In [4]:
```python
import numpy as np
import scipy.optimize
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [7, 3.5]
```

Software Summary

Euler and Runge-Kutta methods for solving ODEs

The Euler method is a first-order numerical method for solving ordinary differential equations (ODEs). It uses a forward difference approximation to estimate the derivative of the solution at each time step and updates the solution accordingly.

Euler's 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 which are 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. 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.

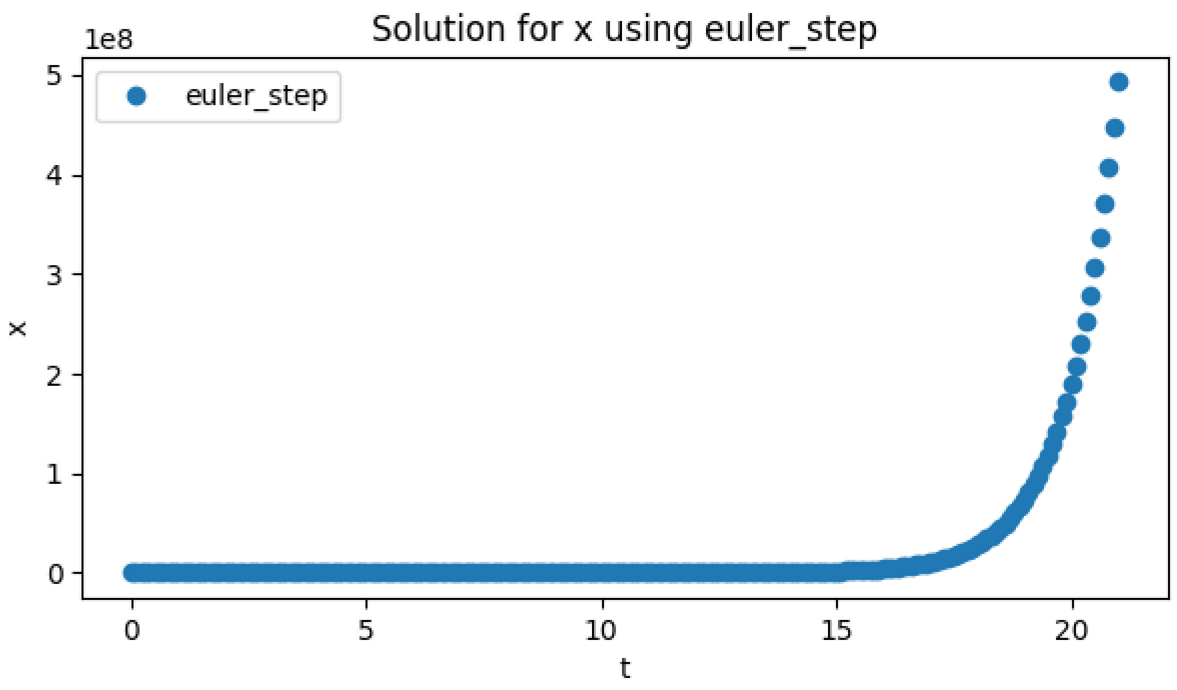
RK4 has a fourth-order global truncation error ($O(h^4)$), so it is more accurate than Euler's method. It is also important to note that RK4 is more robust to stiff ODEs than Euler's method.

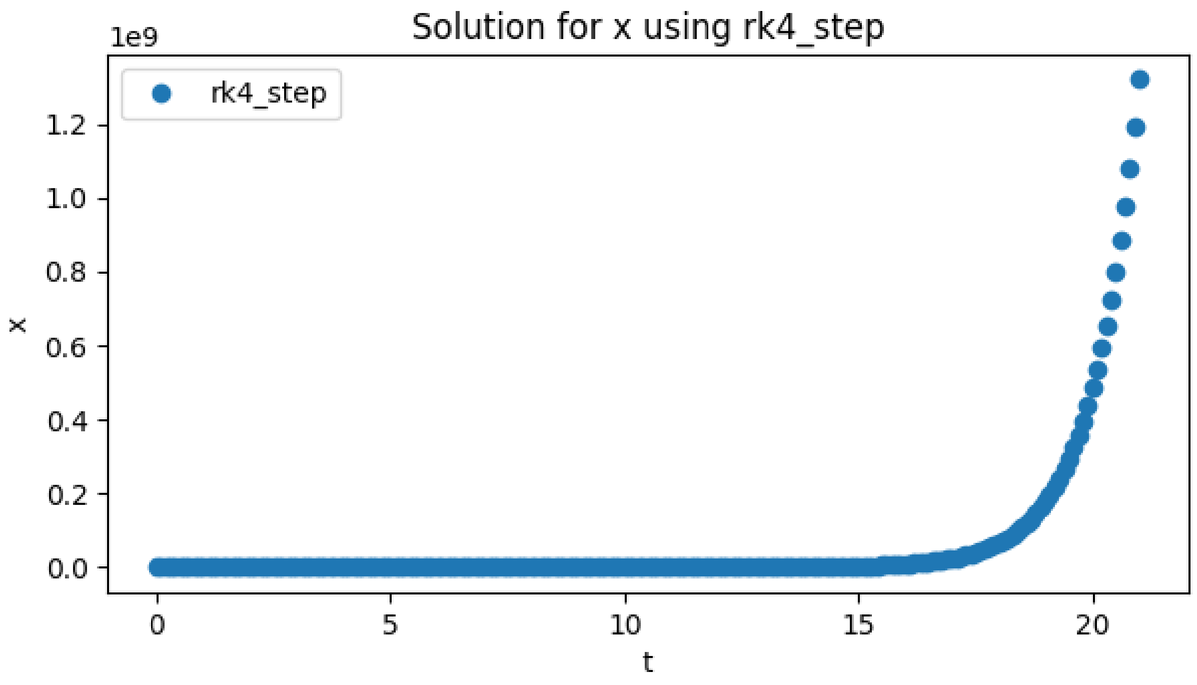
The following code plots the solution and compares the accuracy of Euler's method and RK4 for the following first-order ODE,

$$\frac{dx}{dt} = -x \tag{1}$$

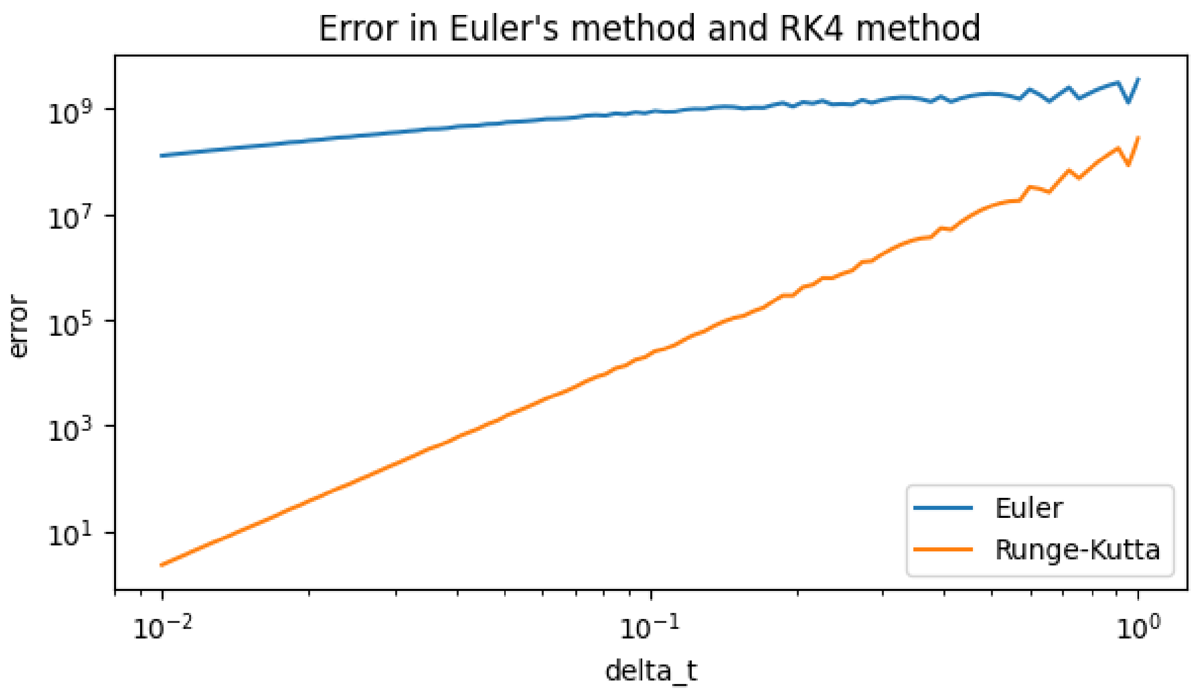
where $x(0) = 1$.

In [5]:
```python
from ODEsolve import plot_euler_rk4, euler_step, rk4_step
# Define the function
f = lambda t, x: x # x' = x ODE
deltat_max = 0.1 # Step size
x0 = np.array([1]) # Initial condition
t0 = 0 # Initial time

plot_euler_rk4(f, x0, t0, deltat_max, euler_step)
plot_euler_rk4(f, x0, t0, deltat_max, rk4_step)
```





In [6]:
```
from ODEsolve import compare_euler_rk4_error

# Define the parameters for the first-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)
```



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.

