Numerical Methods in Computational Fluid Dynamics (CFD)

Let's consider a simple **one-dimensional heat conduction problem** governed by the **heat equation**. This problem is well-suited for demonstrating the three numerical methods because an analytical solution exists and we can focus on the steps for each method.

Problem Setup: 1D Heat Equation

The **one-dimensional heat equation** in its simplest form is given by:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, \quad 0 \leq x \leq L,\ t > 0$$

where:

- $u(x,t)$ is the temperature distribution over time t and space x,
- α is the thermal diffusivity constant.

We will consider a **rod** of length L with boundary conditions $u(0,t) = u(L,t) = 0$ (fixed temperature at the ends of the rod) and an initial condition $u(x,0) = \sin(\pi x)$.

The **analytical solution** for this problem is:

$$u(x,t) = e^{-\alpha \pi^2 t} \sin(\pi x)$$

Now, let's apply the **finite difference**, **finite element**, and **spectral methods** to approximate this solution. We'll assume that $L = 1$, $\alpha = 0.01$, and the time step is small.

Finite Difference Method

The finite difference method (FDM) is one of the simplest and oldest numerical techniques used in CFD. It works by approximating derivatives in partial differential equations (PDEs) using differences between function values at discrete points in space and time.

Discretization:

- **Spatial discretization**: Divide the spatial domain $x \in [0, L]$ into N_x equally spaced grid points with spacing $\Delta x = \frac{L}{N-1}$.
- **Time discretization**: Divide the time domain t into steps of size Δt.

For simplicity, we will use the **explicit scheme** to approximate the time derivative and the **central difference scheme** for the second-order spatial derivative.

Central Difference Approximations:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{(\Delta x)^2}$$

$$\frac{\partial u}{\partial t} \approx \frac{u_i^{n+1} - u_i^n}{\Delta t}$$

Finite Difference Equation:

$$u_i^{n+1} = u_i^n + \frac{\alpha \Delta t}{(\Delta x)^2} \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n \right)$$

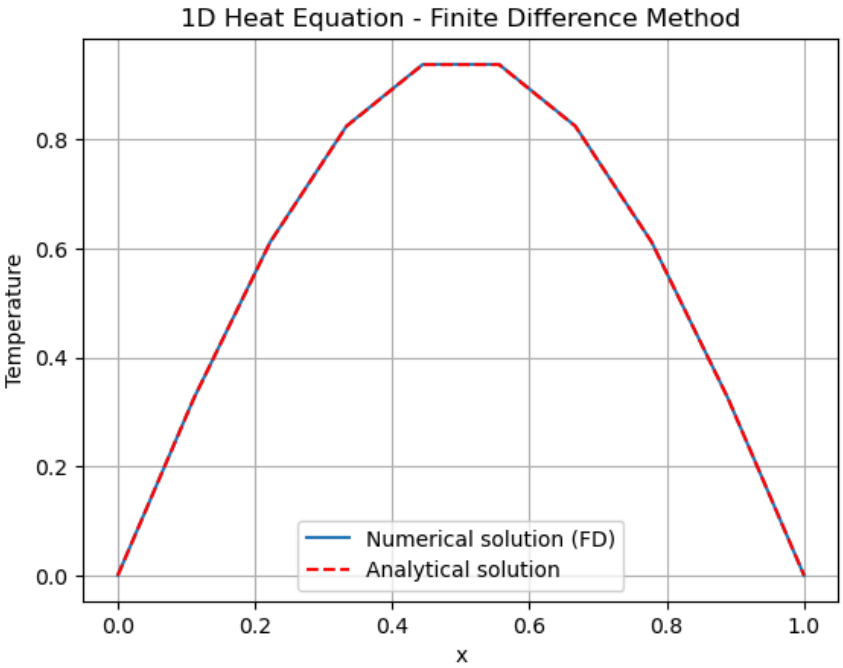
where u_i^n is the temperature at position i and time step n.

