SEMT30002 Scientific Computing and Optimisation

Week 3 Demos: 1D diffusion equations

Matthew Hennessy

- In these demos, we showcase how the explicit and implicit Euler methods can be used to solve diffusion and reaction-diffusion equations.
- We start by importing some packages

In [1]:
```python
# importing packages
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# needed for animations in Jupyter notebook
%matplotlib notebook
```

Example 1 - explicit Euler

- We'll first look at how to use the explicit Euler method to solve the standard diffusion equation given by

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}.$$

- The boundary and initial conditions are given by

$$u(a,t) = \alpha, \quad u(b,t) = \beta, \quad u(x,0) = f(x).$$

- We will take $a = 0$ and $b = 1$; $\alpha = 1$ and $\beta = 0$; and $f(x) = 0$.

Approach

- The explicit Euler method will be implemented using nested `for` loops.
  - One loop is for time points, and the other loop is for grid points
- This approach does not take advantage of vectorisation so it is slow, but it's relatively straightforward to code up
- The first thing we do is define the problem parameters:

In [2]:
```
"""
    Define problem parameters
"""

# Diffusion coefficient
D = 0.5

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

# Dirichlet boundary values
alpha = 1.0
beta = 0.0

# Initial condition using lambda (anonymous) functions
f = lambda x: np.zeros(np.size(x))
```

- The next thing we'll do is define the spatial grid.
- It is helpful to define the spatial grid first because this will determine the maximum time step that can be used with Euler's method

In [3]:
```
# spatial grid
N = 20                              # number of grid points (minus one)
x = np.linspace(a, b, N+1)          # create the grid
x_int = x[1:-1]                     # extract interior grid points
dx = (b - a) / N                    # grid size
```

- We now compute the maximum size of the time step.
- This occurs when $C = D\Delta t/(\Delta x)^2 = 1/2$.
- So we first define $C_{\max} = 1/2$:

In [5]:
```
# Numerical constant that determines stability
C_max = 0.5
```

- From this value of C we can compute the maximum value that the time step can be:

In [6]:
```
dt_max = C_max * dx**2 / D
print(f'The largest time step can be {dt_max:.2e}')
```

```
The largest time step can be 2.50e-03
```

- We'll use a slightly smaller value of the time step to ensure stability.
- Moreover, we'll solve the problem until $t = 1$.
- We now discretise the time variable:

In [7]:
```
# time discretistation
dt = 2.0e-3                          # compute the time step
t_final = 1                          # final time in the simulation
N_time = int(t_final / dt)           # compute the number of time steps
t = dt * np.arange(N_time + 1)       # compute time points

# Recalculate the constant C
C = D * dt / dx**2

# print some info about the time steps
print(N_time, 'time steps will be needed')
```