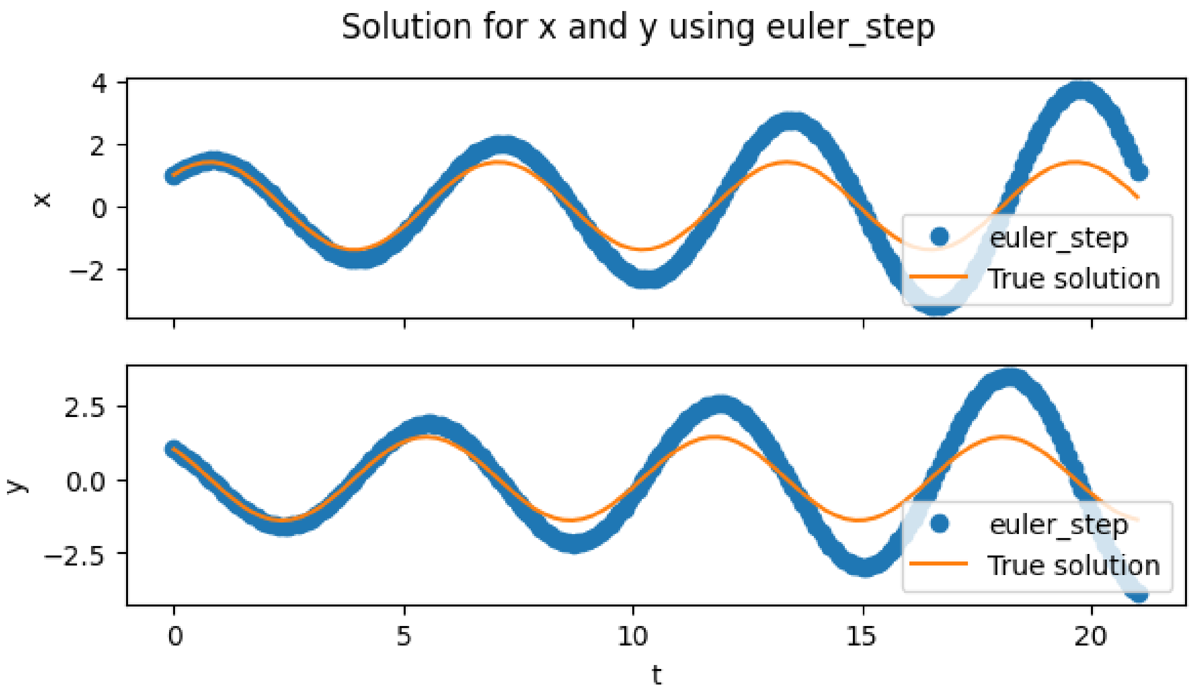
The same function also works for solving second-order ODE. To present this the following ODE will be used,

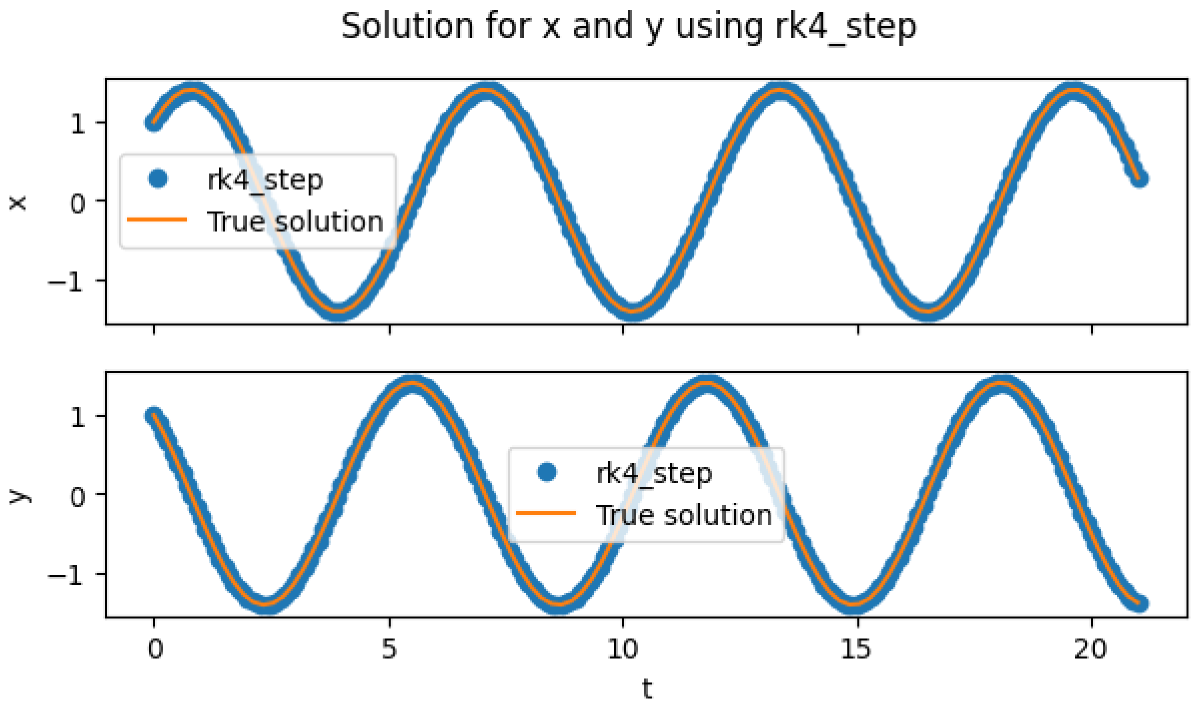
$$\frac{d^2u}{dt^2} = -u, \tag{2}$$

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

In [7]:
```
# 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
```

```
plot_euler_rk4(f, x0, t0, deltat_max, euler_step, true_sol=true_sol)
plot_euler_rk4(f, x0, t0, deltat_max, rk4_step, true_sol=true_sol)
```





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 Method

The numerical shooting method is a numerical method for solving boundary value problems (BVPs) for ODEs. BVPs are differential equations within a domain with defined constraints, called boundary conditions. Numerical shooting is particularly useful for solving BVPs where the boundary conditions are unknown or difficult to determine.

In Shooting_method.py the shooting function solves a one-dimensional ODE, while the shooting2 function solves a two-dimensional ODE. Both functions use the phase_condition and phase_condition2 functions respectively, which define the phase condition for the ODE system.

The solve_to_shooting function is the main function for solving the ODE using the shooting method. If the input initial values are for a two-dimensional system, then solve_to_shooting calls the shooting2 function, otherwise, it calls the shooting function. After finding the initial value for the dependent variable using the shooting method, it calls the solve_to function from the ODEsolve.py file to solve the ODE using the given step function. The function then returns the time and dependent variable values.

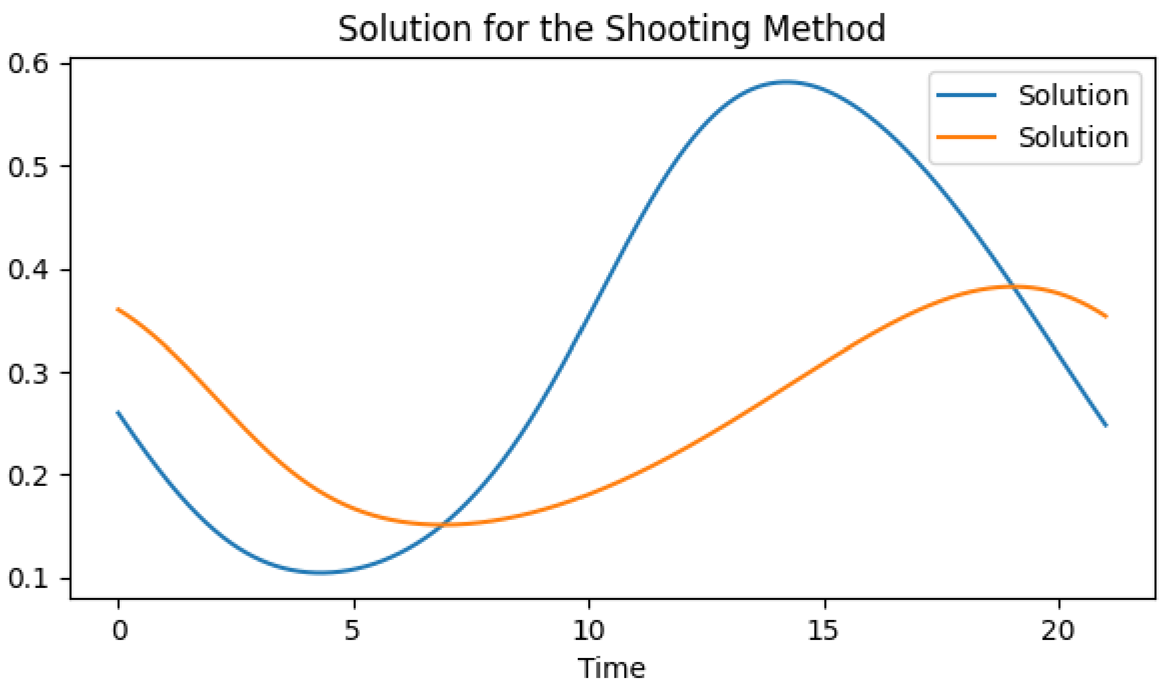
The following code implements the shooting method to solve any BVP in the form $\frac{dx}{dt} = eqn_1, \frac{dy}{dt} = eqn_2$. The simple predator-prey model is described by the following system of ODEs:

