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 [1]:
```
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 $0 \le x \le 10$. We assume $v = 1$.

- Since $v > 0$, we need to impose a boundary condition at $x = 0$.
- We assume that $u(0, t) = 1$.
- The initial condition is taken to be a Gaussian: $u(x, 0) = \exp(-x^2)$.

The speed and spatial grid is first set up, since we use the value of Δx to compute Δt using the CFL condition

In [2]:
```
# 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 [3]:
```
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 [4]:
```
# Array pre-allocation (including the solution at the x = 0 boundary)
u = np.zeros((N + 1, Nt + 1))

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

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

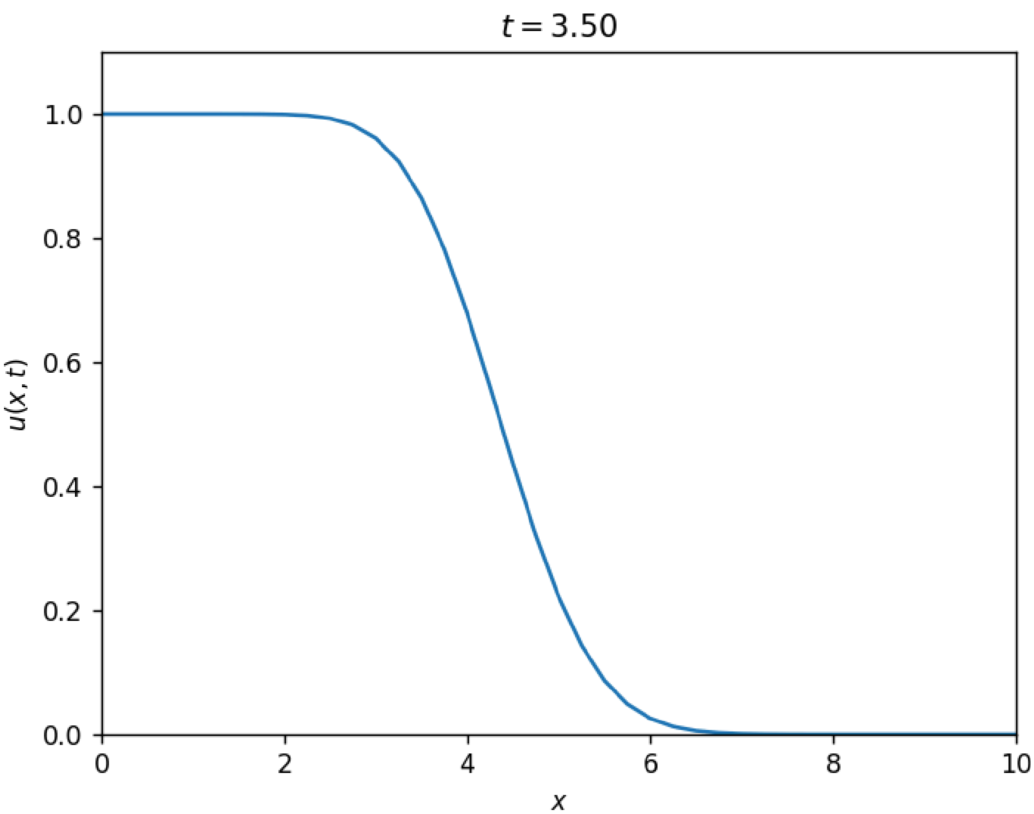
- All of the problem parameters have been defined so we proceed with applying the upwind scheme.
- Since $v > 0$, the upwind scheme is based on *backwards* differencing

