SEMT30002 Scientific Computing and Optimisation

Week 4 Demos: First-order PDEs

Matthew Hennessy

The demos for this week will focus on solving first-order PDEs using upwind methods. We start by importing some basic packages

In [2]:
```python
import numpy as np

import matplotlib.pyplot as plt
import matplotlib.animation as animation

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

Example 1 - The linear advection equation

Here we solve the linear advection equation given by

$$\frac{\partial u}{\partial t} + v\frac{\partial u}{\partial x} = 0$$

on the domain $0 \leq x \leq 10$. We assume $v = 1$.

Since $v > 0$, we need to impose a boundary condition at $x = 0$. We assume that $p(0, t) = 1$.

The initial condition is taken to be a Gaussian: $u(x, 0) = \exp(-x^2)$.

We now define the problem parameters and the spatial discretisation

In [3]:
```python
# Speed
v = 1

# Spatial discretisation
a = 0
b = 10
N = 40
dx = (b - a) / N
x = np.linspace(a, b, N + 1)
```

Now we define the time discretisation. We do this by setting the CFL number to $C = 1/2$. We also use a fixed number of time steps N_t.

In [4]:
```
Nt = 100

C = 0.5
dt = C * dx / v
t = dt * np.arange(Nt + 1)

print(f'The size of the time step is dt = {dt:.2e}')
```

```
The size of the time step is dt = 1.25e-01
```

Now we preallocate the solution array and assign the initial condition and boundary condition at $x = 0$

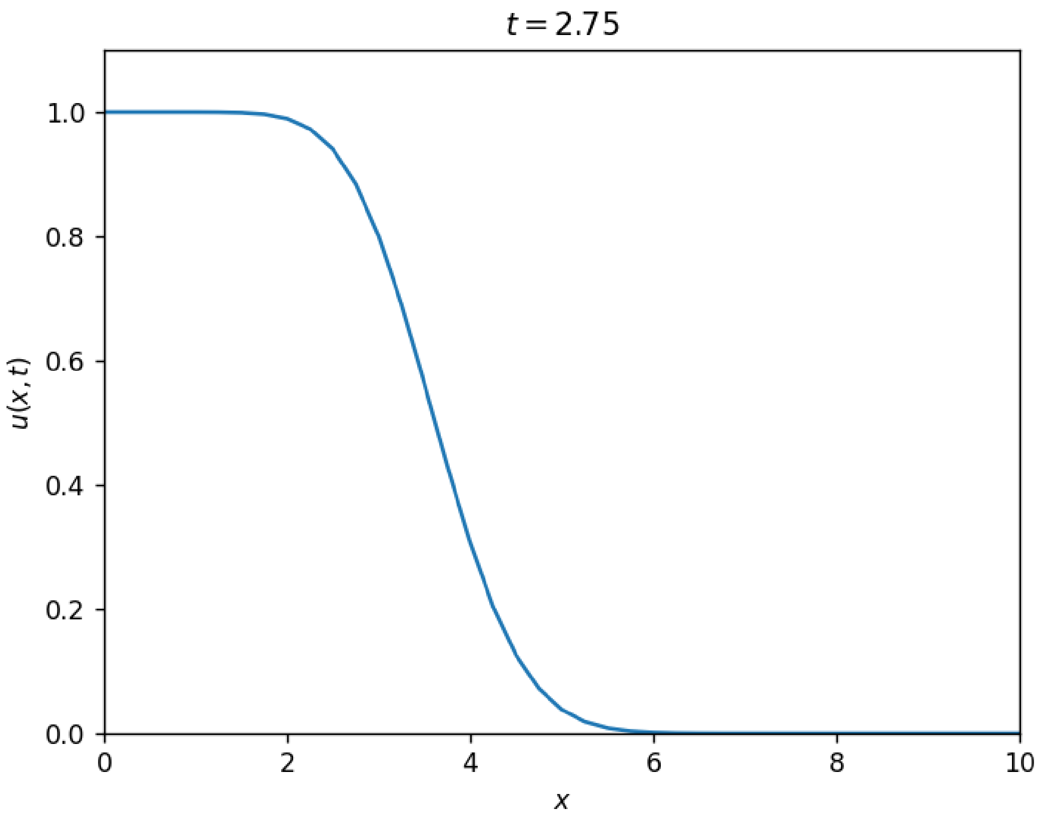
In [5]:
```
# Array pre-allocation
u = np.zeros((N + 1, Nt + 1))

