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

Adaptive Time Step Runge Kutta 45

Adaptive Time Step - Runge Kutta 45

Method:

1. Step forward using RK4 method $y_{RK4,n+1}$
2. Recalculate using a 5th-order Runge Kutta method $y_{RK5,n+1}$
3. Find error between the two methods: $\epsilon = |y_{RK4,n+1} - y_{RK4,n+1}|$
4. Test if $\epsilon > atol + rtol \cdot |y_n|$, then reduce the time step Δt by half and go back to step 1.
    - atol = absolute tolerance
    - rtol = relative tolerannce
5. Test if $\epsilon \leq atol + rtol \cdot |y_n|$, then accept the value $y_{RK4,n+1}$ and go to the next step

Use the SciPy library to implement the RK45 method

1. Load solve_ivp from SciPy library

```
from scipy.integrate import solve_ivp
```

2. Call solve_ivp to perform integration

derivative function

time limits

initial conditions

integration method

```
sol = solve_ivp(deriv_sho, (0,tmax), y0, method='RK45', args=params,
    atol=1e-4,rtol=1e-3)
```

absolute tolerance

relative tolerance

parameters for derivative funvtion

Use the SciPy library to implement the RK45 method

```
sol = solve_ivp(deriv_sho, (0,tmax), y0, method='RK45', args=params,
    atol=1e-4,rtol=1e-3)
```

3. Extract time and solution from the solution object sol:

```
t = sol.t              # extract times
x = sol.y[0,:]         # extract positions
v = sol.y[1,:]         # extract velocities
```

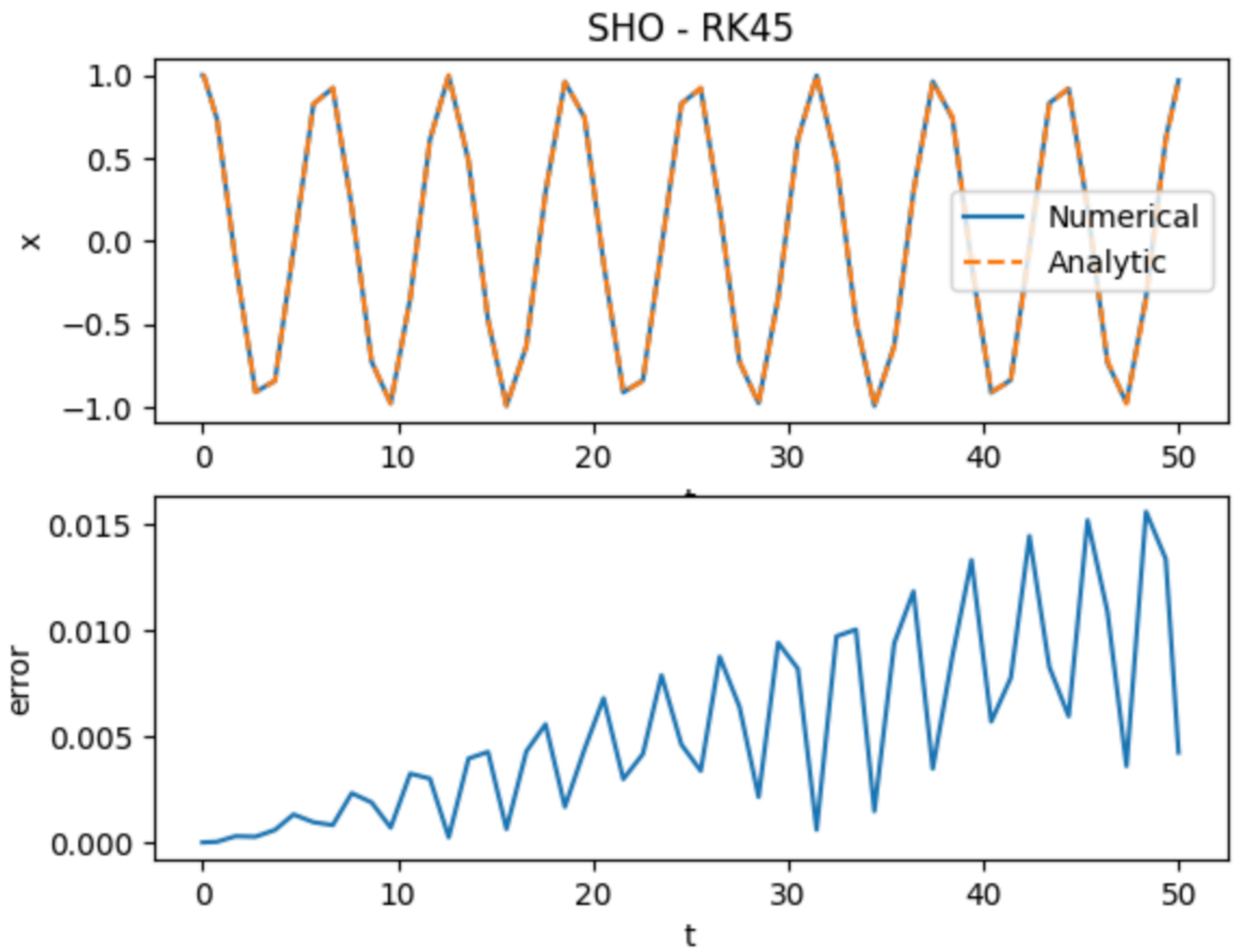
Complete code to implement RK45 integration

Solution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

#########  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 together
y0 = np.array([x0,v0])      # bundle initial conditions together

