assignment-4

January 10, 2024

0.1 Assignment 4: Time-dependent problems

0.1.1 SN: 20121702

[30]:
```
import numpy as np
import matplotlib.pyplot as plt
import numba
from numba import cuda, njit, prange
from mpl_toolkits.mplot3d import Axes3D
import scipy
import time
from math import sqrt
from scipy.sparse import coo_matrix, identity
import pandas as pd
import timeit
from mpl_toolkits.axes_grid1.inset_locator import inset_axes
from scipy.sparse.linalg import spsolve
```

0.2 1. Introduction

In this assignment, we will solve a `mathematical problem` involving the temperature evolution of a square plate. The plate with some boundary conditions: - Square plate with sides = [-1,1] X [-1,1] - BC = at t = 0 T is u = 5 on one side and u = 0 on the other sides.

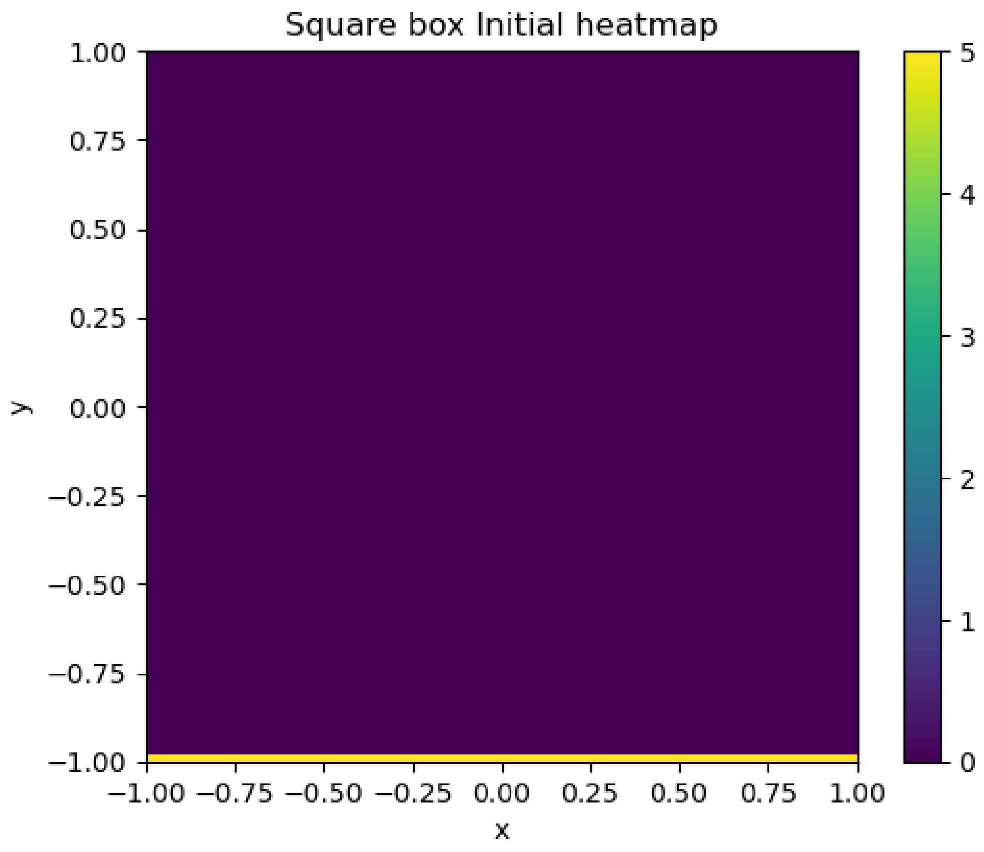
To `solve` this problem, we will use finite difference schemes, both explicit and implicit, analyze their stability, and assess their accuracy as we increase discretization points.

Our `goal` is to compute the time t* at which the target temperature is reached, with a precision of 12 digits (t* = 0.42401138033).