In [5]:
```
"""
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

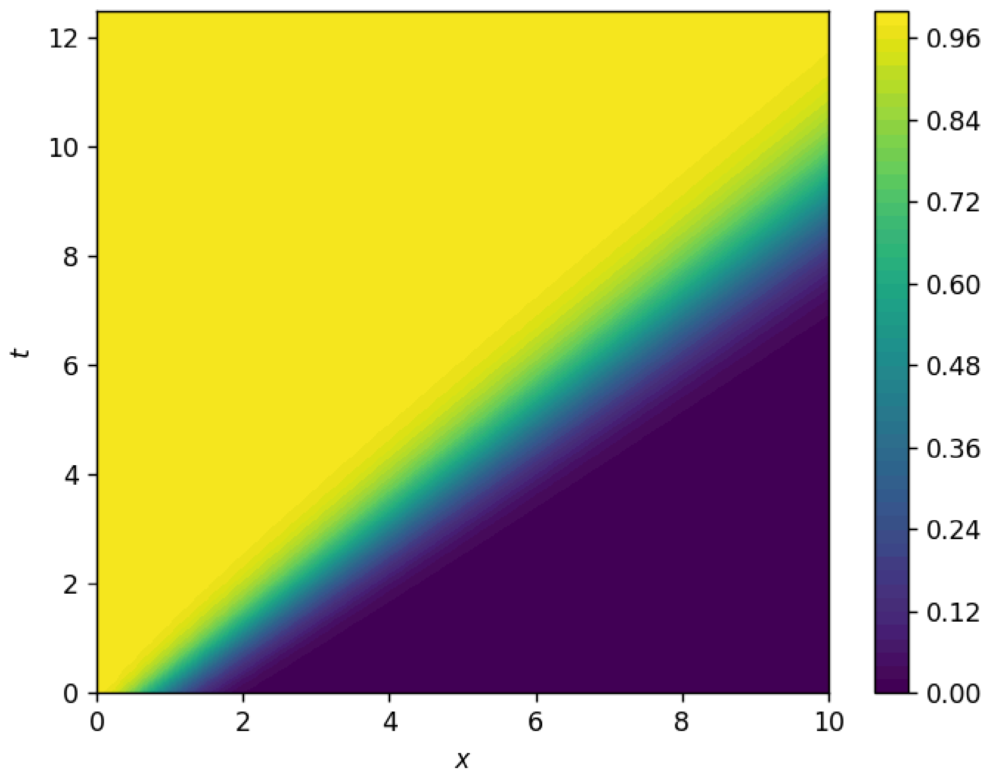
In [6]:
```
"""
    animate the solution
"""
fig, ax = plt.subplots()
ax.set_xlim(0, 10)
ax.set_ylim(0, 1.1)
ax.set_xlabel(f'$x$')
ax.set_ylabel(f'$u(x,t)$')
```

```
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 [7]:
```
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 [8]:

```
"""
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 [9]:
```python
# Array pre-allocation (including the solution at the x = 0 boundary)
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 [10]:
```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 given by

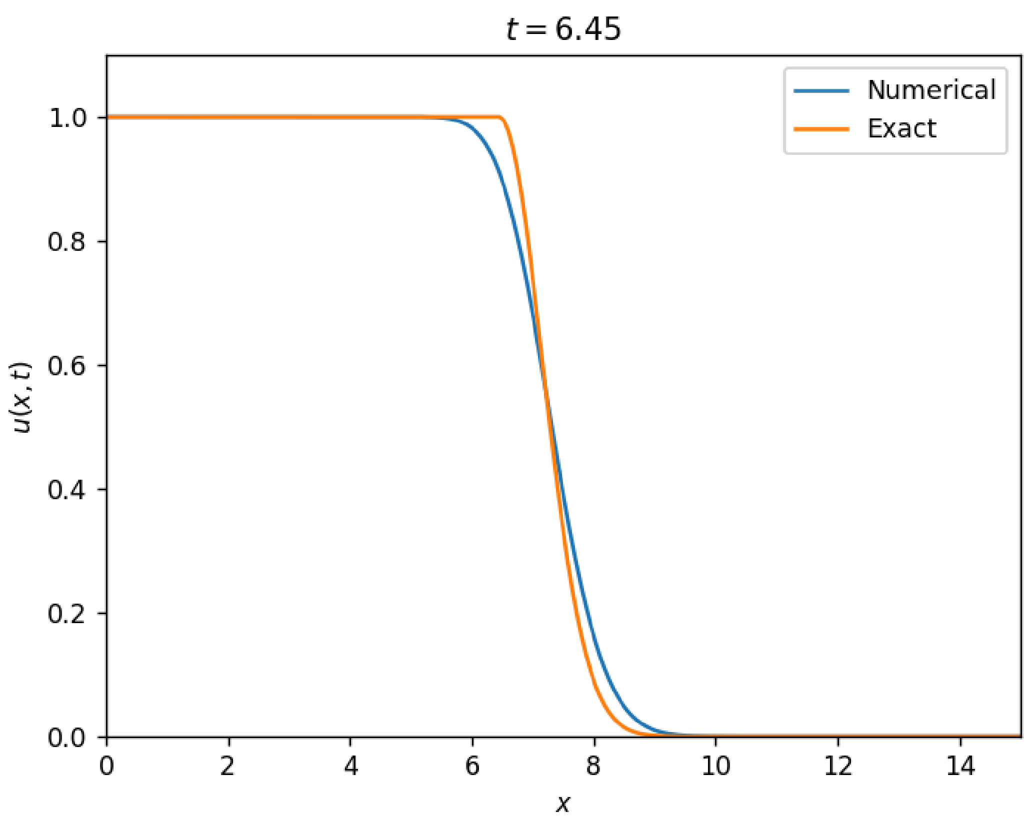
$$u(x,t) = \begin{cases} \exp(-(x-vt)^2), & x > vt, \\ 1, & x < vt \end{cases}$$