#########  Perform RK45 Integration  #########
sol = solve_ivp(deriv_sho, (0,tmax), y0, method='RK45', args=params,
    atol=1e-4,rtol=1e-3)
t = sol.t             # extract times
x = sol.y[0,:]        # extract positions
v = sol.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 - RK45")
```

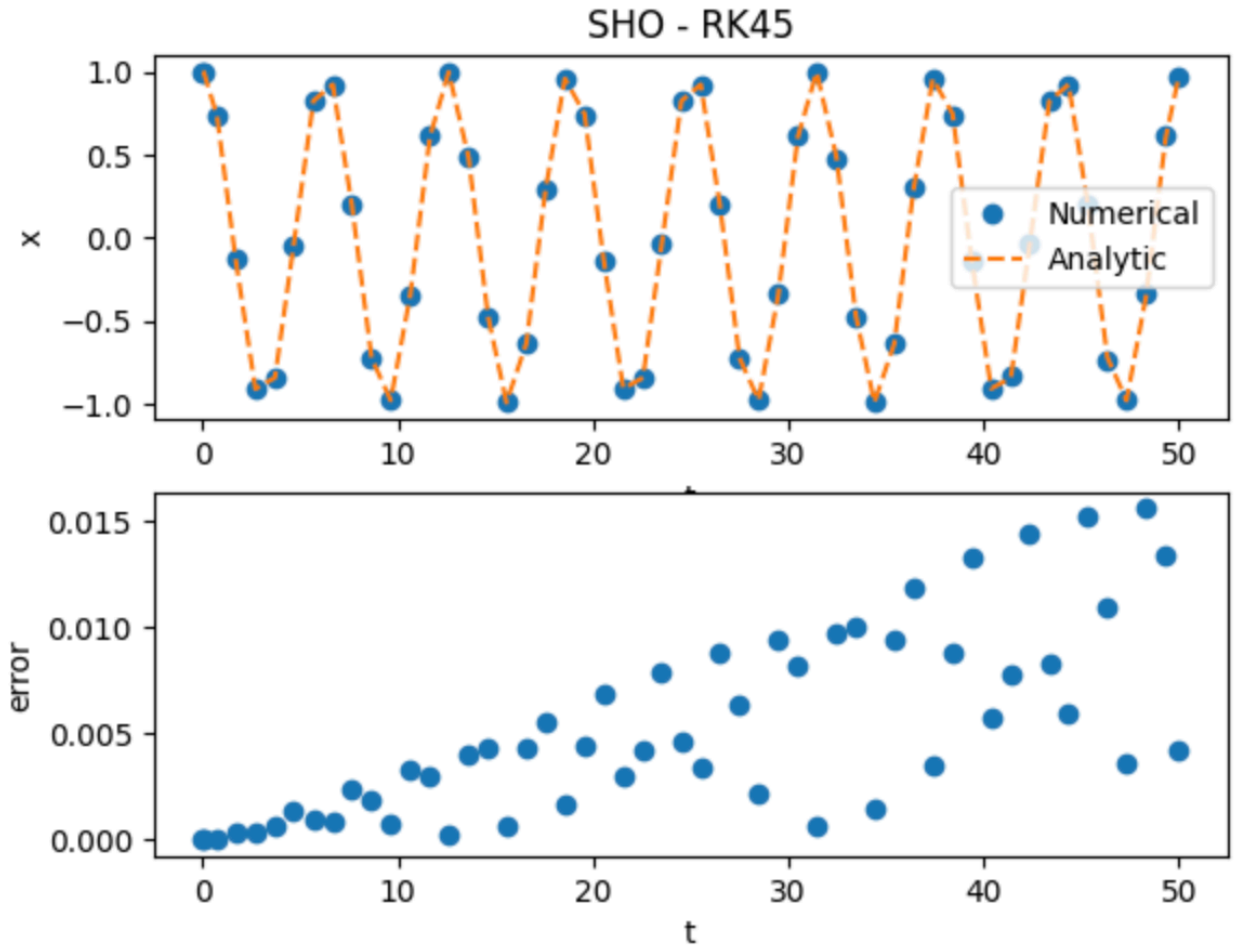
Error is set by the absolute and relative tolerances.

Times are unevenly distributed

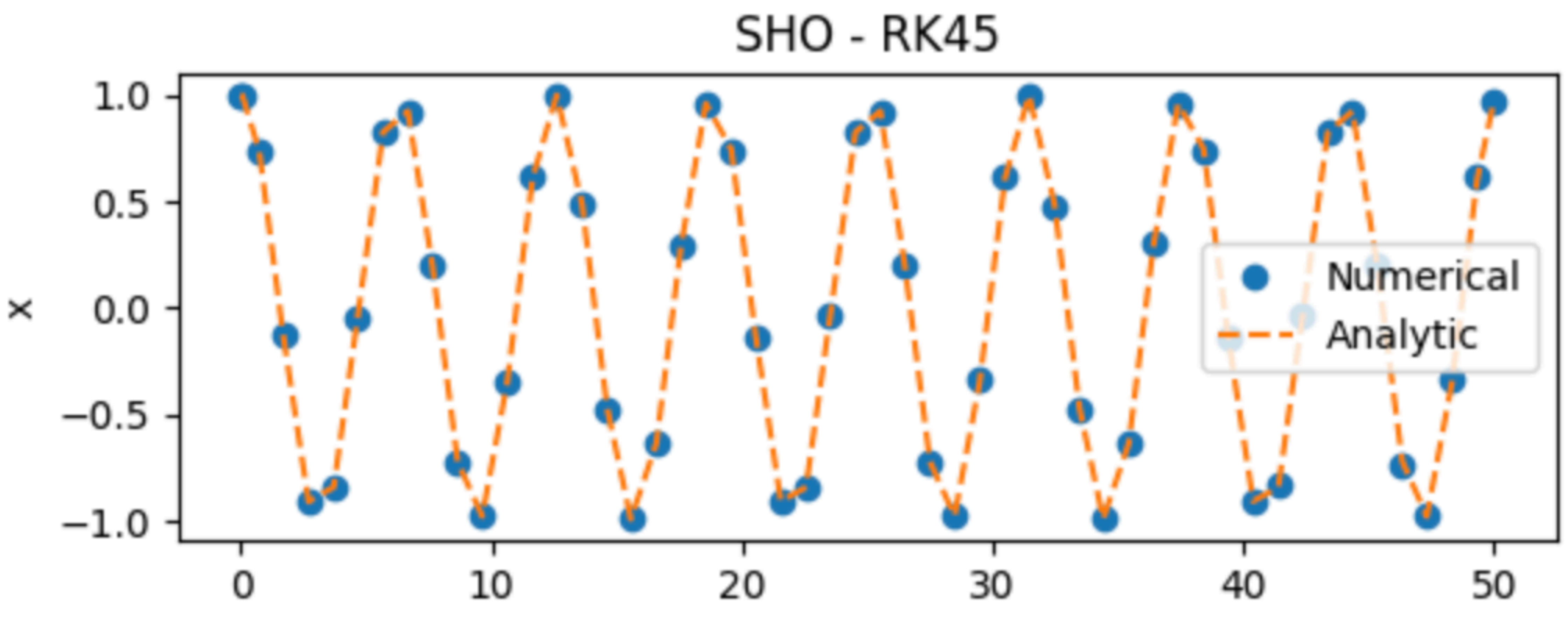


Error is set by the absolute and relative tolerances.

Times are unevenly distributed



Did the RK45 do a good job ?



Two factors:

Accuracy - how well the numerical solution tracks the true solution

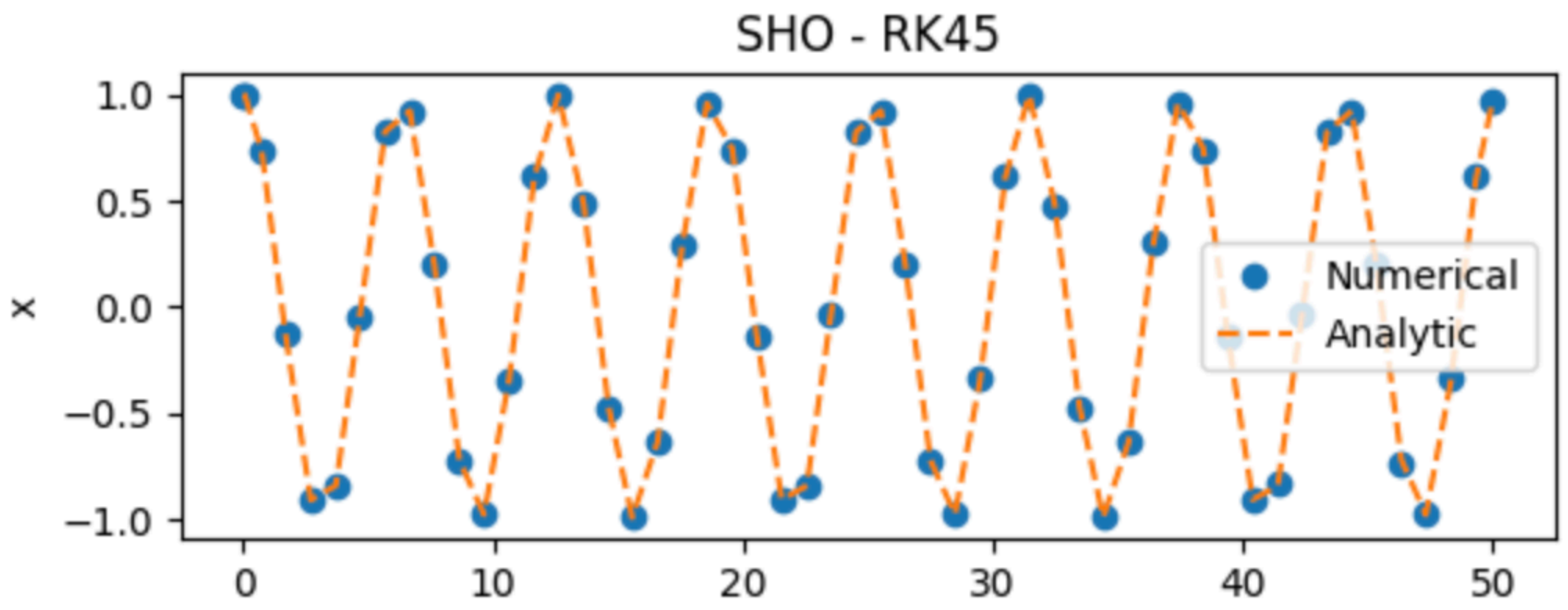
Sampling - how often we sample the solution effects the appearance of the graph.

- The error between the numerical solution and the analytic solution is around 1.5% of the oscillation's amplitude. This error could be reduced by imposing **stricter tolerances**.