$$\frac{dx}{dt} = x \cdot (1 - x) - \frac{a \cdot x \cdot y}{d + x}, \tag{3}$$

$$\frac{dy}{dt} = b \cdot y \cdot \left(1 - \frac{y}{x}\right), \tag{4}$$

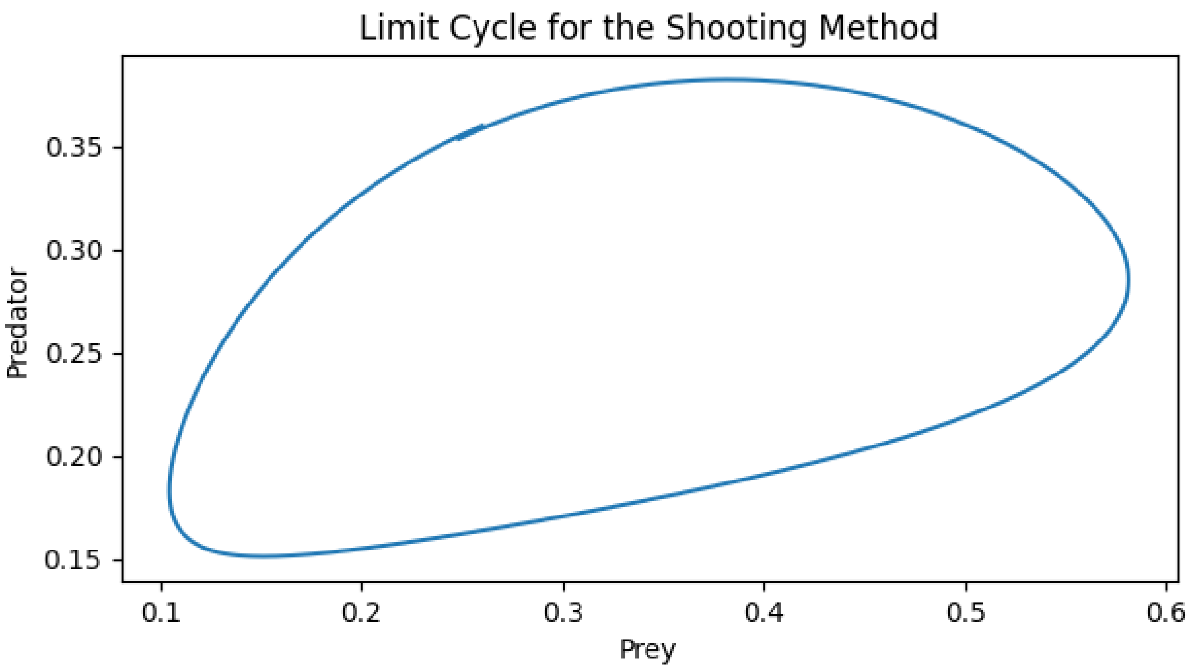
where x(t) represents the population of prey, y(t) represents the population of predators, and a, b, and d are parameters.

In [8]:
```python
from Shooting_method import plot_solution
# 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 = 0.2
    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 = [0.26, 0.26]

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



The code can be tested with different initial conditions and parameter values to study the behaviour of the predator-prey model. Such as the stability or the existence of multiple equilibria. The results can be compared with other numerical methods to validate the accuracy of the method. By plotting the phase portrait of the system when $b < 0.26$ we see there is a stable limit cycle because the prey and predator populations oscillate around the equilibrium point. When $b > 0.26$ the system has a stable equilibrium point because the populations converge to the equilibrium point.

In [9]:
```
from Shooting_method import plot_limit_cycle
# Define initial guess
u0 = [0.26, 0.26]
plot_limit_cycle(ode, u0, deltat_max, rk4_step)
```



Numerical Continuation Method

The numerical continuation method is a method for approximating the solutions to a system of parameterized nonlinear equations.

The system is in the form $F(\mathbf{u}, \lambda) = 0$, where $\mathbf{u}$ is the solution vector and λ is a parameter. The method also requires an initial solution $(\mathbf{u}_0, \lambda_0)$. The method works by solving the system of equations for different values of λ and then adjusting the parameter values until the solutions converge to the desired solutions.

Continuation methods are advantagious when studying bifurcation behaviors or stability as the parameter changes. Two common algorithms are natural parameter continuation and pseudo-arclength continuation.

Natural parameter continuation is the simplest continuation method. It works by updating the parameter value by a small step and then solving the system of equations again to obtain the solution at the updated parameter value.

Psuedo-arclength continuation differs from natural parameter continuation in that it uses the tangent vector of the solution curve as the direction of the parameter update. It then calculates the step size using the following formula,

$$\Delta s = \dot{u}_0^*(u - u_0) + \dot{\lambda}_0^*(\lambda - \lambda_0), \tag{5}$$

where $(\dot{u}_0^*, \dot{\lambda}_0^*)$ is the tangent vector at (u_0, λ_0).

Psuedo-arclength continuation is more complex and computationally expensive than natural parameter continuation, but it is more robust as it can handle situations where the tangent vector becomes small or vanishes, which can occur at bifurcation points.

Natural parameter continuation is more efficient than pseudo-arclength continuation because it does not require the computation of the Jacobian matrix. Though, it is less robust as it fails at turning points.

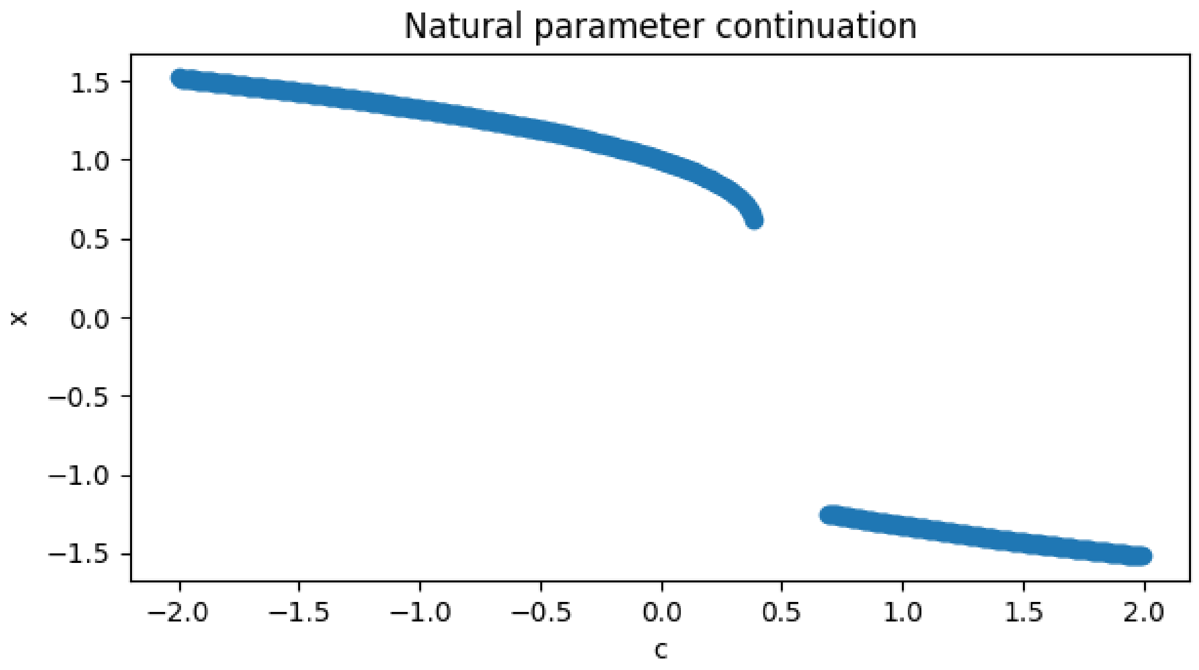
The following code implements natural parameter continuation and pseudo-arclength continuation to solve the following cubic equation,

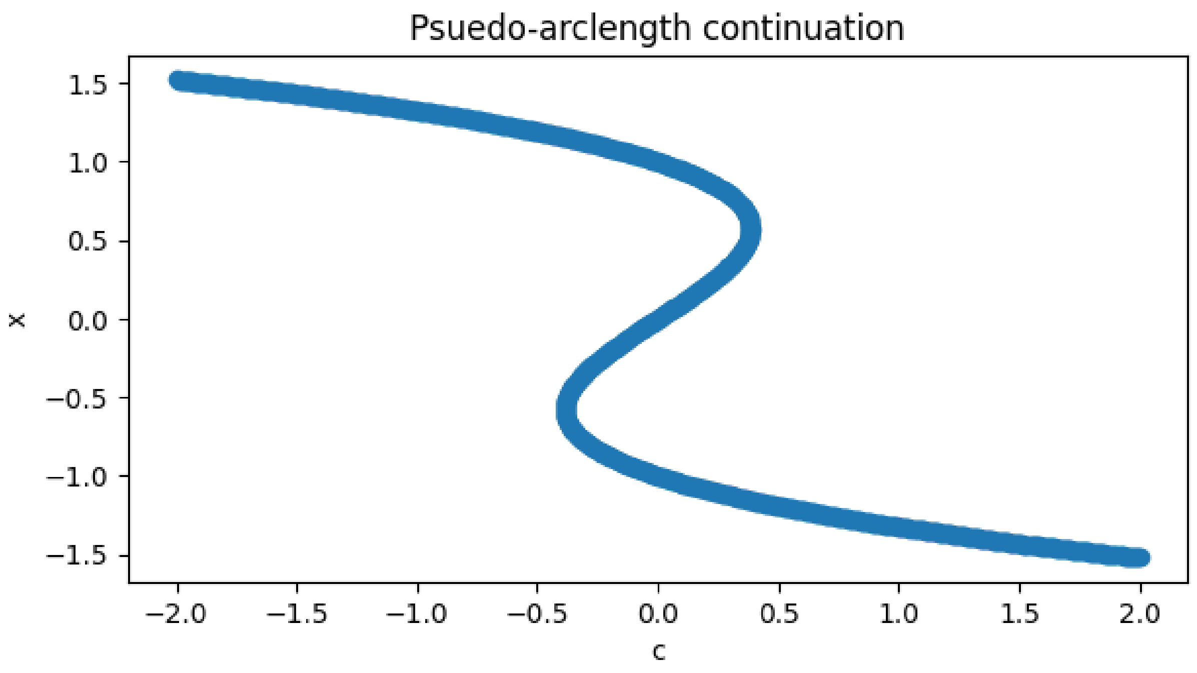
$$x^3 - x + c = 0, \tag{6}$$

where $-2 \leq c \leq 2$.