We will define this as a Python function:

In [11]:
```python
def u_exact(x, t):
    """
    The exact solution to the PDE
    """

    # Evaluate soln at all grid pts
    u = np.exp(-(x - v * t)**2)

    # Find indices for x where x < vt
    ind = x - v * t < 0

    # Set u = 1 where x < vt using the above indices
    u[ind] = 1

    # return the solution
    return u
```

The two solutions are plotted together as an animation:

In [12]:
```
"""
    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]))
plt.legend(("Numerical", "Exact"))

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:

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

where $a < x < b$.

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.
- A boundary condition at the left boundary will be required.
  - We assume that $u(a,0) = 0$.

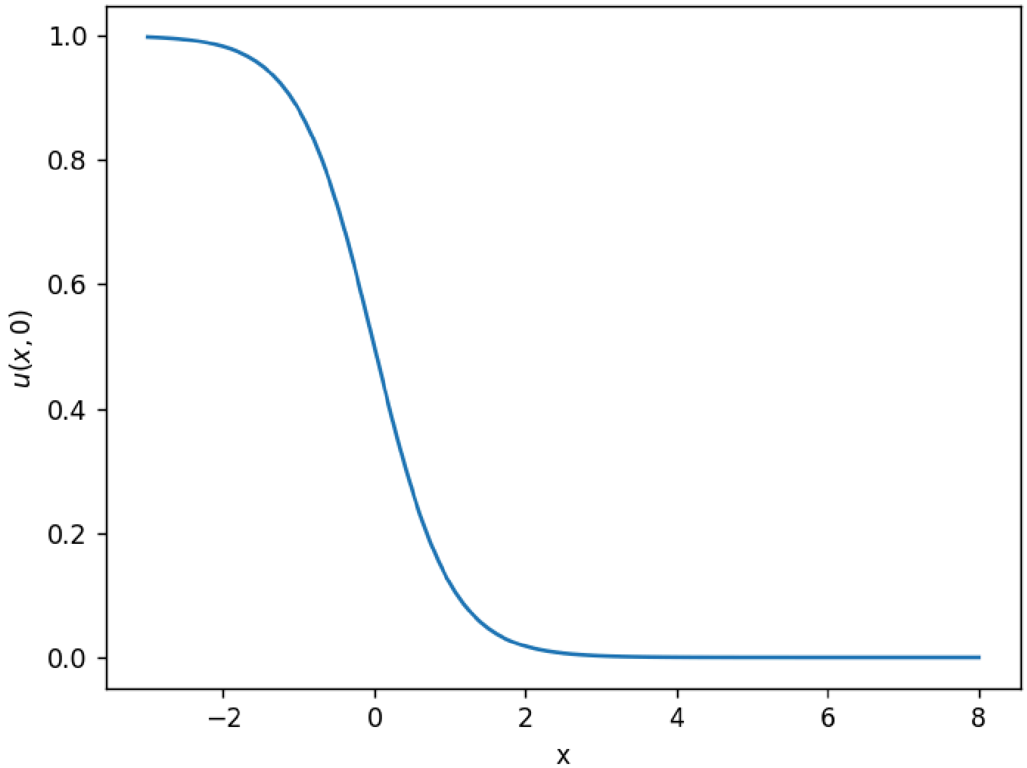
We now define the spatial domain and number of time steps

In [13]:
```
a = -3
b = 8
N = 100

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

Nt = 300
```

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

In [14]:
```
def u_0(x):
    """
    Initial condition function
    """
    return 1/2 * (np.tanh(-x) + 1)
```

In [15]:
```
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 a function of u only:

In [16]:
```
def v(u):
    """
    Computes the speed in the PDE
    """
    return u
```

- We now use the CFL condition to find the time step.
- Since the speed v depends on the solution u, the CFL number will be different at each grid point and at each point in time.
- How can we then ensure the CFL condition will be satisfied for all times?