- The plotted solution looks “jagged” because of poor sampling. We will show ways to use the numerical results to "fill out" the full curve using **interpolation**.

Custom Evaluation Times

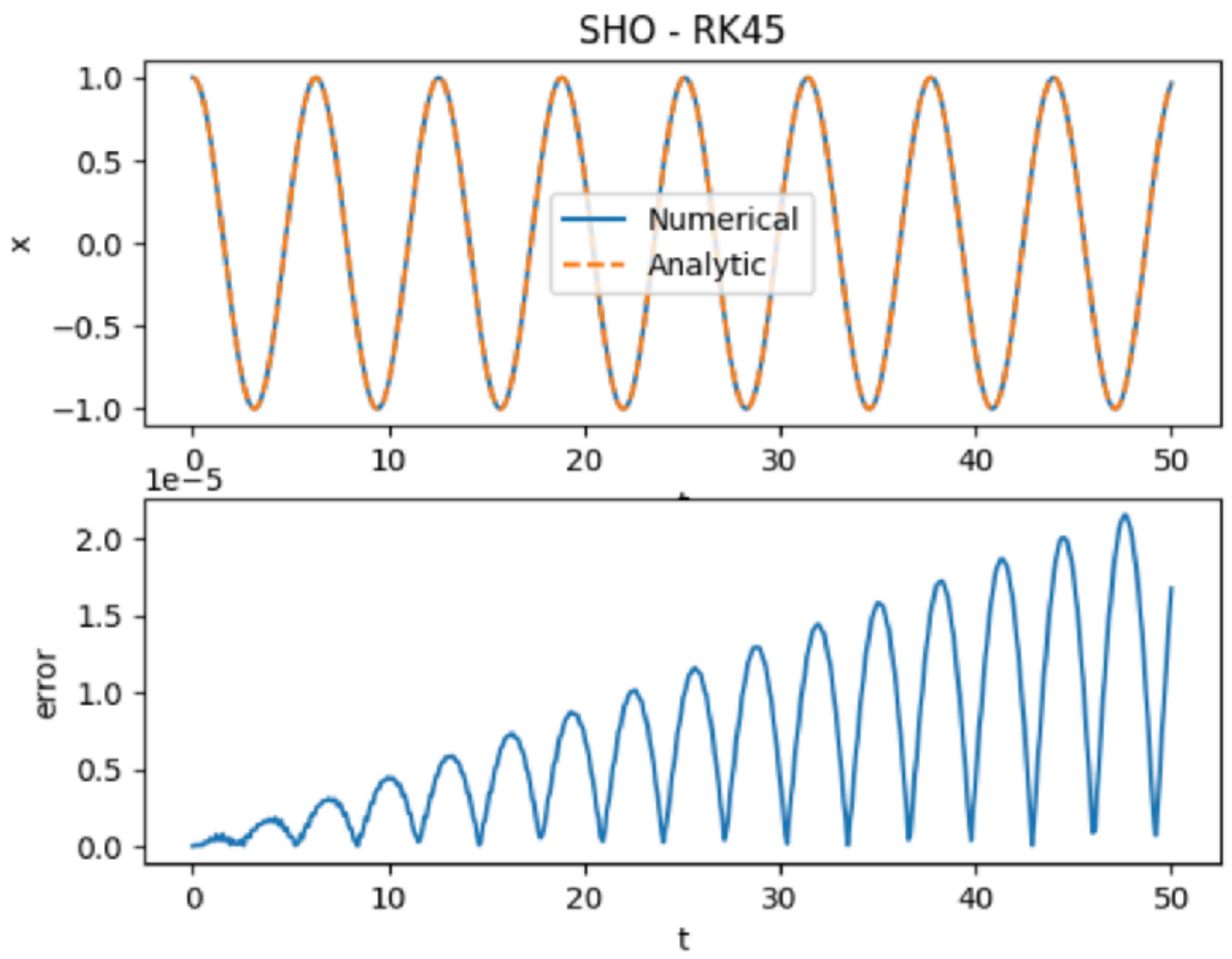
Adaptive time step methods produce solutions at nonuniform times. If you want to generate a solution on uniformly spaced times, you have two options:

- **option 1:** Pass an optional array of times to the `solve_ivp()` function. This option modifies the time-stepping algorithm to include the specified times
- **option 2:** Turn on the `dense_output` option. This option does not affect the numerical integration, but rather allows the user to use interpolation to evaluate the solution at any time they want.



Custom Evaluation Times

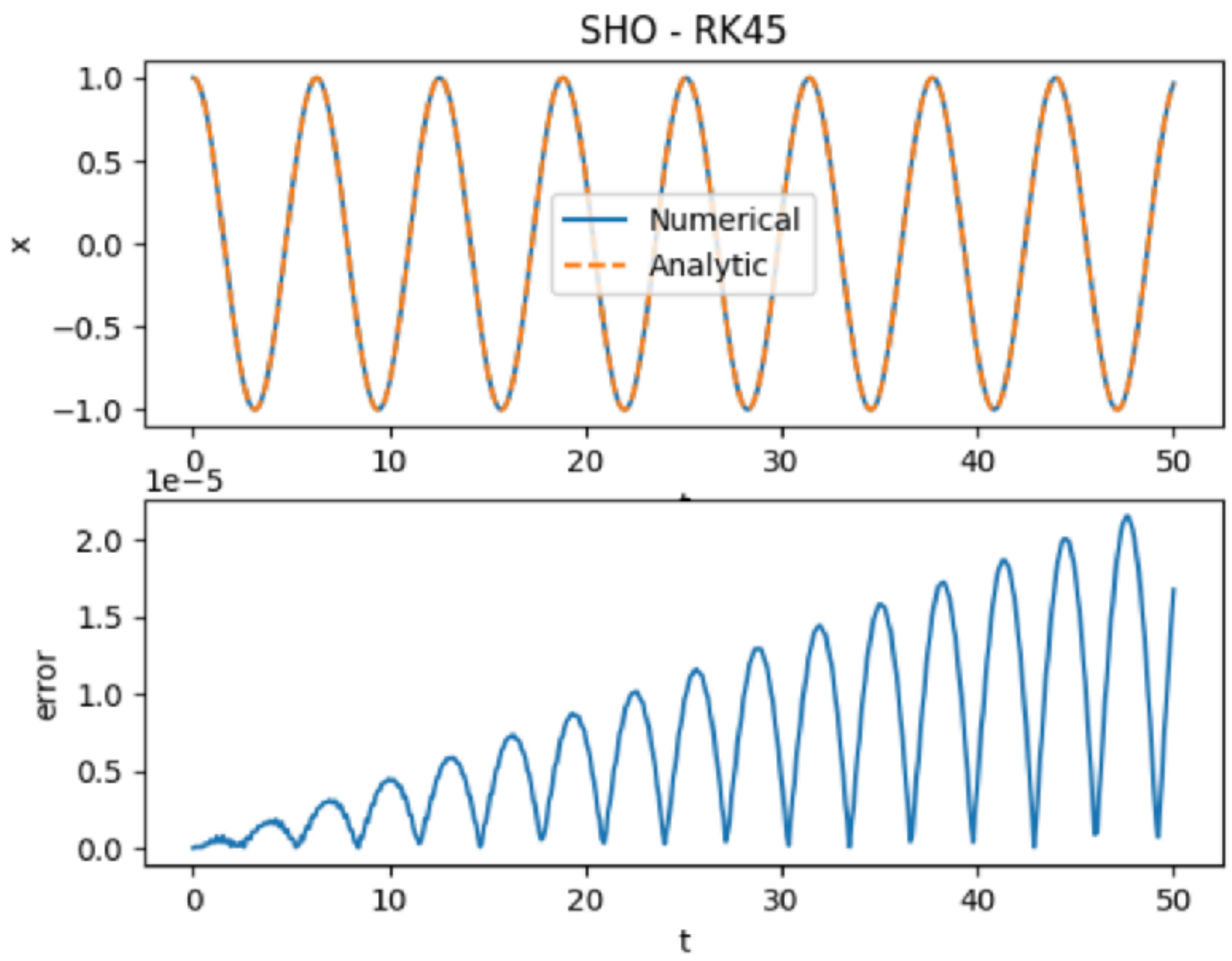
- **option 1:** Pass an optional array of times to the `solve_ivp()` function. This option modifies the time-stepping algorithm to include the specified times




