# 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]:
```
# 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

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

Two Dirichlet boundary conditions will be imposed. These boundary conditions, and the initial condition, are given by

$$u(a,t) = \alpha = 1, \quad u(b,t) = \beta = 0, \quad u(x,0) = f(x) = 0.$$

We will take $a = 0$ and $b = 1$.

The explicit Euler method will be implemented using nested `for` loops. 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]:
```
"""
    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 $C = 1/2$:

In [4]:
```
# Numerical constant that determines stability
C = 0.5
```

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

In [5]:
```
dt_max = C * 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 [6]:
```
# 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')
```

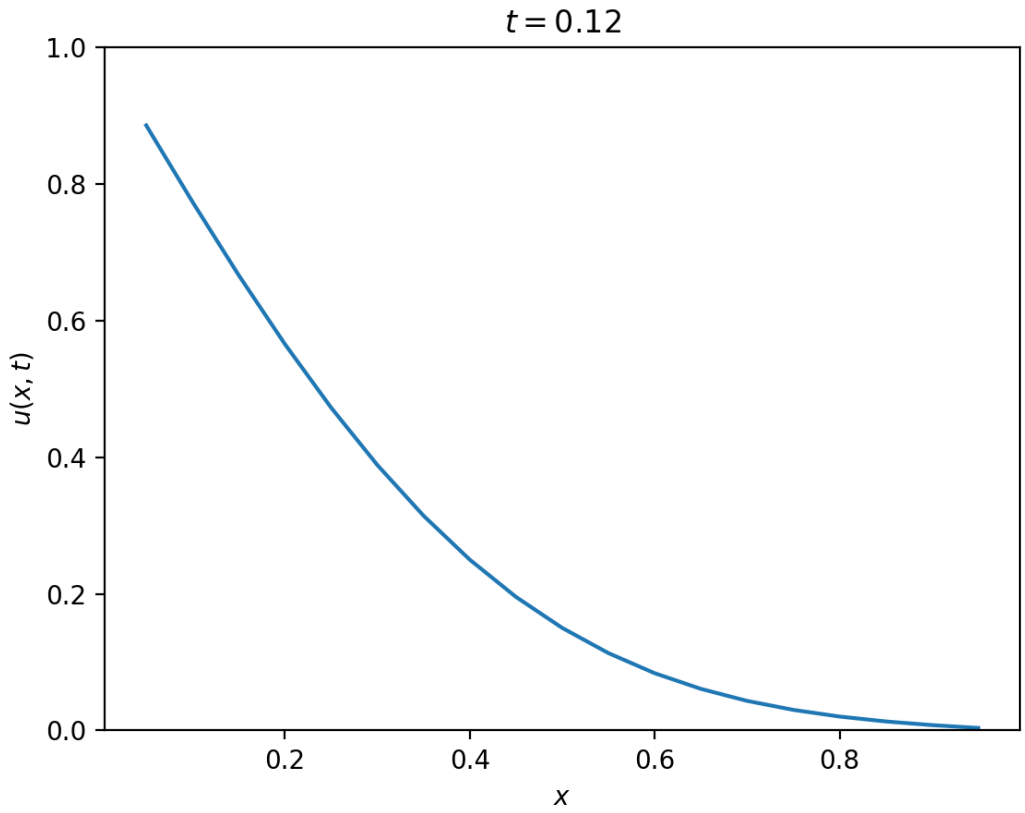
```
500 time steps will be needed
```

Now that all of the variables have been defined, we can start solving the PDE using the explicit Euler method

In [7]:
```
"""
    start 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)
        if 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])
```

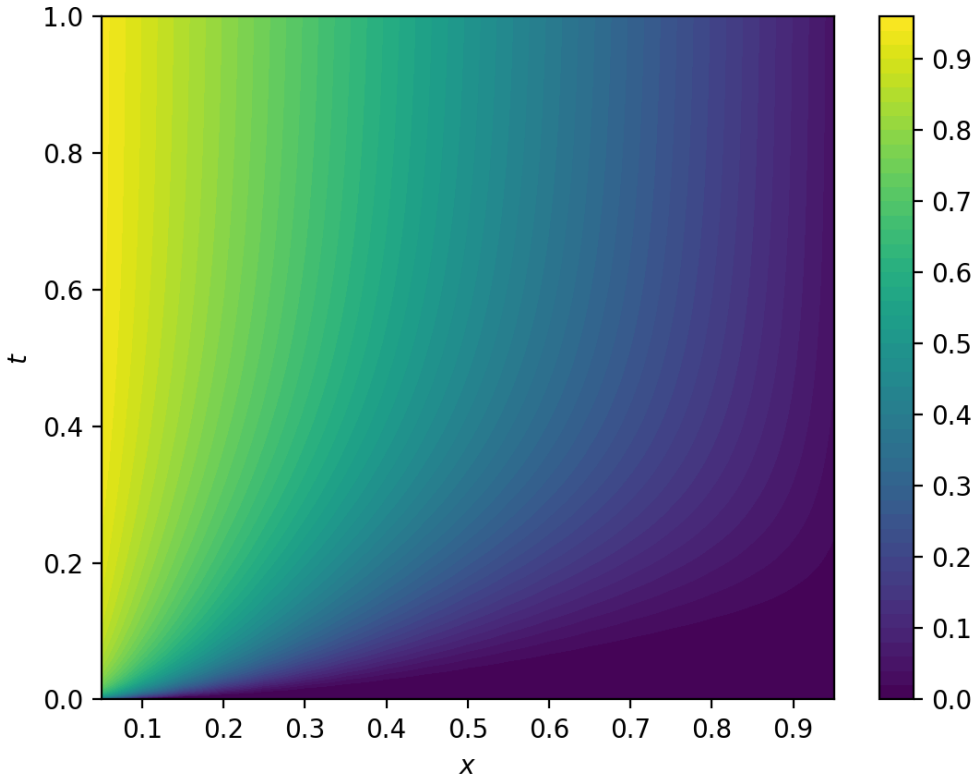
We can use the `animation` module from `matplotlib` to create animations of the solution. This will help with visualisation. We plot $u$ as a function of space ($x$) and animate over time $t$:

In [8]:
```
"""
    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 [9]:
```
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 [10]:
```
C = 1/2
dt_max = C * 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 [11]:
```
# time discretistation
dt = 5.0e-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')
```

