2., delta, distance_definition, maxsteps)
        sol2 = bulirsch_stoer_step(f, sol1[-1][1], t0 + H/2., H/2., delta, distance_definition, maxsteps)
        return sol1 + sol2

    return [[t0+H, R1[n - 1]]]
```

Bulirsch-Stoer method implementation

N-step Bulirsch-Stoer: apply the H step N times:

```python
def bulirsch_stoer(f, x0, t0, nsteps, tmax, delta = 1.e-6, distance_definition = distance_definition_default, maxsubstep
    """Use Bulirsch-Stoer method to integrate for t to tmax using nsteps Bulirsch-Stoer steps
    """
    H = (tmax - t0) / nsteps
    t = np.zeros(nsteps + 1)
    x = np.zeros((len(t), len(x0)))
    t = [t0]
    x = [x0]
    for i in range(0, nsteps):
        bst = bulirsch_stoer_step(f, x[-1], t[-1], H, delta, distance_definition, maxsubsteps)
        [t.append(el[0]) for el in bst]
        [x.append(el[1]) for el in bst]
    return t,x
```
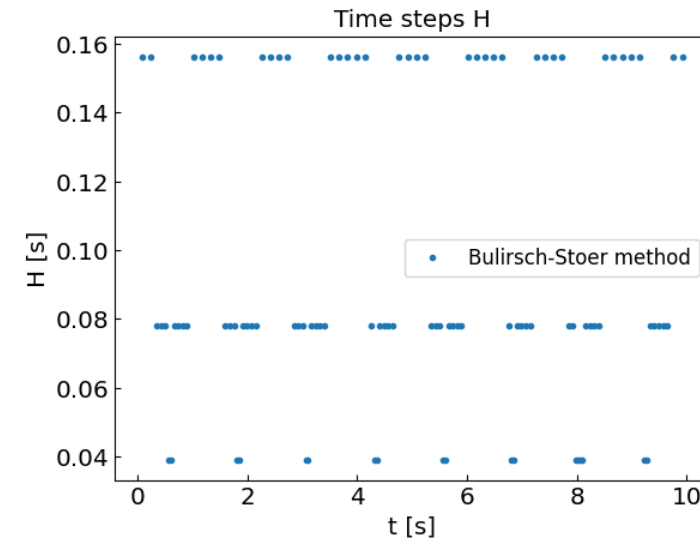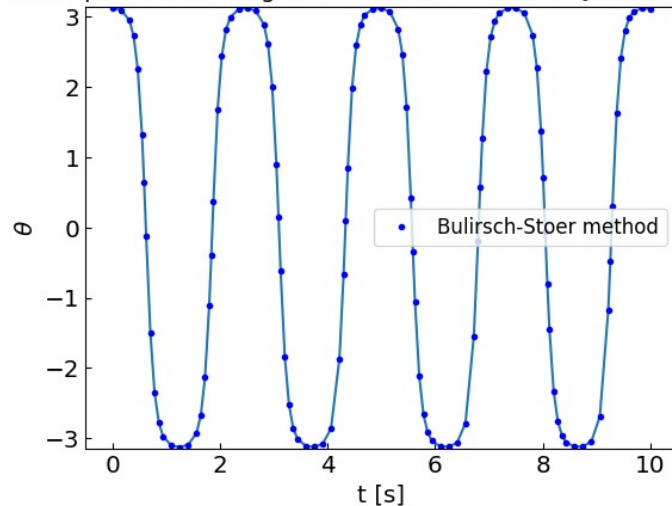
Bulirsch-Stoer method and non-linear pendulum

Apply the method to non-linear pendulum with the initial $H = 10$ (single step) and a maximum of 10 substeps. The method will adjust H as needed.

```python
theta0 = 179. * np.pi / 180.
omega0 = 0.
x0 = np.array([theta0,omega0])
a = 0.
b = 10.0
N = 1
eps = 1.e-8
maxsubsteps = 10

