In [1]:

```python
from sympy import fourier_transform, exp, sqrt, pi, cos, simplify
from sympy.abc import x, k, t, symbols
from sympy import init_printing
init_printing(use_unicode=False, wrap_line=False)

import timeit
import numpy as np
import matplotlib.pyplot as plt
```

Diffusion part 2: matrix approach; implicit and explicit scheme

Repetition...

The one-dimensional diffusion equation is:

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

The first thing you should notice is that —unlike the previous two simple equations we have studied— this equation has a second-order derivative. We first need to learn what to do with it!

Discretizing $\frac{\partial^2 u}{\partial x^2}$

The second-order derivative can be represented geometrically as the line tangent to the curve given by the first derivative. We will discretize the second-order derivative with a Central Difference scheme: a combination of Forward Difference and Backward Difference of the first derivative. Consider the Taylor expansion of u_{i+1} and u_{i-1} around u_i:

$$u_{i+1} = u_i + \Delta x \frac{\partial u}{\partial x}\bigg|_i + \frac{\Delta x^2}{2} \frac{\partial^2 u}{\partial x^2}\bigg|_i + \frac{\Delta x^3}{3!} \frac{\partial^3 u}{\partial x^3}\bigg|_i + O(\Delta x^4)$$

$$u_{i-1} = u_i - \Delta x \frac{\partial u}{\partial x}\bigg|_i + \frac{\Delta x^2}{2} \frac{\partial^2 u}{\partial x^2}\bigg|_i - \frac{\Delta x^3}{3!} \frac{\partial^3 u}{\partial x^3}\bigg|_i + O(\Delta x^4)$$

If we add these two expansions, you can see that the odd-numbered derivative terms will cancel each other out. If we neglect any terms of $O(\Delta x^4)$ or higher (and really, those are very small), then we can rearrange the sum of these two expansions to solve for our second-derivative.

$$u_{i+1} + u_{i-1} = 2u_i + \Delta x^2 \frac{\partial^2 u}{\partial x^2}\bigg|_i + O(\Delta x^4)$$

Then rearrange to solve for $\frac{\partial^2 u}{\partial x^2}\bigg|_i$ and the result is:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} + O(\Delta x^4)$$

Back to discretizing both $\frac{\partial u}{\partial t}$ and $\frac{\partial^2 u}{\partial x^2}$

We can now write the discretized version of the diffusion equation in 1D:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \nu \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}$$

As before, we notice that once we have an initial condition, the only unknown is u_i^{n+1}, so we re-arrange the equation solving for our unknown:

$$u_i^{n+1} = u_i^n + \underbrace{\frac{\nu \Delta t}{\Delta x^2}}_{\beta} (u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

$$u_i^{n+1} = \beta u_{i-1}^n + u_i^n(1 - 2\beta) + \beta u_{i+1}^n$$

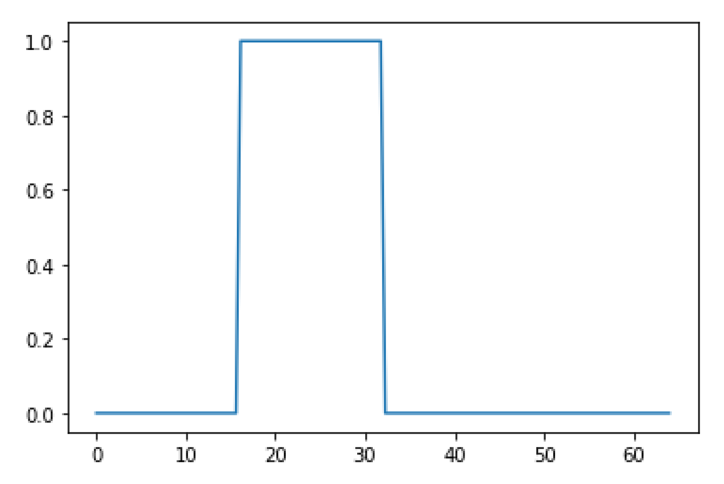
The above discrete equation allows us to write a program to advance a solution in time. But we need an initial condition. Let's continue using our favorite: the hat function. So, at $t = 0$, $u = 1$ in the interval $0.25 \le x \le 0.5$ and $u = 0$ everywhere else. We are ready to number-crunch!