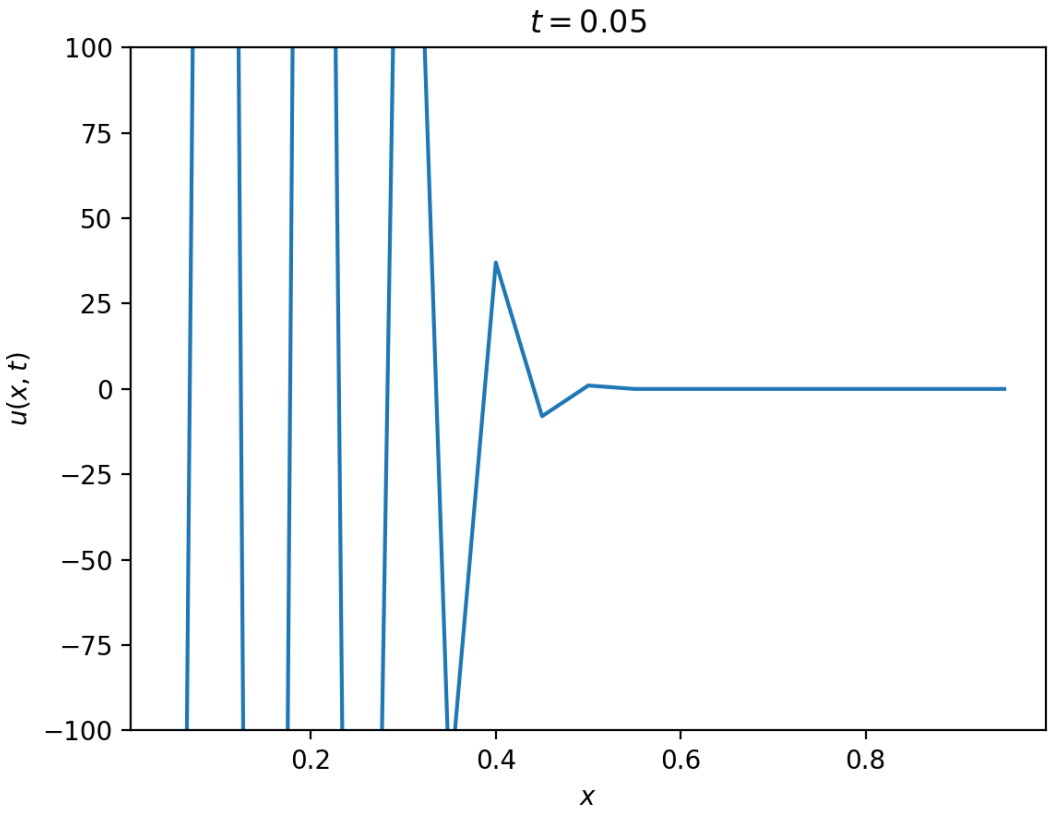
```
200 time steps will be needed
```

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

In [12]:
```
"""
    start 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)
        if 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])
```

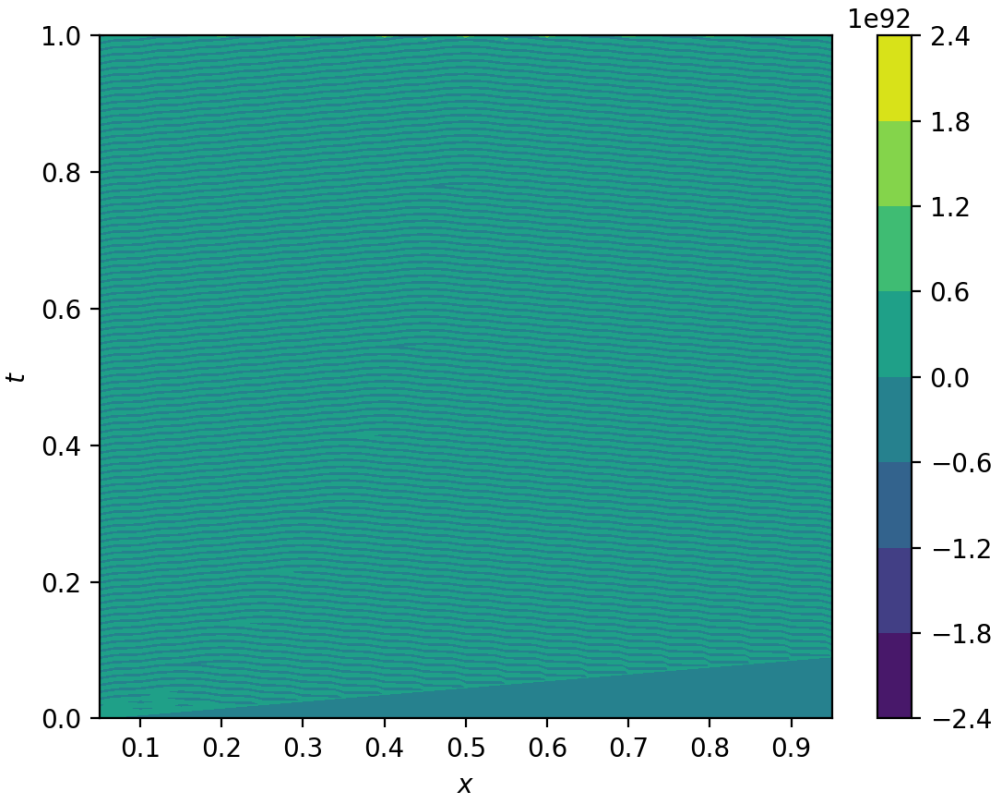
Animating the solution shows that it becomes oscillatory, and the oscillation amplitude grows exponentially in time

In [13]:
```
"""
    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 [14]:
```
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, the standard diffusion equation will be solved. However, now Neumann boundary conditions will be imposed.

The diffusion equation is

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

The boundary and initial conditions will be

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

As in the previous demo, we first define the parameters in the problem and then the spatial grid:

In [15]:
```
"""
    Set up the problem parameters
```

```
"""

# Diffusion coefficient
D = 2

# 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: 1 + np.cos(np.pi * x)
```

In [16]:
```
# 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 [17]:
```
C = 1/2
dt_max = C * 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 [18]:
```
# 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')
```

```
100 time steps will be needed
```

Now we generate the matrix $\mathbf{A}^{NN}$, the boundary condition vector $\boldsymbol{b}^{NN}$, and the identity matrix $\mathbf{I}$.