```
times = np.linspace(0,tmax,500)
sol = solve_ivp(deriv_sho, (0,tmax), y0, method='RK45', t_eval=times,
    args=params, atol=1e-6,rtol=1e-6)
t = sol.t                   # extract times
x = sol.y[0,:]              # extract positions
v = sol.y[1,:]              # extract velocities
```

Custom Evaluation Times

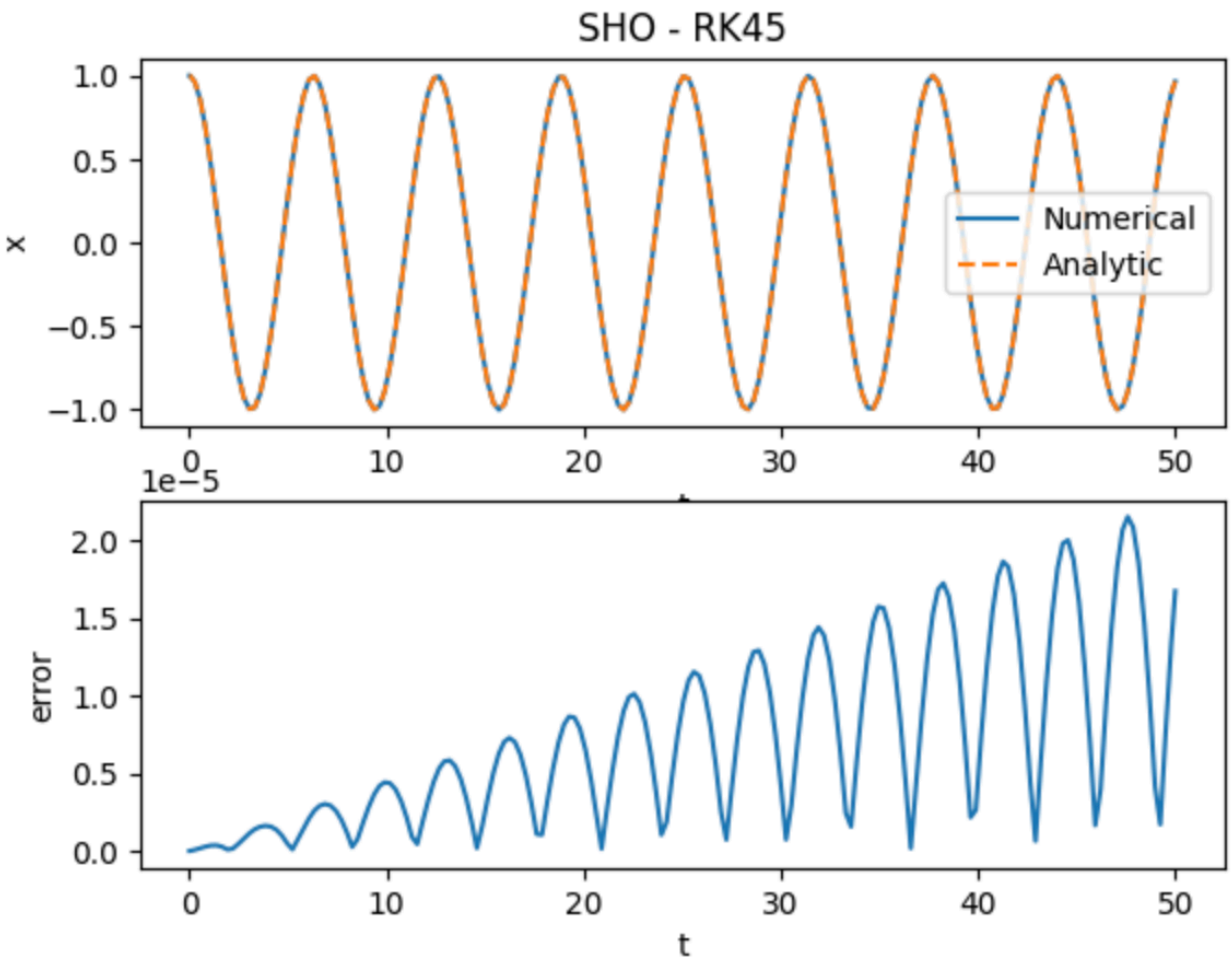
- **option 2:** Turn on the `dense_output` option. This option does not affect the numerical integration, but rather allows the user to use interpolation to evaluate the solution at any time they want.