In [5]:

```python
# %matplotlib inline

nx = 128
domain_length = 64
dx = domain_length / (nx-1)
xspace = np.linspace(0, domain_length, nx)

nt = 200                  # the number of timesteps we want to calculate
nu = 5                    # the value of viscosity
sigma = .2                # sigma is a parameter, we'll learn more about it later
dt = sigma * dx**2 / nu # dt is defined using sigma ... more later!

u_IC = 0*np.ones(nx)             # numpy function ones()
u_IC[int(nx/4):int(nx/2)] = 1    # setting u = 1 between 0.25 and 0.5 as per our I.C.s

plt.plot(xspace, u_IC)
```

Out[5]:

```
[<matplotlib.lines.Line2D at 0x7f5158893c40>]
```



Matrix approach

Explicit, central FD - matrix approach

Notice, that the scheme

$$u_i^{n+1} = \beta u_{i-1}^n + u_i^n(1 - 2\beta) + \beta u_{i+1}^n$$

where $\beta = \frac{\nu \Delta t}{\Delta x^2}$ can be formulated as:

$$\boldsymbol{u}^{n+1} = \mathbb{A} \boldsymbol{u}^n$$

Observe, that $\mathbb{A}$ have a tridiagonal structure:

$$\begin{bmatrix}
A_{0,0} & A_{0,1} & A_{0,2} & & & & & \\
\beta & 1-2\beta & \beta & & & & & \\
0 & \beta & 1-2\beta & \beta & & & & \\
0 & 0 & \beta & 1-2\beta & \beta & & & \\
\dots & & & & & & & \\
\dots & & & & & & & \\
\dots & & & & & & & \\
0 & 0 & 0 & 0 & 0 & \beta & 1-2\beta & \beta \\
0 & 0 & 0 & 0 & 0 & A_{n,n-2} & A_{n,n-1} & A_{n,n}
\end{bmatrix}$$

Hint:

Fill the corners of the matrix using asymmetric stencils:

- forward FD for $A_{0,0}$, $A_{0,1}$ and $A_{0,2}$

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \nu \frac{u_i^n - 2u_{i+1}^n + u_{i+2}^n}{\Delta x^2}$$

$$u_i^{n+1} = (1 + \beta)u_i^n - 2\beta u_{i+1}^n + \beta u_{i+2}^n$$

- backward FD for $A_{n,n}$, $A_{n,n-1}$ and $A_{n,n-2}$

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \nu \frac{u_i^n - 2u_{i-1}^n + u_{i-2}^n}{\Delta x^2}$$

$$u_i^{n+1} = (1 + \beta)u_i^n - 2\beta u_{i-1}^n + \beta u_{i-2}^n$$

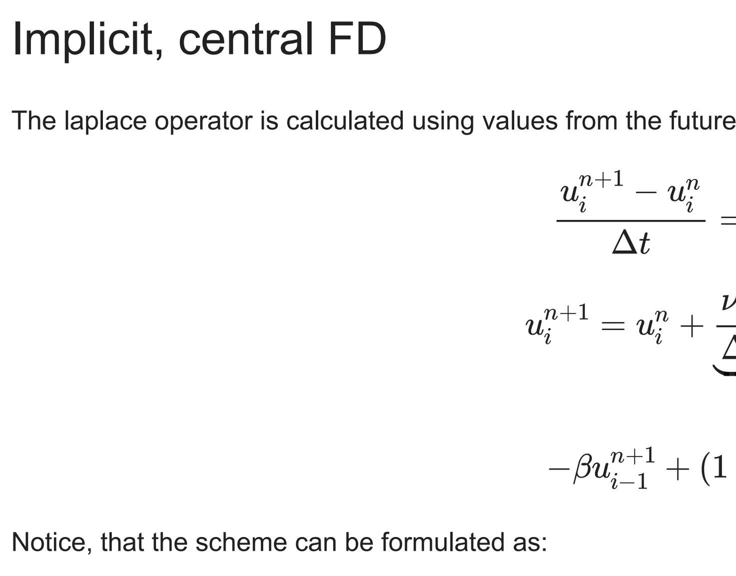
In [3]:

```python
# explicit central FD
import numpy as np
np.set_printoptions(precision=3, suppress=True)

un_ecfd = u_IC.copy()

A = np.zeros((nx, nx))

Beta_FD = dt * nu / (dx**2)
last_index_in_matrix = nx -1

# the BC  - use one sided FD
A[0, 0] = 1+Beta_FD     # forward FD
A[0, 1] = -2*Beta_FD     # forward FD
A[0, 2] = Beta_FD      # forward FD
A[last_index_in_matrix, last_index_in_matrix-2] = Beta_FD   # backward FD
A[last_index_in_matrix, last_index_in_matrix-1] = -2*Beta_FD  # backward FD
A[last_index_in_matrix, last_index_in_matrix] =  1+Beta_FD   # backward FD
for i in range(1, last_index_in_matrix):
    A[i, i-1] = Beta_FD          # left of the diagonal
    A[i, i] = 1 - 2*Beta_FD      # the diagonal
    A[i, i+1] = Beta_FD          # right of the diagonal


w, v = np.linalg.eig(A)  # calculate the eigenvalues and eigenvectors
# plt.scatter(np.arange(0, len(w)), abs(w))  # plot the length of the eigenvalues
# # print(f"determinant A_inv: {np.linalg.det(A_inv)}")
print(f"max(abs(w)): {max(abs(w)):.16f}")

for n in range(nt):  #loop for values of n from 0 to nt, so it will run nt times
    un_ecfd = A@un_ecfd
    # un_ecfd = np.dot(A,un_ecfd) # alternative way of doing the same

plt.plot(xspace, un_ecfd)
```

```
max(abs(w)): 1.0000000018722908
```

Out[3]:

```
[<matplotlib.lines.Line2D at 0x7f51589497c0>]
```



Implicit, central FD

The laplace operator is calculated using values from the future (u^{n+1}).