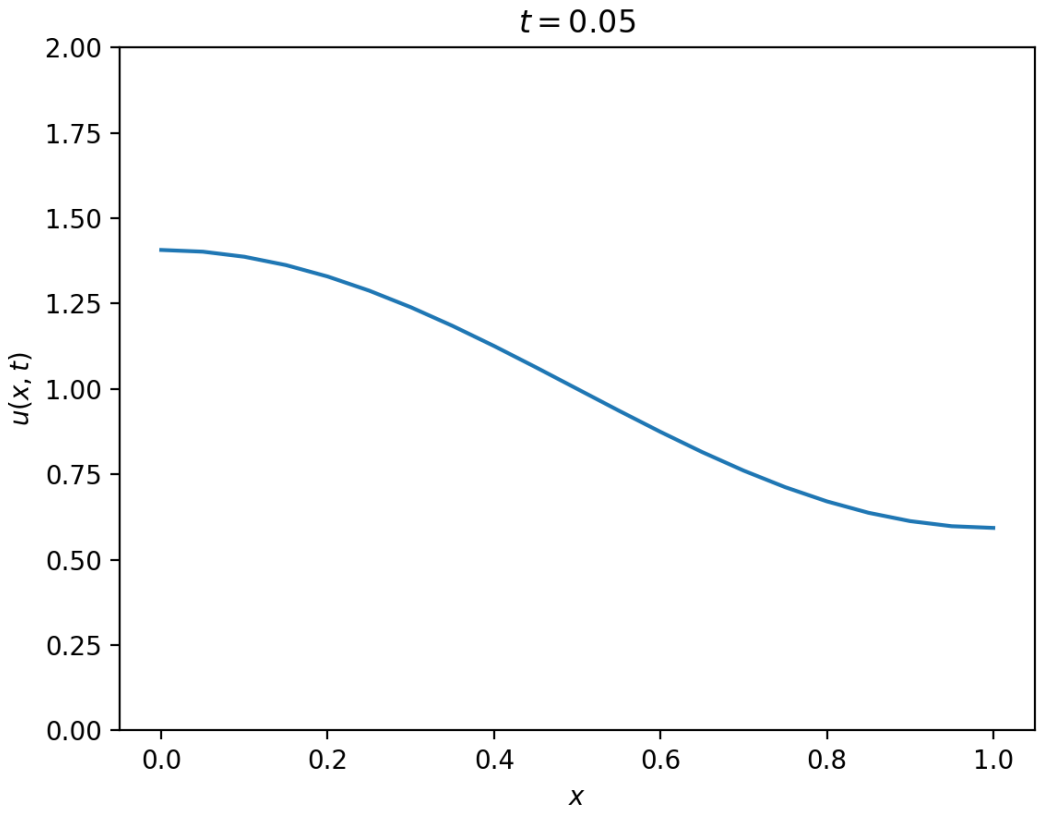
In [19]:
```
A_NN = (np.diag(np.r_[2, np.ones(N-1)], 1) +
        np.diag(-2 * np.ones(N+1)) +
        np.diag(np.r_[np.ones(N-1), 2], -1))
```

```
b_NN = np.zeros(N+1)

I = np.eye(N+1)
```

We now start time stepping using the implicit Euler method

In [20]:
```
"""
    start 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):

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

In [22]:
```
"""
    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

