Runge-Kutta Methods for solving ODEs

ODE Solvers

Runge Kutta

- Euler Method (1st order)
- Midpoint Method (2nd order)
- RK4 (4th order) - SciPy library
- RK45 (adaptive time step 4th and 5th order) - SciPy library - **BEST general purpose solver**

Symplectic

- Euler-Cromer-Aspel Method (1st order)
- Leapfrog (2nd order) - **BEST for Hamiltonian (including Energy-Conserving) systems**

Stiff Solvers

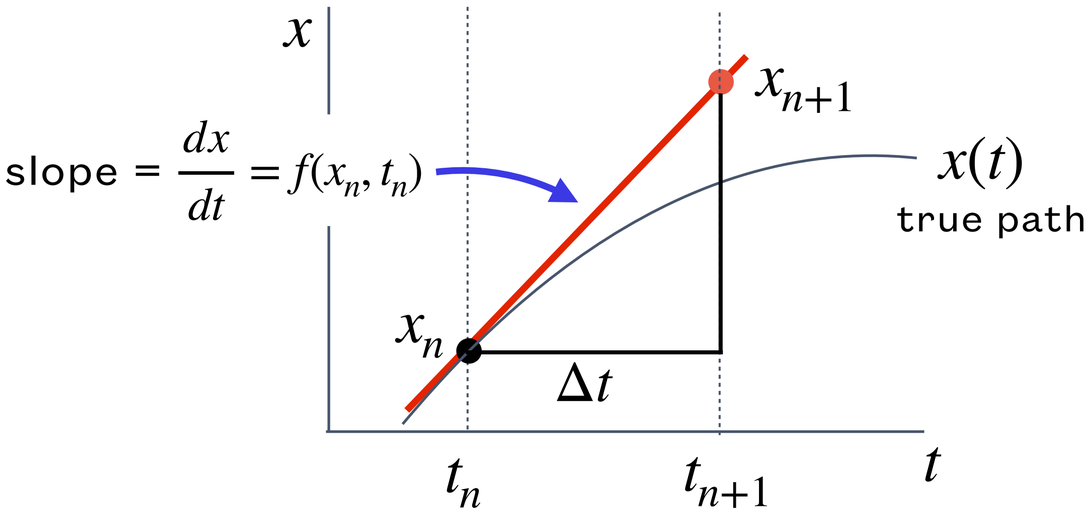
- Radau (5th order) - SciPy library - **Best all-purpose solver for stiff equations**
- BDF (order 1-5) - SciPy library

Euler Method
(1st order Runge Kutta)

Euler Method

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

$$x_{n+1} = x_n + f(x_n, t_n)\Delta t$$



Approximate $x(t)$ as a Taylor series:

$$x(t_n + \Delta t) \approx x_n + \left(\frac{dx}{dt}\right)_{t_n} \Delta t + \left(\frac{d^2x}{dt^2}\right)_{t_n} \Delta t^2 + \ldots$$

The Euler method is equivalent to including the constant term and the linear term

The Euler method is said to be **first-order accurate**

S.H.O. Derivative Function

Returns derivatives:

$$\frac{dx}{dt} = v \qquad \frac{dv}{dt} = -(k/m)x$$

- Passed parameters:
  - t = time
  - y = array containing x & v
  - m = mass
  - k = spring constant
- Returned value:
  - array containing dx/dt & dv/dt

```
def deriv_sho(t, y, m, k):

    # extract variables from y array
    x = y[0]            # position
    v = y[1]            # velocity

    # calculate derivatives
    dxdt = v
    dvdt = -k/m*x

    # return derivatives in a numpy array
    return np.array([dxdt, dvdt])
```

Multi-Variable Euler Function

Performs numerical integration using the Euler method

- Passed parameters:
  - deriv = derivative function
  - y0 = array of initial conditions
  - tmax = max time of integration
  - dt = time step
  - params = array of parameters to pass to deriv() function
- Returned values:
  - t = array of times
  - y = 1D or 2D array containing solution (each column represents different variable)

```
#########  Multi-Variable Euler Integration  #########

def Euler_Vec(deriv, y0, tmax, dt, params):

    #########  Create Arrays  #########

    # determine the number of variables in the system from initial
    nvar = 1 if not isinstance(y0, np.ndarray) else y0.size

    N = int(tmax/dt)+1          # number of steps in simulation
    y = np.zeros((N,nvar))      # array to store y values
    t = np.zeros(N)             # array to store times

    if nvar == 1:
        y[0] = y0               # assign initial value if single var
    else:
        y[0,:] = y0             # assign vector initial values if mu

    #########  Loop to implement the Euler update rule  #########

    for n in range(N-1):
        f = deriv(t[n], y[n], *params)
        y[n+1] = y[n] + f*dt
        t[n+1] = t[n] + dt

    return t, y
```

Plot solution and error

Plots numerical solution and analytic solution in upper plot.
Plots error in the lower plot.