# Impose the initial condition
u[:, 0] = np.exp(-x**2)

# Impose the boundary condition
u[0, :] = 1
```

All of the problem parameters have been defined so we proceed with applying the upwind scheme. Since $v > 0$, the upwind scheme is based on *backwards* differencing

In [6]:
```
"""
Upwind scheme based on backwards differencing
"""

# Loop over time
for n in range(Nt):

    # Loop over all grid points except for the left-most grid point (x[0])
    for i in range(1, N+1):
        u[i, n+1] = (1 - C) * u[i, n] + C * u[i-1, n]
```

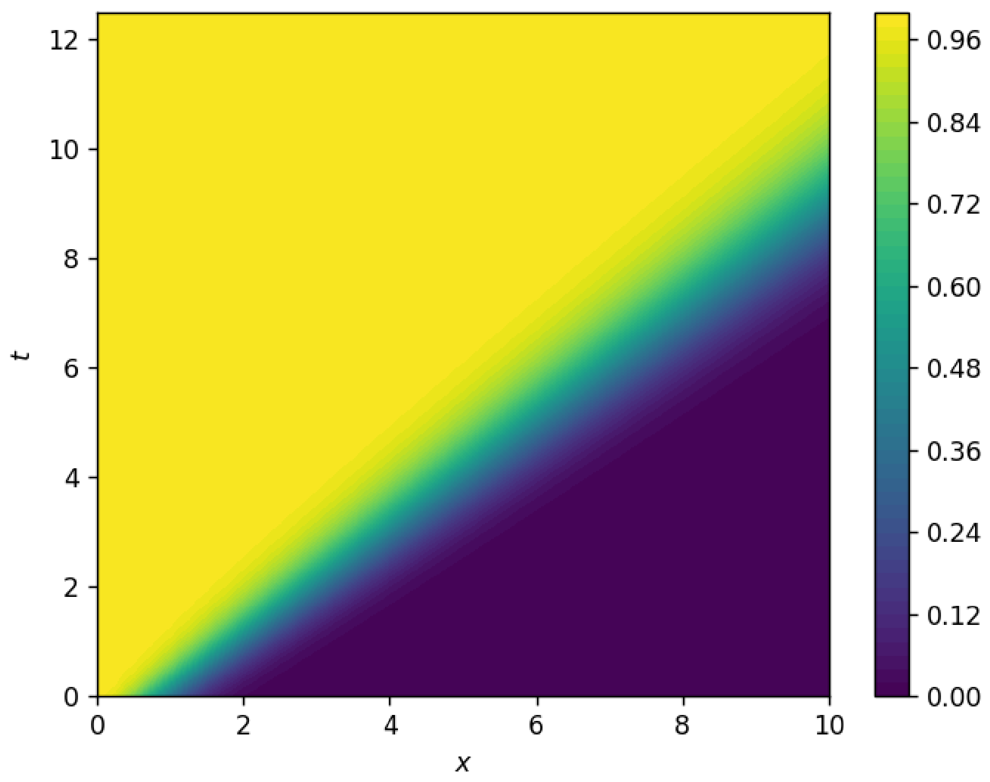
We now animate the solution using the following code:

In [7]:
```
"""
    animate the solution
"""
fig, ax = plt.subplots()
ax.set_xlim(0, 10)
ax.set_ylim(0, 1.1)
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=Nt, blit=True, interval=2
plt.show()
```



A filled contour plot in this case is particularly insightful. The widening of the region between $u = 0$ and $u = 1$ illustrates the artificial spreading of the solution due to numerical diffusion.

In [9]:
```
plt.contourf(x, t, u.T, 50)
plt.xlabel('$x$')
plt.ylabel('$t$')
plt.colorbar()
plt.show()
```



Example 2 - numerical diffusion

In this example, the same PDE as above will be solved, but a larger spatial domain will be used to showcase the artifical spreading of the solution caused by numerical diffusion

In [11]:
```
"""
Define problem parameters
"""

