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 will be used 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 [94]:
```
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 [95]:
```
# 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.
- 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 $\boldsymbol{F}(\boldsymbol{u})$.
- SciPy's `root` function 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 [96]:
```
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 [109…
```
# set the initial guess
u_guess = x_int
```

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

In [110…
```
# 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 [111…
```
print(f'Did the solution converge? {sol.success}')
```

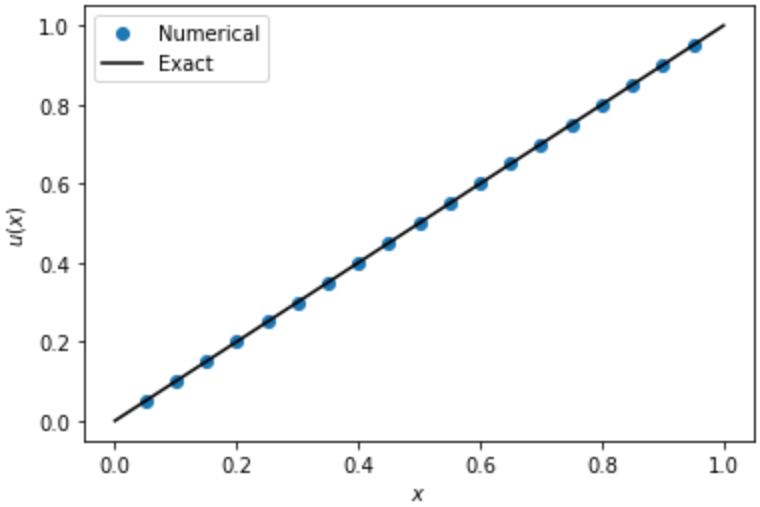
```
Did the solution converge? True
```

- 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 [112…
```
# 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 [11]:
```
# 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 first thing we do is create the grid. We'll use $N = 10$ (11 grid points in total)

In [44]:
```
# create the grid
N = 10
dx = 1 / N
x = np.linspace(0, 1, N+1)
```

- The next step is to create the matrix $\mathbf{A}^{DD}$
- We first set the matrix to an array of zeros
- We then use a for loop to loop through each row and assign values to the columns

In [46]:
```
# pre-allocate
A_DD = np.zeros((N-1, N-1))

# loop through each row
for n in range(N-1):
```

```
    # first row
    if n == 0:
        A_DD[n, n] = -2
        A_DD[n, n+1] = 1

    # last row
    elif n == N - 2:
        A_DD[n, n-1] = 1
        A_DD[n, n] = -2

    # other rows
    else:
        A_DD[n, n-1] = 1
        A_DD[n, n] = -2
        A_DD[n, n+1] = 1
```

- Now we create the boundary condition vector $\boldsymbol{b}^{DD}$.
- Recall, the boundary conditions are $u(0) = 0$ and $u(1) = 0$
- This means the $\boldsymbol{b}^{DD}$ is just a vector of zeros

In [52]:
```
b_DD = np.zeros(N-1)
```

- The last part of building the linear system is to construct the vector $\boldsymbol{q}$.
- For this problem, the source term is $q(x) = 1$.
- This means the vector $\boldsymbol{q}$ is just a vector of ones

In [48]:
```
q = np.ones(N-1)
```

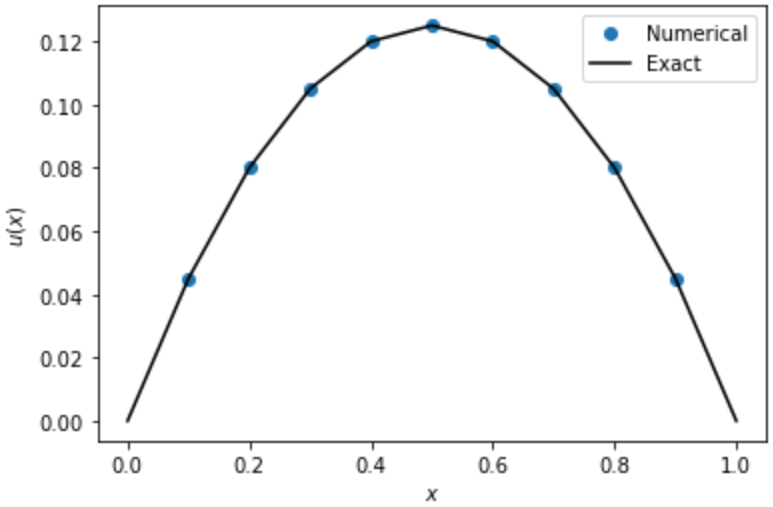
- Now that $\mathbf{A}^{DD}$, $\boldsymbol{b}^{DD}$, and $\boldsymbol{q}$ can been defined, we can solve the linear system
- This is done using NumPy's `linalg.solve` function

In [49]:
```
# now solve the linear system
u = np.linalg.solve(A_DD, -b_DD - dx**2 * q)
```

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 [50]:
```
# The exact solution
u_exact = 1/2 * x * (1 - x)
```

In [51]:
```
# 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()
```



Code optimisations

Can anyone think of a way to optimise this code?