sol = bulirsch_stoer(fpendulum, x0, a, N, b, eps, error_definition_pendulum, maxsubsteps)
```
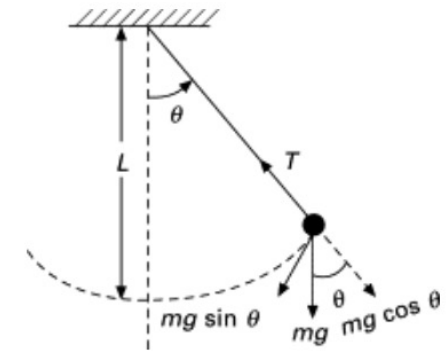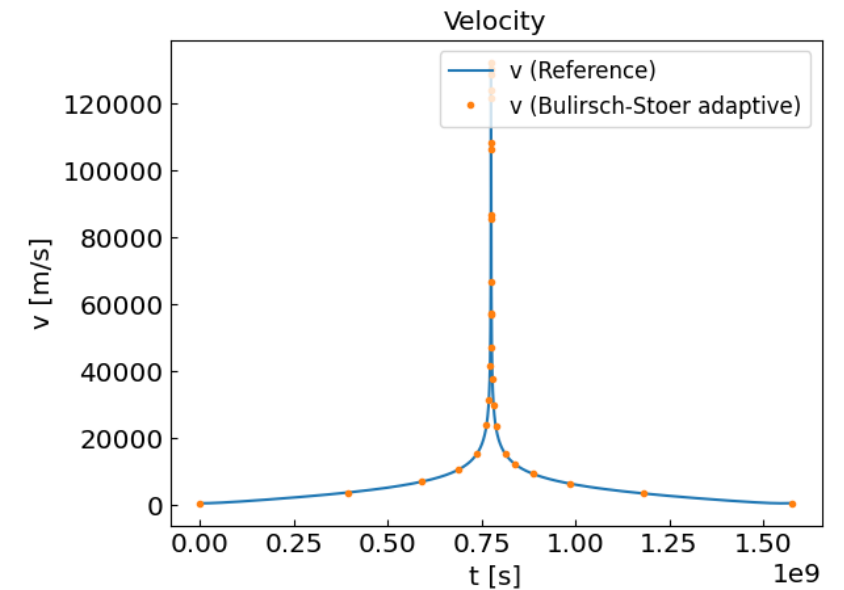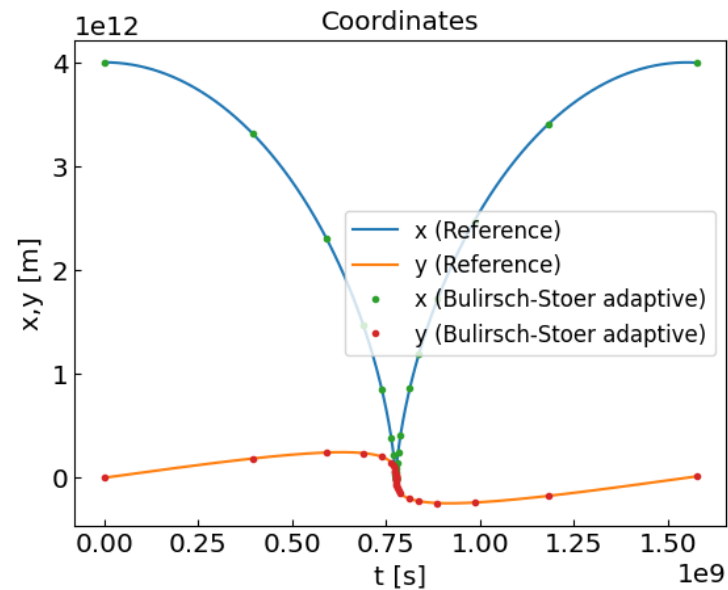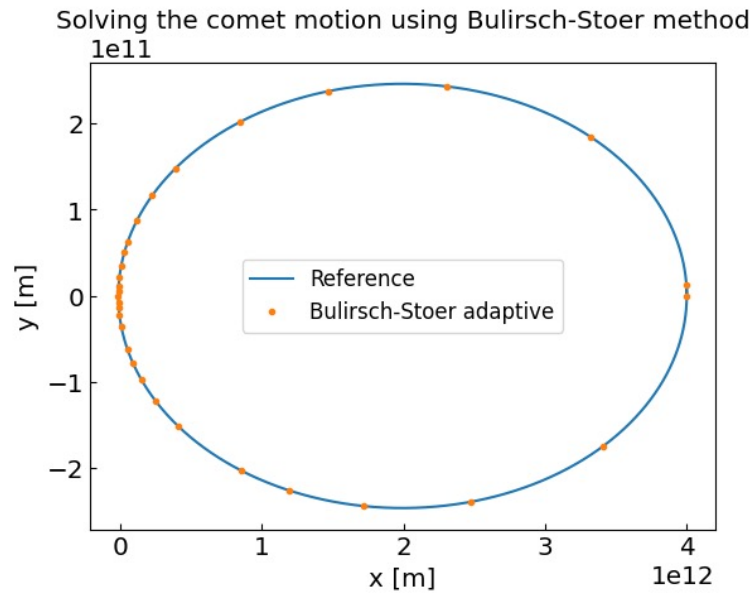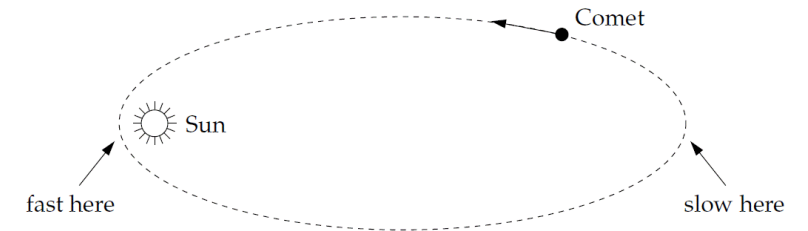
Bulirsch-Stoer method and the comet motion

Same for the comet motion. Accuracy: 1 km per day

