```
In [32]:  import numpy as np
          import scipy.optimize
          import matplotlib.pyplot as plt
          plt.rcParams['figure.figsize'] = [8, 3]
```

Euler and Runge-Kutta methods for solving ODEs

The Euler method is a first-order numerical method for solving ODEs. It uses a forward difference approximation to estimate the derivative of the solution at each time step and updates the solution accordingly. The basic formula for Euler's method is,

$$y_{n+1} = y_n + h \cdot f(t_n, y_n) \tag{1}$$

where, y_n is the approximate solution at time t_n, h is the time step and $f(t_n, y_n)$ is the derivative of the solution at time t_n.

Euler method has a first-order global truncation error ($O(h)$), which means that the error in the approximation is proportional to the time step. It is simple to implement but may not be accurate for stiff ODEs with rapidly changing or oscillatory solutions.

Runge-Kutta methods are a family of numerical methods for solving ODEs. These methods work by approximating the solution of an ODE at discrete time steps, also known as time discretization. The most commonly used Runge-Kutta method is the fourth-order Runge-Kutta (RK4).

The RK4 method uses a weighted average of four derivative evaluations to estimate the derivative of the solution at each time step. The basic formula for RK4 is,

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

where y_n is the approximate solution at time t_n, h is the time step, and k_1, k_2, k_3, k_4 are the weighted derivatives of the solution at time t_n. The weighted derivatives are calculated as follows,

$$k_1 = f(t_n, y_n), \tag{3}$$
$$k_2 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} \cdot k_1 \right), \tag{4}$$
$$k_3 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} \cdot k_2 \right), \tag{5}$$
$$k_4 = f\left(t_n + h, y_n + h \cdot k_3 \right). \tag{6}$$

RK4 has a fourth-order global truncation error ($O(h^4)$), so it is more accurate than Euler's method. It is also important to note that RK4 is more robust to stiff ODEs than Euler's method.

The following code compares the accuracy of Euler method and RK4 for solving the following first-order ODE,

$$\frac{dx}{dt} = -x \tag{7}$$

where $x(0) = 1$.

In [33]:
```python
from ODEsolve import compare_euler_rk4_error

# Define the parameters for the forst-order ODE
x0 = 1 # Initial condition
t0 = 0 # Initial time
delta_t_values = np.logspace(-2, 0, 100) # Step sizes
f = lambda t, x: x # x' = x ODE
# Compare the error in Euler's method and Runge-Kutta's method for different step s
compare_euler_rk4_error(f, x0, t0, delta_t_values)
```

Error in Euler's method and RK4 method
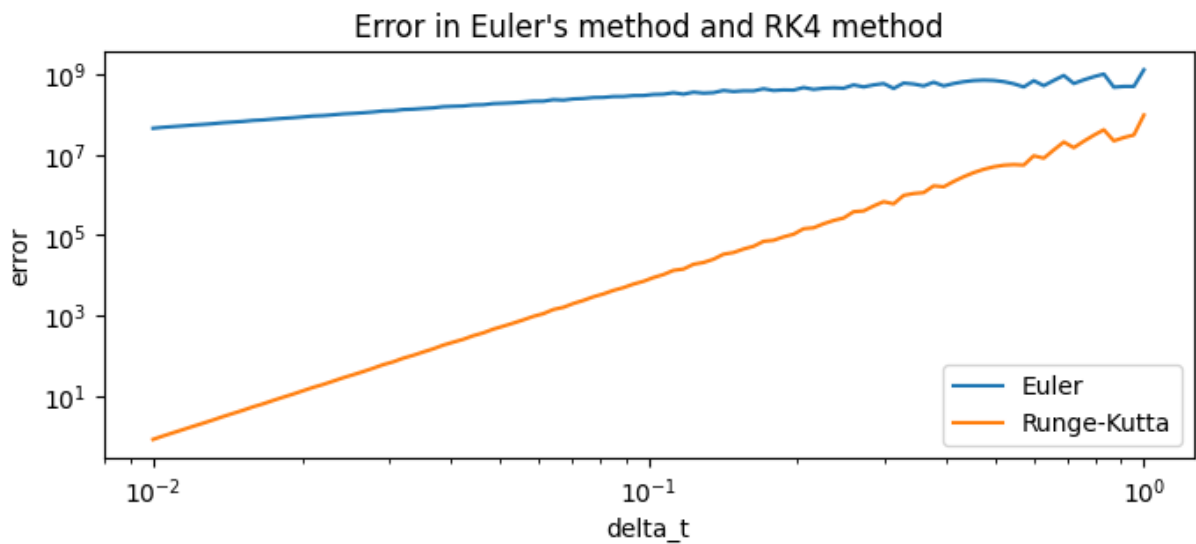
The plot above shows that RK4 is more accurate than Euler's method for different time steps. Because it is a fourth-order method, it uses higher-order polynomial approximations to estimate the slope of the solution curve at each time step. On the other hand, RK4 is more computationally expensive than Euler's method because it requires four derivative evaluations at each time step.