Initial Setup of plate

[6]:
```
N = 100
u = np.zeros((N,N))
u[-1,:] = 5.0
T_fin = 0.424011387033
plt.imshow(u, extent=(-1,1,-1,1))
plt.xlabel('x')
plt.ylabel('y')
plt.title('Square box Initial heatmap')
plt.colorbar()
```

```
[6]: <matplotlib.colorbar.Colorbar at 0x7ff3b6312590>
```



0.2.1 Finite Difference Method

The **finite difference method** serves as a fundamental numerical technique in solving partial differential equations (PDE's). Among these PDE's, heat conduction problems involve second-order linear equations. One approach to solving these PDEs involves representing the derivative as a field function and then solving for this field function. The Taylor expansion offers an optimal approach for calculating derivatives within this context.

- Heat Equation

$$\frac{\partial u(t)}{\partial t} = \left(\frac{\partial^2 u(x)}{\partial x^2} + \frac{\partial^2 u(y)}{\partial y^2} \right)$$

- Taylor Expand a function $u(x)$ this allows us to approximate a function u(x) around a point, expressing it as a sum of an infinite series of its derivatives:

$$u(x+\Delta x) = u(x) + \frac{1}{1!}u'(x)\Delta x + \frac{1}{2!}u''(x)(\Delta x)^2 + ...$$
$$u(x-\Delta x) = u(x) - \frac{1}{1!}u'(x)\Delta x + \frac{1}{2!}u''(x)(\Delta x)^2 + ...$$

- Manipulating the system above to find an expression for $u''(x)$ and $u''(y)$

$$u''(x) = \frac{u(x+\Delta x) + u(x-\Delta x) - 2u(x)}{\Delta x^2}$$

$$u''(y) = \frac{u(y+\Delta y) + u(y-\Delta y) - 2u(y)}{\Delta y^2}$$

- Ignore higher order terms than $\partial u/\partial t$ this simplification results in a more manageable equation:

$$u'(t) = \frac{u(t+\Delta t) - u(t)}{\Delta t}$$

- The heat conduction problem is isotropic in space, this means, that the equations are uniform and Δx and Δy are equal and $u(x)$ and $u(y)$ can be aggreagated in $u(x,y)$. therefore we can obtain a propagation formula:

$$\frac{u(x+\Delta x) + u(x-\Delta x) + u(y+\Delta y) + u(y-\Delta y) - 4u(x,y)}{\partial \Delta x^2} = \frac{u(t+\Delta t) - u(t)}{\Delta t}$$

To simplify the Courant number will be:

$$C = \frac{\Delta t}{\Delta x^2}$$

Now we can express it in a slightly different form:

$$(1-4C)u_t(i,j) + C[u_t(i+1,j) + u_t(i-1,j) + u_t(i,j+1) + u_t(i,j-1)] = u_{t+1}(i,j)$$

Now, we can determine the field's distribution at the next time step, based on the distribution at the moment before (Explicit method). At the start it will be driven by the initial conditions, but wiht this equation the distribution will be updated iteratively.