The following steps are:

1. Initialize $u(x,0) = \sin(\pi x)$.

2. Update the temperature at each grid point using the finite difference equation.
3. Repeat for each time step.

In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Parameters for the problem
L = 1.0         # Length of the rod
T = 0.5         # Total time
alpha = 0.01    # Thermal diffusivity
Nx = 10         # Number of spatial points
Nt = 500        # Number of time steps
dx = L / (Nx - 1)  # Spatial step size
dt = T / Nt     # Time step size
r = alpha * dt / dx**2  # Stability condition parameter

# Create grid
x = np.linspace(0, L, Nx)
u = np.sin(np.pi * x)  # Initial condition: u(x,0) = sin(pi*x)
u_new = np.zeros_like(u)  # Array to store new temperature values

# Boundary conditions
u[0] = 0  # u(0, t) = 0
u[-1] = 0  # u(L, t) = 0

# Time-stepping loop
for n in range(Nt):
    # Update the temperature using the finite difference formula
    for i in range(1, Nx-1):
        u_new[i] = u[i] + r * (u[i+1] - 2*u[i] + u[i-1])

    # Update for the next time step
    u[:] = u_new[:]

# Plot the final solution
plt.plot(x, u, label="Numerical solution (FD)")
plt.plot(x, np.exp(-alpha * np.pi**2 * T) * np.sin(np.pi * x), 'r--', label="Analytical solution")
plt.xlabel('x')
plt.ylabel('Temperature')
plt.title('1D Heat Equation - Finite Difference Method')
plt.legend()
plt.grid(True)
plt.show()
```



Finite Element Method

The finite element method (FEM) is widely used in engineering, particularly for structural analysis, but it also has applications in CFD. FEM divides the problem domain into small sub-domains (elements), typically using unstructured meshes. It then approximates the solution by using test functions (often polynomials) over these elements.

Discretization:

- **Mesh**: Divide the domain $x \in [0, L]$ into N elements, where each element is defined by nodes at the element boundaries. For this example, we use approximate trial functions (**linear basis functions**).

We represent $u(x,t)$ as a sum of basis functions $\phi_i(x)$ over the nodes:

$$u(x,t) \approx \sum_{i=1}^{N} U_i(t)\phi_i(x)$$

where $U_i(t)$ are the unknowns coefficients at the nodes.

The basis functions are defined over a mesh of elements and have the following properties:

1. Each basis function $\phi_i(x)$ is associated with a specific node x_i.
2. $\phi_i(x)$ is **1** at node x_i, and **0** at all other nodes.
3. $\phi_i(x)$ is linear within each element, non-zero only within the element adjacent to the node, and zero elsewhere.

For 1D problems, the basis functions are typically "tent functions", which are zero everywhere except in the interval covering the two neighboring nodes. For example:

$$\phi_i(x) = \begin{cases} \dfrac{x - x_{i-1}}{x_i - x_{i-1}} & \text{for } x \in [x_{i-1}, x_i], \\ \dfrac{x_{i+1} - x}{x_{i+1} - x_i} & \text{for } x \in [x_i, x_{i+1}], \\ 0 & \text{otherwise.} \end{cases}$$