The following code implements Euler's method and RK4 for solving the following second-order ODE,

$$\frac{d^2u}{dt^2} = -u \tag{8}$$

where $u = (x, y)$, $x(0) = 1$ and $y(0) = 1$.

In [34]:
```python
from ODEsolve import solve_euler_rk4, euler_step, rk4_step
# Define the 2nd order ODE x'' = -x
def f(t,u):
    x = u[0]
    y = u[1]
    dydt = -x
    dxdt = y
```

```
    return np.array([dxdt, dydt])

# Define the true solution for the 2nd order ODE x'' = -x
def true_sol(t):
    x = np.sin(t) + np.cos(t)
    y = np.cos(t) - np.sin(t)
    return np.array([x, y])

x0 = np.array([1, 1]) # Initial condition, x = 1, y = 1
deltat_max = 0.1 # Step size

# Solve for x and y using Euler's method and Runge-Kutta's method
solve_euler_rk4(f, x0, t0, deltat_max, euler_step, true_sol=true_sol)
solve_euler_rk4(f, x0, t0, deltat_max, rk4_step, true_sol=true_sol)
```
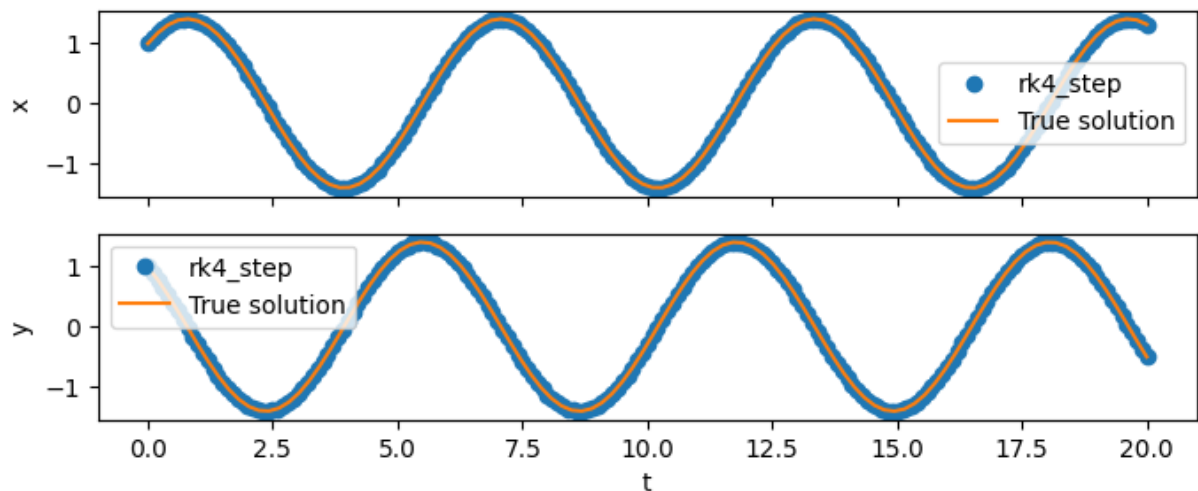