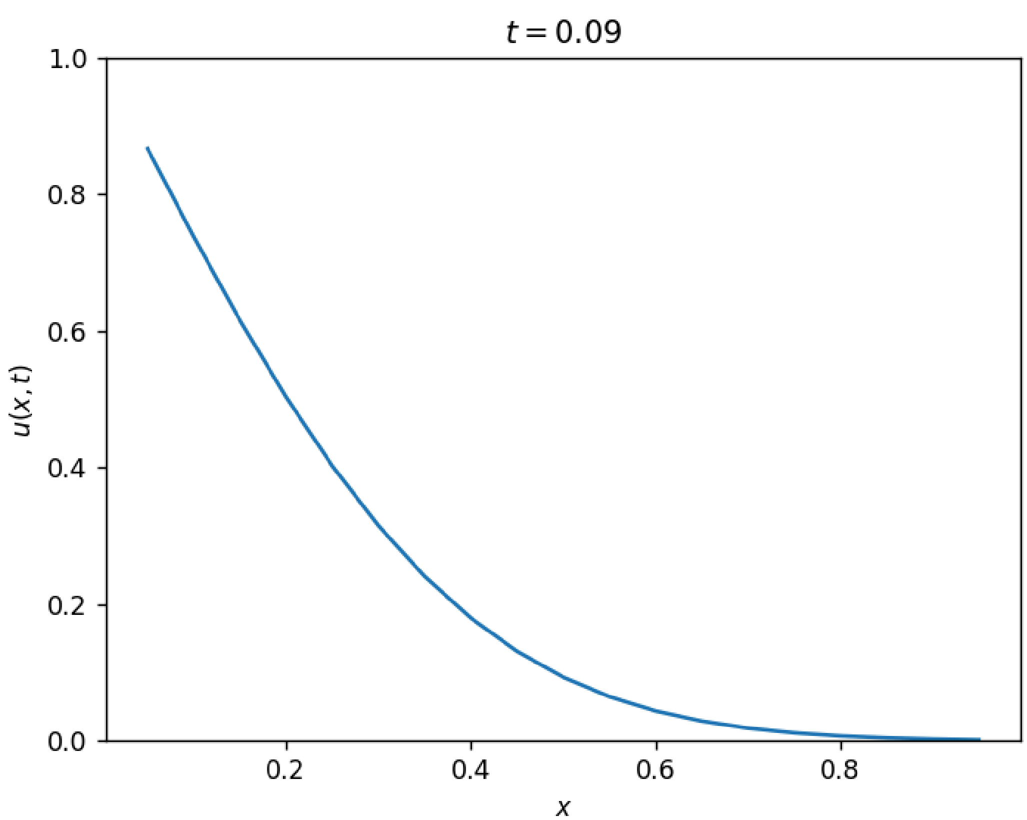
```
500 time steps will be needed
```

- Now that all of the variables have been defined, we can start solving the PDE using the explicit Euler method
- Recall that a `for` loop will be used to update the solution at each interior grid point
- We only loop over interior grid points because two Dirichlet boundary conditions are imposed

In [8]:
```
"""
    Time stepping using Euler's method
"""

# Pre-allocate the solution array.  Rows are for space, columns for time
u = np.zeros((N - 1, N_time + 1))

# Set the first column of the solution array to the initial condition
u[:, 0] = f(x_int)

# loop over the time steps
for n in range(N_time):

    # loop over the grid
    for i in range(0, N-1):
        if i == 0:
            u[0, n+1] = u[0, n] + C * (u[1, n] - 2 * u[0, n] + alpha)
        elif 0 < i and i < N-2:
            u[i, n+1] = u[i, n] + C * (u[i+1,n] - 2 * u[i, n] + u[i-1, n])
        else:
            u[N-2, n+1] = u[N-2, n] + C * (beta - 2 * u[N-2, n] + u[N-3, n])
```

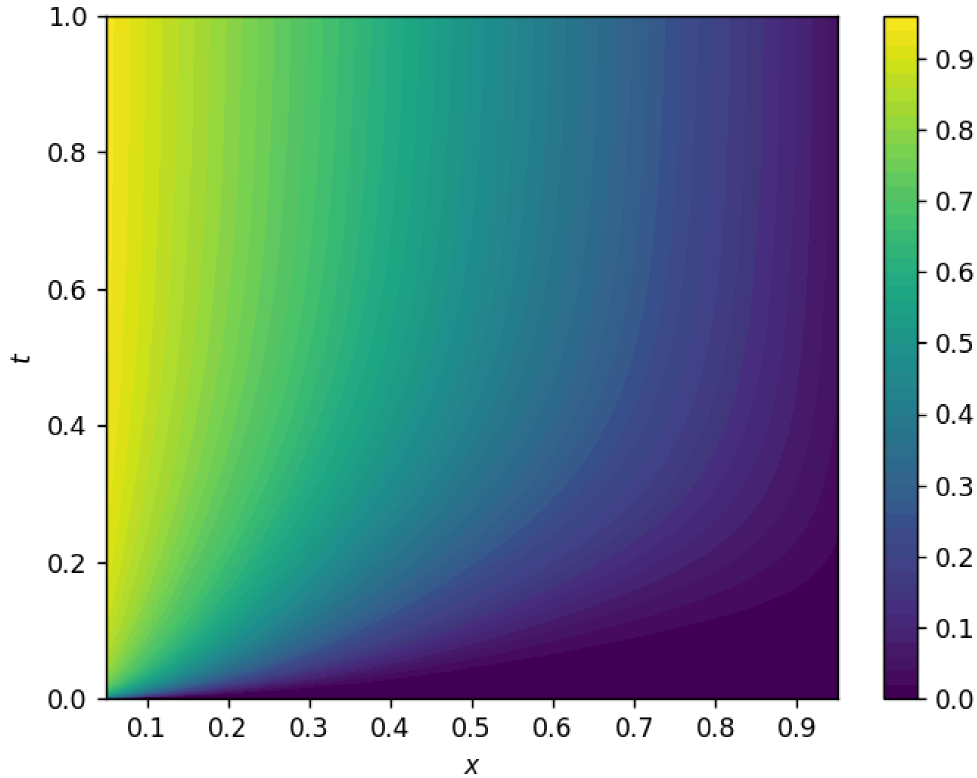
- We can use the `animation` module from `matplotlib` to create animations of the solution.
- We plot u as a function of space (x) and animate over time t
- If using VS Code with native Python files (.py), animations can be created without the `animation` module using the `plt.pause` and `plt.clf` functions from Matplotlib