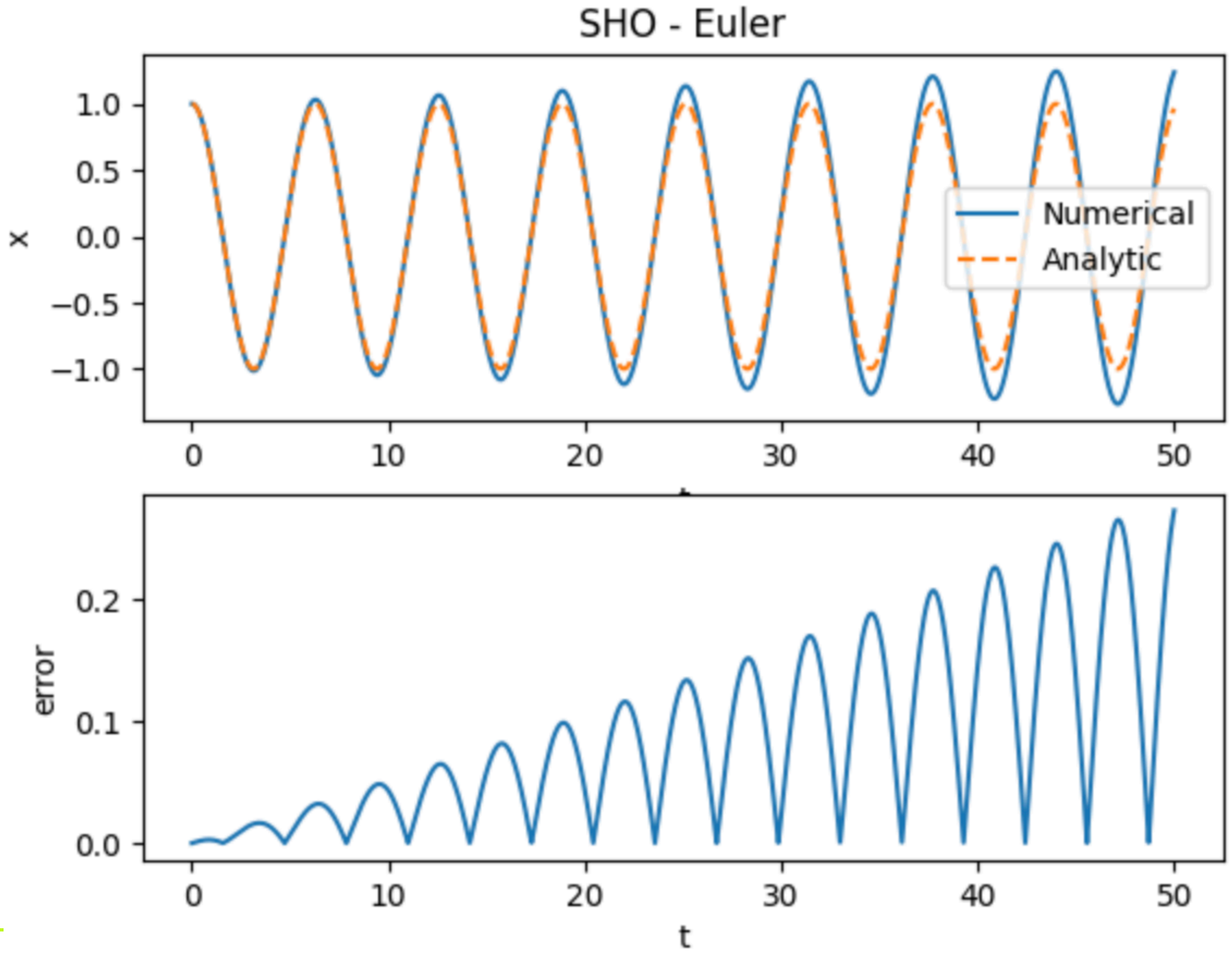
- Passed parameters:
  - x = solution dor position
  - y_true = analytic solution
  - t = time values
  - title =string containing title
- Returned values:
  - None

```
def plot_solution(x, x_true, t, title):

    err = np.abs(x-x_true)      # calculate numerical error

    #########  Plot Solution  #########

    plt.subplot(2,1,1)                              # upper s
    plt.plot(t, x, label='Numerical')               # plot po
    plt.plot(t, x_true, '--', label='Analytic')     # analyti

    plt.xlabel('t')             # label the x and y axes
    plt.ylabel('x')
    plt.title(title)            # give the plot a title
    plt.legend()                # display the legend

    #########  Plot Error  #########

    plt.subplot(2,1,2)          # lower subplot
    plt.plot(t, err)            # plot position
    plt.xlabel('t')             # label the x and y axes
    plt.ylabel('error')

    plt.show()                          # display the plot
```

Put it all together: Use the Euler Method to numerically integrate the Simple Harmonic Oscillator

```
#########  Parameters  #########

m    = 1          # mass
k    = 1          # spring constant
tmax = 50         # maximum time
dt   = 0.01       # time step
x0   = 1          # initial position
v0   = 0          # initial velocity

params = np.array([m,k])    # bundle derivative parameters
y0 = np.array([x0,v0])      # bundle initial conditions to


#########  Perform Euler Integration  #########

t, y =  Euler(deriv_sho, y0, tmax, dt, params)

x = y[:,0]        # extract positions
v = y[:,1]        # extract velocities

#########  Analytic Solution  #########

omega = np.sqrt(k/m)
x_true = x0 * np.cos(omega*t)

#########  Plot Solution  #########

plot_solution(x, x_true, t, "SHO - Euler")
```

With a time step of $\Delta t = 0.01$, the absolute error is around 0.3 after 8 oscillations.



Midpoint Method
(2nd order Runge Kutta)