```
x0 = [4.e12,0.,0.,500.]
a = 0.
b = 50. * 365. * 24. * 60. * 60.
N = 1
delta = 1. * 1.e3 / (365. * 24. * 60. * 60.)
sol = bulirsch_stoer(fcomet, x0, a, N, b, delta, error_definition_comet)
```

SIR model

The SIR model is the simplest model for infection disease dynamics in the population. The population is split into susceptible (S), infected (I), and recovered/immune (R) parts.

The SIR equations read:

$$\frac{dS}{dt} = -\beta SI,$$

$$\frac{dI}{dt} = \beta SI - \gamma I,$$

$$\frac{dR}{dt} = \gamma I.$$

```python
gam = 1./10.      # 10 days recovery rate
beta = 1./4.      # 4 days to infect other person
# R0 = beta/gam   # basic reproduction factor
kappa = 1. / 90. # immunity lasts for 90 days

def fSIR(xin, t):
    S = xin[0]
    I = xin[1]
    R = 1. - S - I
    return np.array([-beta * S * I, beta * I * S - gam * I])
```

Here β is the infection rate and γ is the recovery rate.

The ratio $R_0 = \beta/\gamma$ is basic reproduction number.

Given that $S + I + R = 1 = const$ at all times,
one only needs to solve two ODEs, e.g. dS/dt and dI/dt.

SIR model

Solve the SIR model equations using e.g. Bulirsch-Stoer method

```python
t0 = 0.
tend = 365.

I0 = 1.e-5             # Initial fraction of infected
x0 = [1. - I0, I0]    # Initial conditions

delta = 1.e-9 # The desired accuracy per day
N = 50         # Minimum number of steps
sol = bulirsch_stoer(fSIR, x0, t0, N, tend, delta)
```
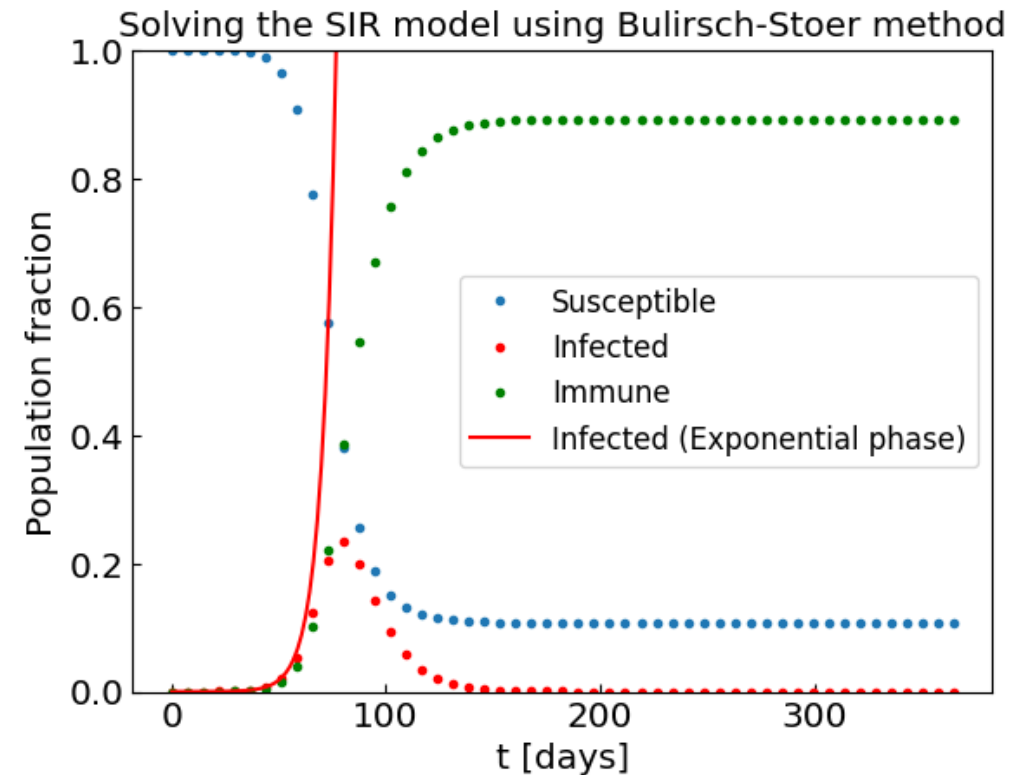
SIR model

Solve the SIR model equations using e.g. Bulirsch-Stoer method