Weak Form:

Multiply the original **strong form** of the heat equation by a test function $v(x)$, integrate over the domain $[0, L]$, and apply integration by parts to the second-order derivative.

$$\begin{aligned} \int_0^L v\frac{\partial u}{\partial t}\,dx &= \alpha \int_0^L v\frac{\partial^2 u}{\partial x^2}\,dx \\ &= \alpha\left[v\frac{\partial u}{\partial x}\right]_0^L - \alpha\int_0^L \frac{\partial v}{\partial x}\frac{\partial u}{\partial x}\,dx \end{aligned}$$

The boundary term $\left[\dfrac{\partial u}{\partial x}\right]_0^L$ vanishes because $v(x) = 0$ at $x = 0$ and $x = L$ (it satisfies the same boundary conditions as $u(x)$).

Thus, the **weak form** becomes:

$$\int_0^L v\frac{\partial u}{\partial t}\,dx = -\alpha\int_0^L \frac{\partial v}{\partial x}\frac{\partial u}{\partial x}\,dx$$

Matrix Construction:

To approximate $u(x,t)$, we use a **trial function** $u_h(x,t)$, expressed as a linear combination of basis functions $\phi_i(x)$:

$$u_h(x,t) = \sum_{i=1}^{N} U_i(t)\phi_i(x)$$

Similarly, the test function $v(x)$ will be chosen from the same set of basis functions:

$$v(x) = \phi_j(x)$$

Substituting these into the weak form gives:

$$\int_0^L \phi_j(x)\sum_{i=1}^{N}\frac{dU_i(t)}{dt}\phi_i(x)\,dx = -\alpha\int_0^L \frac{\partial\phi_j(x)}{\partial x}\sum_{i=1}^{N} U_i(t)\frac{\partial\phi_i(x)}{\partial x}\,dx$$

1. Mass Matrix

Let's see the left-hand side term involving the time derivative. Since the time derivative acts only on $U_i(t)$, we factor it out:

$$\sum_{i=1}^{N} \frac{dU_i(t)}{dt} \int_0^L \phi_j(x)\phi_i(x)\,dx$$

The **mass matrix** $\mathbf{M}$ is defined as the matrix of integrals of the products of the basis functions:

$$m_{ji} = \int_0^L \phi_j(x)\phi_i(x)\,dx$$

which represents the coupling between different basis functions over the domain. It is symmetric and often sparse, especially for local basis functions like piecewise linear functions. For **linear basis functions** (piecewise linear over each element), this matrix will typically have non-zero values only for neighboring nodes, leading to a **tridiagonal structure**.

$$\mathbf{M} = \begin{pmatrix} m_{11} & m_{12} & 0 & \dots & 0 \\ m_{12} & m_{22} & m_{23} & \dots & 0 \\ 0 & m_{23} & m_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & m_{NN} \end{pmatrix}$$

For **linear basis functions**, these integrals are easy to calculate since $\phi_i(x)$ is a linear function. For example, on a uniform mesh, these integrals result in:

$$m_{ii} = \int_{x_i}^{x_{i+1}} \phi_i(x)^2\,dx = \frac{\Delta x}{3}$$

$$m_{i,i+1} = \int_{x_i}^{x_{i+1}} \phi_i(x)\phi_{i+1}(x)\,dx = \frac{\Delta x}{6}$$

2. **Stiffness Matrix**

The **stiffness matrix** $\mathbf{K}$ comes from the right-hand side of the weak form, which involves the spatial derivative. This term simplifies to:

$$-\alpha \sum_{i=1}^{N} U_i(t) \int_0^L \frac{\partial \phi_j(x)}{\partial x} \frac{\partial \phi_i(x)}{\partial x}\,dx$$

The entries of the stiffness matrix are defined as:

$$k_{ji} = \int_0^L \frac{\partial \phi_j(x)}{\partial x} \frac{\partial \phi_i(x)}{\partial x}\,dx$$

This matrix represents the coupling of the derivatives of the basis functions. Like the mass matrix, the stiffness matrix is also symmetric and sparse for local basis functions. For **linear basis functions** on a uniform mesh, the stiffness matrix will have a **tridiagonal structure**.

$$\mathbf{K} = \begin{pmatrix} k_{11} & k_{12} & 0 & \dots & 0 \\ k_{12} & k_{22} & k_{23} & \dots & 0 \\ 0 & k_{23} & k_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & k_{NN} \end{pmatrix}$$

Similarly, the stiffness matrix integrals are calculated over each element. Since $\phi_i(x)$ is linear, its derivative is constant over each element. Therefore, these integrals are straightforward to compute. For a uniform mesh, these integrals yield:

$$k_{ii} = \int_{x_i}^{x_{i+1}} \left(\frac{d\phi_i(x)}{dx}\right)^2 dx = \frac{2}{\Delta x}$$

$$k_{i,i+1} = \int_{x_i}^{x_{i+1}} \frac{d\phi_i(x)}{dx} \frac{d\phi_{i+1}(x)}{dx}\,dx = -\frac{1}{\Delta x}$$

Assemble the mass and stiffness matrices based on the basis functions, which yields the system of equations:

$$\mathbf{M}\frac{d\boldsymbol{U}}{dt} = -\alpha \mathbf{K}\boldsymbol{U}$$

This system of **ODE**s represents the semi-discrete form of the heat equation, meaning it is discretized in space but still continuous in time.

3. **Implicit Euler**

To solve this system of ODEs, we must discretize it in time. One of the simplest and commonly used approaches is the **implicit Euler** method, which is a first-order time-stepping scheme.

We approximate the time derivative at time step $n+1$ as:

$$\frac{\boldsymbol{U}^{n+1} - \boldsymbol{U}^n}{\Delta t} \approx \left.\frac{d\boldsymbol{U}}{dt}\right|_{n+1}$$

Substitute this into the original ODE:

$$\mathbf{M}\frac{\boldsymbol{U}^{n+1} - \boldsymbol{U}^n}{\Delta t} = -\alpha \mathbf{K}\boldsymbol{U}^{n+1}$$

Rearranging this equation gives us the following system to solve for $\boldsymbol{U}^{n+1}$ at each time step:

$$(\mathbf{M} + \alpha \Delta t \mathbf{K})\boldsymbol{U}^{n+1} = \mathbf{M}\boldsymbol{U}^n$$

At each time step, we need to solve the following **linear system** for $\boldsymbol{U}^{n+1}$:

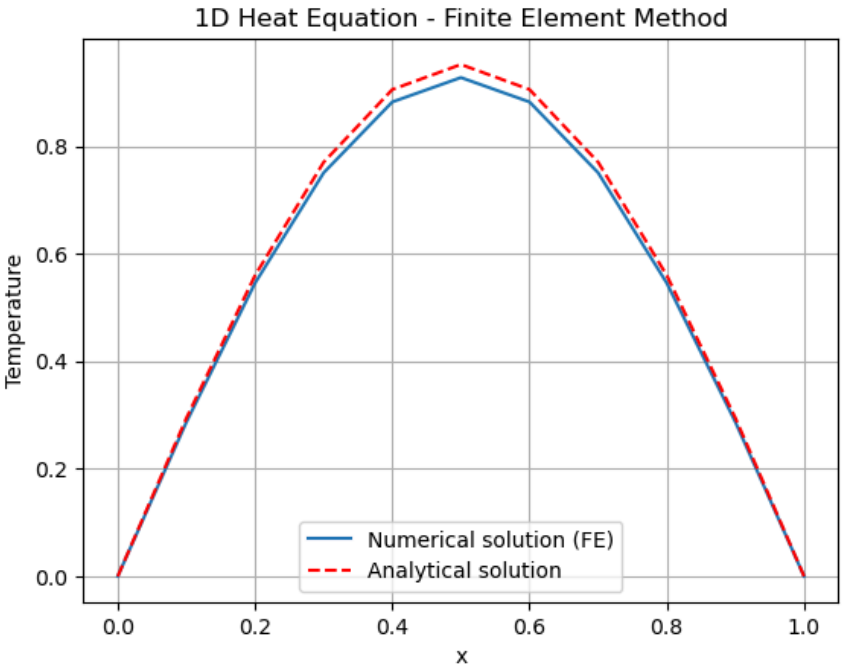
$$\mathbf{A}\boldsymbol{U}^{n+1} = \boldsymbol{b}$$

where:

- $\mathbf{A} = \mathbf{M} + \alpha \Delta t \mathbf{K}$ (system matrix),
- $\boldsymbol{b} = \mathbf{M}\boldsymbol{U}^n$ (right-hand side, where $\boldsymbol{U}^n$ is the solution from the previous time step).

In [2]:
```
from scipy.sparse import diags, csc_matrix
from scipy.sparse.linalg import spsolve

