SEMT30002 Scientific Computing and Optimisation

Week 2 Demos: ODE BVPs

Matthew Hennessy

The demos this week will showcase how to solve ODE boundary value problems (BVPs) using the finite difference method.

First problem - Dirichlet boundary conditions

The first problem we'll solve is

$$\frac{d^2u}{dx^2} = 0, \quad u(a) = \alpha, \quad u(b) = \beta$$

Method

We'll discretise the problem using finite differences. The algebraic system of equations will be constructed using `for` loops. SciPy's `root` function to solve the algebraic system. To use the `root` function, we need to write the discrete system in the form $\boldsymbol{F}(\boldsymbol{u}) = \boldsymbol{0}$.

To get started, we first import the required packages:

In [1]:
```
from scipy.optimize import root
import numpy as np
import matplotlib.pyplot as plt
```

We'll now specify the problem parameters. Let's take $a = 0$, $b = 1$, $\alpha = 0$, and $\beta = 1$. We'll use 21 grid points to approximate the solution; this means setting $N = 20$. These parameters will now be coded into Python variables:

In [2]:
```
# Number of grid points (minus one)
N = 20

# Start and end of the domain
a = 0
b = 1

# Dirichlet boundary condition values
```

```
alpha = 0.0
beta = 1.0

# Create the grid points, including the points on the boundary
x = np.linspace(a, b, N+1)

# Calculate the grid spacing
dx = (b - a) / N

# extract the interior grid points
x_int = x[1:-1]
```

A Python function that builds the discretised ODE will now be defined. Remember that when using SciPy's `root` function (or Matlab's `fsolve`), the discretised ODE must be written as an algebraic system of equations that has the form $\boldsymbol{F}(\boldsymbol{u}) = \boldsymbol{0}$. We need to code up a function that computes $\boldsymbol{F}(\boldsymbol{u})$. The solver will then compute $\boldsymbol{u}$ such that $\boldsymbol{F}(\boldsymbol{u}) = \boldsymbol{0}$.

In the code below, we use the reduced algebraic system that incorporates the Dirichlet boundary conditions directly into discrete ODE. This system has $N-1$ unknowns.

In [3]:
```
def dirichlet_problem(u, N, dx, alpha, beta):
    """
    Builds the algebraic system that arises from discretising the ODE.
    Note that Python indexing here is not ideal
    because it does not match the mathematical formulation of the
    problem (which uses 1 as the first index)
    """

    # Pre-allocate an array to store the algebraic system
    F = np.zeros(N-1)

    # The discrete ODE at the first interior grid point
    F[0] = (u[1] - 2*u[0] + alpha) / dx**2

    # The discrete ODE at the last interior grid point
    F[N-2] = (beta - 2 * u[N-2] + u[N-3]) / dx**2

    # The discrete ODE at all the other interior grid points
    for i in range(1, N-2):
        F[i] = (u[i+1] - 2 * u[i] + u[i-1]) / dx**2

    return F
```

Having defined a Python function that generates the discretised ODE, we now solve this using SciPy's `root` function.

The first step involves setting an initial guess for the solver. This can be a critical step; as a poor initial guess of the solution may cause the solver to diverge (not find a solution).

In [4]:
```
# set the initial guess
u_guess = 0.1 * x_int
```

We now attempt to solve the problem using `root`:

In [5]:
```
# try to solve the problem using root
sol = root(dirichlet_problem, u_guess, args = (N, dx, alpha, beta))
```

The solver ran, but did it find a solution? We can use the `success` attribute in the `sol` object to check:

In [6]:
```
print(f'Did the solution converge? {sol.success}')
```

```
Did the solution converge? True
```

If the value is False, then the solver did not converge and a solution was not found. In this case, change the initial guess of the solution and try again.

If the value is True, then the converge did converge and a solution was found. If this is the case, then we'll now extract the solution from the `sol` object. This will be the solution at the interior grid points,