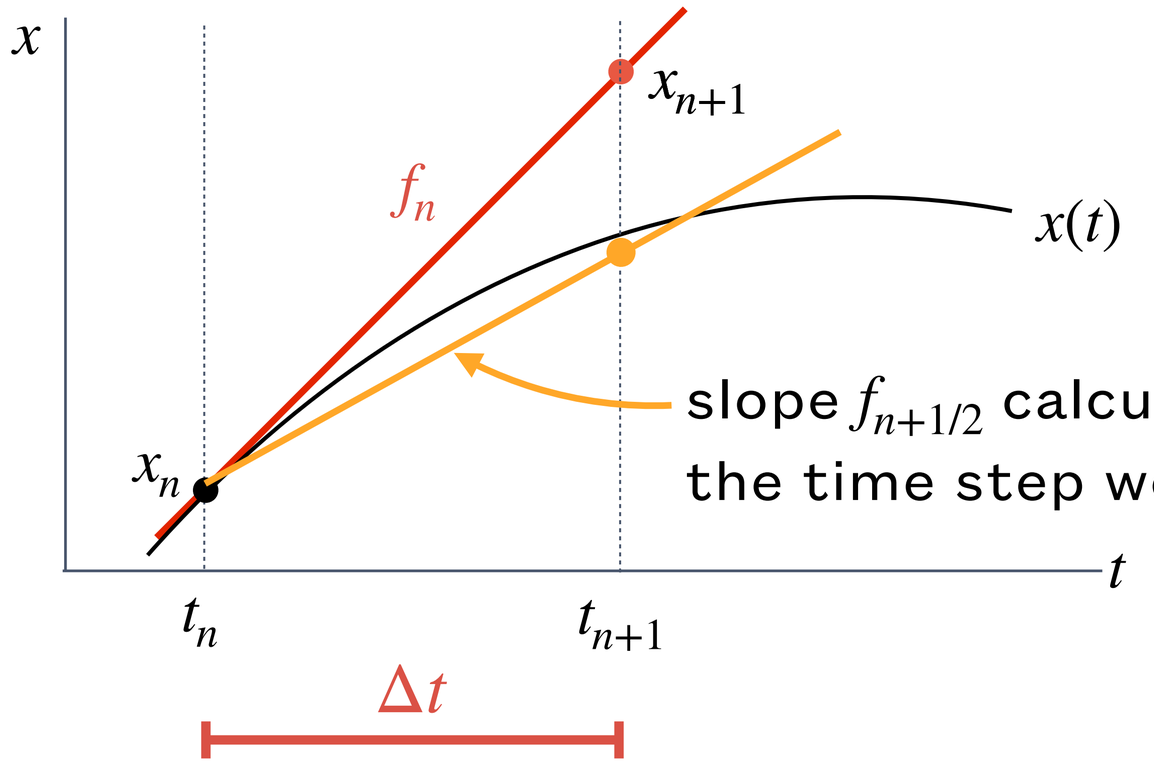
How to improve Euler method?

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

$$x_{n+1} = x_n + f_n \Delta t$$

$$f_n = f(x_n, t_n)$$

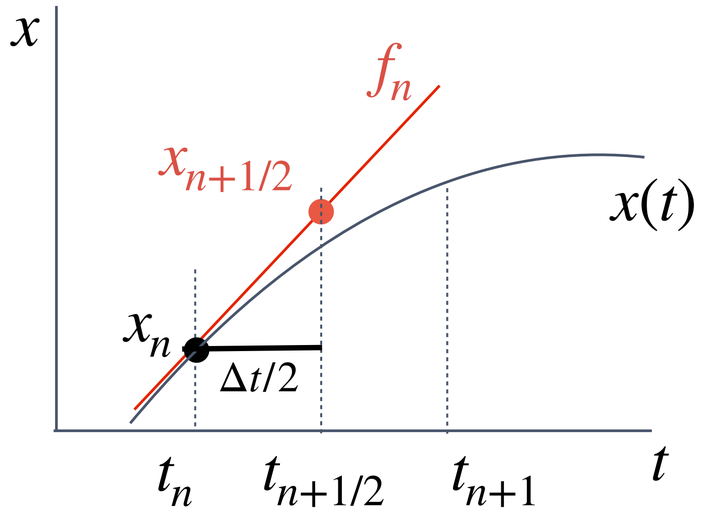
slope f_n calculated from x_n is too steep

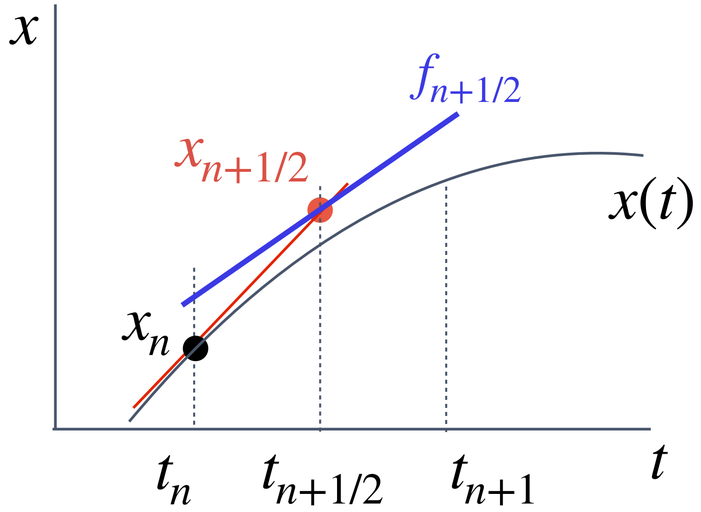


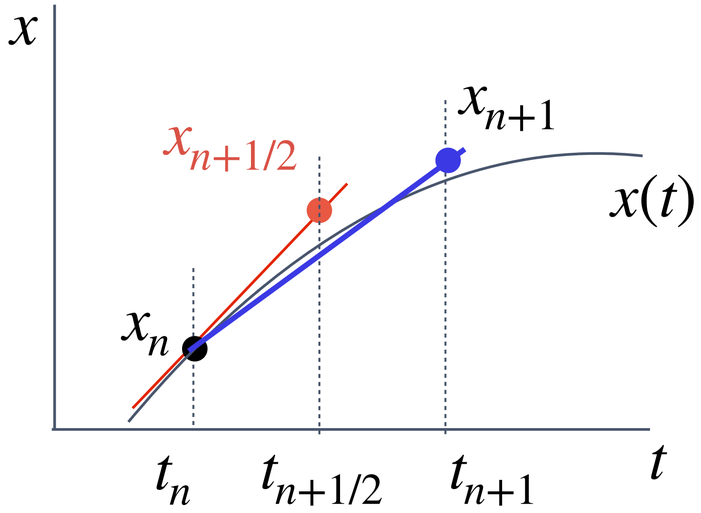
slope $f_{n+1/2}$ calculated in the middle of the time step would be better