# Parameters for the problem
L = 1.0         # Length of the rod
T = 0.5         # Total time
alpha = 0.01    # Thermal diffusivity
Nx = 10         # Increased number of elements (nodes are Nx+1)
Nt = 500        # Number of time steps
dx = L / Nx     # Spatial step size
dt = T / Nt     # Time step size

# Generate the grid points (nodes)
x = np.linspace(0, L, Nx + 1)

# Initial condition: u(x,0) = sin(pi * x)
u = np.sin(np.pi * x)
u_new = np.zeros_like(u)  # Array to store new temperature values

# Boundary conditions
u[0] = 0  # u(0, t) = 0
u[-1] = 0  # u(L, t) = 0

# Stiffness matrix K (tridiagonal)
main_diag_K = 2 / dx * np.ones(Nx - 1)
off_diag_K = -1 / dx * np.ones(Nx - 2)

# Mass matrix M (tridiagonal)
main_diag_M = 2 * dx / 6 * np.ones(Nx - 1)
off_diag_M = dx / 6 * np.ones(Nx - 2)

# Use scipy.sparse.diags to create the tridiagonal matrices
K = diags([off_diag_K, main_diag_K, off_diag_K], [-1, 0, 1])
M = diags([off_diag_M, main_diag_M, off_diag_M], [-1, 0, 1])

# Convert to CSR format to be compatible with spsolve
K = csc_matrix(K)
M = csc_matrix(M)

# Time-stepping loop
for n in range(Nt):
    # Right-hand side: M * u
    rhs = M @ u[1:Nx]  # Interior nodes only (excluding boundary)