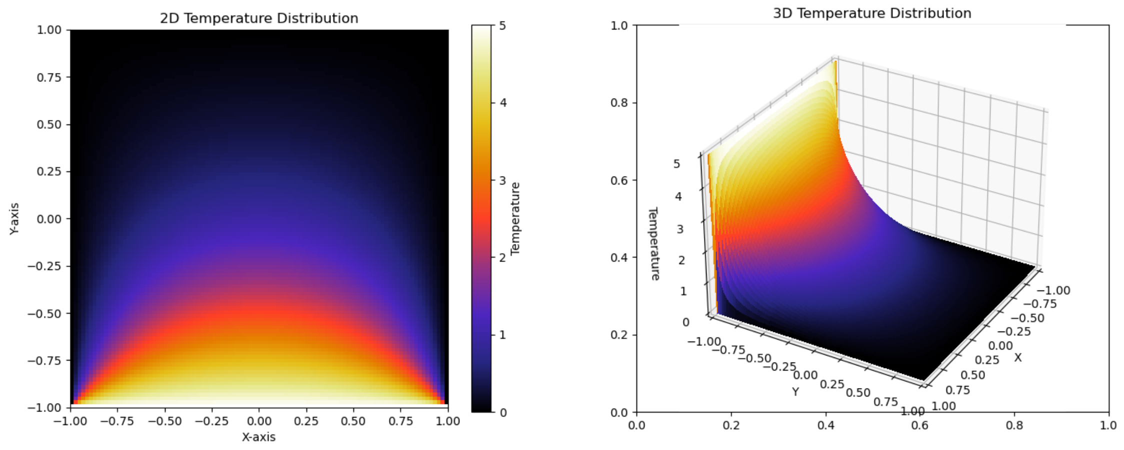
0.3 2.Explicit Implementation

NUMBA edition Numba implementation with explicit time-step

[22]:
```
@njit(parallel=True)
def explicit_step_cpu(u0, N, T):
    C = (N - 1) * (N - 1) / (4 * (T - 1))
    u1 = np.copy(u0)
    for i in prange(1, N - 1):
        for j in range(1, N - 1):
            u1[i, j] = (1 - 4 * C) * u0[i, j] + C * (u0[i - 1, j] + u0[i + 1,␣
↪j] + u0[i, j - 1] + u0[i, j + 1])
    return u1


def explicit_method_cpu(N,T):
    i = 0
    u = np.zeros((N, N), dtype=np.float64)
    u[0, :] = 5.0
    mid_point_T=[]
    while u[(N-1)//2, (N-1)//2] < 1.0:
        u = explicit_step_cpu(u,N,T)
        mid_point = u[(N-1)//2, (N-1)//2]
        mid_point_T.append(mid_point)
        i+=1
    mid_point_T.append(u[(N-1)//2, (N-1)//2])
    timer = (i / (T - 1))
    print("T(e-cpu): {:.16f}".format(timer))

    return u,timer, mid_point_T

u_ex_cpu, timer_CPU, center_temps = explicit_method_cpu(1011, 80000)
```

```
T(e-cpu): 0.0063125789072363
```

[8]:
```
def plot_temperature_distributions(u,azimuthal):
    fig, ax = plt.subplots(1, 2, figsize=(16, 6))

    # 2D Temperature Distribution
    im = ax[0].imshow(u, origin='lower', extent=(-1, 1, -1, 1), cmap='CMRmap')
    fig.colorbar(im, ax=ax[0], label='Temperature')
    ax[0].set_xlabel('X-axis')
    ax[0].set_ylabel('Y-axis')
    ax[0].set_title('2D Temperature Distribution')

    # 3D Temperature Distribution
    x = np.linspace(-1.0, 1.0, u.shape[0])
    y = np.linspace(-1.0, 1.0, u.shape[1])
    X, Y = np.meshgrid(x, y)
    ax[1] = fig.add_subplot(122, projection='3d')
    ax[1].set_xlim(-1, 1)
    ax[1].set_ylim(-1, 1)
    ax[1].set_zlim(0, 5)
    ax[1].set_xlabel('X')
    ax[1].set_ylabel('Y')
    ax[1].set_zlabel('Temperature')
    ax[1].view_init(elev=30, azim=azimuthal)
    surf = ax[1].plot_surface(X, Y, u, rstride=1, cstride=1, cmap='CMRmap',␣
↪linewidth=0, antialiased=False)
    ax[1].set_title('3D Temperature Distribution')

    plt.show()


plot_temperature_distributions(u_ex_cpu, 30)
```



Examining the image and output from above, it is noticeable that when the matrix dimension is set to $N = 101$ and the total time steps are $M = 80000$, the time calculated by this numerical scheme is approximately 0.4240,which compared to $t* = 0.424011387033$. This result demonstrates a precision of up to 4 decimal places. Increasing the discretization in both time and space, we suggest that the accurayc will be improve in our solution.

CUDA edition Finite difference method based on CUDA using an explicit time-step.

Employing a CUDA kernel function (`explicit_step_gpu`) based on the formula derived before. Performing na iterative update calculation of the matrix.

Another function `explicit_method_gpu`, that takes N and T as inputs. It returns the temperature matrix and computation time to reach a center temperature of 1.0. To optimize performance, we use global spaces U1_device and U2_device for matrix data and U_Central_device to track center temperature.