In [9]:
```
"""
    animate the solution
"""
fig, ax = plt.subplots()
ax.set_ylim(0, 1)
ax.set_xlabel(f'$x$')
ax.set_ylabel(f'$u(x,t)$')

line, = ax.plot(x_int, u[:, 0])

def animate(i):
    line.set_data((x_int, u[:, i]))
    ax.set_title(f'$t = {t[i]:.2f}$')
    return line

ani = animation.FuncAnimation(fig, animate, frames=N_time, blit=True, interv
plt.show()
```



We can also create a contour plot to see the entire spatio-temporal evolution of the solution:

In [10]:
```
plt.contourf(x_int, t, u.T, 50)
plt.xlabel('$x$')
plt.ylabel('$t$')
plt.colorbar()
plt.show()
```



Numerical blow up - choosing the wrong time step

- Although the explicit Euler method is simple, there is a strict restriction on the maximum size of the time step.
- This demo will now look at what happens when a time step that is too large is used with explicit Euler.

- Recall that the max time step is given by:

In [11]:
```
C_max = 1/2
dt_max = C_max * dx**2 / D
print(f'The time step can be at most {dt_max:.2e}')
```

```
The time step can be at most 2.50e-03
```

- We'll now set the time step to be larger than this.
- Again, we'll attempt to solve the PDE until $t = 1$.

In [12]:
```
# time discretistation
dt = 5.0e-3                         # compute the time step
t_final = 1                         # final time in the simulation
```

```python
N_time = int(t_final / dt)        # compute the number of time steps
t = dt * np.arange(N_time + 1)    # compute time points

# Recalculate the constant C
C = D * dt / dx**2

# print some info about the time steps
print(N_time, 'time steps will be needed')
print(f'The value of C is {C:.2f}')
```

```
200 time steps will be needed
The value of C is 1.00
```

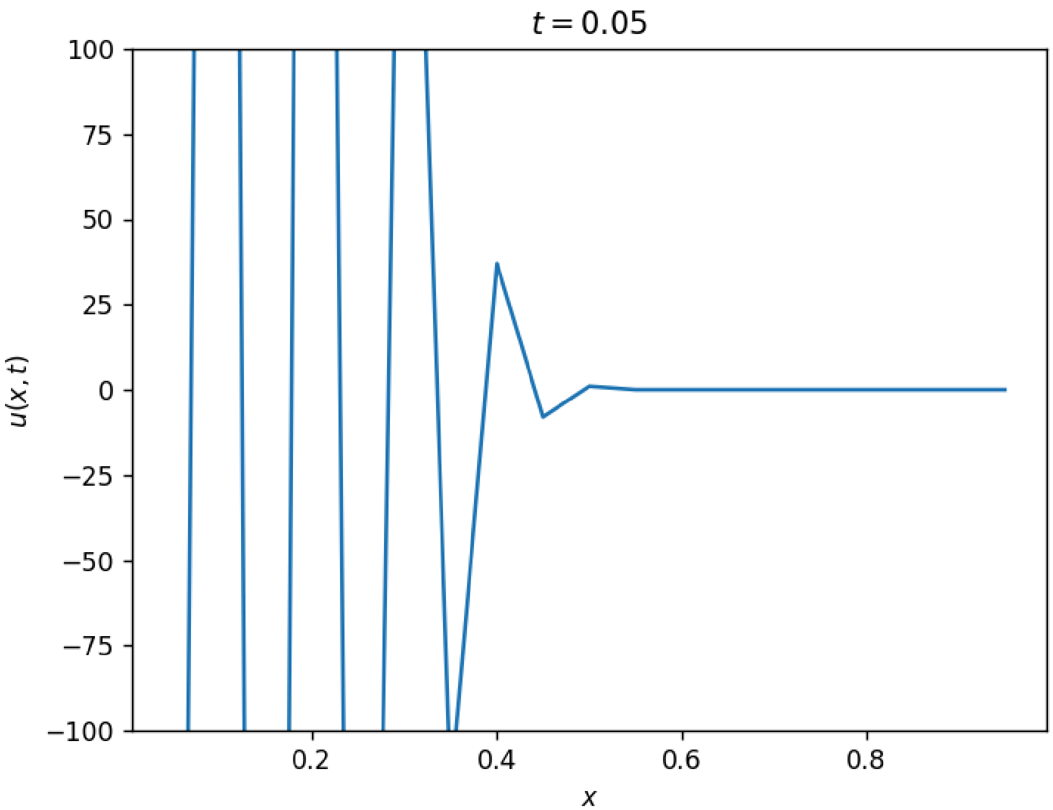
- Now we'll re-solve the problem using explicit Euler:

In [13]:
```python
"""
    Time stepping using Euler's method
"""

# Pre-allocate the solution array.  Rows are for space, columns for time
u = np.zeros((N-1, N_time + 1))

# Set the first column of the solution array to the initial condition
u[:, 0] = f(x_int)

# loop over the time steps
for n in range(N_time):

    # loop over the grid
    for i in range(0, N-1):
        if i == 0:
            u[0, n+1] = u[0, n] + C * (u[1, n] - 2 * u[0, n] + alpha)
        elif 0 < i and i < N-2:
            u[i, n+1] = u[i, n] + C * (u[i+1,n] - 2 * u[i, n] + u[i-1, n])
        else:
            u[N-2, n+1] = u[N-2, n] + C * (beta - 2 * u[N-2, n] + u[N-3, n])
```

- Animating the solution shows that it becomes oscillatory, and the oscillation amplitude grows exponentially in time

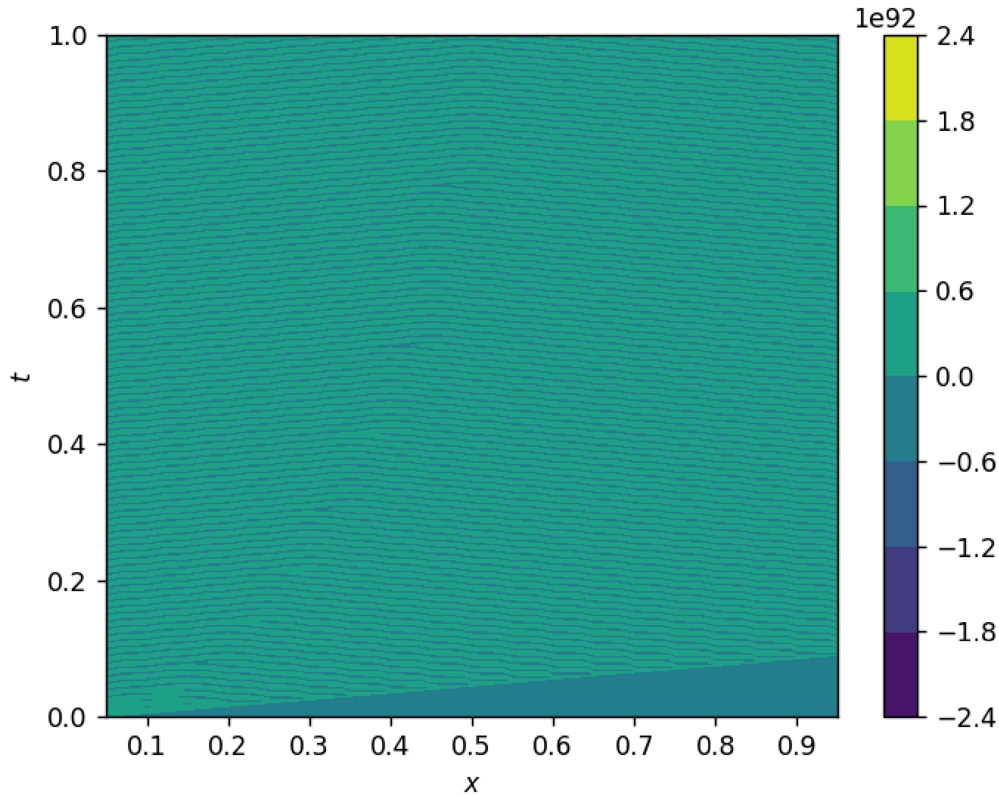
In [14]:
```python
"""
    animate the solution
"""
fig, ax = plt.subplots()
ax.set_ylim(-100, 100)
ax.set_xlabel(f'$x$')
ax.set_ylabel(f'$u(x,t)$')

line, = ax.plot(x_int, u[:, 0])

def animate(i):
    line.set_data((x_int, u[:, i]))
    ax.set_title(f'$t = {t[i]:.2f}$')
```

```
    return line

ani = animation.FuncAnimation(fig, animate, frames=30, blit=True, interval=1
plt.show()
```



A contour plot also slows that the magnitude of the solution is becoming extremely large (this is why we say the solution is blowing up)

In [15]:
```
plt.contourf(x_int, t, u.T)
plt.xlabel('$x$')
plt.ylabel('$t$')
plt.colorbar()
plt.show()
```



Example 2 - implicit Euler

- If we want to solve the diffusion equation with larger time steps, $\Delta t > (\Delta x)^2/2D$, then the implicit Euler method can be used.
- This involves solving a system of algebraic equations at each time step.