```
In [10]: from Numerical_continuation import plot_continuation
         # Define the initial guess and the parameter range
         x0 = 1
         c0 = -2
         c1 = 2

         # Define the function
         def f(x, c):
             return x**3 - x + c

         # Plot the continuation diagrams
         plot_continuation(f, x0, c0, c1, 'natural')
         plot_continuation(f, x0, c0, c1, 'arclength')
```

```
Solver failed when c was in the range 0.38877755511022016 to 0.6853707414829655
```





The plots display the advantage of using pseudo-arclength continuation over natural parameter continuation. The natural parameter continuation method fails at the turning points, while the pseudo-arclength continuation method does not.

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.

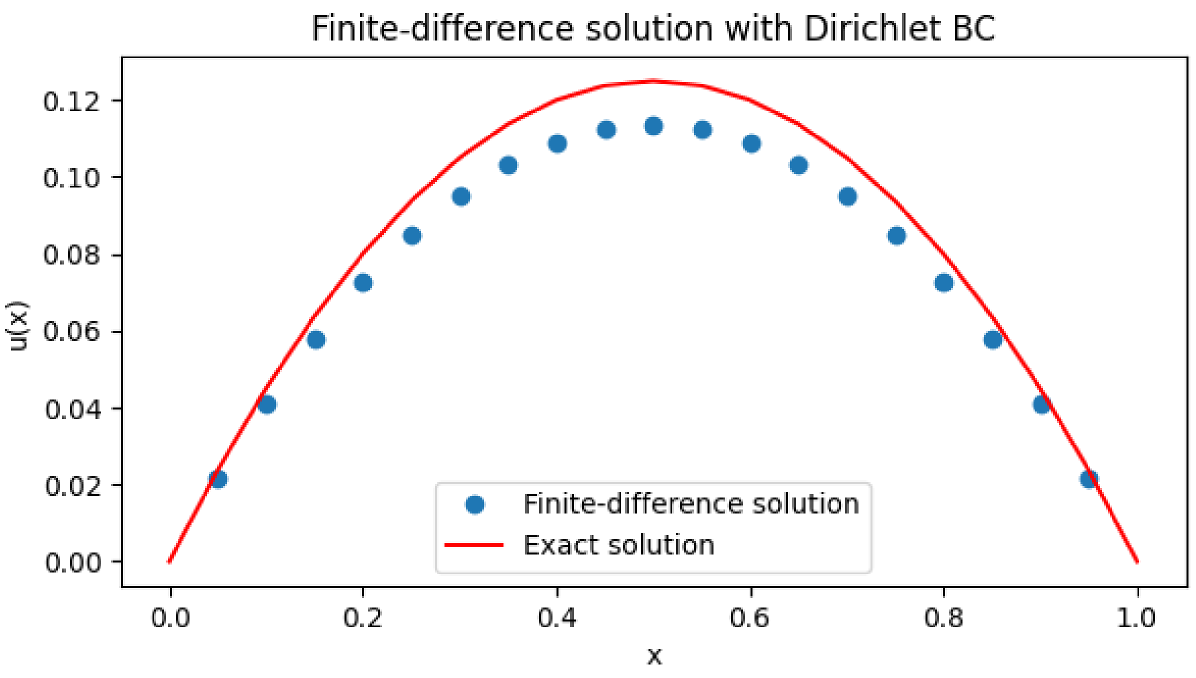
Three types of boundary conditions can be applied to a BVP; Dirichlet, Neumann, and Robin. A Dirichlet boundary condition is where the boundary value is the solution. A Neumann boundary condition is where the derivative of the solution is the boundary value. A Robin boundary condition is a combination of Dirichlet and Neumann boundary conditions where the boundary value is the derivative of the solution plus a function.

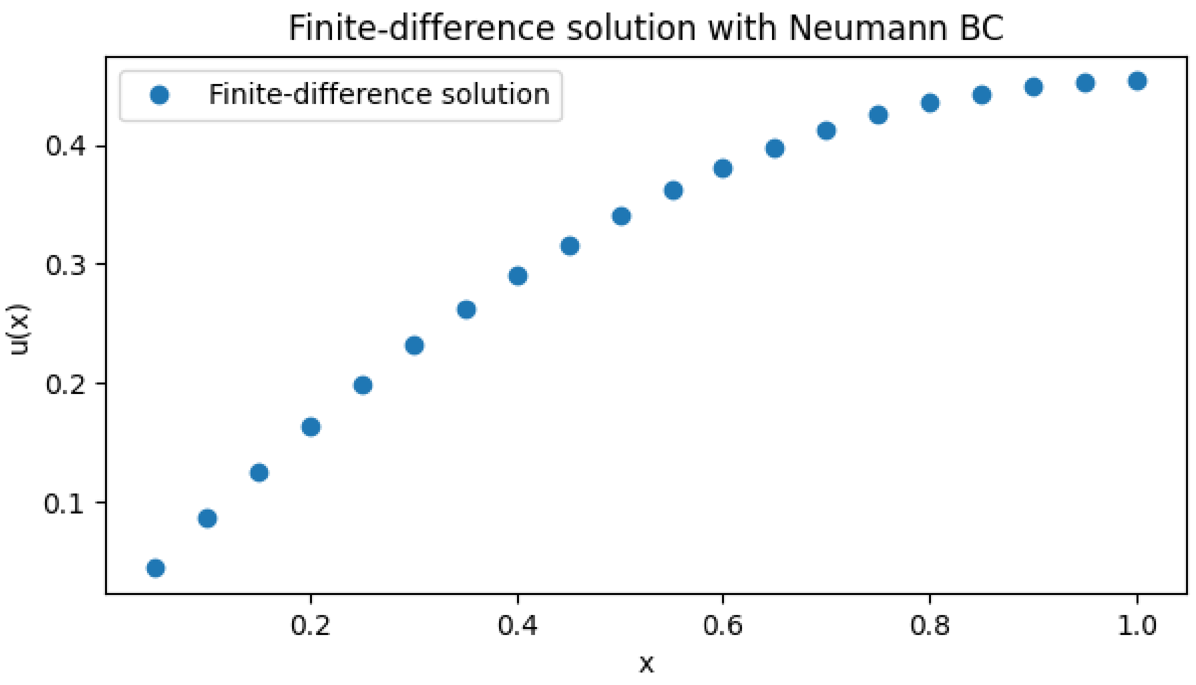
To illustrate the use of finite differences with these boundary conditions, we can consider the BVP,

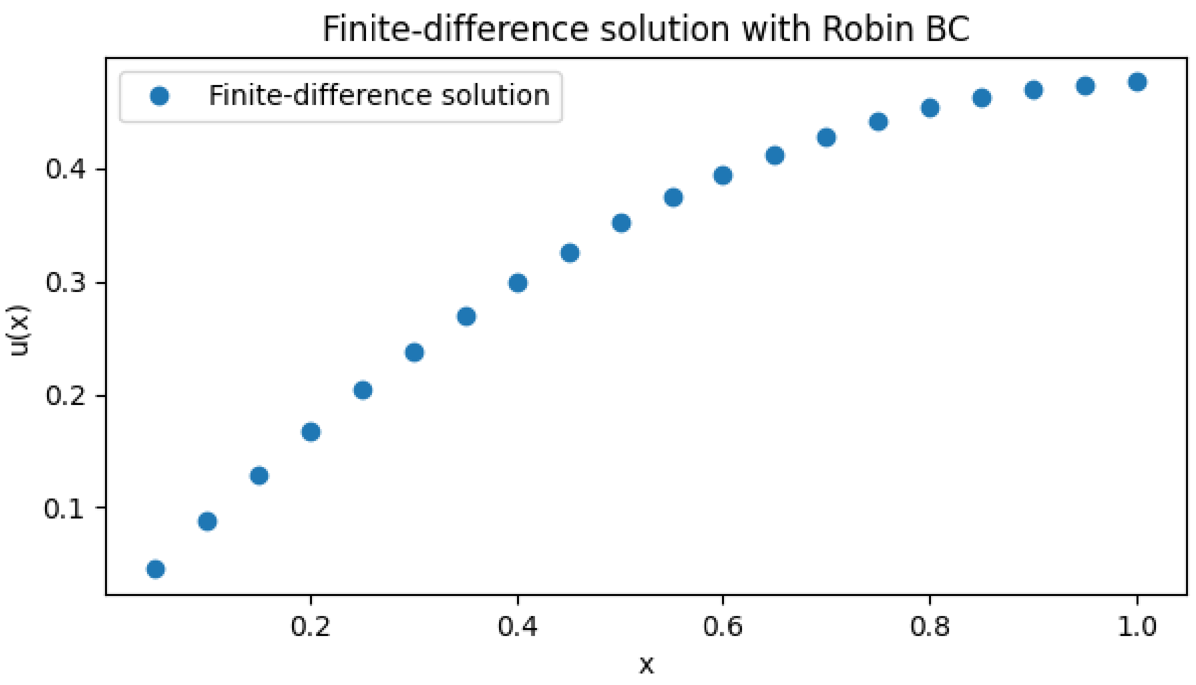
$$\frac{d^2u}{dx^2} + q(x) = 0, u(a) = 0, u(b) = 0. \tag{7}$$

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 [11]: from BVPsolve import solve_bvp
N = 21; a = 0; b = 1
alpha = 0; beta = 0; gamma = 1; delta = 0
def q(x):
    return 1
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)
```