```
sol = solve_ivp(deriv_sho, (0,tmax), y0, method='RK45', dense_output=True,
    args=params, atol=1e-6,rtol=1e-6)

# Extract solution on a regular grid of time values
t = np.linspace(0, tmax, 500)       # define time array
y = sol.sol(t)                      # create a 2D solution array
x = y[0,:]                          # extract position from the solution arr
```

Error is set by the absolute and relative tolerances.

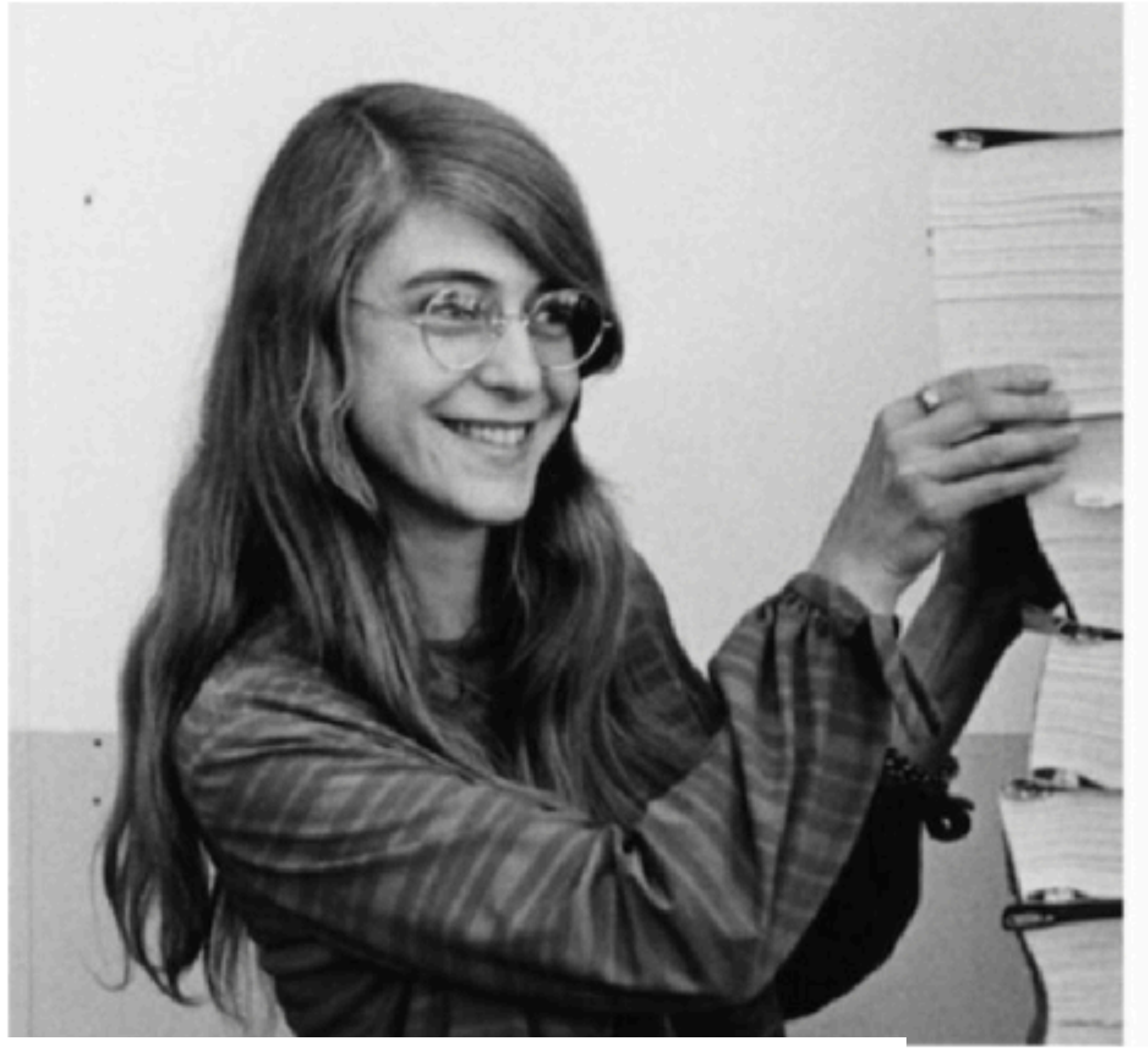
Times are unevenly distributed



Application: Chaos Theory

Edward Lorenz
(1917-2008)

- MIT Meteorologist
- Founder of Chaos Theory



Margaret Hamilton
(1936-)

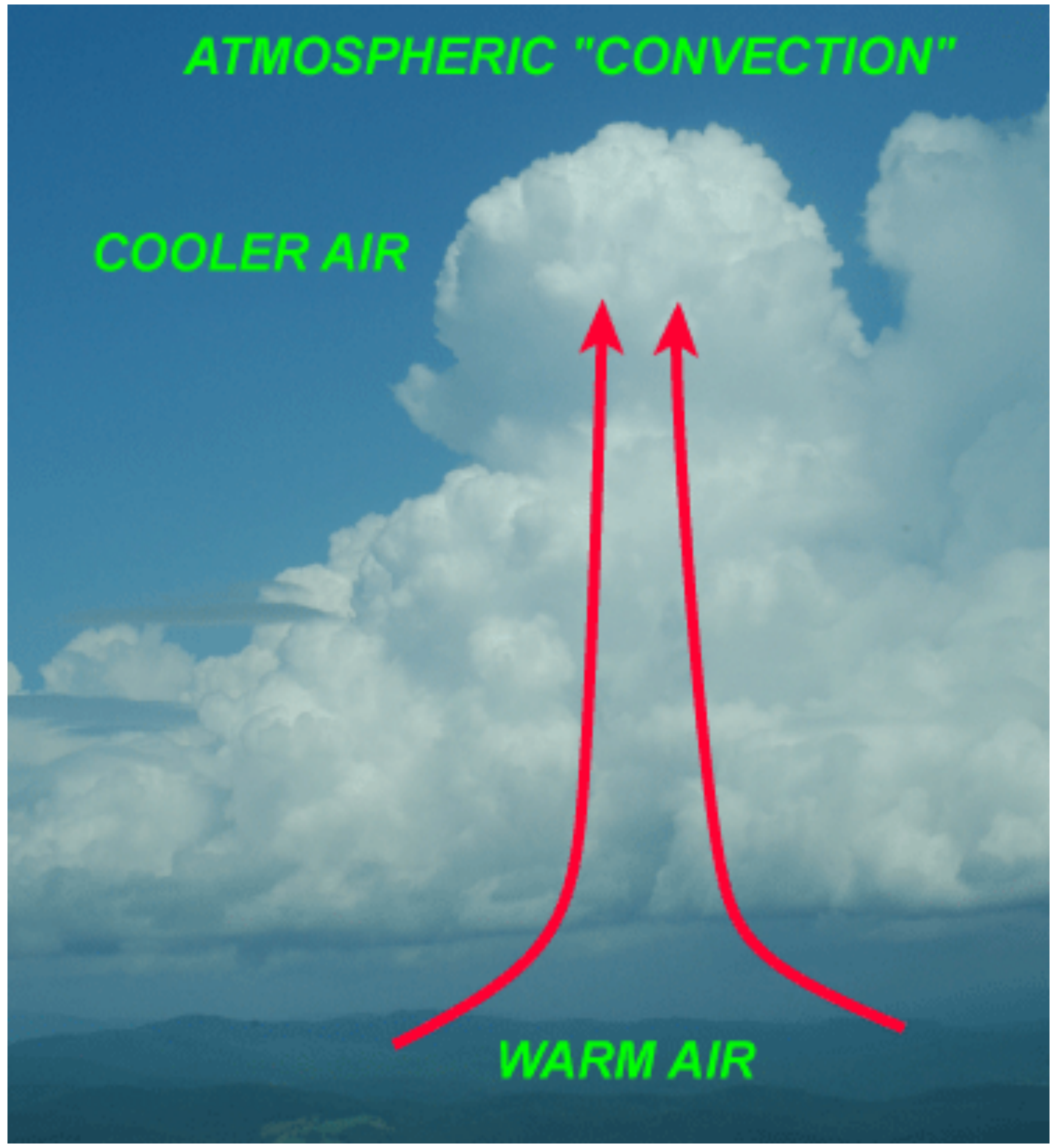
- Director MIT Instrument laboratory
- Developed software for NASA's Apollo Guidance Computer