- In this demo we will solve the standard diffusion equation

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}.$$

- The boundary and initial conditions will be

$$\left.\frac{\partial u}{\partial x}\right|_{x=0} = 0, \qquad \left.\frac{\partial u}{\partial x}\right|_{x=1} = 0, \qquad u(x,0) = 1 + \cos(2\pi x)$$

- Unlike in other demos, two Neumann boundary conditions are being applied to this problem

- We first define the problem parameters and then the grid

In [16]:
```
"""
    Set up the problem parameters
```

```
"""

# Diffusion coefficient
D = 2

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

# Initial condition using lambda (anonymous) functions
f = lambda x: 1 + np.cos(2 * np.pi * x)
```

In [17]:
```
# spatial grid
N = 20                              # number of grid points (minus one)
x = np.linspace(a, b, N+1)          # create the grid
dx = (b - a) / N                    # grid size
```

- Now we calculate the largest time step that would be possible using the **explicit** Euler method

In [18]:
```
C_max = 1/2
dt_max = C_max * dx**2 / D
print(f'The time step can be at most {dt_max:.2e}')
```

```
The time step can be at most 6.25e-04
```

- We will use a time step that more than 10 times larger than this.
- This would lead to blow-up if the explicit Euler method was used
- However, the implicit Euler method will remain stable

In [19]:
```
# time discretistation
dt = 1e-2                           # compute the time step
t_final = 1                         # final time in the simulation
N_time = int(t_final / dt)          # compute the number of time steps
t = dt * np.arange(N_time + 1)     # compute time points

# Recalculate the constant C
C = D * dt / dx**2

# print some info about the time steps
print(N_time, 'time steps will be needed')
print(f'The value of C is {C:.2f}')
```

```
100 time steps will be needed
The value of C is 8.00
```

- We will use NumPy's `linalg.solve` function to solve the linear algebraic system
- Now we generate the matrix $\mathbf{A}^{NN}$

In [20]:
```
# Pre-allocate A^NN
A_NN = np.zeros((N+1, N+1))
```

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

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

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

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

- We also generate the boundary condition vector $\boldsymbol{b}^{NN}$, and the identity matrix $\mathbf{I}$.
- The boundary conditions are of the form $\partial u/\partial x = 0$, so $\boldsymbol{b}^{NN}$ is just an array of zeros.

In [21]:
```
b_NN = np.zeros(N+1)
I = np.eye(N+1)
```

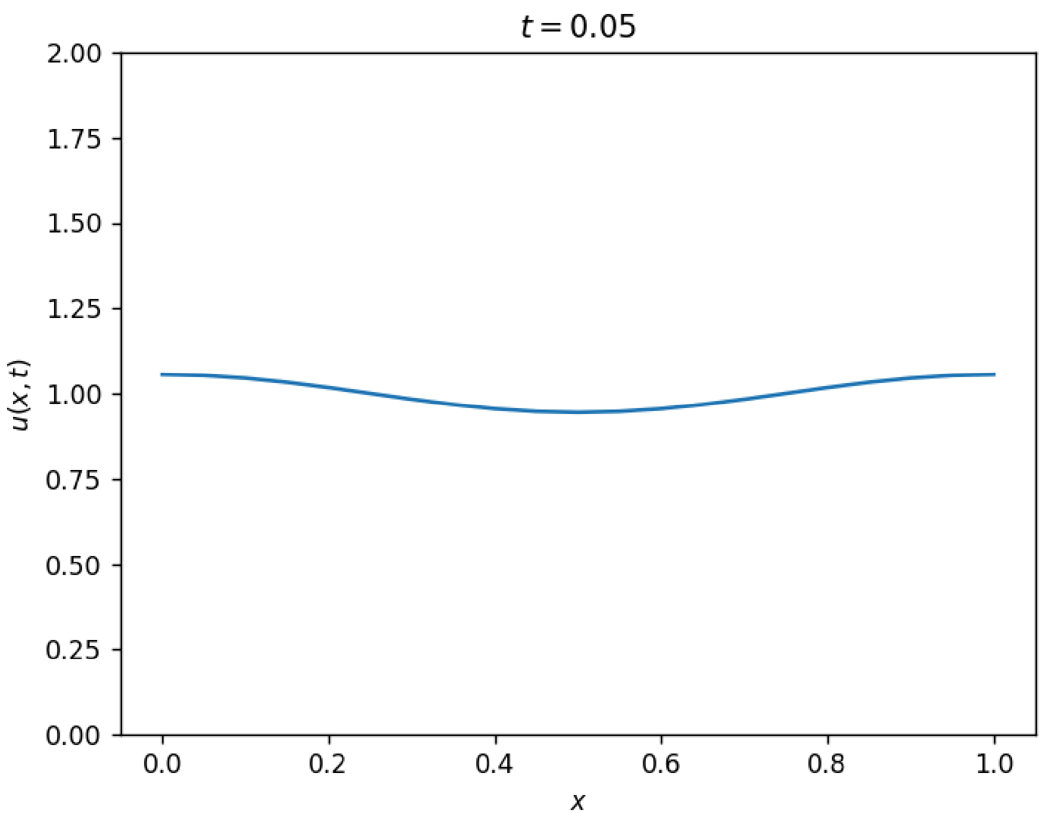
We now start time stepping using the implicit Euler method