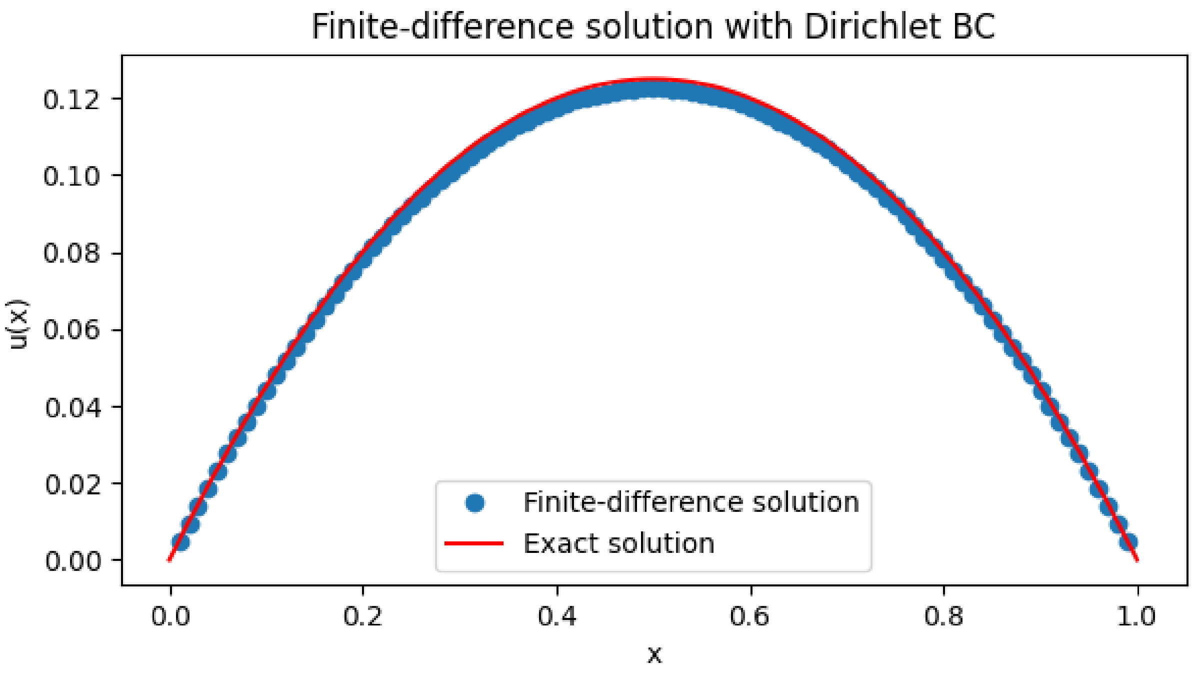






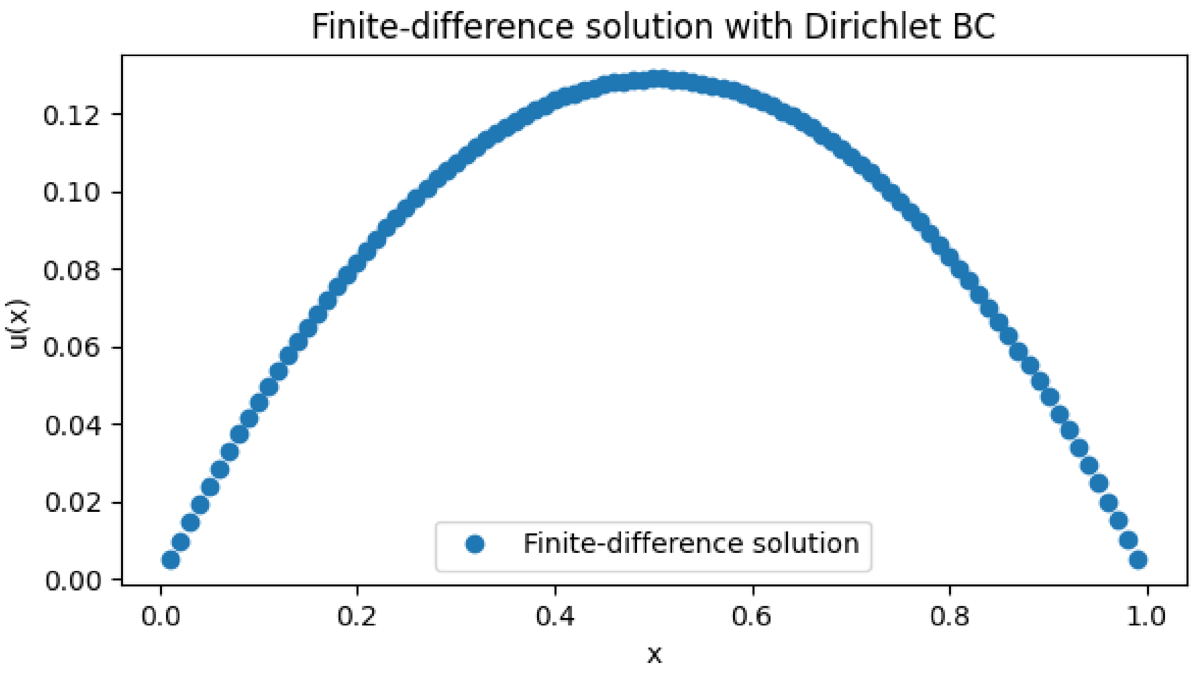
For each type of boundary condition, the numerical approximation has a significant truncation error when $N = 21$. By setting $N = 101$ the step size is much smaller therefore, the error is much smaller.

In [12]:
```
N = 101
solve_bvp(N, a, b, alpha, beta, q, "dirichlet")
```



The BVP solver also works when the source term $q(x)$ is a function of x. For example, we can evaluate the Bratu equation with $q(x) = e^{0.1x}$.

In [13]:
```
def q(x):
    return np.exp(0.1*x)
solve_bvp(N, a, b, alpha, beta, q, "dirichlet")
```



Explicit Euler, Implicit Euler, and Crank-Nicolson Methods for PDEs

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,

$$u_{x,t+1} = u_{x,t} + \frac{D\Delta t}{(\Delta x)^2} * \left(u_{x+1,t} - 2u_{x,t} + u_{x-1,t}\right), \tag{8}$$

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,

$$\frac{D\Delta t}{(\Delta x)^2} > \frac{1}{2}, \tag{9}$$

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. 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.

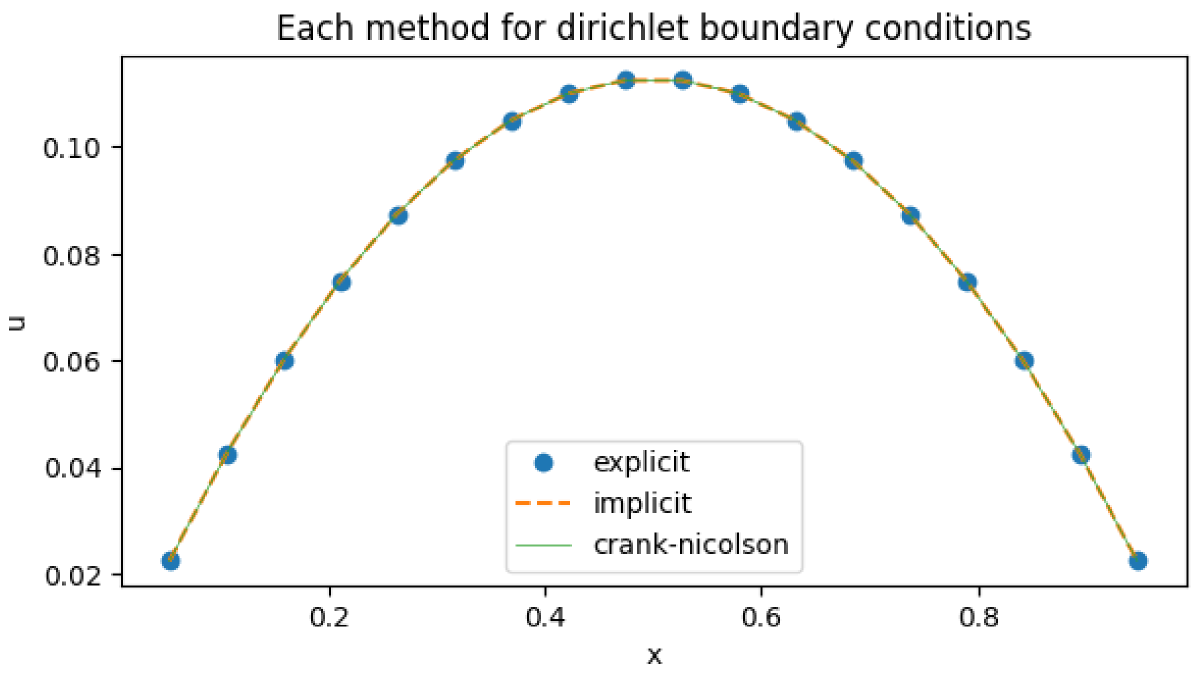
The Crank-Nicolson method is a second-order method that approximates the derivative in time using a central difference. It uses a central difference approximation for both the first and second derivatives. This means that the method is unconditionally stable and is suitable for solving stiff ODEs. It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