Midpoint Method: estimate the slope at the midpoint of the time step

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







$$x_{n+1/2} = x_n + f_n \frac{\Delta t}{2}$$

$$f_n = f(x_n, t_n)$$

$$f_{n+1/2} = f(x_{n+1/2}, t_{n+1/2})$$

$$x_{n+1} = x_n + f_{n+1/2}\Delta t$$

Midpoint Method: estimate the slope at the midpoint of the time step

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

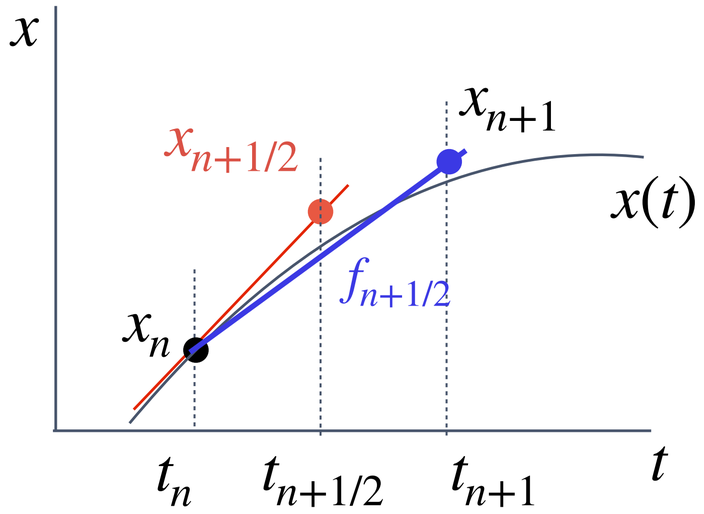
Update rule:

$$x_{n+1/2} = x_n + f_n \frac{\Delta t}{2}$$

$$x_{n+1} = x_n + f_{n+1/2}\Delta t$$

$$f_n = f(x_n, t_n)$$

$$f_{n+1/2} = f(x_{n+1/2}, t_{n+1/2})$$



Midpoint Method is 2nd-Order Accurate

$$x_{n+1/2} = x_n + v_n \frac{\Delta t}{2}$$

$$v_{n+1/2} = v_n + a_n \frac{\Delta t}{2}$$

$$x_{n+1} = x_n + v_{n+1/2} \Delta t$$

$$v_{n+1} = v_n + a_{n+1/2} \Delta t$$

$$x_{n+1} = x_n + \left(v_n + a_n \frac{\Delta t}{2} \right) \Delta t$$

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n \Delta t^2$$

Compare to Taylor series for $x(t)$:

$$x(t_n + \Delta t) \approx x_n + \left(\frac{dx}{dt} \right)_{t_n} \Delta t + \left(\frac{d^2x}{dt^2} \right)_{t_n} \Delta t^2 + \ldots$$

Midpoint method includes the $O(\Delta t^2)$ term

Midpoint Method in Action

Euler rule

$$x_{n+1} = x_n + f(x_n, t_n)\Delta t$$

```
for n in range(N-1):
    f = deriv(t[n], y[n], *params)
    y[n+1] = y[n] + f*dt
    t[n+1] = t[n] + dt
```

Midpoint rule:

$$y_{n+1/2} = y_n + f_n \frac{\Delta t}{2} \qquad f_n = a y_n$$

$$y_{n+1} = y_n + f_{n+1/2}\Delta t \qquad f_{n+1/2} = a y_{n+1/2}$$

```
for n in range(N-1):
    f =              # ???
    y_half =         # ???
    f =              # ???
    y[n+1] =         # ???
    t[n+1] = t[n] + dt
```

Modify the Euler_vec() function to apply the Midpoint Method

```
def Midpoint(deriv, y0, tmax, dt, params):

    #########  Create Arrays  #########

    # determine the number of variables in the system from initial conditions
    nvar = 1 if not isinstance(y0, np.ndarray) else y0.size

    N = int(tmax/dt)+1          # number of steps in simulation
    y = np.zeros((N,nvar))      # array to store y values
    t = np.zeros(N)             # array to store times

    if nvar == 1:
        y[0] = y0               # assign initial value if single variable
    else:
        y[0,:] = y0             # assign vector initial values if multivariable

    #########  Loop to implement the Midpoint update rule  #########

    for n in range(N-1):
        f =          # YOUR CODE HERE     # evaluate derivatives on whole step
        y_half =     # YOUR CODE HERE     # half step
        f =          # YOUR CODE HERE     # evaluate derivatives on half step
        y[n+1] =     # YOUR CODE HERE     # whole step
        t[n+1] = t[n] + dt

    return t, y
```

Modify the Euler_vec() function to apply the Midpoint Method

```
def Midpoint(deriv, y0, tmax, dt, params):

    #########  Create Arrays  #########

    # determine the number of variables in the system from initial conditions
    nvar = 1 if not isinstance(y0, np.ndarray) else y0.size

    N = int(tmax/dt)+1          # number of steps in simulation
    y = np.zeros((N,nvar))      # array to store y values
    t = np.zeros(N)             # array to store times

    if nvar == 1:
        y[0] = y0               # assign initial value if single variable
    else:
        y[0,:] = y0             # assign vector initial values if multivariable

    #########  Loop to implement the Euler update rule  #########

    for n in range(N-1):
        f = deriv(t[n], y[n], *params)          # evaluate derivatives on whole step
        y_half = y[n] + f * dt/2                # half step
        f = deriv(t[n]+dt/2, y_half, *params)   # evaluate derivatives on half step
        y[n+1] = y[n] + f*dt                    # whole step
        t[n+1] = t[n] + dt

    return t, y
```