    # System matrix: A = M + alpha * dt * K
```

```
    A = M + alpha * dt * K

    # Solve for the next time step: (M + alpha * dt * K) u_new = M * u
    u_new[1:Nx] = spsolve(A, rhs)

    # Update boundary conditions explicitly (keep them zero)
    u_new[0] = 0
    u_new[-1] = 0

    # Update for the next time step
    u[:] = u_new[:]

# Plot the final solution
plt.plot(x, u, label="Numerical solution (FE)")
plt.plot(x, np.exp(-alpha * np.pi**2 * T) * np.sin(np.pi * x), 'r--', label="Analytical solution")
plt.xlabel('x')
plt.ylabel('Temperature')
plt.title('1D Heat Equation - Finite Element Method')
plt.legend()
plt.grid(True)
plt.show()
```



Spectral Method

The Spectral Method is a powerful technique used in numerical solutions of PDEs, especially for problems with smooth solutions and periodic boundary conditions. It leverages global basis functions, such as trigonometric polynomials (Fourier series) or orthogonal polynomials (like Chebyshev or Legendre polynomials), to approximate the solution of the PDE. Since spectral methods exhibit exponential convergence for smooth solutions, they achieve high accuracy with fewer grid points compared to methods like FDM or FEM.

Discretization:

- **Basis Functions**: In the spectral method, we approximate the solution using **global basis functions** (e.g., sine or cosine functions for periodic boundary conditions). Since our initial condition is $u(x,0) = \sin(\pi x)$, we can use a sine basis.

We first discretize the spatial domain into N_x grid points x_i where $i = 1, 2, \ldots, N_x$.

Explanation:

For real-valued functions, it's more common to use a cosine series (or sine series for Dirichlet boundary conditions). For simplicity, assume our function is expanded using sines, which automatically satisfies boundary conditions like $u(0,t) = u(L,t) = 0$.

$$u(x,t) = \sum_{n=1}^{N} \hat{u}_n(t) \sin\left(\frac{n\pi x}{L}\right)$$

where $\hat{u}_n(t)$ are the Fourier coefficients that we need to solve for, and N is the highest Fourier mode we use to approximate the solution.

The key advantage of using Fourier series is that derivatives in physical space become algebraic operations in Fourier space. Thus, if we take the Fourier transform of the heat equation, we obtain:

$$\frac{d\hat{u}_n(t)}{dt} = -\alpha k_n^2 \hat{u}_n(t)$$

where $k_n = \frac{2\pi n}{L}$.

This is a simple **ODE** for each Fourier coefficient $\hat{u}_n(t)$, which can be solved analytically:

$$\hat{u}_n(t) = \hat{u}_n(0) e^{-\alpha k_n^2 t}$$

Here, $\hat{u}_n(0)$ is the initial Fourier coefficient, which we can compute by taking the Fourier transform of the initial condition $u(x, 0)$.

After solving for the Fourier coefficients at time t, we apply the **Inverse Fourier Transform** to recover the solution $u(x, t)$ in physical space. We use the Inverse FFT (iFFT) to reconstruct the solution in physical space.

Notes:

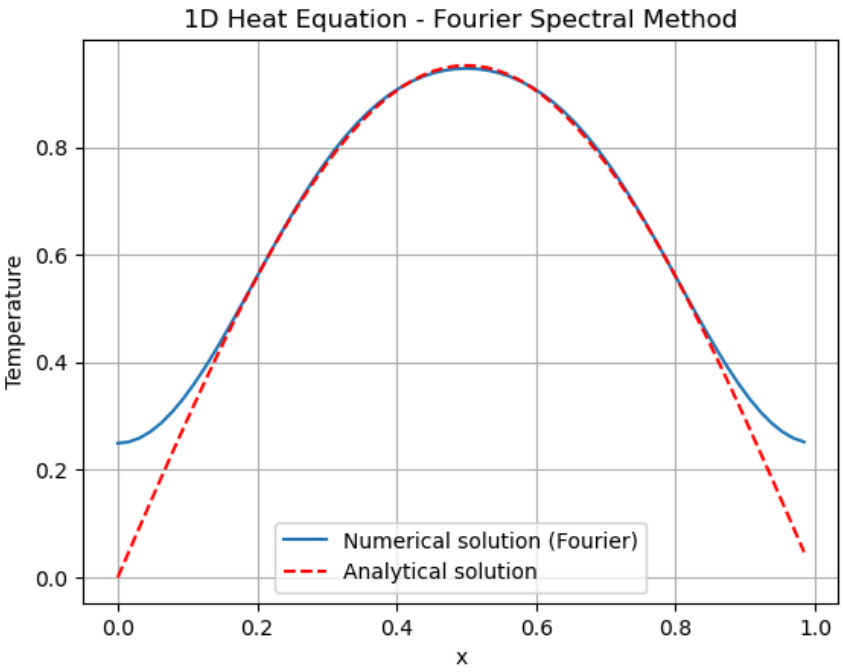
Fourier spectral methods are designed to work naturally with periodic problems, where the solution repeats itself at the boundaries. When applying Fourier transforms to non-periodic problems, like the 1D heat equation with fixed boundary conditions, aliasing issues arise, which means the numerical solution can "wrap around" from $x = L$ to $x = 0$, creating artifacts near the boundaries.

An alternative approach is to use Chebyshev polynomials in the spectral method. Chebyshev polynomials are non-periodic and especially useful in domains that are not naturally periodic, such as the 1D heat equation with **Dirichlet** or **Neumann** boundary conditions. When using Chebyshev polynomials, we can compute **spectral differentiation matrices** that allow us to take derivatives of the solution accurately in **Chebyshev space**. These matrices are used to approximate derivatives and can be efficiently implemented using fast algorithms.

- **Chebyshev nodes**: Use the Chebyshev-Gauss-Lobatto nodes for the spatial grid.
- **Spectral differentiation matrix**: Compute the differentiation matrix in Chebyshev space.
- **Time-stepping**: Use an explicit or implicit time-stepping method to evolve the solution in time.

In [3]:
```
# Parameters for the problem
L = 1.0         # Length of the rod
T = 0.5         # Total time
alpha = 0.01    # Thermal diffusivity
Nx = 64         # Number of spatial grid points (this also limits the number of Fourier modes)
Nt = 500        # Number of time steps
dt = T / Nt     # Time step size
N = 2           # Number of Fourier modes to use