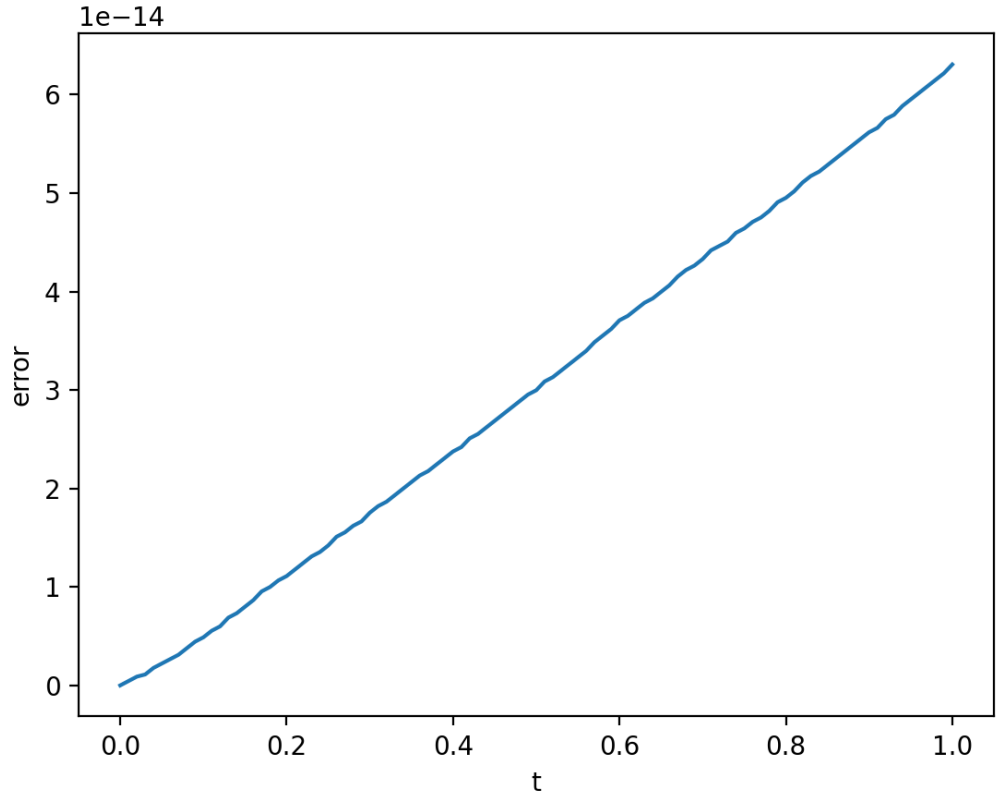
$$\int_0^1 u(x,t)\,dx = 1$$

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 [23]:
```
# 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 [24]:
```
# Plot the results
plt.plot(t, np.abs(mean_u - 1))
plt.xlabel('t')
plt.ylabel('error')
plt.show()
```




## Example 3 - the Allen-Cahn equation

In this demo, the Allen--Cahn equation will be numerically solved. The Allen-Cahn equation describes how a homogeneous mixture of two materials, such as oil and water, separates into regions that are pure in each phase.

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 thermal fluctuations in the system. The boundary conditions are

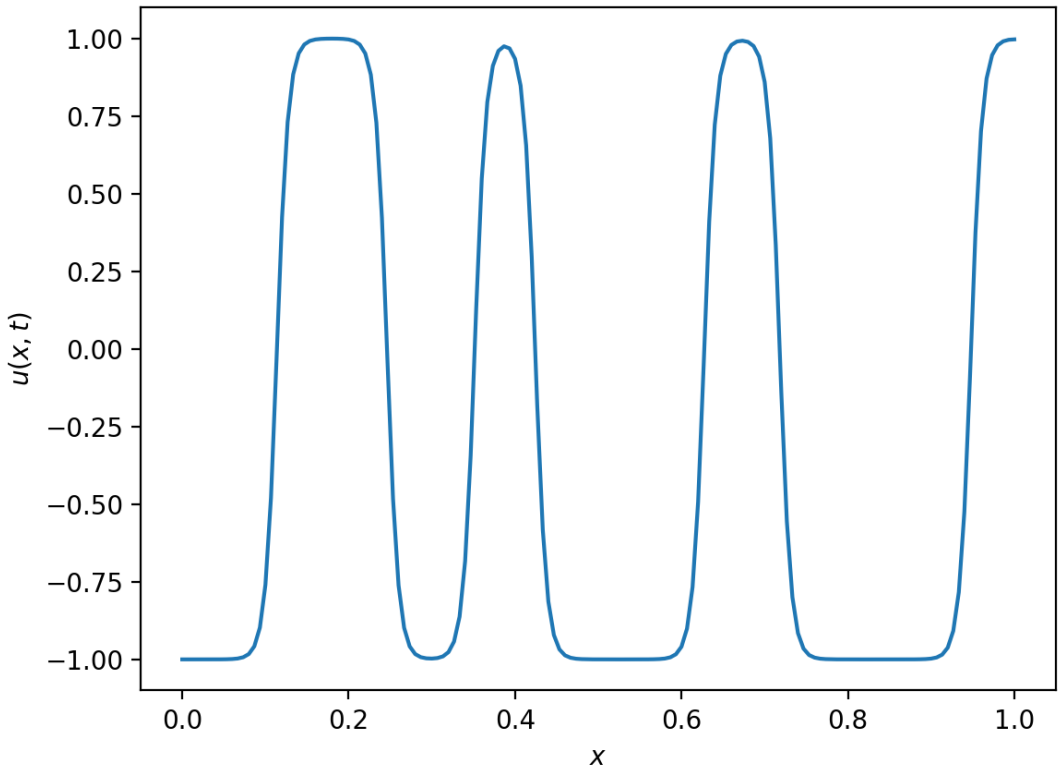
$$\left.\frac{\partial u}{\partial x}\right|_{x=0,1} = 0.$$