Solution for x and y using euler_step
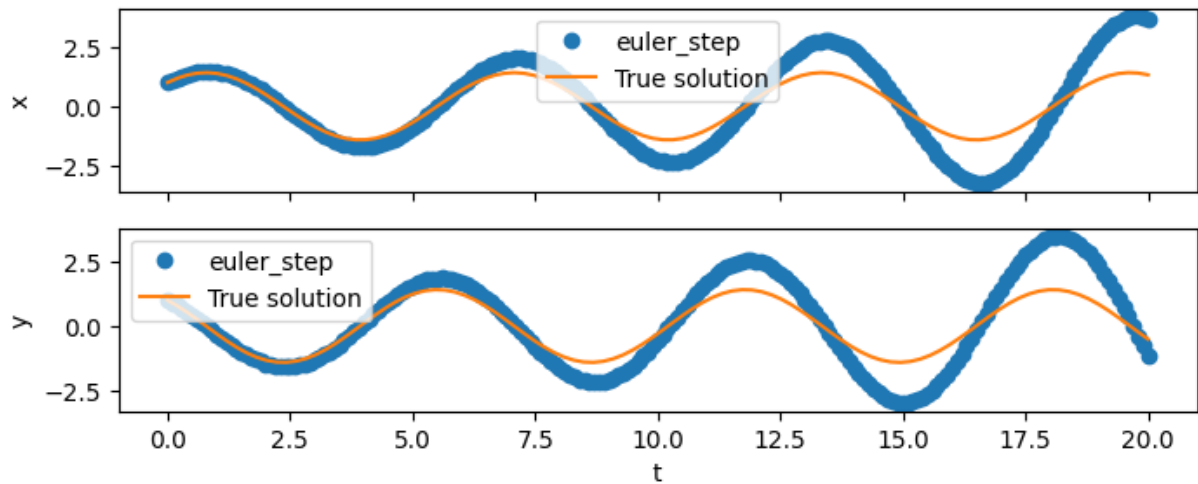
Solution for x and y using rk4_step

The plot above shows that, for a step size of $h = 0.1$, RK4 is very accurate while Euler's method is not. This is because of the aforementioned higher-order polynomial approximations used by RK4.

Numerical Shooting

It is a two-step process. In the first step, the initial guess for the solution is obtained by solving the initial value problem (IVP) with the given boundary conditions. In the second step, the initial guess is used to solve the BVP by applying a correction to the initial guess. The correction is obtained by solving the IVP with the given boundary conditions and the initial guess. The correction is then added to the initial guess to obtain the solution of the BVP. The process is repeated until the solution converges.

The numerical shooting method is a numerical method for solving boundary value problems (BVPs) for ODEs. Numerical shooting is particularly useful for solving BVPs where the boundary conditions are unknown or difficult to determine. The basic idea behind numerical shooting is to transform a BVP into an IVP by "shooting" the solution from one boundary towards the other, and then adjust the initial conditions until the desired boundary conditions are satisfied. This is done by solving the IVP using a numerical ODE solver, such as the Runge-Kutta method, and adjusting the initial conditions until the solution satisfies the boundary conditions.

The shooting method involves the following steps:

1. Guess the initial conditions: A guess is made for the unknown initial conditions that would satisfy the boundary conditions.
2. Solve the IVP: The IVP is solved using the guessed initial conditions. The solution trajectory is obtained from the IVP.
3. Adjust the initial conditions: The initial conditions are adjusted based on the solution trajectory.
4. Repeat until convergence: Steps 2 and 3 are repeated until the initial conditions converge to the values that satisfy the boundary conditions.

The code provided above implements the shooting method to solve a specific BVP for the predator-prey model. The model is described by the following system of ODEs:

$$\frac{dx}{dt} = x \cdot (1 - x) - \frac{a \cdot x \cdot y}{d + x}, \tag{9}$$

$$\frac{dy}{dt} = b \cdot y \cdot \left(1 - \frac{y}{x}\right), \tag{10}$$

where x(t) represents the population of prey, y(t) represents the population of predators, and a, b, and d are parameters.

The code uses the shooting method to find the limit cycle of the predator-prey model by solving the system of ODEs with guessed initial conditions and adjusting them until the limit cycle is obtained. The limit cycle is then plotted using matplotlib to visualize the behavior of the predator-prey populations over time.

The code can be tested with different initial conditions and parameter values to study the effect of different parameters on the behavior of the predator-prey model, such as the stability of the limit cycle or the existence of multiple limit cycles. The results can be

compared with analytical solutions or other numerical methods to validate the accuracy and efficiency of the shooting method for solving the predator-prey model.