# speed
v = 1

# spatial discretisation
a = 0
b = 15
N = 200
dx = (b - a) / N
x = np.linspace(a, b, N + 1)

# time discretisation
Nt = 300
C = 0.5
dt = C * dx / v
```

```python
t = dt * np.arange(Nt + 1)
print(f'dt = {dt:.2e}')
```

```
dt = 3.75e-02
```

The solution array is pre-allocated and the initial/boundary condition imposed

In [12]:
```python
# Pre-allocation
u = np.zeros((N + 1, Nt + 1))

# Impose the initial condition
u[:, 0] = np.exp(-x**2)

# Impose the boundary condition
u[0, :] = 1
```

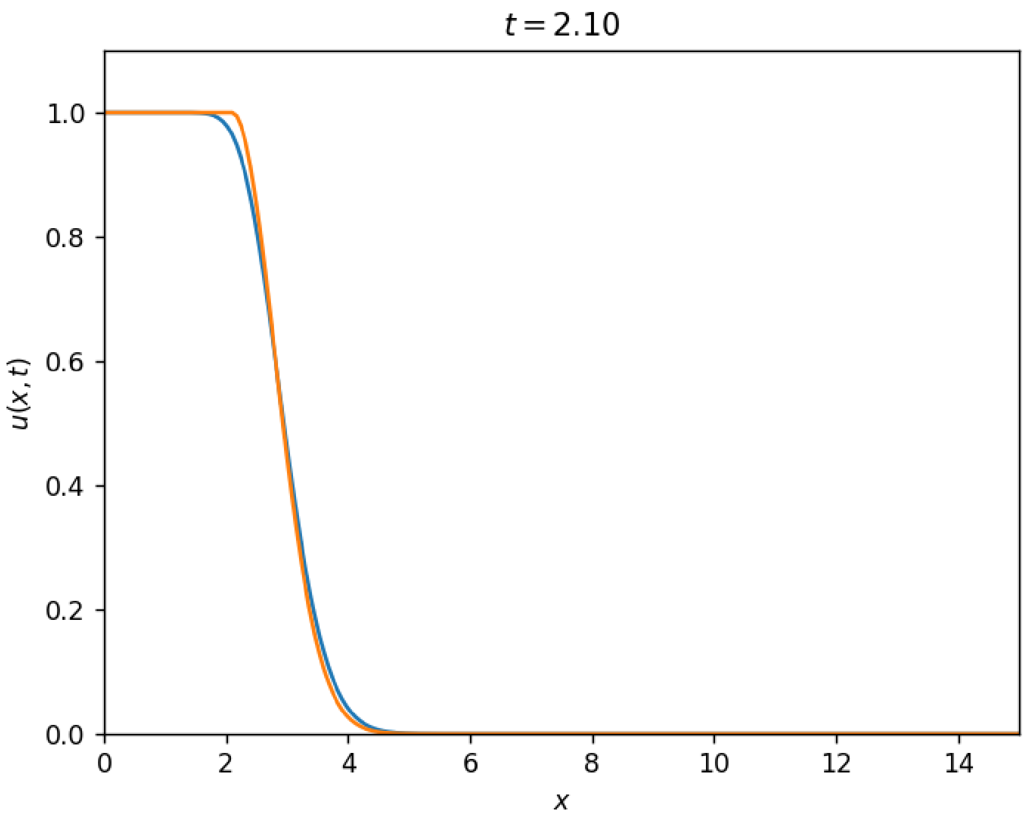
In [13]:
```python
"""
Solve using the unwind scheme
"""

# Loop over time steps
for n in range(Nt):

    # Loop over grid points
    for i in range(1, N+1):
        u[i, n+1] = (1 - C) * u[i, n] + C * u[i-1, n]
```

For this problem, there is an exact solution. We will define this as a Python function

In [14]:
```python
def u_exact(x, t):
    """
    The exact solution to the PDE
    """

    u_exact = np.exp(-(x - v * t)**2)