For PDEs of the form

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

if the CFL condition is initially satisfied, then it will be satisfied for all time

- We set the initial CFL number to $C = 0.5$.
- 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 [17]:
```
C = 0.5
max_speed = np.max(v(u_0(x)))
dt = C * dx / max_speed
print(f'The time step dt = {dt:.2e}')
```

```
The time step dt = 5.51e-02
```

Now we pre-allocate the solution, assign the initial and boundary conditions

In [21]:
```
# Pre-allocation
u = np.zeros((N + 1, Nt + 1))

# initial condition
u[:, 0] = u_0(x)

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

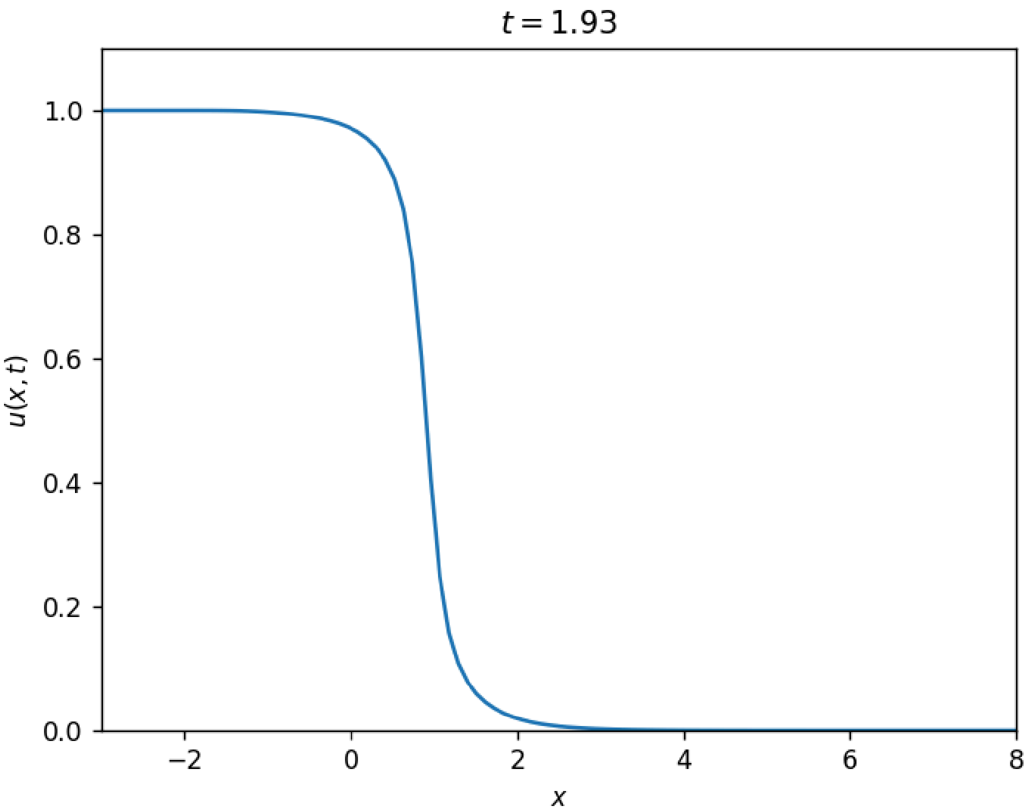
- The next step is to solve the problem using the upwinding scheme
- Backwards differencing is used because $v > 0$

In [22]:
```
"""
Solve using the upwind 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] = u[i, n] - dt * v(u[i,n]) * (u[i, n] - u[i-1, n]) / dx
```

Now the solution is animated

In [27]:
```
"""
    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(x, u[:, 0])

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

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



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