In [35]:
```python
from Shooting_method import plot_limit_cycle
# Define the predator-prey ODE where b > 0.26
def ode(t, u):
    x = u[0] # Number of prey
    y = u[1] # Number of predators
    a = 1; d = 0.1; b = 0.5
    dxdt = x*(1-x) - (a*x*y)/(d + x)
    dydt = b*y*(1 - (y/x))
    return np.array([dxdt, dydt])

# Define parameters for the predator-prey ODE
deltat_max = 0.01
u0 = [0.3,0.3]

plot_limit_cycle(ode, u0, deltat_max, rk4_step)
```

0.4

Limit Cycle for the Shooting Method with b = 0.5
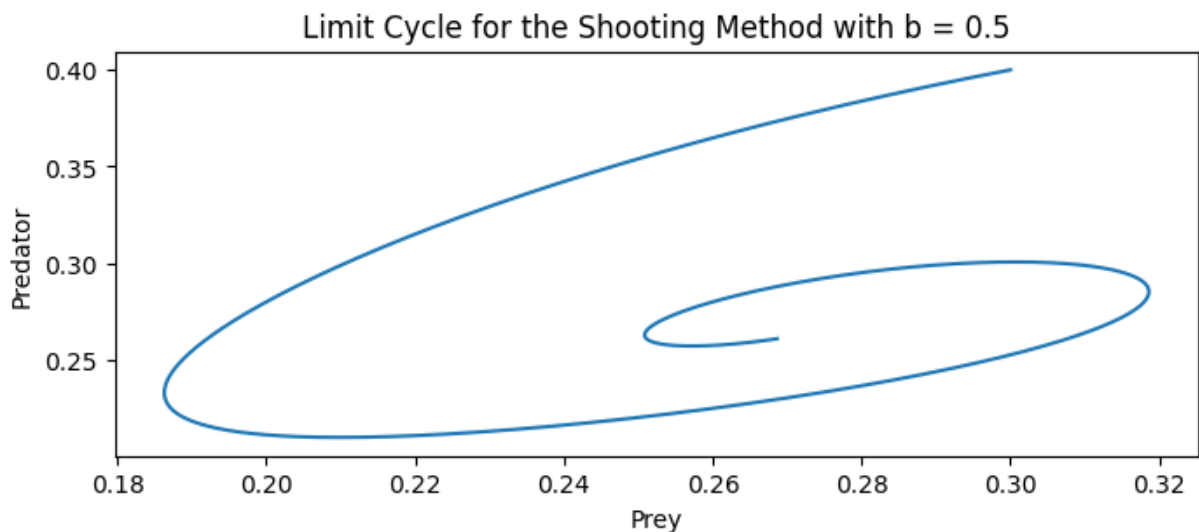
sTABLE NODE FOR GREATER THAN 0.26

Method of Lines

The method of lines (MOL) is a numerical method to approximate solutions to partial differential equations (PDEs) on a uniform discrete grid. It works by discretizing the spatial domain using finite differences. This then allows us to write the PDE as an initial value problem (IVP) or a boundary value problem (BVP) in ODEs, which can be solved using a numerical method. The method of lines is a general method that can be used to solve any PDE. However, it is most commonly used to solve PDEs with constant coefficients. The error for MOL is proportional to the step size in the spatial domain, so increasing the number of points in the grid makes the approximation more accurate.