    # Set u = 1 if x < v * t
    ind = x - v * t < 0
    u_exact[ind] = 1

    # return the solution
    return u_exact
```

In [19]:
```python
"""
    animate the solution
"""
fig, ax = plt.subplots()
ax.set_xlim(a, b)
ax.set_ylim(0, 1.1)
ax.set_xlabel(f'$x$')
ax.set_ylabel(f'$u(x,t)$')

line_0, line_1 = ax.plot(x, u[:, 0], x, u_exact(x, t[0]))
```

```
def animate(i):
    line_0.set_data((x, u[:, i]))
    line_1.set_data((x, u_exact(x, t[i])))
    ax.set_title(f'$t = {t[i]:.2f}$')
#     ax.set_legend()
    return [line_0, line_1]

ani = animation.FuncAnimation(fig, animate, frames=Nt, blit=True, interval=2
plt.show()
```



Example 3 - The inviscid Burgers' equation

Now we solve one of the most famous nonlinear first-order PDEs called Burgers' equation:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0.$$

Notice that the speed of the solution depends on the solution itself.

For this PDE, if the initial condition $u(x,0) > 0$ for all x, then the solution $u(x,t) > 0$ for all time t. We will assume that $u(x,0) > 0$. This means the speed will always be positive, so a boundary condition at the left boundary will be required. We assume that $u(a,0) = 0$.

Burgers' equation is solved using a Python class that implements the upwind scheme presented in the videos

In [20]:
```
import finite_diff as fd
from pde_solvers import FirstOrderPDE
```

We now define the spatial domain:

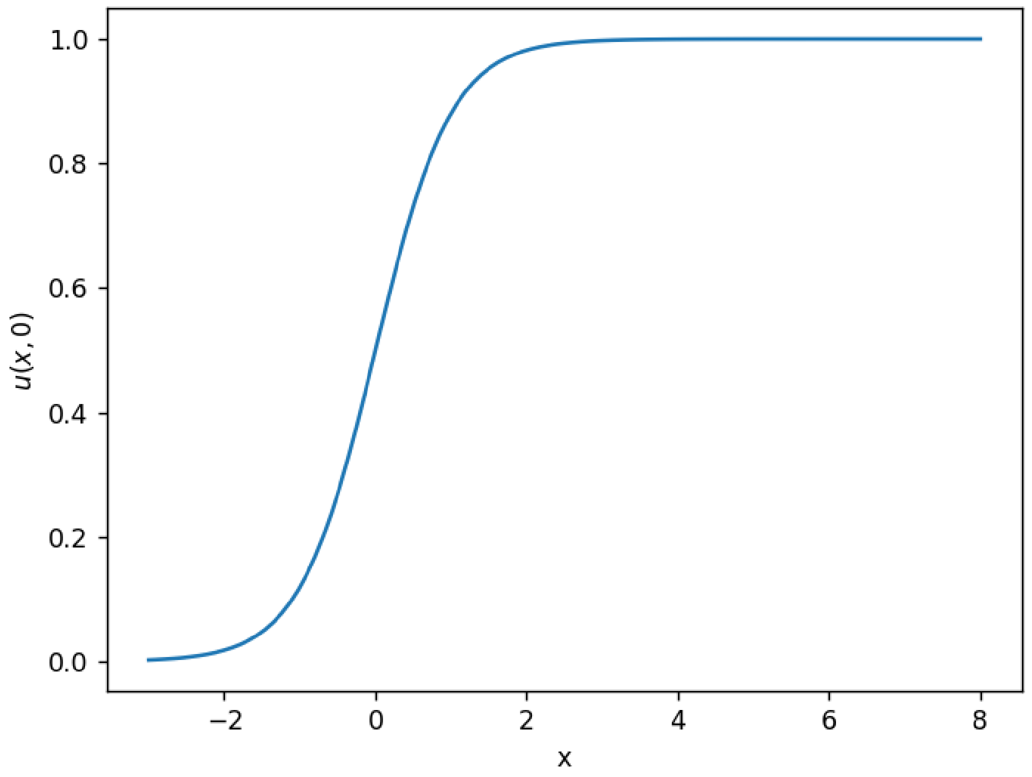
In [21]:
```
a = -3
b = 8
N = 100