- Coined term "software engineer"

Ellen Fetter

- Programmer
- Worked on Numerical Chaos Theory with Lorenz

Lorenz System

Lorenz was looking for a simplified system of equations to model convection in the atmosphere, when he discovered this set of equations:



$$\frac{dx}{dt} = \sigma(y - x)$$

x = flow rate of convection

$$\frac{dy}{dt} = x(r - z) - y$$

y = Temp difference

$$\frac{dz}{dt} = xy - bz$$

z = nonlinearity of Temp profile

$\sigma =$ Prandtl number
$r =$ Rayleigh number
$b =$ scale of the flow

Lorenz invented **Chaos Theory** to describe the dynamics of this system

Dynamical Systems

Consider the dynamics of systems that have both:

- dissipation (energy removed from the system)
- external driving (energy added to system)

These systems often have "**attractors**" which are values toward which the system evolves over time. These attractors have different forms:

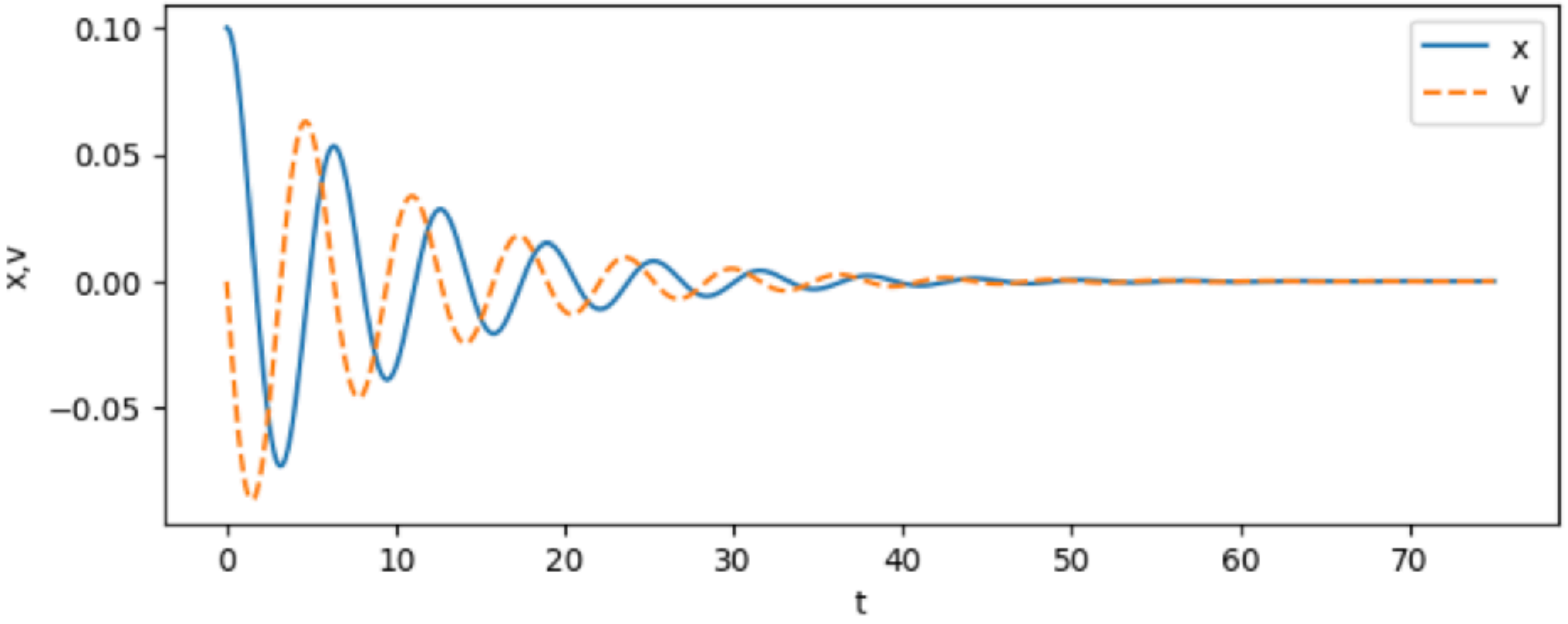
- fixed (equilibrium) points - 0D
- limit cycles - 1D
- strange attractors (fractal dimension)

Dynamical Systems

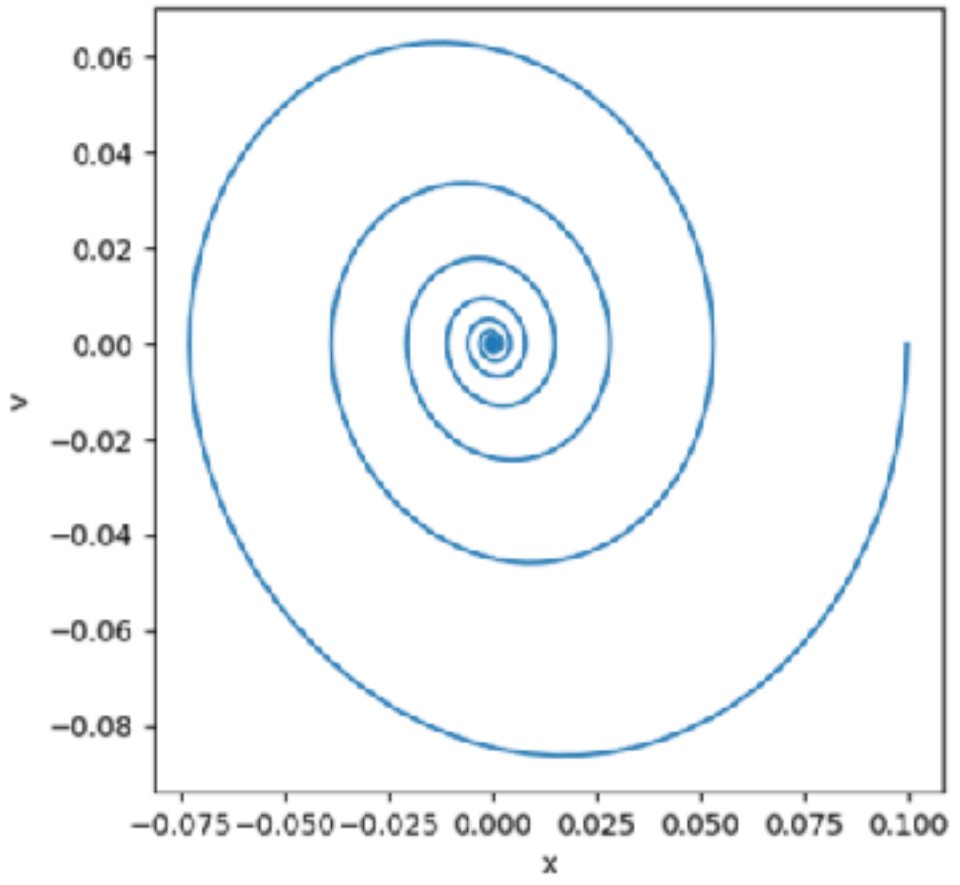
Example: Van der Pol oscillator: $$\frac{d^2x}{dt^2} - \mu(1 - x^2)\frac{dx}{dt} + x = 0$$

Fixed Point

$\mu = 0.8$

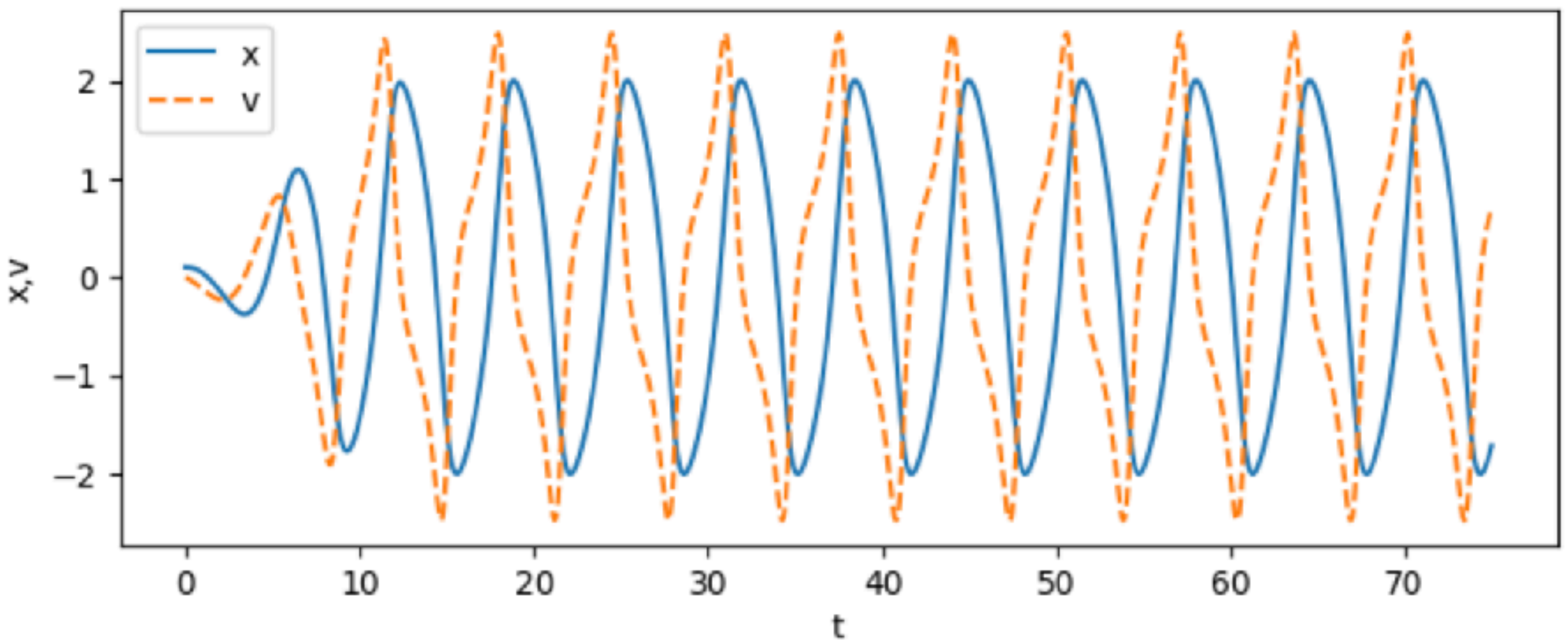


approaches single point

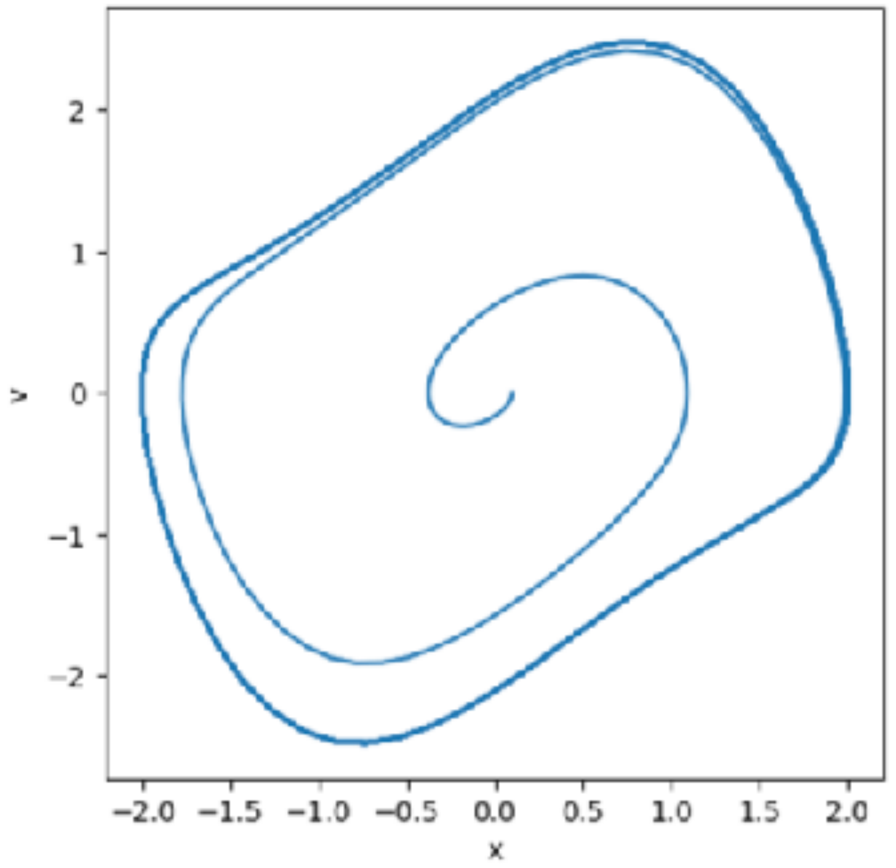


Limit Cycle

$\mu = -0.2$



closed orbit in phase space



Lorenz System

Perform numerical integration to solve for $x(t), y(t), z(t)$:

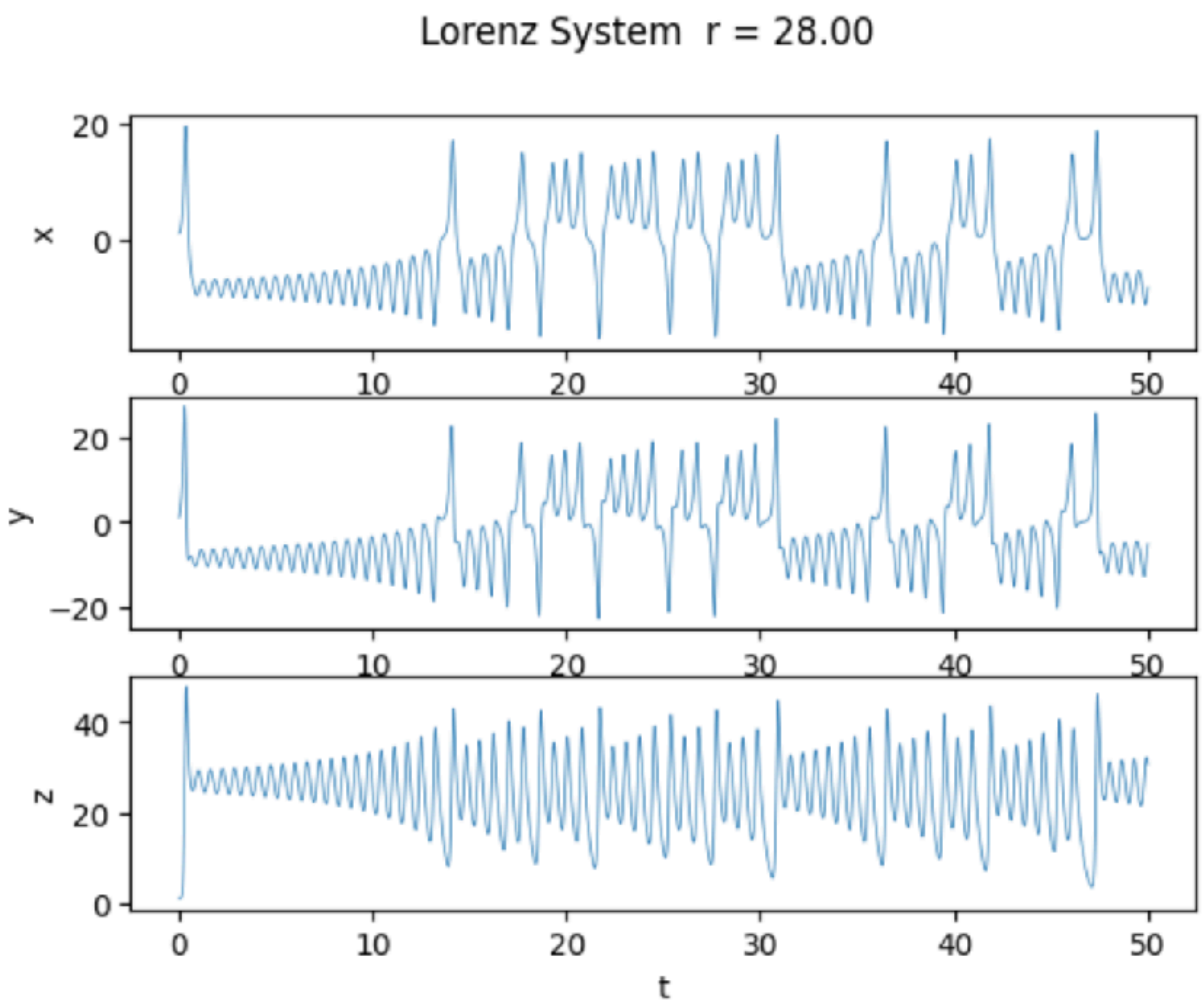
- Create a derivative function
- Use the RK45 method