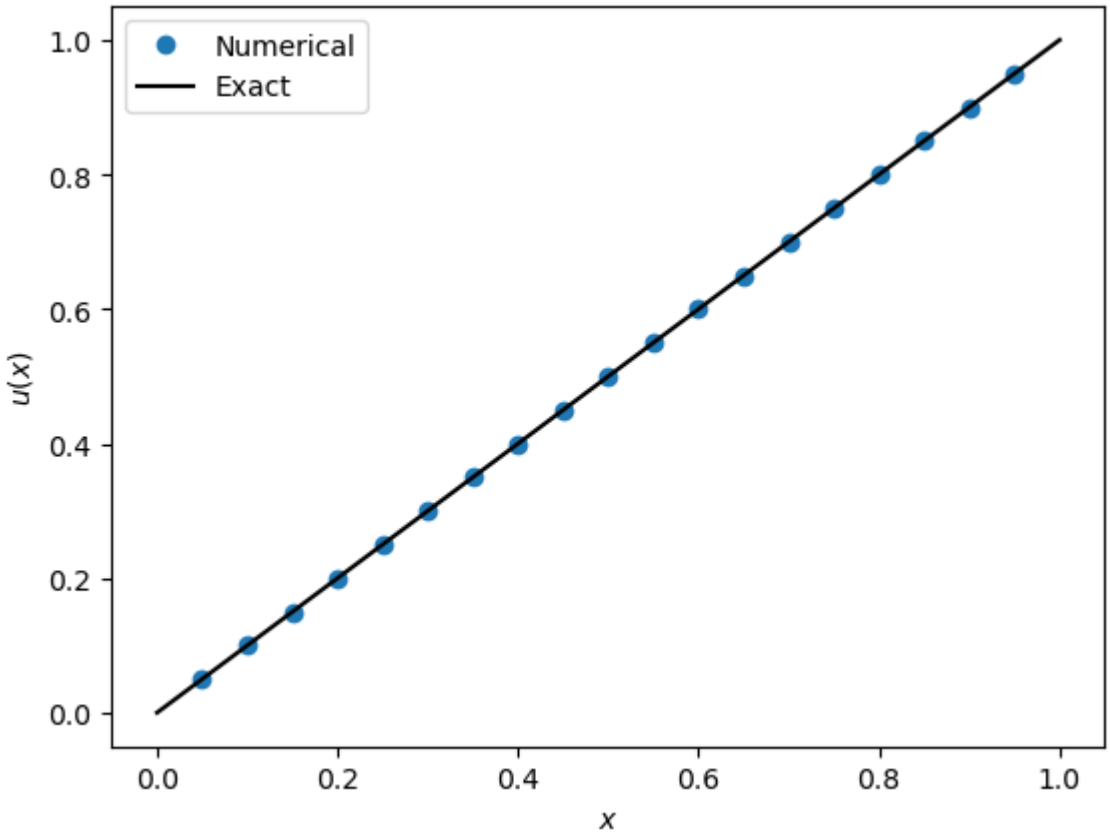
In [7]:
```
# extract solution at the interior grid points
u_int = sol.x
```

Now we'll plot the numerical solution and compare it against the exact solution. For this problem, the exact solution is given by $u(x) = x$.

In [8]:
```
# plot the numerical solution
plt.plot(x_int, u_int, 'o', label="Numerical")

# add the exact solution
plt.plot(x, x, 'k', label = "Exact")

# add labels to the axes
plt.xlabel('$x$')
plt.ylabel('$u(x)$')
plt.legend()
plt.show()
```



Second problem - source term

Now we'll add a source term into the problem and numerically solve

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

We'll set $q(x) = 1$, $a = 0$, and $b = 1$.

Method

This problem will be solved with NumPy. This means the discretised system will be formulated as a matrix-vector system of the form

$$\mathbf{A}^{DD}\boldsymbol{u} = -\boldsymbol{b}^{DD} - (\Delta x)^2\boldsymbol{q}$$

The code to build the discretised problem is contained in a handmade module that I've called `finite_diff`. This module has classes for the finite-difference grid and boundary conditions and functions for creating the matrix $\mathbf{A}^{DD}$ and the vector $\boldsymbol{b}^{DD}$.

In [9]:
```
# import my home-made module for using the finite difference method
from finite_diff import *
```

The first step is to create the spatial grid

In [10]:
```
# create the finite-difference grid with 11 points (N = 10)
grid = SpaceGrid(N = 10, a = 0, b = 1)
dx = grid.dx
x = grid.x
```

Now we create the two Dirichlet boundary conditions with $\alpha = 0$ and $\beta = 0$.

In [11]:
```
# create two Dirichlet boundary conditions
bc_left = BoundaryCondition('Dirichlet', 0.0)
bc_right = BoundaryCondition('Dirichlet', 0.0)
```

A function is now called that will create the matrix $\mathbf{A}^{DD}$ and the boundary condition vector $\boldsymbol{b}^{DD}$.