# Generate the spatial grid points
x = np.linspace(0, L, Nx, endpoint=False)

# Initial condition: u(x,0) = sin(pi * x)
u = np.sin(np.pi * x)

# Fourier transform of the initial condition
u_hat = np.fft.rfft(u)

# Wavenumbers for the spectral method (only up to N modes)
k = np.fft.rfftfreq(Nx, d=(x[1] - x[0])) * 2 * np.pi
k = k[:N+1]  # Take only the first N modes (including the zeroth mode)

# Keep only the first N Fourier modes, discard higher modes
u_hat = u_hat[:N+1]

# Time-stepping loop
for n in range(Nt):
    # Update the Fourier coefficients for the first N modes using the analytical time evolution
    u_hat *= np.exp(-alpha * (k**2) * dt)

# Transform back to physical space using only the first N Fourier modes
u_final = np.fft.irfft(u_hat, n=Nx)

# Plot the final solution
```

```
plt.plot(x, u_final, label=f"Numerical solution (Fourier)")
plt.plot(x, np.exp(-alpha * np.pi**2 * T) * np.sin(np.pi * x), 'r--', label="Analytical solution")
plt.xlabel('x')
plt.ylabel('Temperature')
plt.title(f'1D Heat Equation - Fourier Spectral Method')
plt.legend()
plt.grid(True)
plt.show()
```



In [4]:

```
def chebyshev(N):
    """
        Function to compute the Chebyshev nodes and differentiation matrix
    """
    x = np.cos(np.pi * np.arange(N + 1) / N)  # Chebyshev-Gauss-Lobatto nodes in [-1, 1]
    c = np.ones(N + 1)
    c[0] = 2
    c[-1] = 2
    c[1:N] = 1
    X = np.tile(x, (N + 1, 1))
    dX = X - X.T
    D = (c * (1 / c).T) / (dX + np.eye(N + 1))
    D = D - np.diag(np.sum(D, axis=1))
    return D, x

# Parameters
alpha = 0.01  # Thermal diffusivity
T = 0.5       # Total time
Nt = 500      # Number of time steps
N = 40        # Number of Chebyshev nodes (polynomial degree)
dt = T / Nt   # Time step size

# Get Chebyshev differentiation matrix and nodes
D, x_cheb = chebyshev(N)

# Map the Chebyshev nodes from [-1, 1] to [0, 1]
x = (x_cheb + 1) / 2

# Initial condition: u(x,0) = sin(pi * x) on [0, 1]
u = np.sin(np.pi * x)

# Apply boundary conditions (u(0) = u(1) = 0)
u[0] = 0
u[-1] = 0

# Crank-Nicolson scheme for time-stepping
I = np.eye(N + 1)
A = I - 0.5 * alpha * dt * D @ u  # Crank-Nicolson system matrix
B = I + 0.5 * alpha * dt * D @ u

# Time-stepping loop
for n in range(Nt):
    # Update u using Crank-Nicolson (solve A * u_new = B * u_old)
```

```
    u_new = np.linalg.solve(A, B @ u)
    u_new[0] = 0  # Enforce boundary condition at x=0
    u_new[-1] = 0  # Enforce boundary condition at x=1
    u = u_new

# Exact solution for comparison
u_exact = np.exp(-alpha * np.pi**2 * T) * np.sin(np.pi * x)

# Plot the results
plt.plot(x, u, label="Numerical solution (Chebyshev)")
plt.plot(x, u_exact, 'r--', label="Analytical solution")
plt.xlabel('x')
plt.ylabel('Temperature')
plt.title('1D Heat Equation - Chebyshev Spectral Method')
plt.legend()
plt.grid(True)
plt.show()
```