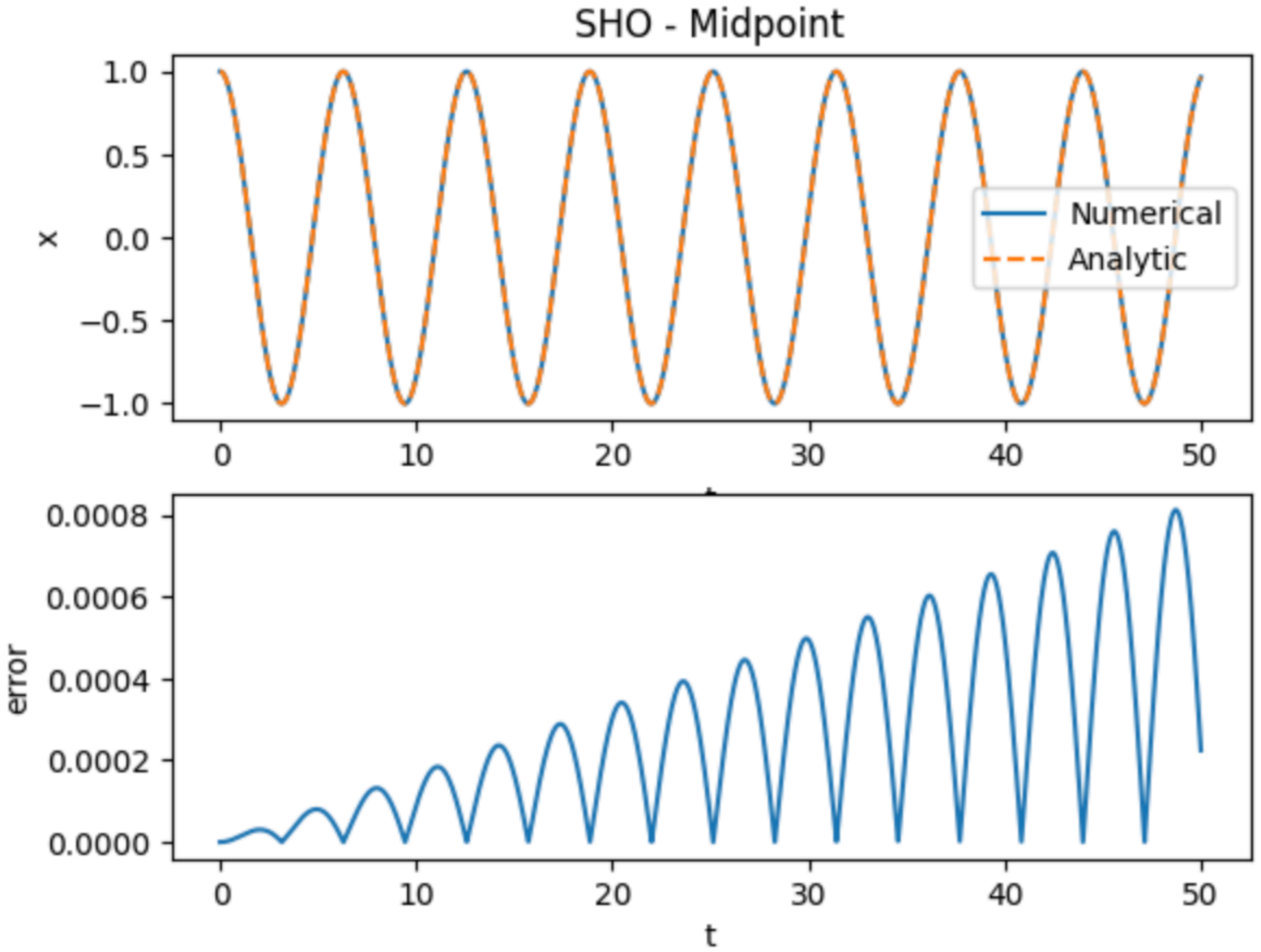
Solution

Apply Midpoint Method to the Simple Harmonic Oscillator

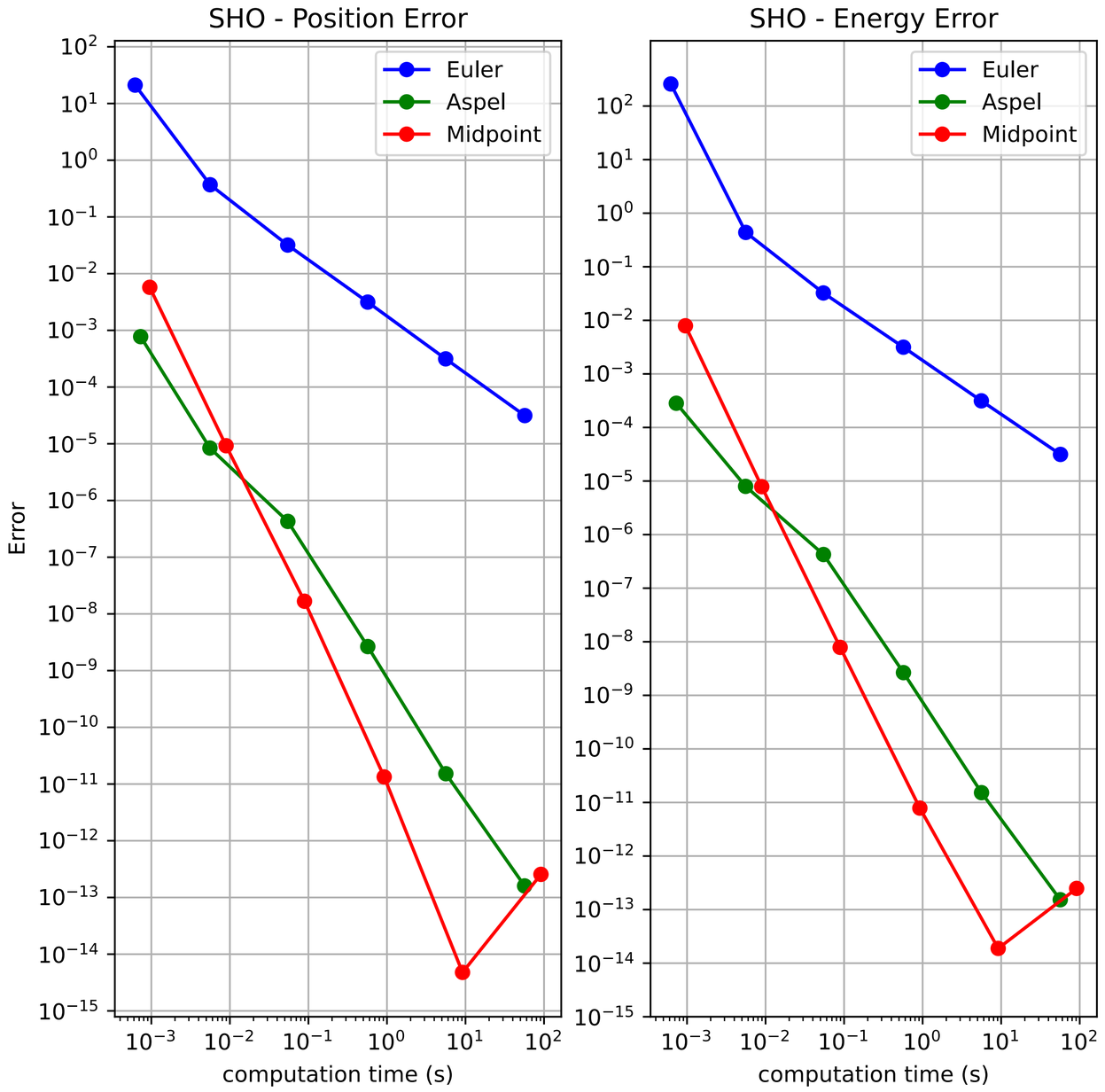
Run your code with the same parameters used for the Euler integration:

- dt = 0.01
- tmax = 50

Error reduced from 0.3 to 4×10^{-4}

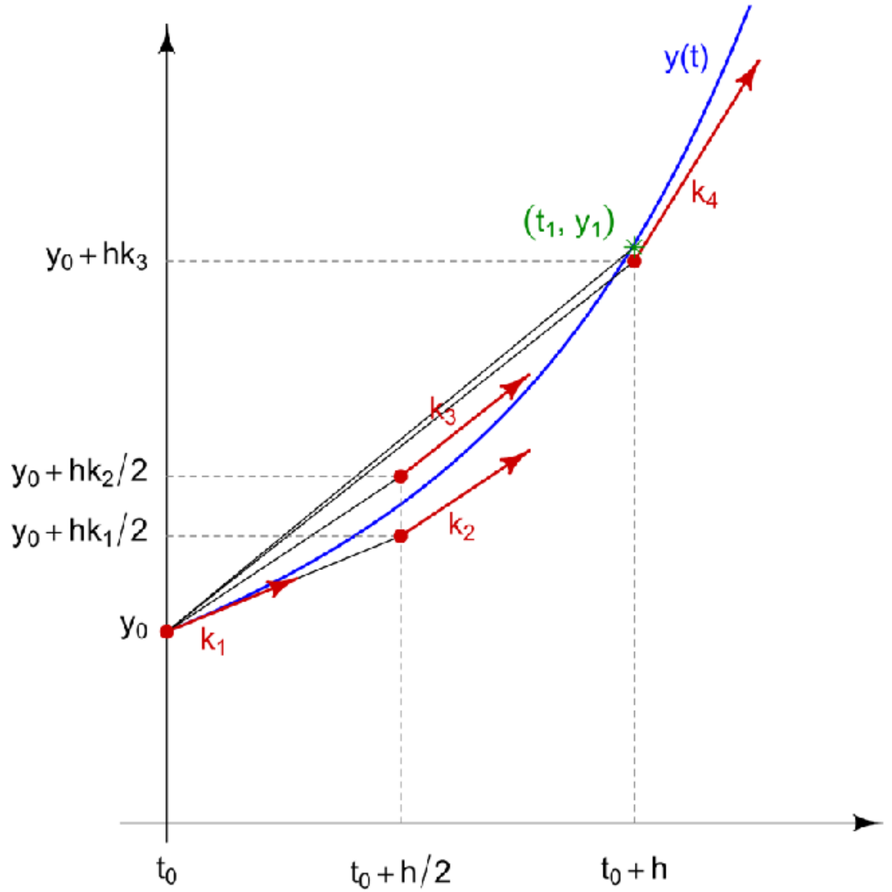


Midpoint Method is comperable to the Euler-Cromer-Aspel method at large time steps, and 2-3 orders of magnitude more accurate at small time steps



4th order Runge Kutta

Runga-Kutta RK4 Method (4th-Order Accurate)



$$k_1 = f(t_n, y_n)$$

$$k_2 = f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{k_1}{2}\Delta t\right)$$

$$k_3 = f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{k_2}{2}\Delta t\right)$$

$$k_4 = f\left(t_n + \Delta t, y_n + k_3\Delta t\right)$$

$$y_{n+1} = y_n + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

Runga-Kutta RK4 Method (4th-Order Accurate)

$$k_1 = f(t_n, y_n)$$

$$k_2 = f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{k_1}{2}\Delta t\right)$$

$$k_3 = f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{k_2}{2}\Delta t\right)$$

$$k_4 = f\left(t_n + \Delta t, y_n + k_3 \Delta t\right)$$

$$y_{n+1} = y_n + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

```
for n in range(N-1):
    k1 =          #  ??
    k2 =          #  ??
    k3 =          #  ??
    k4 =          #  ??
    y[n+1] =      #  ??

    t[n+1] = t[n] + dt
```

Modify the midpoint() function to apply the RK4 Method

```
def RK4(deriv, y0, tmax, dt, params):

    #########  Create Arrays  #########

    # determine the number of variables in the system from initial conditions
    nvar = 1 if not isinstance(y0, np.ndarray) else y0.size

    N = int(tmax/dt)+1          # number of steps in simulation
    y = np.zeros((N,nvar))      # array to store y values
    t = np.zeros(N)             # array to store times

    if nvar == 1:
        y[0] = y0               # assign initial value if single variable
    else:
        y[0,:] = y0             # assign vector initial values if multivariable

    #########  Loop to implement the Euler update rule  #########

    for n in range(N-1):
        k1 =          # YOUR CODE HERE
        k2 =          # YOUR CODE HERE
        k3 =          # YOUR CODE HERE
        k4 =          # YOUR CODE HERE
        y[n+1] =      # YOUR CODE HERE

        t[n+1] = t[n] + dt

    return t, y
```