In [12]:
```
# create the matrix A^{DD} and the vector b^{DD} from the slides
A_DD, b_DD = construct_A_and_b(grid, bc_left, bc_right, matrix_type="dense")
```

Now we create a Python function that evaluates the source term in the 1D Poisson equation

In [13]:
```
# create a Python function for the source term
def q(x):
    return np.ones(np.size(x))
```

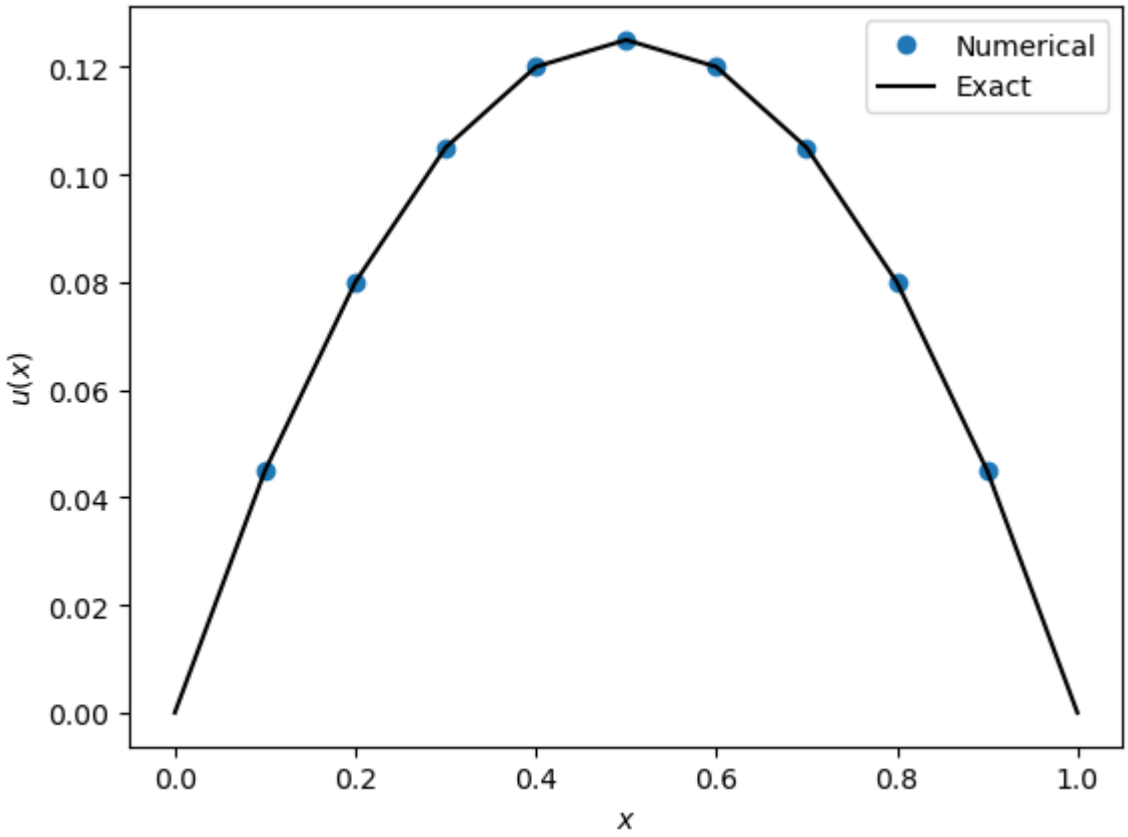
We now have all of the ingredients to solve the problem. This is done using NumPy's `linalg.solve` function:

In [14]:
```
# now solve the linear system
u = np.linalg.solve(A_DD, -b_DD - dx**2 * q(x[1:-1]))
```

Now we'll plot the numerical solution and compare it to the exact solution. In this case, the exact solution is given by $u(x) = 0.5x(1 - x)$.

In [15]:
```
# The exact solution
u_exact = 1/2 * x * (1 - x)

# Plotting
plt.plot(x[1:-1], u, 'o', label = 'Numerical')
plt.plot(x, u_exact, 'k', label = 'Exact')
plt.xlabel(f'$x$')
plt.ylabel(f'$u(x)$')
plt.legend()
plt.show()
```



Third problem - a Neumann boundary condition

Now we'll add a Neumann boundary condition and numerically solve

$$\frac{d^2u}{dx^2} + q(x) = 0, \quad u(a) = 0, \quad \left.\frac{du}{dx}\right|_{x=b} = 0.$$

Again, we'll set $q(x) = 1$, $a = 0$, and $b = 1$ and solve this problem with NumPy. The discrete system can be written as

$$\mathbf{A}^{DN}\boldsymbol{u} = -\boldsymbol{b}^{DN} - (\Delta x)^2\boldsymbol{q}$$