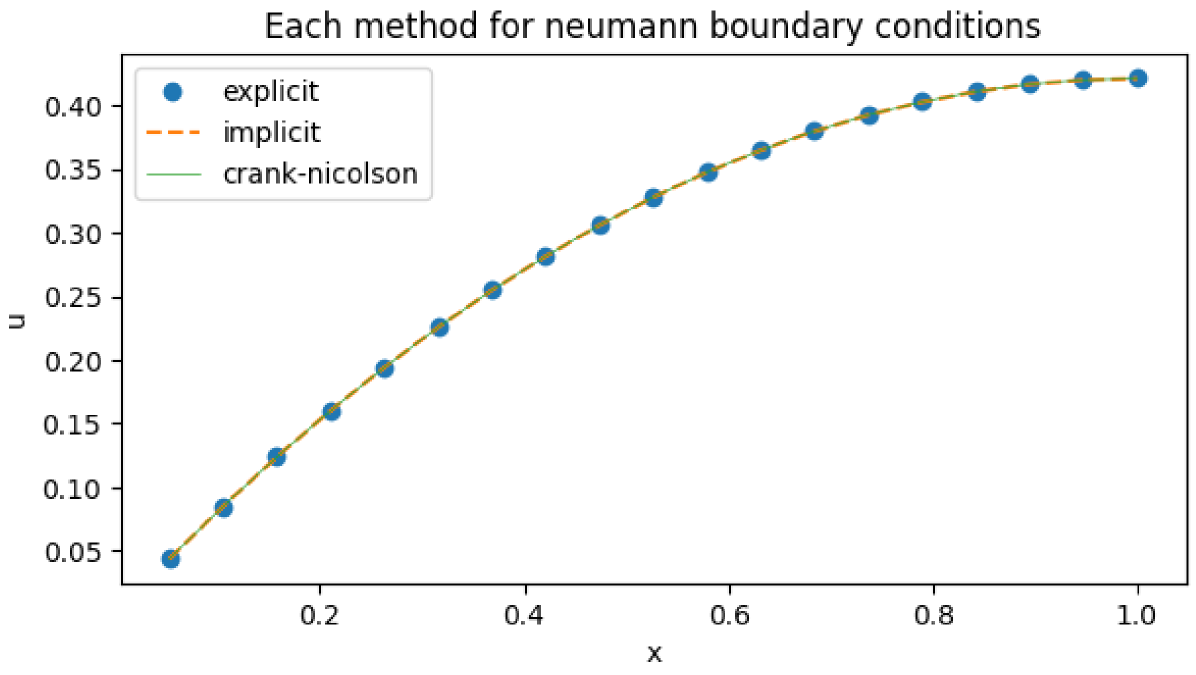
To illustrate the use of each method, we can consider the second-order PDE,

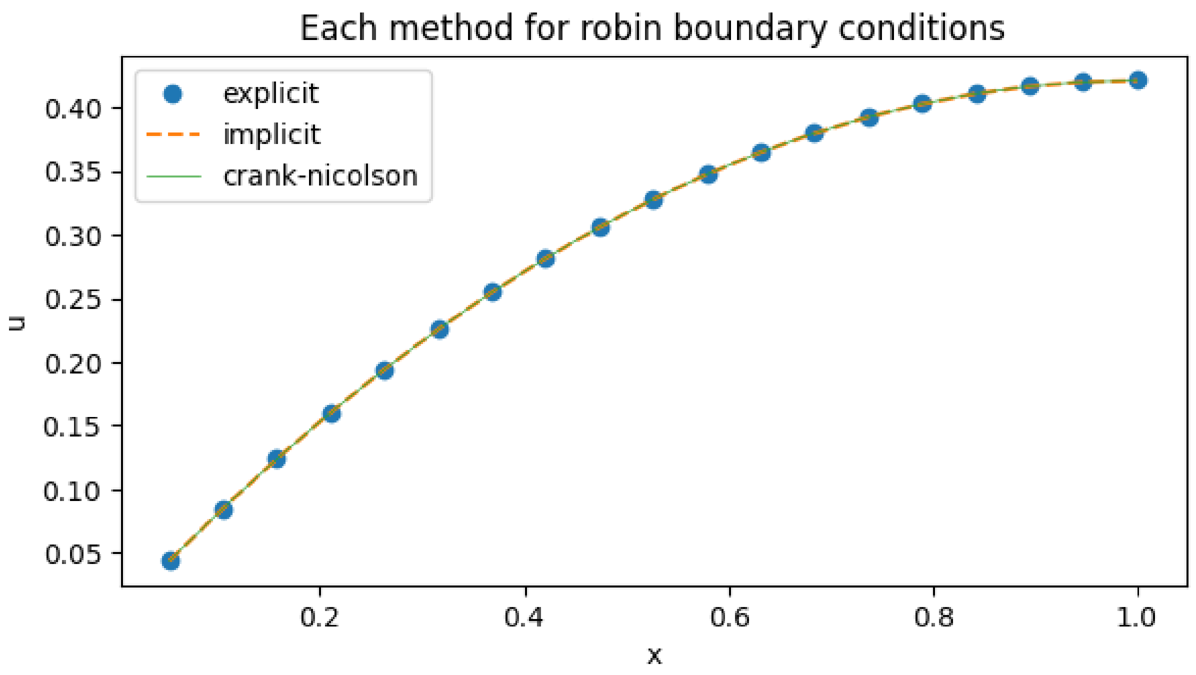
$$\frac{du}{dt} = D\frac{d^2u}{dx^2} + q(x) \tag{10}$$

where D is the diffusion coefficient and $q(x)$ is the source term. We will set $D = 1$ for simplicity. The initial condition is $u(x, 0) = 0$ and the boundary conditions are $u(0, t) = 0$ and $u(1, t) = 0$. The code uses the same boundary types as the previous section. The first set of plots shows the solutions for $q(x) = 1$, the second set of plots shows the solutions for $q(x) = e^{0.1x}$ and the third set of plots shows the solutions for $q(x, u) = (1 - u) \cdot e^{-x}$.

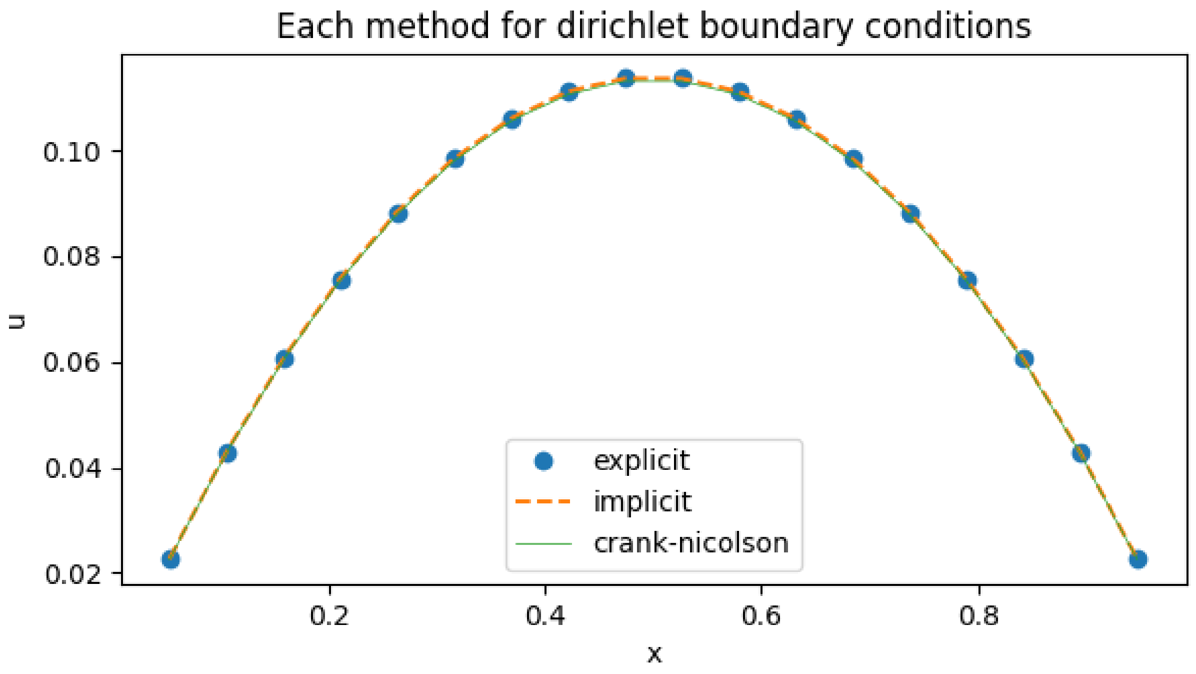
In [14]:
```python
from PDEsolve import plot_method_forall_bctypes

# define the initial condition
def u0(x, t):
    return 0.0

def q(x, t, u, mu):
    return 1.0

# define the source term for the

# Define the parameters for the PDE
N = 20; a = 0.0; b = 1.0; D = 1; u0 = u0; t_max = 1.0; dt = 0.001
plot_method_forall_bctypes(N, a, b, D, u0, t_max, dt, q, "dirichlet")
plot_method_forall_bctypes(N, a, b, D, u0, t_max, dt, q, "neumann")
plot_method_forall_bctypes(N, a, b, D, u0, t_max, dt, q, "robin")
```