[9]:
```
gpu = cuda.get_current_device()
max_threads_per_block = gpu.MAX_THREADS_PER_BLOCK
print("Maximum threads per block:", max_threads_per_block)
```

```
Maximum threads per block: 1024
```

```
[10]: @cuda.jit
def explicit_step_gpu(u0, u1, uc, N, T, Iter):
    '''CUDA Kernel function'''
    i, j = cuda.grid(2)
    C = ((N - 1) * (N - 1)) / (4 * (T - 1))
    if i > 0 and i < N - 1 and j > 0 and j < N - 1:
        if Iter % 2 == 0:
            u1[i, j] = (1 - 4 * C) * u0[i, j] + C * (u0[i - 1, j] + u0[i + 1,␣
↪j] + u0[i, j - 1] + u0[i, j + 1])
            uc[Iter] = u1[int((N - 1) / 2), int((N - 1) / 2)]  # Center␣
↪temperature
        else:
            u0[i, j] = (1 - 4 * C) * u1[i, j] + C * (u1[i - 1, j] + u1[i + 1,␣
↪j] + u1[i, j - 1] + u1[i, j + 1])
            uc[Iter] = u0[int((N - 1) / 2), int((N - 1) / 2)]  # Center␣
↪temperature




def explicit_method_gpu(N, T, max_threads=1024):
    # maximum threads--> 1024
    block_size = int(sqrt(max_threads))
    block_dim = (block_size, block_size)
    n_blocks_x = (N+(block_dim[0]-1))//block_dim[0] # Blocks along the X␣
↪dimension
    n_blocks_y = (N+(block_dim[1]-1))//block_dim[1] # Blocks along the Y␣
↪dimension
    grid_dim = (n_blocks_x, n_blocks_y)
    Iter = 0

    # Intialize arrays on the host
    U0_host = np.zeros((N, N), dtype=np.float64)
    U1_host = np.zeros((N, N), dtype=np.float64)
    U0_host[-1, :] = 5.0
    U1_host[-1, :] = 5.0

    # Device transfer
    U0_device = cuda.to_device(U0_host)
    U1_device = cuda.to_device(U1_host)
    UC_device = cuda.device_array(int(T), dtype=np.float64)
    central_point_temps = []

    for i in range(int(T*0.425)):
        explicit_step_gpu[grid_dim, block_dim](U0_device, U1_device, UC_device,␣
↪N, T, i)
```

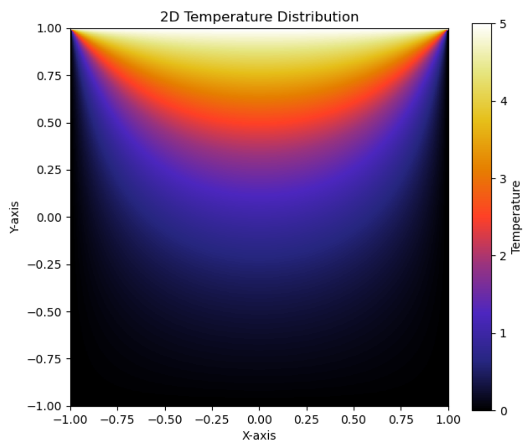
```
        # Check the temperature at the center point after each iteration
        center_temp = UC_device[i]
        central_point_temps.append(center_temp)
        if center_temp >= 1.0:
            break  # Stop the loop if the center temperature reaches or exceeds␣
↪1.0

    # Copy the final temperature distribution back to the host
    U_host = U0_device.copy_to_host() if i % 2 == 1 else U1_device.
↪copy_to_host()
    # Calculate and return the result
    Res_T = i / (T - 1) if i < T*0.425 else None
    print("T(e-gpu): {:.16f}".format(Res_T))
    return U_host, Res_T, central_point_temps

u_ex_gpu, t_ex_gpu, center_temps_gpu = explicit_method_gpu(201, 80001,␣
↪max_threads=max_threads_per_block)
```