Modify the midpoint() function to apply the RK4 Method

```
def RK4(deriv, y0, tmax, dt, params):

#########  Create Arrays  #########

# determine the number of variables in the system from initial conditions
nvar = 1 if not isinstance(y0, np.ndarray) else y0.size

N = int(tmax/dt)+1          # number of steps in simulation
y = np.zeros((N,nvar))      # array to store y values
t = np.zeros(N)             # array to store times

if nvar == 1:
    y[0] = y0               # assign initial value if single variable
else:
    y[0,:] = y0             # assign vector initial values if multivariable

#########  Loop to implement the Euler update rule  #########

for n in range(N-1):
    k1 = deriv(t[n], y[n], *params) * dt
    k2 = deriv(t[n], y[n]+0.5*k1, *params) * dt
    k3 = deriv(t[n], y[n]+0.5*k2, *params) * dt
    k4 = deriv(t[n], y[n]+k3, *params) * dt
    y[n+1] = y[n] + (1/6)*(k1 + 2*k2 + 2*k3 + k4)

    t[n+1] = t[n] + dt

return t, y
```

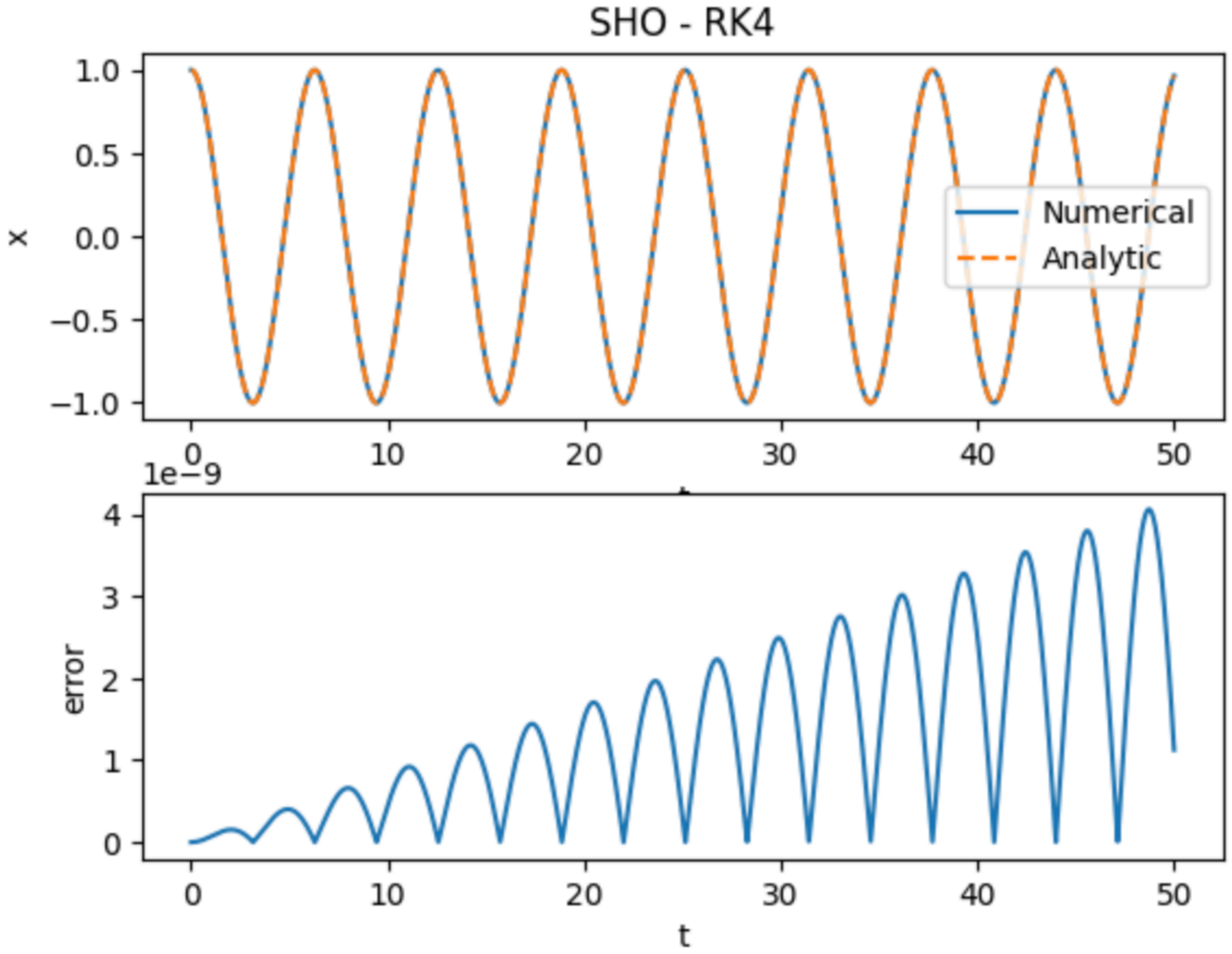
Solution

Apply RK4 Method to the Simple Harmonic Oscillator

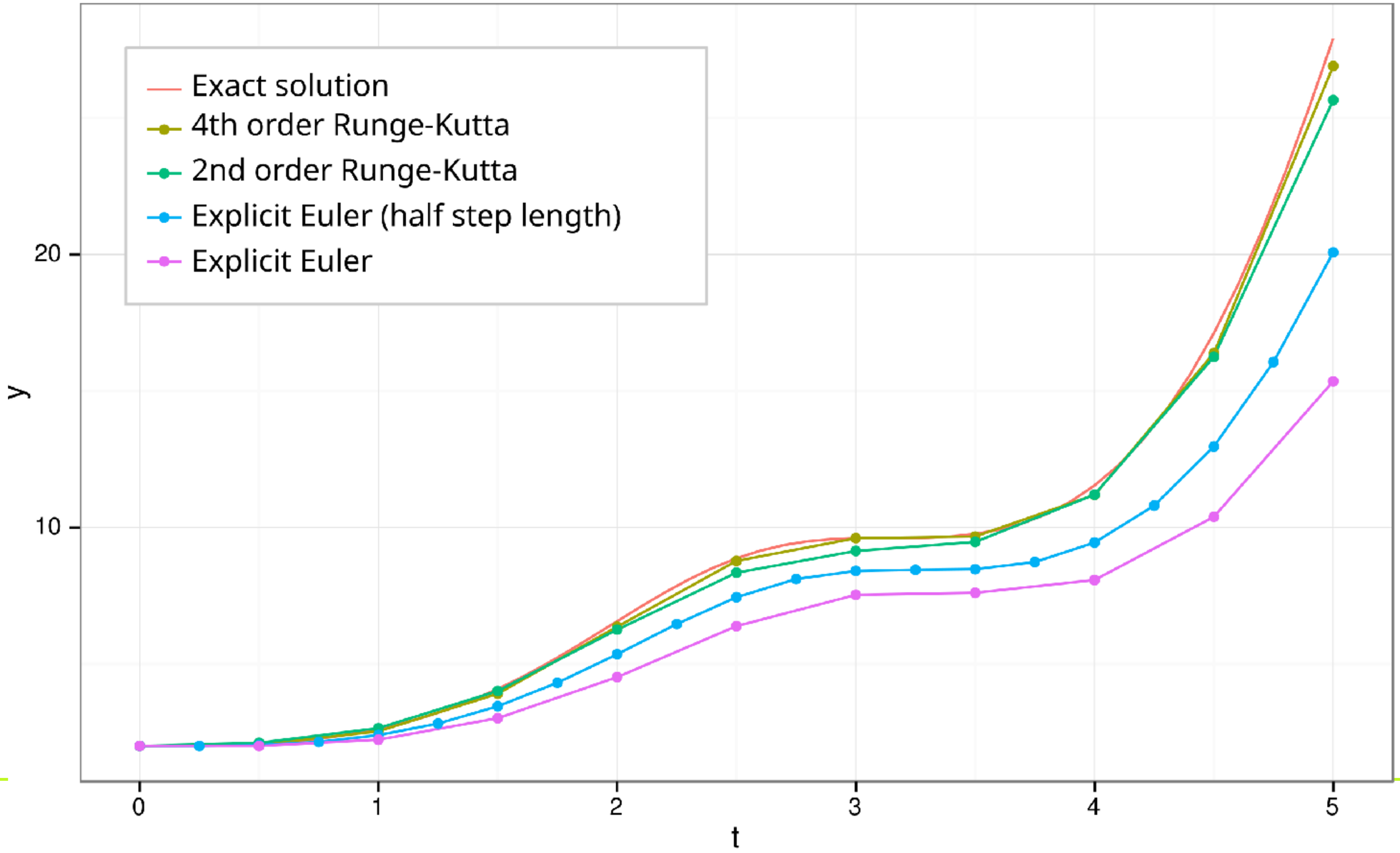
Run your code with the same parameters used for the Euler integration:

- dt = 0.01