BVPs are differential equations within a domain with defined constraints, called boundary conditions. Hence the solution to the PDE must satisfy these boundary conditions. There are
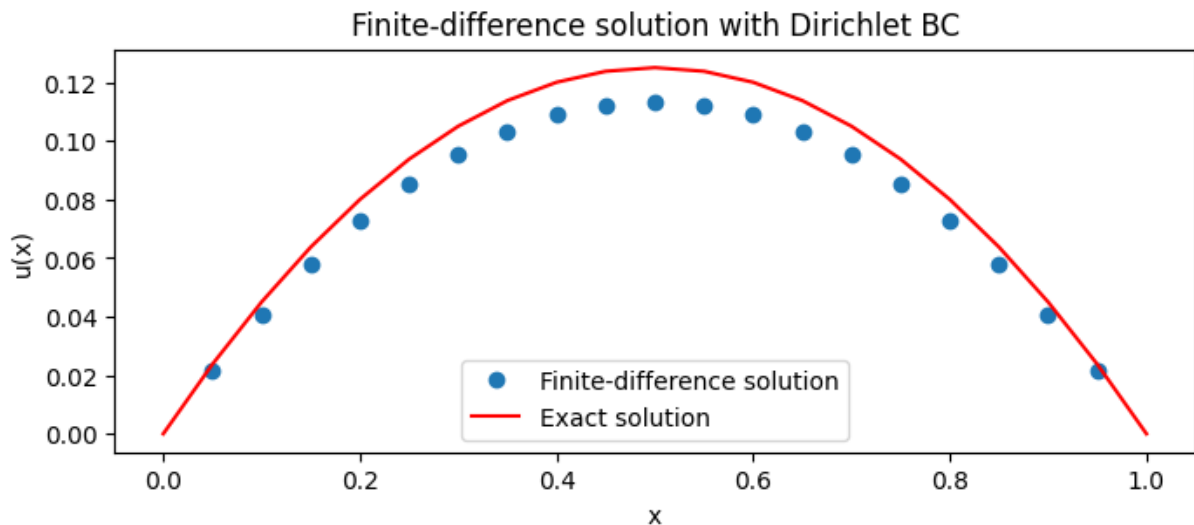
three types of boundary conditions that can be applied to a BVP; Dirichlet, Neumann, and Robin. A Dirichlet boundary condition is where the boundary value is the solution. A Neumann boundary condition is where the derivative of the solution is the boundary value. A Robin boundary condition is a combination of Dirichlet and Neumann boundary conditions where the boundary value is the derivative of the solution plus a function.
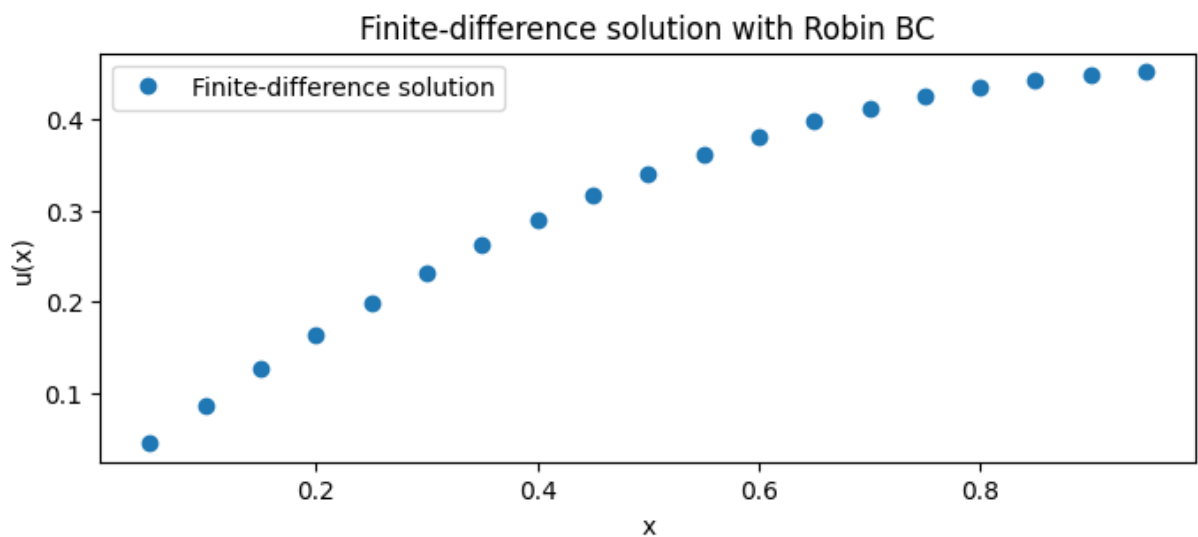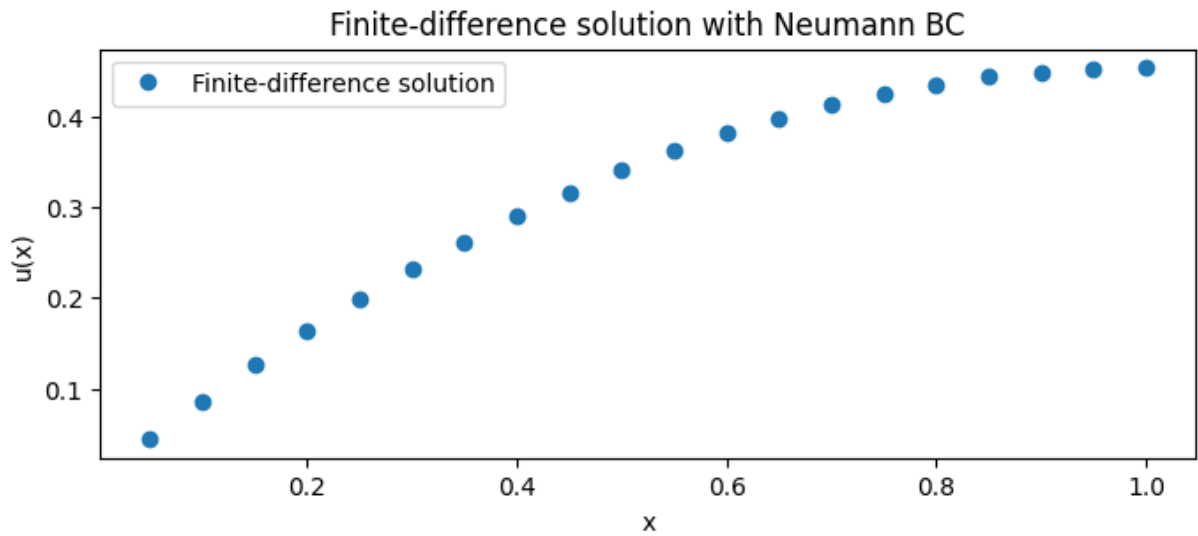
To illustrate the use of finite differences with these boundary conditions, we can consider the BVP,