dx = (b - a) / N
x = np.linspace(a, b, N+1)
```

Case 1 - increasing initial condition

Now we assume that the initial condition has a wave profile that increases as x increases. We define the initial condition as a Python function and then plot it

In [30]:
```
def u_0(x):
    return 1/2 * (np.tanh(x) + 1)
```

In [31]:
```
plt.plot(x, u_0(x))
plt.xlabel('x')
plt.ylabel('$u(x,0)$')
plt.show()
```



We now define a Python function to evaluate the wave speed, which is assumed to be a function of x, t, and u:

```
In [24]: def v(x, t, u):
             return u
```

We now use the CFL condition to find the time step. This is a bit tricky since the speed depends on the solution u, so the CFL number will be different at each grid point and at each point in time. However, for this problem, if the CFL condition is initially satisfied, then it will be satisfied for all time.

We set the initial CFL number to $C = 0.5$. Then, we find the maximum of the initial speed by evaluating the speed using the initial condition. From this, the time step can be obtained as $\Delta t = C \Delta x / \max\{v\}$.

```
In [34]: C = 0.5
         dt = C * dx / np.max(v(x, 0, u_0(x)))
         print(f'The time step dt = {dt:.2e}')
```

```
The time step dt = 5.50e-02
```

Now the grid and boundary condition is defined.

```
In [41]: grid = fd.SpaceTimeGrid(N = N, a = a, b = b, t_0 = 0, T = 10, dt = 5e-3)

         bcs = {
```

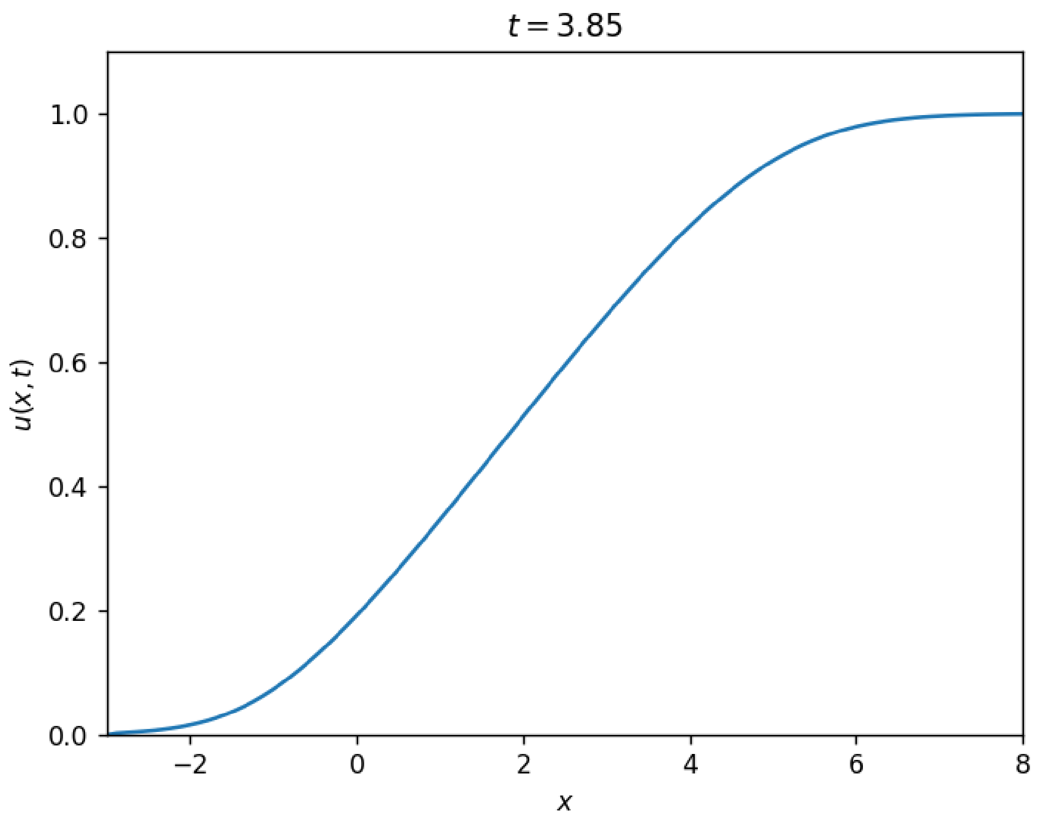
```
    'left': fd.BoundaryCondition('Dirichlet', lambda t: 0)
}
```

Now we define an object to store the PDE and then solve it using the upwind scheme:

In [36]:
```
pde = FirstOrderPDE(grid, bcs, u_0, v)
u = pde.solve()
```

Now the solution is animated

In [29]:
```
"""
    animate the solution
"""
fig, ax = plt.subplots()
ax.set_xlim(a, b)
ax.set_ylim(0, 1.1)
ax.set_xlabel(f'$x$')
ax.set_ylabel(f'$u(x,t)$')