- Interpolate onto a uniform time array
- Standard parameter values:
  - $\sigma = 10$
  - $b = 8/3$
  - $r = 28$
- Plots:
  - time series: $x(t), y(t), z(t)$
  - phase space: (x, y, z)

$$\frac{dx}{dt} = \sigma(y - x)$$

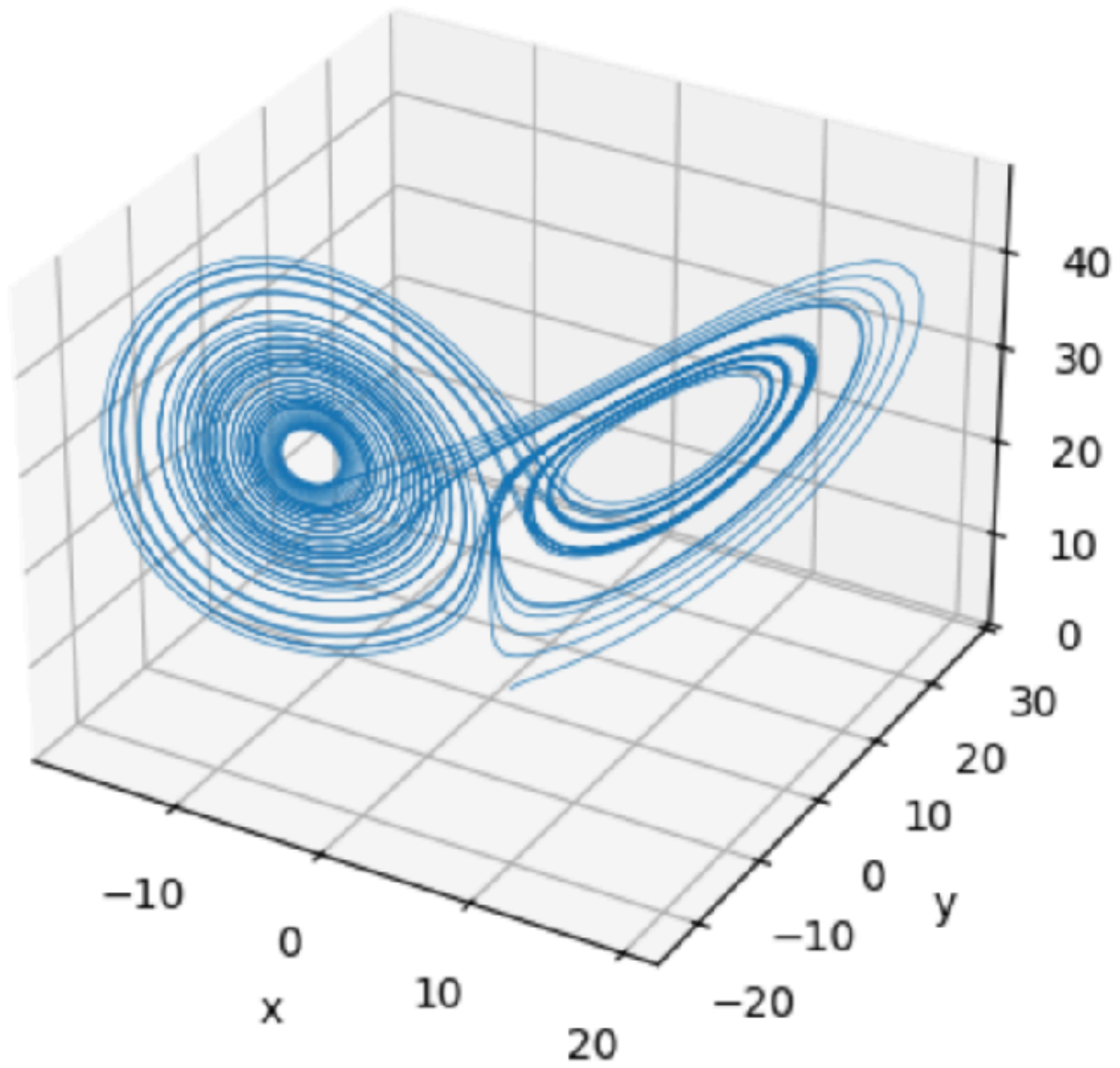
$$\frac{dy}{dt} = x(r - z) - y$$

$$\frac{dz}{dt} = xy - bz$$

Lorenz System



Dynamics do not repeat (aperiodic)

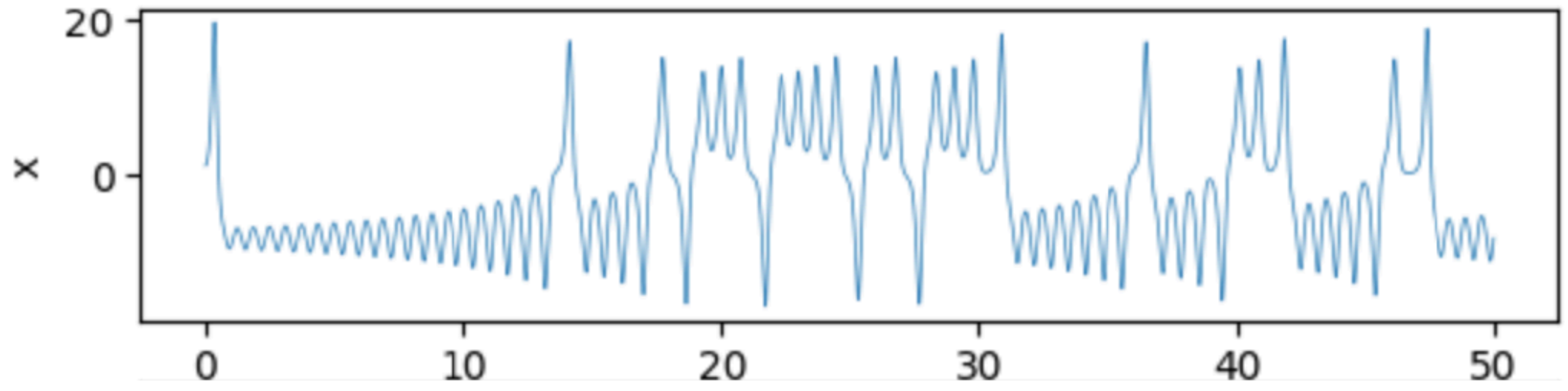


Strange attractor is neither a fixed point nor a limit cycle. It has fractal dimension

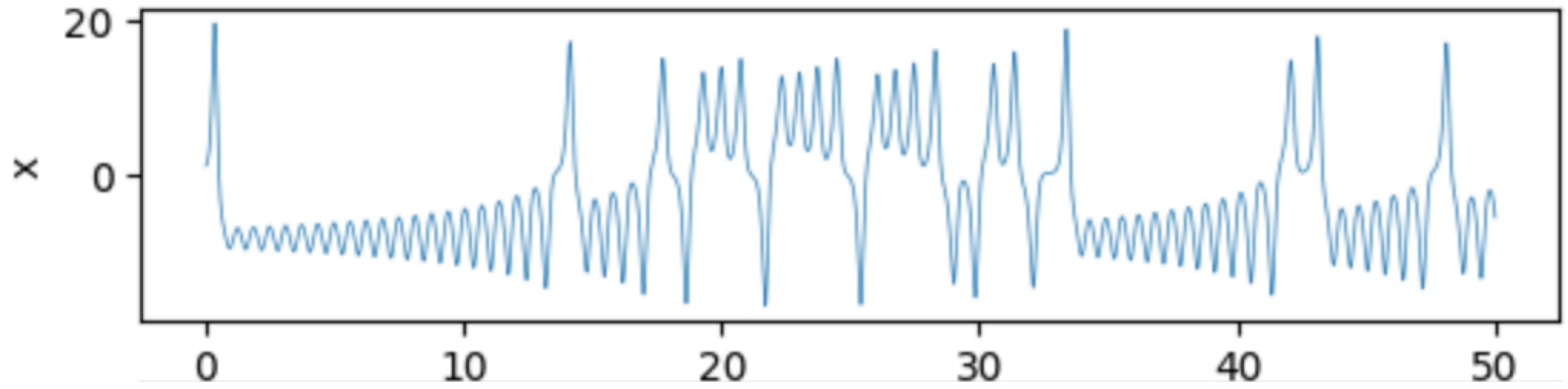
Sensitivity to Initial Conditions

Small changes to the initial conditions can lead to large changes later

$$(x_0, y_0, z_0) = (1,1,1)$$



$$(x_0, y_0, z_0) = (1,1,1.000001)$$



$$\Delta r \propto e^{\lambda t}$$