$$d^2u/dx^2 + q(x) = 0, u(a) = 0, u(b) = 0. \tag{11}$$

We will set $q(x) = 1$, $a = 0$, and $b = 1$ for simplicity. By applying Dirichlet boundary conditions and obtaining different solutions we can see how the approximations differ from the exact solution $u_{\text{exact}} = \frac{1}{2} \cdot x \cdot (1 - x)$. For Neumann and Robin boundary conditions, the approximations are not similar to the exact solution but are similar to one another. This is because the Neumann and Robin boundary conditions impose constraints on the derivative of the solution at the boundary, rather than providing the values of the solution itself.
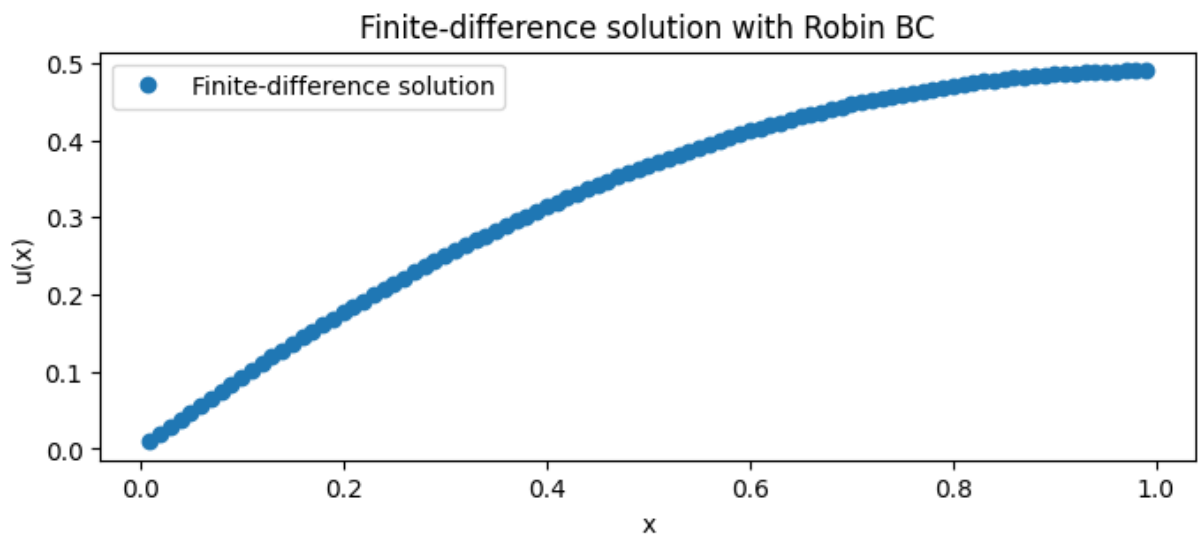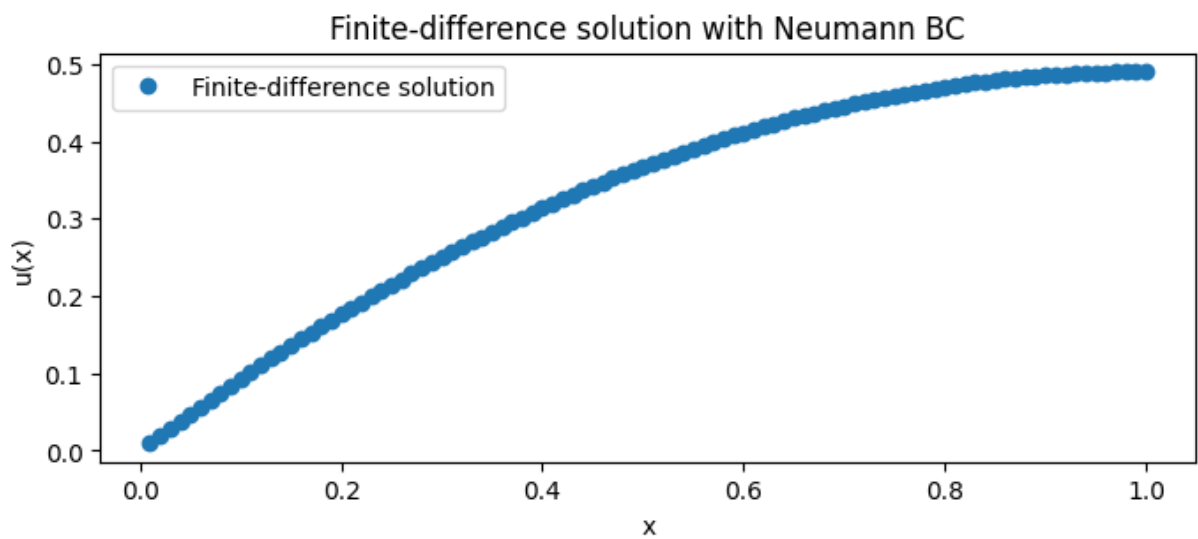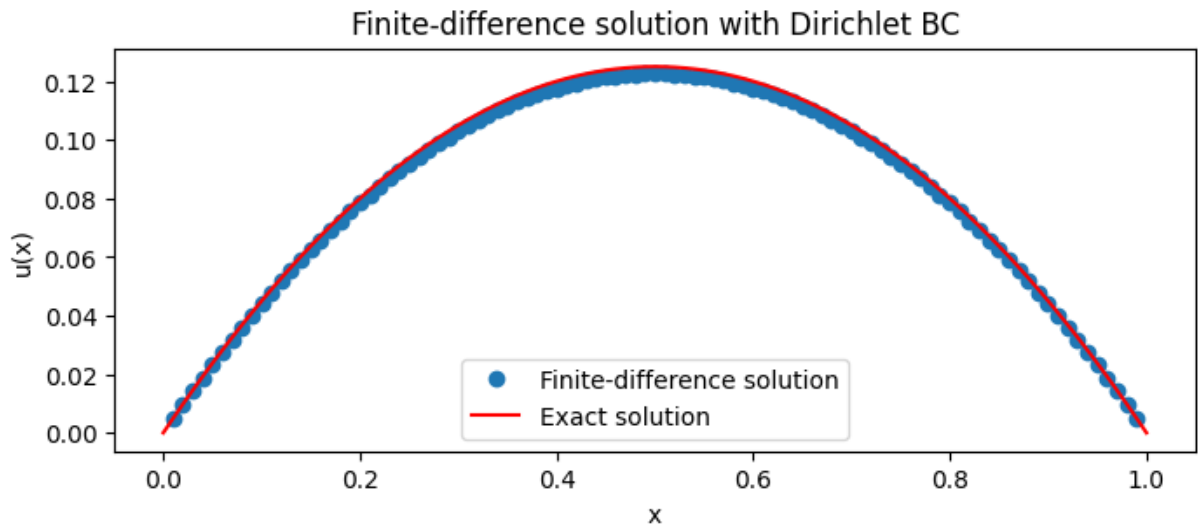
In [36]:
```python
from BVPsolve import solve_bvp
N = 21; a = 0; b = 1
alpha = 0; beta = 0; gamma = 1; delta = 0
q = lambda x: np.ones(np.size(x))
solve_bvp(N, a, b, alpha, beta, q, "dirichlet")
solve_bvp(N, a, b, alpha, beta, q, "neumann", delta=delta)
solve_bvp(N, a, b, alpha, beta, q, "robin", gamma=gamma)
```


Finite-difference solution with Dirichlet BC

Finite-difference solution with Neumann BC

Finite-difference solution with Robin BC

For each type of boundary condition, the numerical approximation has a significant truncation error when $N = 21$. By setting $N = 101$ the step size is much smaller therefore, the error is much smaller.