In [22]:
```
"""
    Time stepping using implicit Euler's method
"""

# Pre-allocate the solution array.  Rows are for space, columns for time
u = np.zeros((N + 1, N_time + 1))

# Set the first column of the solution array to the initial condition
u[:, 0] = f(x)

# loop over the time steps
for n in range(N_time):

    # Solve the linear system
    u[:, n+1] = np.linalg.solve(I - C * A_NN, C * b_NN + u[:, n])
```

- As before, we animate the solution

In [23]:
```
"""
    animate the solution
"""
fig, ax = plt.subplots()
ax.set_ylim(0, 2)
```

```
ax.set_xlabel(f'$x$')
ax.set_ylabel(f'$u(x,t)$')

line, = ax.plot(x, u[:, 0])

def animate(i):
    line.set_data((x, u[:, i]))
    ax.set_title(f'$t = {t[i]:.2f}$')
    return line

ani = animation.FuncAnimation(fig, animate, frames=30, blit=True, interval=1
plt.show()
```



Validation

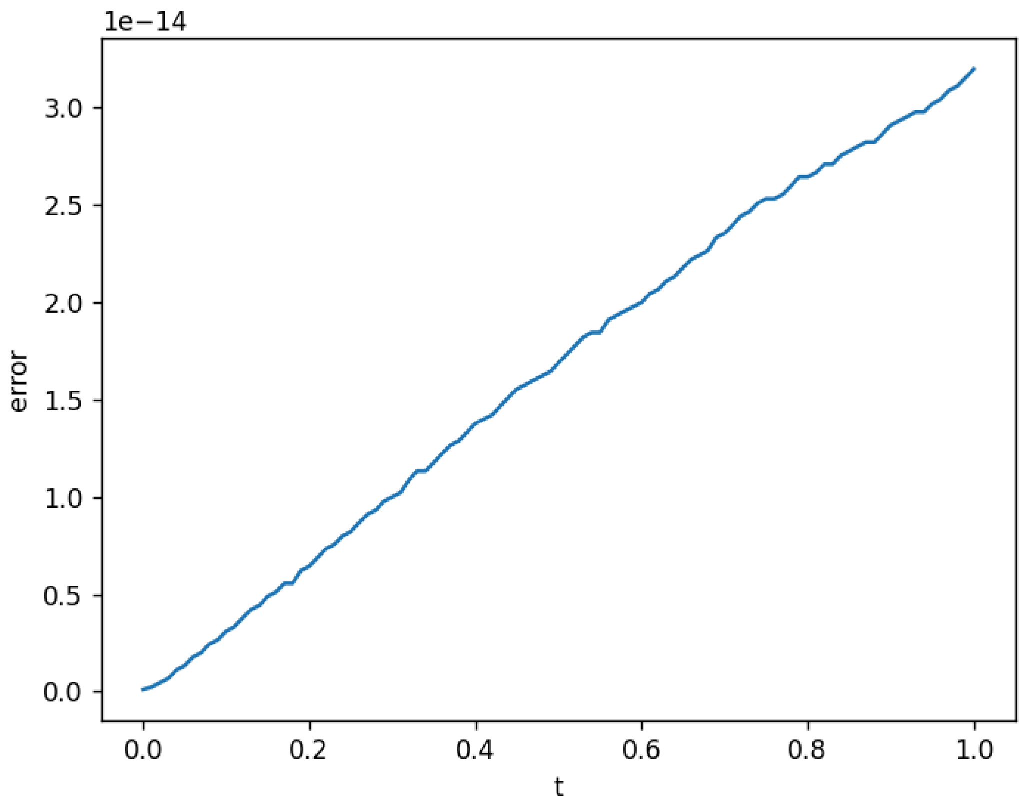
- For this problem it turns out that

$$\int_0^1 u(x,t)\,dx = 1$$

  for all time.
- In this case, the integral of u is called the mean of u
- We can check this is true using NumPy's `trapz` function as a means of validating the code