line_0, = ax.plot(grid.x, u[:, 0])

def animate(i):
    line_0.set_data((grid.x, u[:, i]))
    ax.set_title(f'$t = {grid.t[i]:.2f}$')
    return line_0

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



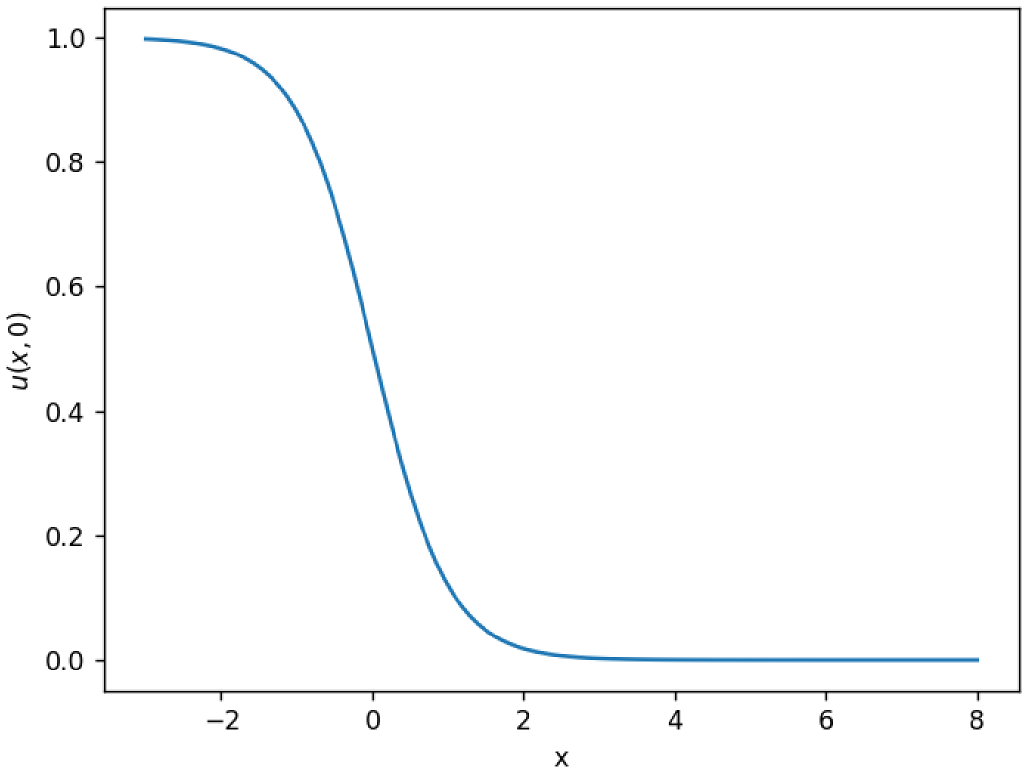
Case 2 - decreasing initial condition

Now we consider a more interesting case that occurs when the speed is positive, $v = u > 0$, but the initial condition decreases with x.