In [37]:
```python
N = 101
solve_bvp(N, a, b, alpha, beta, q, "dirichlet")
solve_bvp(N, a, b, alpha, beta, q, "neumann", delta=delta)
solve_bvp(N, a, b, alpha, beta, q, "robin", gamma=gamma)
```

Finite-difference solution with Dirichlet BC

Finite-difference solution with Neumann BC

Finite-difference solution with Robin BC

Explicit Euler, Implicit Euler, and Crank-Nicolson Methods for PDEs using Method of Lines

PDEs can be converted to a system of ODEs using the method of lines and can therefore be solved using numerical methods for ODEs.

The explicit Euler method, as mentioned earlier, is a first-order method that approximates the derivative in time using a forward difference. But because a system of ODEs is being solved, the central difference approximation is also used to evaluate the second derivative in space. The basic formula for the explicit Euler method therefore becomes,

$$u_{x,t+1} = u_{x,t} + \frac{D\Delta t}{(\Delta x)^2} * (u_{x+1,t} - 2u_{x,t} + u_{x-1,t}), \tag{12}$$

where $u_{x,t}$ is the approximate solution at time t and space x, Δt is the time step, Δx is the space step, and D is the diffusion coefficient.

This is the simplest method for solving a system of ODEs but it has some drawbacks. The method is first-order accurate in time and second-order accurate in space making it the least accurate of the three methods. It is also unstable if,

$$\frac{D\Delta t}{(\Delta x)^2} > \frac{1}{2}, \tag{13}$$

which can be restrictive for small values of Δx. The explicit Euler method is also unsuitable for solving stiff ODEs because it is a first-order method.

The implicit Euler method is a first-order method that approximates the derivative in time using a backward difference instead. This means that the method is unconditionally stable. However, it is computationally more expensive than the explicit Euler method when computing the next time step. The basic formula for the implicit Euler method is,

$$u_{x,t+1} = u_{x,t} + \frac{D\Delta t}{(\Delta x)^2} * (u_{x+1,t+1} - 2u_{x,t+1} + u_{x-1,t+1}). \tag{14}$$

Because, like explicit Eluer, the implicit Euler method is a first-order method, it is not suitable for solving stiff ODEs either. Also like explicit Euler, the implicit Euler method is first-order accurate in time and second-order accurate in space so it is still not very accurate.

The Crank-Nicolson method is a second-order method that approximates the derivative in time using a central difference. This means that the method is unconditionally stable and is suitable for solving stiff ODEs. It is also more accurate than the other two methods too because it is second-order accurate in time and space. The basic formula for the Crank-Nicolson method is,

$$u_{x,t+1} = u_{x,t} + \frac{D\Delta t}{2(\Delta x)^2} * (u_{x+1,t+1} - 2u_{x,t+1} + u_{x-1,t+1}) + \frac{D\Delta t}{2(\Delta x)^2} * (u_{x+1,t} - 2u_{x,t}$$

where all variables are the same as in the implicit Euler method. Crank-Nicolson uses a central difference approximation for both the first and second derivatives.

To illustrate the use of each method, we can consider the second-order PDE for heat diffusion,

$$\frac{du}{dt} = \kappa \frac{d^2 u}{dx^2}, 0 < x < L, 0 < t < T, \tag{16}$$

with zero-temperature boundary conditions, $u = 0$ at $x = 0, L, t > 0$, and the initial temperature function, $u_I = \sin(\frac{\pi x}{L})$.

The code uses a uniform grid with N_x grid points in the x-direction and N_t grid points in the t-direction. The step size in the x-direction is $h = L/(N_x - 1)$ and the step size in the t-direction is $k = T/(N_t - 1)$. The code uses a forward difference approximation for the first derivative and a central difference approximation for the second derivative. The code also uses a forward difference approximation for the time derivative.