In [24]:
```python
# Pre-allocate space for the integral (mean) of u
mean_u = np.zeros(N_time+1)

# Loop over each time point
for n in range(N_time+1):

    # calculate the integral
    mean_u[n] = np.trapz(u[:, n], x)
```

In [25]:
```python
# Plot the results
plt.figure()
plt.plot(t, np.abs(mean_u - 1))
plt.xlabel('t')
plt.ylabel('error')
plt.show()
```



Example 3 - the Allen-Cahn equation

- The Allen-Cahn equation is a nonlinear reaction-diffusion equation given by

$$\frac{\partial u}{\partial t} = \varepsilon^2 \frac{\partial^2 u}{\partial x^2} + u - u^3$$

- Here, u is like a concentration, with $u = -1$ corresponding to pure water, $u = 1$ corresponding to pure oil, and $u = 0$ corresponding to a 50:50 mix of oil and water

- The initial condition will be $u(x, 0) = u_0(x)$, where $u_0(x) = 0 + \text{noise}$.
- The noise in the initial condition represents the fact there will be small fluctuations in the system.
- Noise is generated through an array of random numbers
- The boundary conditions are

$$\left.\frac{\partial u}{\partial x}\right|_{x=0,1} = 0.$$

Approach

- This PDE will be solved using the implicit Euler method
  - this requires solving a nonlinear algebraic system at each time step
  - the nonlinear system is solved using Newton's method
- This will be done using some modules I've created for solving PDEs
  - You don't have to do this, but I want to show you what is possible

In [26]:
```
# import my modules for the unit
import finite_diff as fd
from pde_solvers import DiffusionEquation
```

- The boundary conditions are defined using a Python class:

In [27]:
```
# Boundary conditions
bc_left = fd.BoundaryCondition('Neumann', lambda t: 0)
bc_right = fd.BoundaryCondition('Neumann', lambda t: 0)
```

- Then the space and time points are created using another class

In [28]:
```
# Define the grid (space and time)
grid = fd.SpaceTimeGrid(N = 150, a = 0, b = 1, t_0 = 0, T = 50, dt = 1e-1)
```

- The parameters in the problem are defined
- The only parameter is ε^2, which is equivalent to the diffusion coefficient

In [29]:
```
# Define the parameter values
eps = 1e-2
pars = {"D": eps**2}
```

- Now Python functions that evaluate the initial condition and source terms are defined

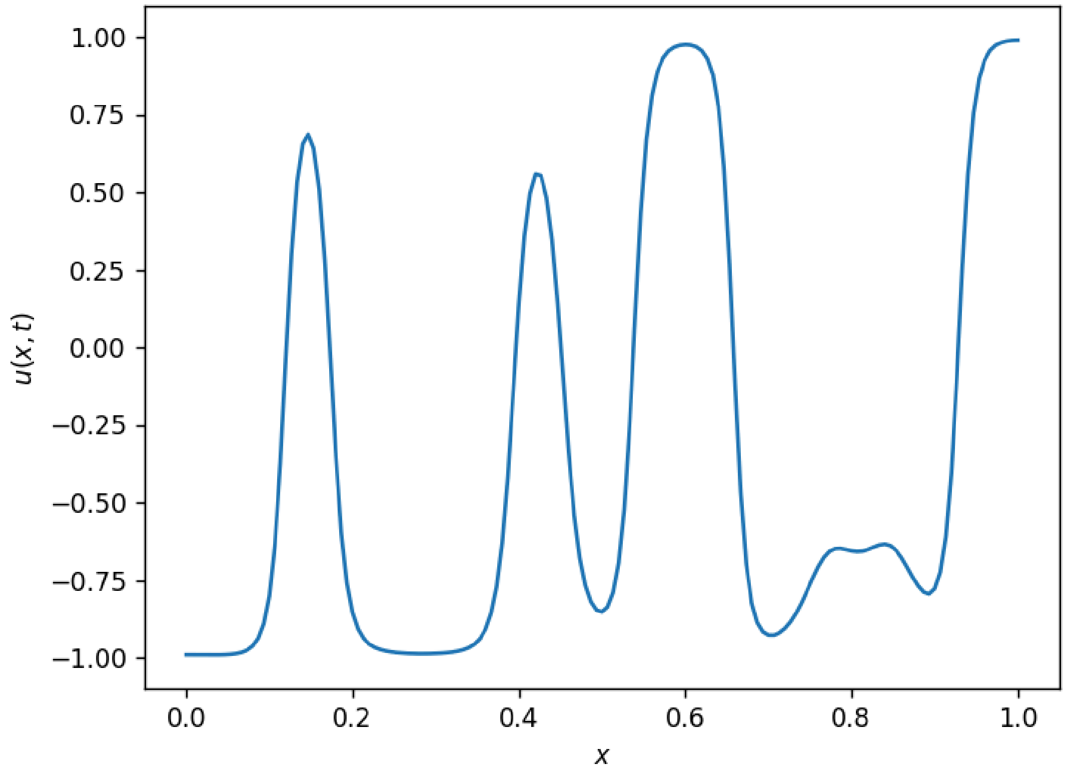
In [30]:
```
# Initial condition
def u_0(x, pars):
    return 1e-3 * (2 * np.random.random(grid.N+1) - 1)

# Source term
def q(u, x, pars):
    return u - u**3

# Derivative of source term wrt u (needed for Newton's method)
def q_prime(u, x, pars):
    return 1 - 3 * u**2
```