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \nu \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2}$$

$$u_i^{n+1} = u_i^n + \underbrace{\frac{\nu \Delta t}{\Delta x^2}}_{\beta} (u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1})$$

$$-\beta u_{i-1}^{n+1} + (1 + 2\beta)u_i^{n+1} - \beta u_{i+1}^{n+1} = u_i^n$$

Notice, that the scheme can be formulated as:

$$\mathbb{A} \boldsymbol{u}^{n+1} = \boldsymbol{u}^n$$

Observe, that $\mathbb{A}$ have a tridiagonal structure:

$$\begin{bmatrix}
A_{0,0} & A_{0,1} & A_{0,2} & & & & & \\
-\beta & 1+2\beta & -\beta & & & & & \\
0 & -\beta & 1+2\beta & -\beta & & & & \\
0 & 0 & -\beta & 1+2\beta & -\beta & & & \\
\dots & & & & & & & \\
\dots & & & & & & & \\
\dots & & & & & & & \\
0 & 0 & 0 & 0 & 0 & -\beta & 1+2\beta & -\beta \\
0 & 0 & 0 & 0 & 0 & A_{n,n-2} & A_{n,n-1} & A_{n,n}
\end{bmatrix}$$

Hint:

Fill the corners of the matrix using asymmetric stencils:

- forward FD for $A_{0,0}$, $A_{0,1}$ and $A_{0,2}$

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \nu \frac{u_i^{n+1} - 2u_{i+1}^{n+1} + u_{i+2}^{n+1}}{\Delta x^2}$$

$$u_i^{n+1}(1 - \nu \underbrace{\frac{\Delta}{\Delta x^2}}_{\beta}) + 2\beta u_{i+1}^{n+1} - \beta u_{i+2}^{n+1} = u_i^n$$

- backward FD for $A_{n,n}$, $A_{n,n-1}$ and $A_{n,n-2}$

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \nu \frac{u_i^{n+1} - 2u_{i-1}^{n+1} + u_{i-2}^{n+1}}{\Delta x^2}$$

$$u_i^{n+1}(1 - \nu \underbrace{\frac{\Delta}{\Delta x^2}}_{\beta}) + 2\beta u_{i-1}^{n+1} - \beta u_{i-2}^{n+1} = u_i^n$$

In [4]:

```python
# implicit central FD
import numpy as np
np.set_printoptions(precision=3, suppress=True)

un_icfd = u_IC.copy()
A = np.zeros((nx, nx))

Beta_FD = dt * nu / (dx**2)
# nt += 100
last_index_in_matrix = nx -1

# the BC  - use one sided FD
A[0, 0] = 1-Beta_FD     # forward FD
A[0, 1] = 2*Beta_FD     # forward FD
A[0, 2] = -Beta_FD      # forward FD
A[last_index_in_matrix, last_index_in_matrix-2] = -Beta_FD   # backward FD
A[last_index_in_matrix, last_index_in_matrix-1] = 2*Beta_FD  # backward FD
A[last_index_in_matrix, last_index_in_matrix] =  1-Beta_FD   # backward FD

for i in range(1, last_index_in_matrix):
    A[i, i-1] = -Beta_FD        # left of the diagonal
    A[i, i] = 1 + 2*Beta_FD     # the diagonal
    A[i, i+1] = -Beta_FD         # right of the diagonal


A_inv = np.linalg.inv(A)

w, v = np.linalg.eig(A_inv)
# plt.scatter(np.arange(0, len(w)), abs(w)) # plot the eigenvalues
# # print(f"determinant A_inv: {np.linalg.det(A_inv)}")
print(f"max(abs(w)): {max(abs(w)):.16f}")

for n in range(nt):  #loop for values of n from 0 to nt, so it will run nt times
    un_icfd = A_inv@un_icfd
    # alternative way of doing the same:
    #     un_icfd = np.dot(A_inv,un_icfd)
    #     b = un_icfd.copy()
    #     un_icfd = np.linalg.solve(A, b) # u(t+1)

plt.plot(xspace, un_icfd)
```

```
max(abs(w)): 1.0000000000000238
```

Out[4]:

```
[<matplotlib.lines.Line2D at 0x7f51588ba6d0>]
```