The initial condition is defined using a similar wave profile as above but now it decreases. We also plot it below

In [37]:
```python
def u_0(x):
    return 1/2 * (np.tanh(-x) + 1)
```

In [38]:
```python
plt.plot(x, u_0(x))
plt.xlabel('x')
plt.ylabel('$u(x,0)$')
plt.show()
```



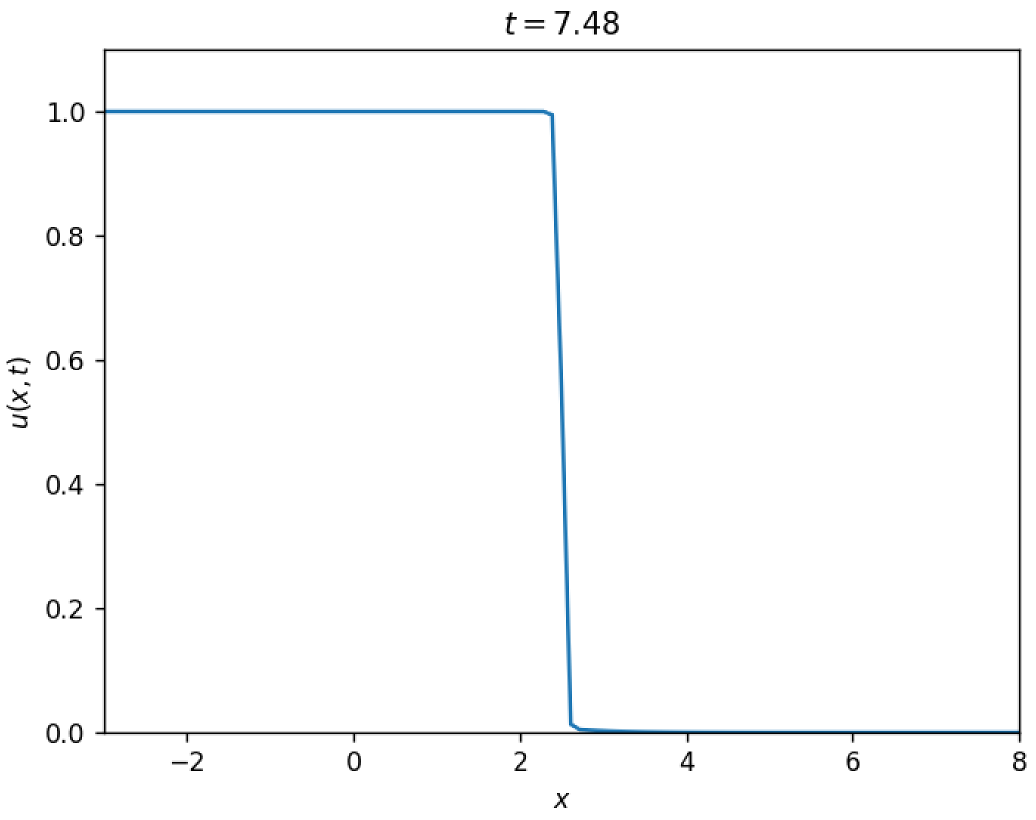
We also change the boundary condition so that it is consistent with the intial condition. We now set $u(a,t) = 1$.

In [42]:
```
bcs = {
    'left': fd.BoundaryCondition('Dirichlet', lambda t: 1)
}
```

Now we redefine the `pde` object using the new initial condition, solve it, and animate the solution

In [43]:
```
pde = FirstOrderPDE(grid, bcs, u_0, v)
u = pde.solve()
```

In [44]:
```
"""
    animate the solution
"""
fig, ax = plt.subplots()
ax.set_xlim(a, b)
ax.set_ylim(0, 1.1)
ax.set_xlabel(f'$x$')
ax.set_ylabel(f'$u(x,t)$')

line_0, = ax.plot(grid.x, u[:, 0])

def animate(i):
    line_0.set_data((grid.x, u[:, i]))
    ax.set_title(f'$t = {grid.t[i]:.2f}$')
    return line_0

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



The solution now develops a discontinuity. These discontinuities are called **shocks** and they can occur in first-order PDEs when the speed depends on the solution.