This demo uses a class that I've created for 1D Poisson equations, along with the same classes shown above for creating spatial grids and boundary conditions

In [16]:
```
from ode_solvers import Poisson1D
from finite_diff import *
```

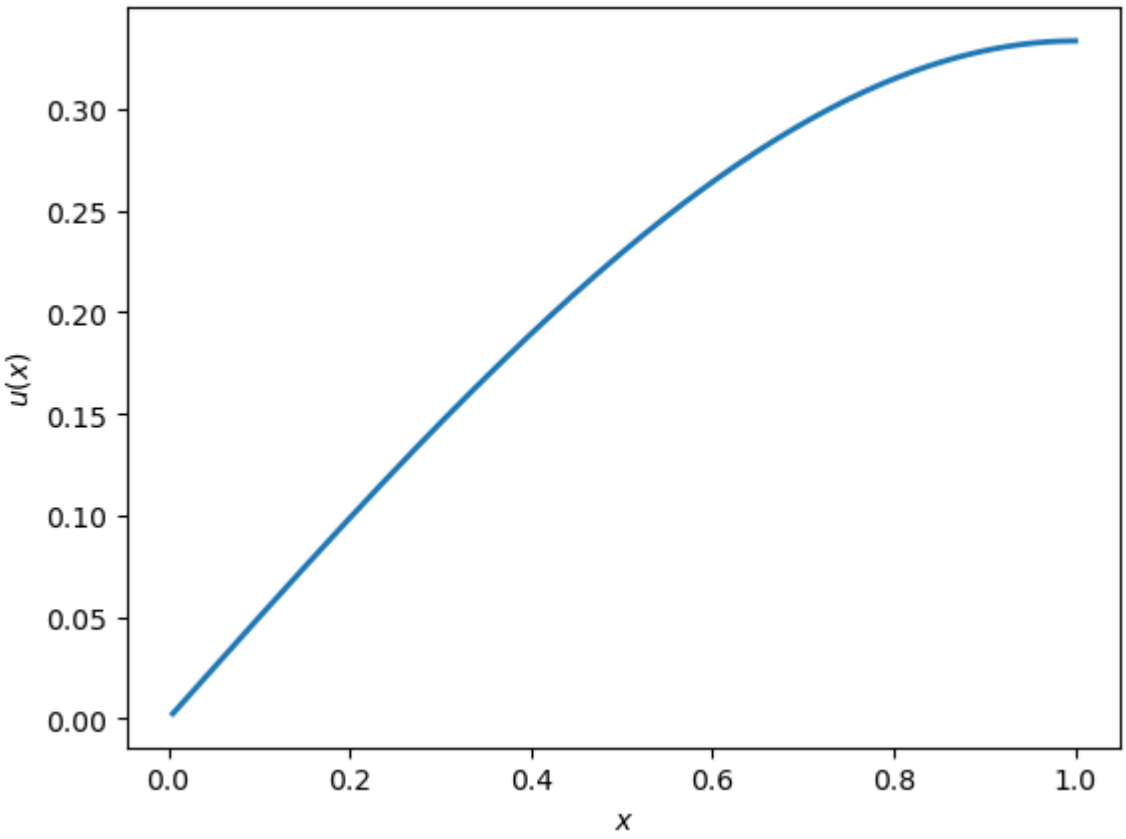
In [17]:
```
# update the right boundary condition to Neumann
bc_left = BoundaryCondition('Dirichlet', 0.0)
bc_right = BoundaryCondition('Neumann', 0.0)

# set up the grid
N = 199
grid = SpaceGrid(N, 0, 1, bc_left, bc_right)
x = grid.x
```

```
# set up the source term
def q(u, x, pars):
    return x
```

In [18]:
```
# create the Poisson eqn and solve
ode = Poisson1D(grid, bc_left, bc_right, q)
u = ode.solve()

# plot
plt.plot(x, u, lw = 2)
plt.xlabel('$x$')
plt.ylabel('$u(x)$')
plt.show()
```



`root` vs `linalg.solve`

In this demo, we solve the third problem using SciPy's `root` function and NumPy's `linalg.solve` function so that we can compare their speeds:

In [19]:
```
%timeit -r 5 -n 4 ode.solve(method = 'root')
```

```
25.1 ms ± 2.94 ms per loop (mean ± std. dev. of 5 runs, 4 loops each)
```

In [20]:
```
%timeit -r 5 -n 50 ode.solve(method = 'dense_la')
```

```
1.26 ms ± 347 µs per loop (mean ± std. dev. of 5 runs, 50 loops each)
```

Using NumPy's `linalg.solve` is about 20 times faster