- Now the discretised equation is constructed using a Python class
  - The `DiffusionEquation` class generates the algebraic system for diffusion equations
  - All of the data structures are then stored in an object
- The algebraic system is then solved by calling the `solve` method

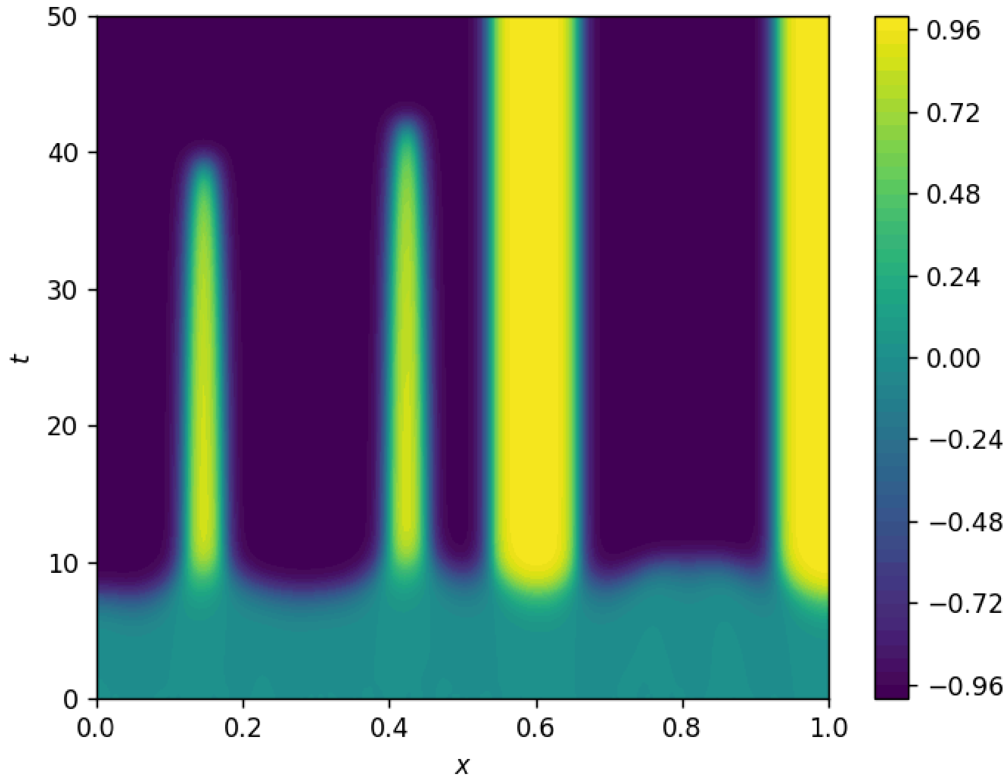
In [31]:
```
# Define the PDE and solve
pde = DiffusionEquation(grid, bc_left, bc_right, u_0, pars, q, q_prime)
u = pde.solve(method = "implicit_Euler")
```

- Now the solution is visualised

In [32]:
```
# Animate the solution
fig, ax = plt.subplots()
ax.set_ylim(-1.1, 1.1)
ax.set_xlabel('$x$')
ax.set_ylabel('$u(x,t)$')

line, = ax.plot(grid.x, u[:, 0])

def animate(i):
    line.set_data((grid.x, u[:, i]))
    return line

ani = animation.FuncAnimation(fig, animate, frames=grid.Nt, blit=True, inter
plt.show()
```



In [33]:
```
plt.figure()
plt.contourf(grid.x, grid.t, u.T, 50)
plt.xlabel('$x$')
plt.ylabel('$t$')
plt.colorbar()
plt.show()
```



50
40
30
20
10
0
t
0.0
0.2
0.4
0.6
0.8
1.0
x
0.96
0.72
0.48
0.24
0.00
−0.24
−0.48
−0.72
−0.96