```python
t0 = 0.
tend = 365.

I0 = 1.e-5           # Initial fraction of infected
x0 = [1. - I0, I0]   # Initial conditions

delta = 1.e-9 # The desired accuracy per day
N = 50           # Minimum number of steps
sol = bulirsch_stoer(fSIR, x0, t0, N, tend, delta)
```


Solving the SIR model using Bulirsch-Stoer method

One can clearly see the initial exponential phase of the epidemic, and its end once a sufficient fraction of the population obtained immunity.

Modified SIR model

What if the immunity disappears with time?
Introduce the loss of immunity rate κ

$$\frac{dS}{dt} = -\beta SI + \kappa R,$$
$$\frac{dI}{dt} = \beta SI - \gamma I,$$
$$\frac{dR}{dt} = \gamma I - \kappa R.$$

```python
gam = 1./10.       # 10 days recovery rate
beta = 1./4.       # 4 days to infect other person
# R0 = beta/gam  # basic reproduction factor
kappa = 1. / 90. # immunity lasts for 90 days

def fSIR(xin, t):
    S = xin[0]
    I = xin[1]
    R = 1. - S - I
    # print(xin)
    return np.array([-beta * S * I + immu * (1. - S - I), beta * I * S - gam * I])

t0 = 0.
tend = 365.

I0 = 1.e-5
x0 = [1. - I0, I0]

delta = 1.e-9 # The desired accuracy per day
N = 50
sol = bulirsch_stoer(fSIR, x0, t0, N, tend, delta)
```

Modified SIR model

What if the immunity disappears with time?
Introduce the loss of immunity rate κ

$$\frac{dS}{dt} = -\beta SI + \kappa R,$$
$$\frac{dI}{dt} = \beta SI - \gamma I,$$
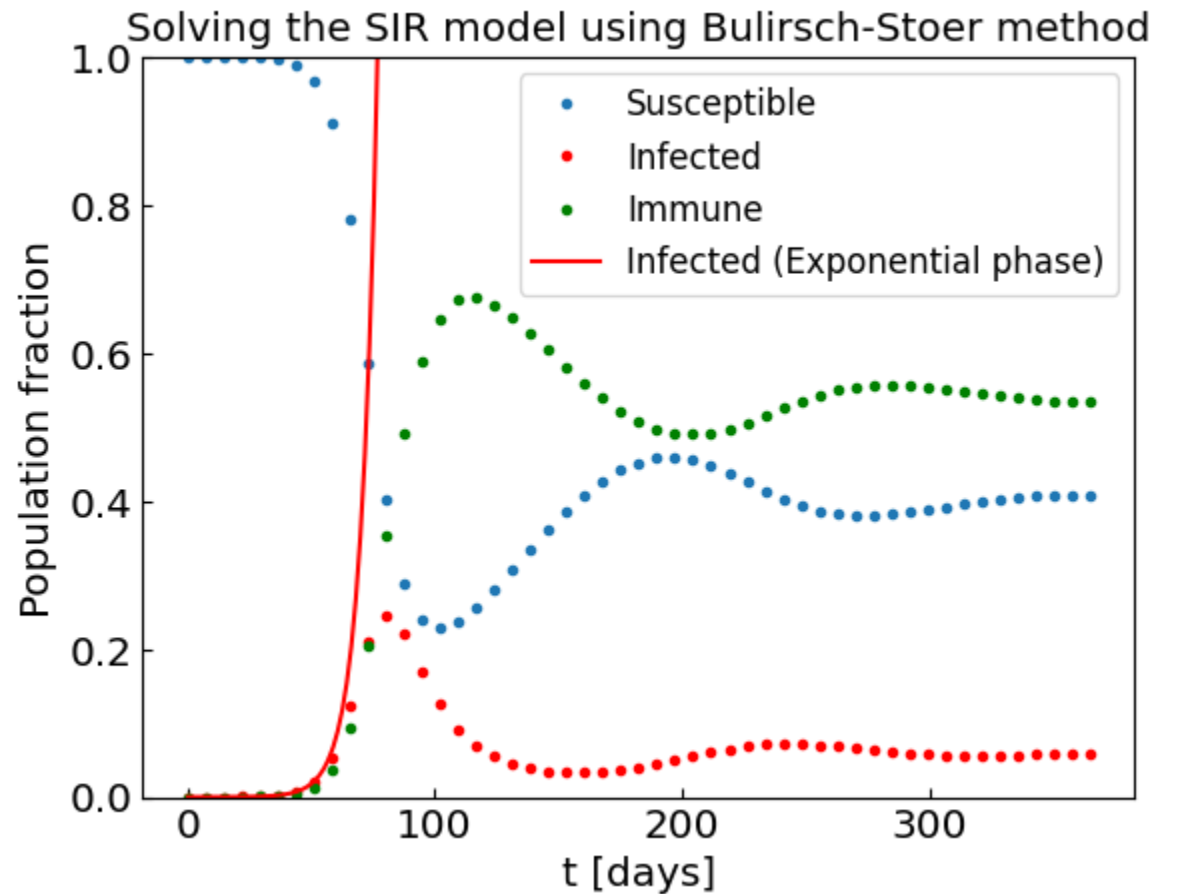$$\frac{dR}{dt} = \gamma I - \kappa R.$$