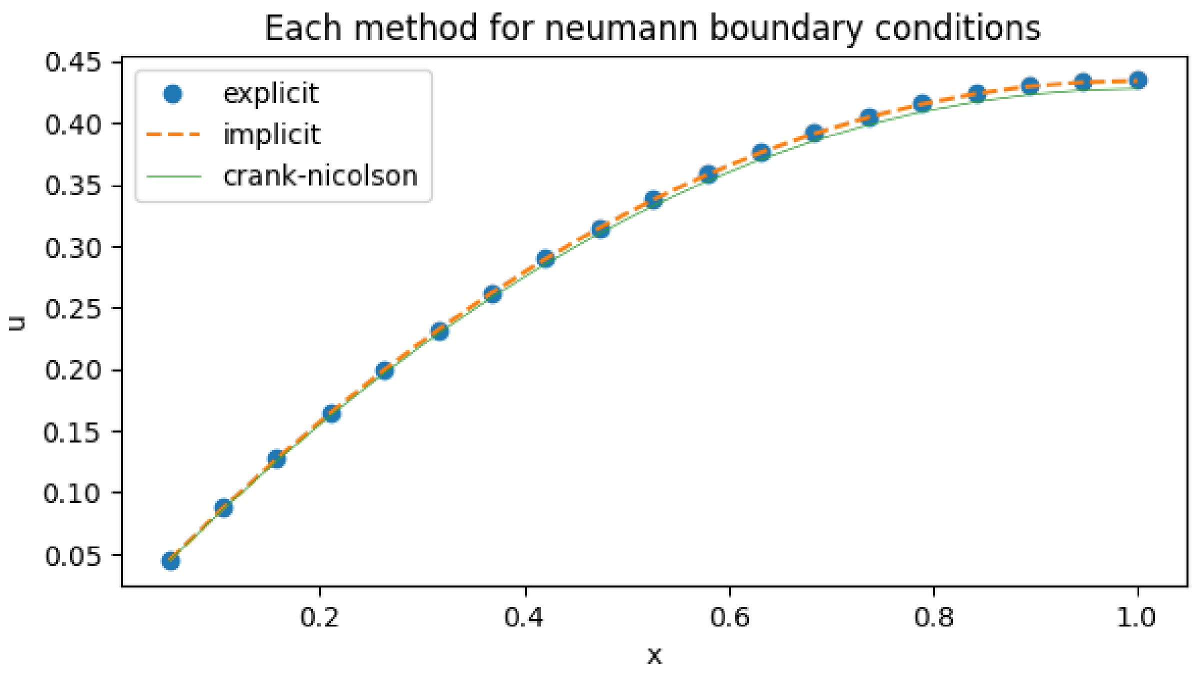

Each method for dirichlet boundary conditions
u
x
0.02
0.04
0.06
0.08
0.10
0.2
0.4
0.6
0.8
explicit
implicit
crank-nicolson

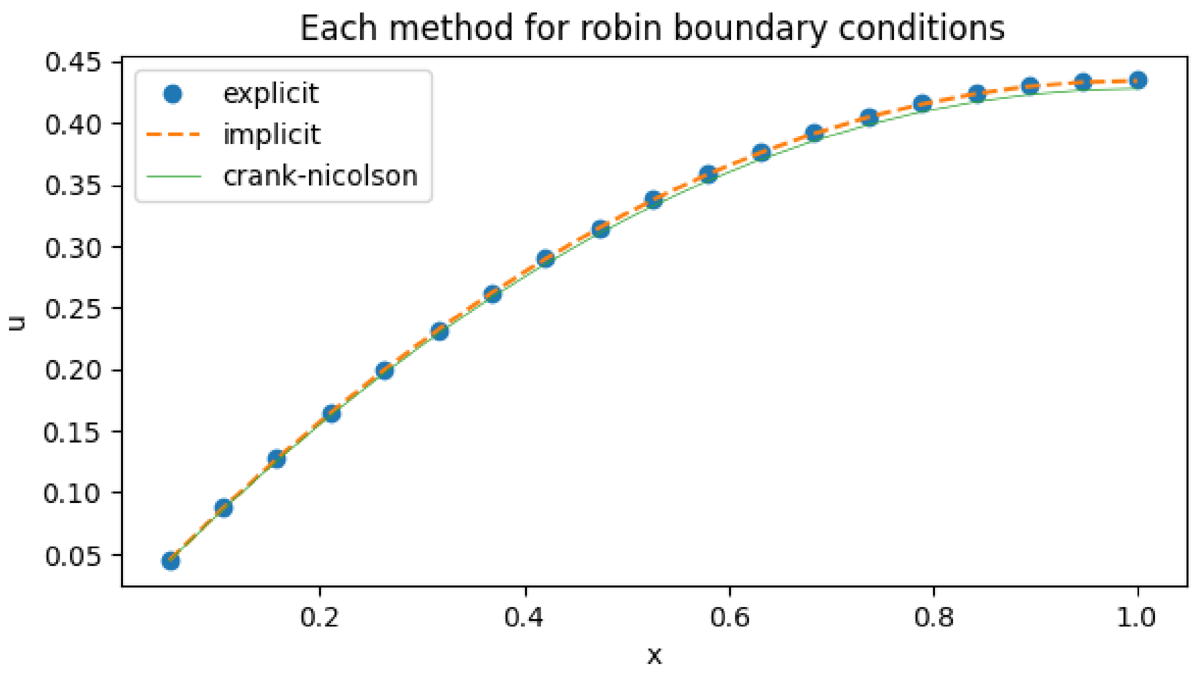

Each method for neumann boundary conditions
u
x
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.2
0.4
0.6
0.8
1.0
explicit
implicit
crank-nicolson



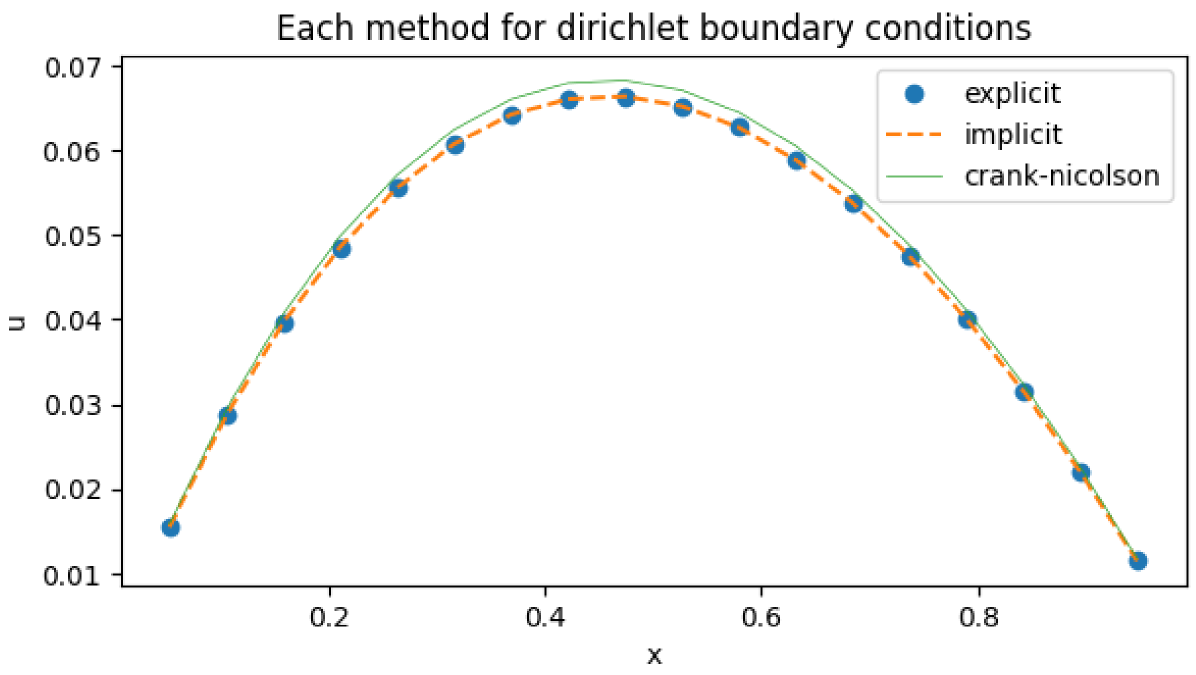
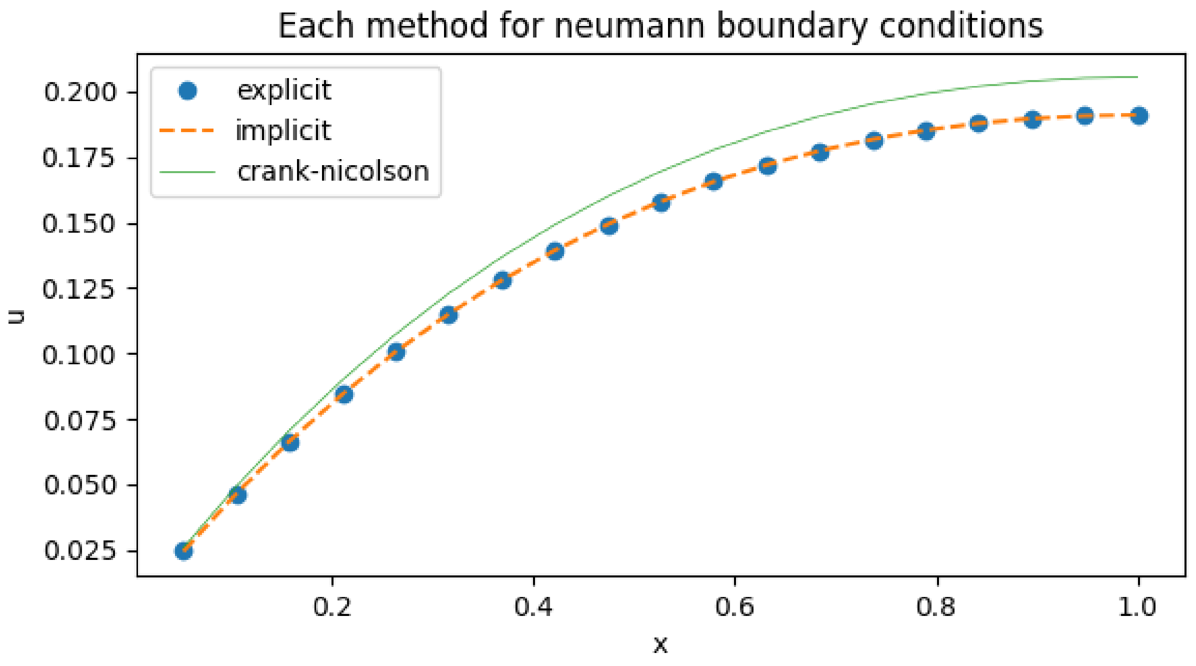
In [15]:
```
def q(x, t, u, mu):
    return np.exp(0.1*u)
plot_method_forall_bctypes(N, a, b, D, u0, t_max, dt, q, "dirichlet")
plot_method_forall_bctypes(N, a, b, D, u0, t_max, dt, q, "neumann")
plot_method_forall_bctypes(N, a, b, D, u0, t_max, dt, q, "robin")
```