- tmax = 50

Error reduced from 0.3 to 4×10^{-9}



Comparison of Runge Kutta Methods

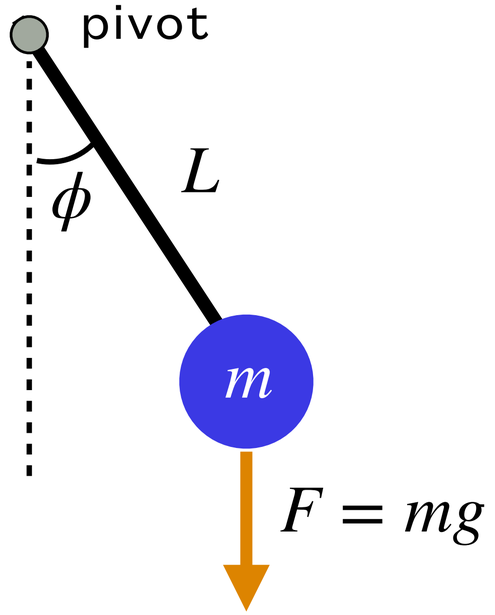


Application: Simple Pendulum

Simple Pendulum

Consider a pendulum consisting of a mass m attached to a pivot with a massless rod

- m = mass
- L = distance from pivot to mass
- ϕ = angle from point where mass hangs straight down



Rotational version of Newton's 2nd law: $\tau = I\alpha$

- Torque of weight of the mass around pivot: $\tau = -(L\sin\phi)mg$
- Rotational inertia of point mass $I = mL^2$

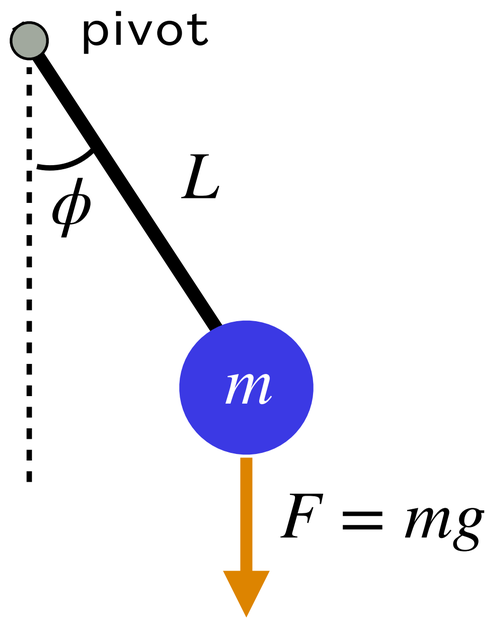
Solve for the angular acceleration: $-(L\sin\phi)mg = mL^2\alpha \rightarrow \alpha = -\frac{g}{L}\sin\phi$

$$\boxed{\frac{d^2\phi}{dt^2} = -\frac{g}{L}\sin\phi}$$

Simple Pendulum

Our 2nd-order equation of motion may be broken up into two 1st-order equations:

$$\frac{d\phi}{dt} = \omega \qquad \frac{d\omega}{dt} = -\frac{g}{L}\sin\phi$$



where we introduced the angular velocity ω.

Thus, we have a pair of coupled, first-order ODEs with variables ϕ and ω.

Solve this system of equations using your RK4 integration scheme for $\omega_0 = 0$ and:

$$\phi_0 = 45^\circ, 90^\circ, 135^\circ, 179^\circ$$