- Notice the tridiagonal matrix $\mathbf{A}^{DD}$ is the sum of three diagonal matrices
- NumPy's `diag` function can be used to build a diagonal matrix
- This allows $\mathbf{A}^{DD}$ to be built using vectorisation

Checking convergence

- Checking the rate of convergence is useful because it provides reassurance that the numerical implementation is working correctly
- Convergence can be checked by computing the difference between the exact solution and numerical solution at the same grid point as the grid is refined (Δx becomes smaller or N increases).

- Consider the again the second problem:

$$\frac{d^2u}{dx^2} + 1 = 0, \quad u(0) = 0, \quad u(1) = 0.$$

- We will numerically solve this problem for increasing values of N
- The numerical and exact solutions at the midpoint of the domain, $x = 1/2$, will be compared
- The exact value of the solution at the midpoint is $u(1/2) = 0.125$.

- We define Python a function to create all of the arrays

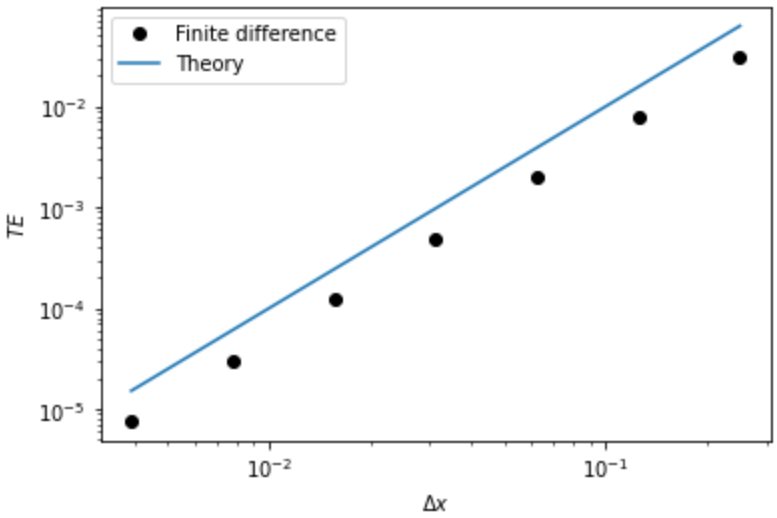
In [85]:
```
def create_arrays(N):

    x = np.linspace(0, 1, N+1)
    b = np.zeros(N-1); q = np.ones(N-1); A = np.zeros((N-1, N-1))

    for n in range(N-1):
        if n == 0:
            A[n, (n, n+1)] = (-2, 1)
        elif n == N - 2:
            A[n, (n-1, n)] = (1, -2)
        else:
            A[n, (n-1, n, n+1)] = (1, -2, 1)

    return x, A, b, q
```

- We now solve the problem for increasing values of N
- We will consider values of N that are powers of 2
- This choice will ensure that $x = 1/2$ lands on a grid point
- For each value of N, we calculate the truncation error $TE = |u_{N/2} - 0.125|$
- Since we've used a second-order discretisation, we expect $TE = O(1/N^2)$

In [93]:
```
# Define the values of N
N = np.array([4, 8, 16, 32, 64, 128, 256])

# Pre-allocate an array to store TE
TE = np.zeros(len(N))

# Loop over the values of N
for n in range(len(N)):

    # build data and arrays
    dx = 1 / N[n]
    x, A, b, q = create_arrays(N[n])

    # solve
    u = np.linalg.solve(A, -b - dx**2 * q)

    # Compute truncation error
    TE[n] = abs(u[int(N[n]/2)] - 0.125)
```

- Now we plot the results

In [92]:
```
plt.loglog(1/N, TE, 'ko', label = 'Finite difference')
plt.loglog(1/N, 1/N**2, label = 'Theory')
plt.xlabel('$\\Delta x$'); plt.ylabel('$TE$'); plt.legend(); plt.show()
```



- The plot shows that TE decreases quadratically with Δx
- This agrees with theory, so the code is likely correct