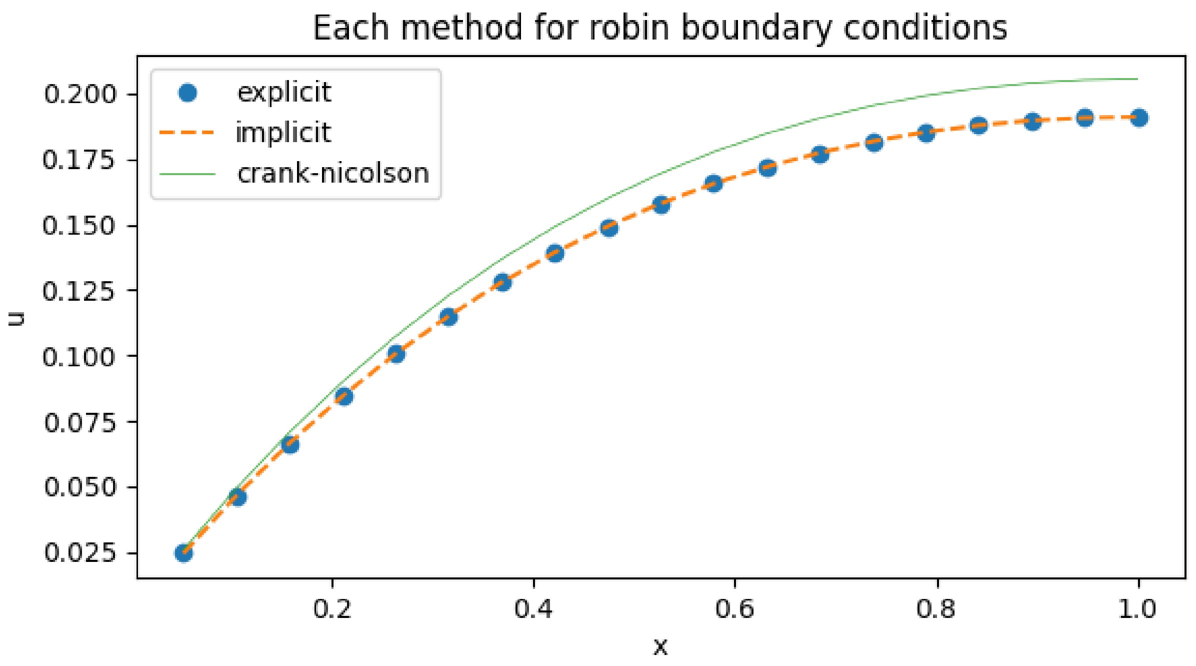






In [16]:
```
def q(x, t, u, mu):
    return (1-u)*np.exp(-x)

plot_method_forall_bctypes(N, a, b, D, u0, t_max, dt, q, "dirichlet")
plot_method_forall_bctypes(N, a, b, D, u0, t_max, dt, q, "neumann")
plot_method_forall_bctypes(N, a, b, D, u0, t_max, dt, q, "robin")
```


Each method for dirichlet boundary conditions
explicit
implicit
crank-nicolson
u
x
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.2
0.4
0.6
0.8

Each method for neumann boundary conditions
explicit
implicit
crank-nicolson
u
x
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.2
0.4
0.6
0.8
1.0



The plots show that the Crank-Nicolson method is more accurate than the other two methods.

Software Design Decisions

ODE Solver

In the file 'ODEsolve.py' I separated the functions into smaller ones that do simpler tasks in this file because of the uses each could have in other solvers. For example, the euler_step and rk4_step functions are used in the shooting method solver. The modularity of the ODE solver is important as it allows the user to input their own one or two-dimensional ODEs and initial conditions. The step function is also taken as input so the user can choose or even create their step function to use in the solver.

The code is very readable and easy to understand because of the use of descriptive variable names and comments. Additionally, the functions are properly documented with docstrings explaining their inputs, outputs, and purpose. This makes it easy for the user to understand the code.

The ODE solver is also reliable by raising errors when the user inputs invalid arguments. For example, if the user inputs a step function that is not callable, the solver will raise a TypeError. Every error message is descriptive and explains what the user did wrong which makes it easy for the user to fix their mistake. I attempted unit testing for the ODE solver but I was not able to get it to work. I would like to learn how to do this in the future because it would make the code much more reliable. The only limitation to the solver is that the ODE must be one or two-dimensional but I believe this is a reasonable limitation. Other than that, the solver is very robust and can solve any ODE as long as the user inputs the correct arguments.

Numerical Shooting Method

The file Shooting_method.py contains functions for solving one and two-dimensional ODEs using the shooting method. The code is easy to read and understand because each function is well-named and has a docstring explaining its purpose, inputs, and outputs. Each function is succinct and does one specific task. This makes the code modular and easy to use in other solvers if needed. The code is also reliable because the arguments are checked for validity before the code is executed. Overall the file is well-organized and efficient at solving ODEs. However, the code is not very robust because it can only solve one and two-dimensional ODEs.

Numerical continuation

The file 'Numerical_continuation.py' uses larger functions than previous files because these functions are not used in other solvers. The functions are well-named and explain their purpose to the user effectively. The two main functions perform natural parameter continuation and pseudo-arclength continuation. Both use the 'root' function within Scipy. I chose to use this function over 'fsolve' because it can keep track of the success and failure of the solver. This allowed me to output the range at which the solver failed when using the natural parameter method. I also started using Numpy arrays instead of lists in this file because they are more capable of doing mathematical operations. They are also quicker than lists which is important when using large arrays. I was unable to figure out how to use continuation on a two-dimensional ODE so the code is limited to one-dimensional ODEs. As a result of this, I felt it was not necessary to do unit testing for this file.

BVP solver and MOL class

For the BVP solver, I decided to use classes so that all the information to create boundary conditions was in one place. This turned out to be effective as the PDE solver also uses the same class. The file 'MethodOfLines_class.py' contains two classes. The 'Grid' class creates a discrete grid and the 'BoundaryCondition' class creates the boundary conditions. There is also a function to construct A and b matrices that the 'BoundaryCondition' class can manipulate depending on the boundary type. I chose to do it this way because it allows the user to create their own boundary conditions by creating a new instance of the 'BoundaryCondition' class. The file 'BVPsolve.py' is well-organized and allows the user complete freedom to create their own boundary conditions because there are 9 different arguments to modify. The code uses Numpy's 'linalg' module to solve the system of equations. The code is also reliable because it checks for invalid arguments before executing the code. I feel this is my best code because it is very robust and can solve any BVP as long as the user inputs the correct arguments. I tried performing unit testing for this file but I was not able to get it to work.

PDE solver

The file 'PDEsolve.py' contains two functions, one for solving a PDE and one for plotting the solution. The 'solve_pde' function is longer than I hoped for but I was not able to find a way to make it shorter or split it into smaller functions. The function is well-documented so it is easy for the user to understand what each argument does. The function is also reliable because there are lots of checks for invalid arguments. I created a solver for just explicit Euler called 'EEmethod.py' that worked well, but it was messy and slow. So, like the BVP solver, the PDE solver uses the MOL class to create the matrices and Numpy's 'linalg' module to solve the system of equations. This code is also very robust because it has provided a solution to every different source term I have tried. I chose not to do unit testing for this file because I ran out of time and was unsuccessful in my attempts for other files.

Reflective Learning Log

Throughout this unit, I have gained a wealth of knowledge regarding software engineering principles. As well as how to tackle the challenges involved in developing functional and well-documented code. One of the most valuable lessons I have learned is the importance of thoroughly understanding the problem before beginning the code. At the start of the course, I was rushing through sections that ended up causing a lot of hassle later down the line. By mapping out precisely what needs to be done beforehand, the issues became much clearer towards the end of the course. In addition, planning enabled me to consider the program's structure. Furthermore, I have discovered that writing docstrings for the code as I go along is an effective way to approach this task. Initially, I had planned to complete this step at the end, but I soon realized that it is much more beneficial doing it as I went. Describing what the function does, the inputs it requires, and the outputs it produces helped me view the function as part of the bigger picture.

Technically, I have gained valuable abilities. I have learned how useful classes can be when done correctly. Moreover, I have improved my use of git. By committing changes regularly as soon as a piece of code works, I can keep a log of my work and rectify any future errors by reverting to prior commits.

This course has also encouraged me to think more critically about the quality of my code. I now consider how different functions interact and pass values to one another and how to make it more user-friendly for others to use and develop. These skills will undoubtedly serve me well in future professional endeavours and when revisiting old code projects.

If allowed to start the unit again, I would thoroughly plan what I want out of each function. I would also seek more assistance from the TAs early on to help me understand how functions should interact. Doing this would give me the time to perform unit testing on all my code, which is a skill I would like to learn. To increase modularity, I would use more classes. To make my code more reliable, I would make each function more general so it is possible to solve higher-dimension differential equations. Finally, to improve efficiency I would use sparse matrices for BVPs and PDEs as well as implement Numpy arrays throughout the functions.