This PDE will be solved using the implicit Euler method - this requires solving a nonlinear algebraic system at each time step

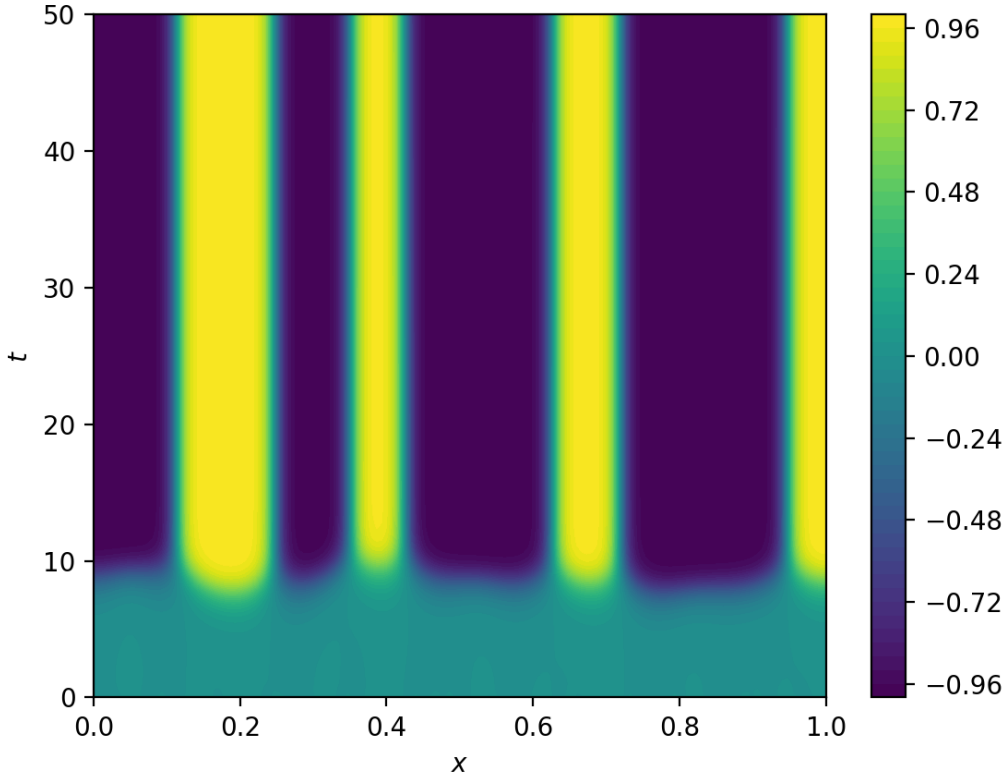
In [23]:
```
# Boundary conditions
bc_left = fd.BoundaryCondition('Neumann', lambda t: 0)
bc_right = fd.BoundaryCondition('Neumann', lambda t: 0)

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

# 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

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

In [24]:
```
# Define the PDE and solve
pde = DiffusionEquation(grid, bc_left, bc_right, u_0, pars, q)
u = pde.solve(method = "implicit_Euler", matrix_type='dense')
```

In [25]:
```
# 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 [28]:
```
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