```python
gam = 1./10.      # 10 days recovery rate
beta = 1./4.      # 4 days to infect other person
# R0 = beta/gam  # basic reproduction factor
kappa = 1. / 90. # immunity lasts for 90 days

def fSIR(xin, t):
    S = xin[0]
    I = xin[1]
    R = 1. - S - I
    # print(xin)
    return np.array([-beta * S * I + immu * (1. - S - I), beta * I * S - gam * I])

t0 = 0.
tend = 365.

I0 = 1.e-5
x0 = [1. - I0, I0]

delta = 1.e-9 # The desired accuracy per day
N = 50
sol = bulirsch_stoer(fSIR, x0, t0, N, tend, delta)
```


Solving the SIR model using Bulirsch-Stoer method

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).
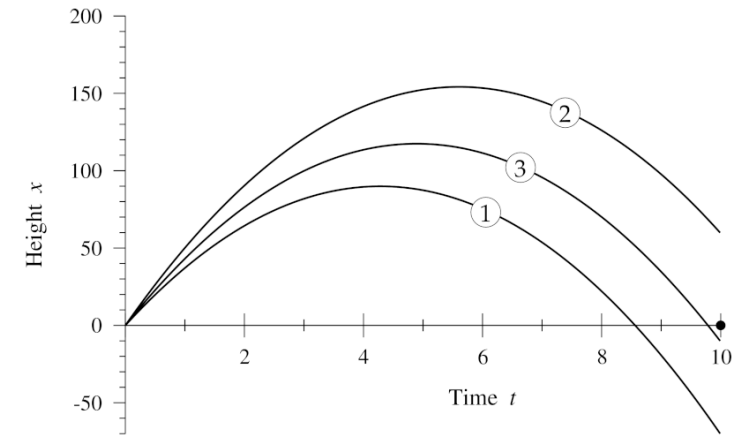
Shooting method for vertically thrown object

Search for v_0 using bisection method and solve the intermediate ODEs using RK4

```python
g = 9.81 # m/s^2
# ODEs
def fball(xin,t):
    x = xin[0]
    v = xin[1]
    return np.array([v,-g])

# Initial and final times
t0 = 0.
tend = 10.
# Number of RK4 steps
Nrk4 = 100
hrk4 = (tend - t0) / Nrk4

# Desired accuracy for v0
accuracy_v0 = 1.e-10 # m/s

v0min = 0.01    # m/s
v0max = 1000.0 # m/s

def fbisection(v0):
    x0 = [0., v0]
    return ode_rk4_multi(fball, x0, t0, hrk4, Nrk4)[1][-1][0]

v0sol = bisection_method(fbisection, v0min, v0max, accuracy_v0)
print("The required initial velocity is",v0sol,"m/s")
```

```
The required initial velocity is 49.0500000000017 m/s
```