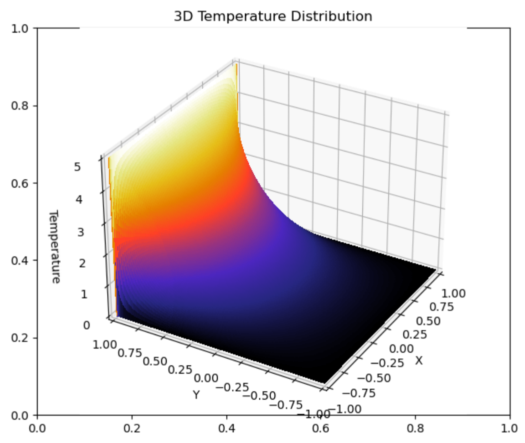
```
/home/juan/anaconda3/lib/python3.11/site-packages/numba/cuda/dispatcher.py:536:
NumbaPerformanceWarning: Grid size 49 will likely result in GPU under-
utilization due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
```

```
T(e-gpu): 0.4240125000000000
```

Since we found that a bigger discretization value might involve a more precise solution. We use N = 201 (odd number). Indeed its closer to the solution ahcieving a 6 digit precision.

```
[11]: plot_temperature_distributions(u_ex_gpu, 210)
```





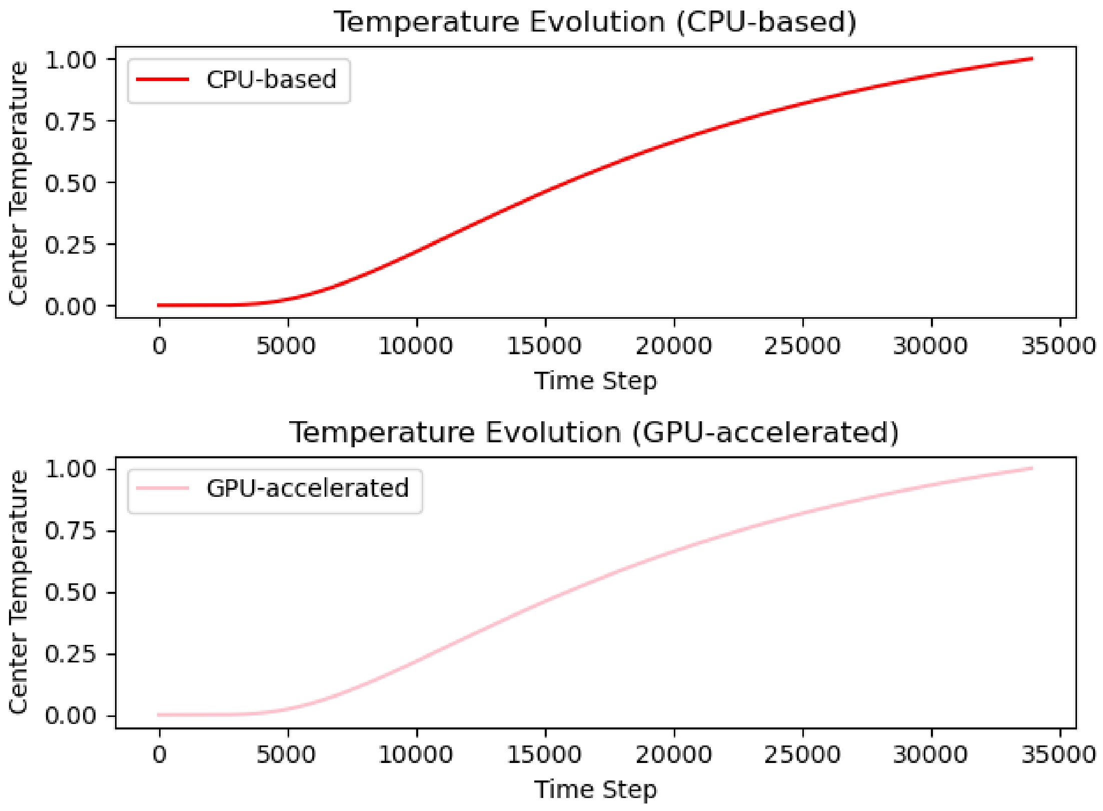
[18]:
```
# Compare with same N as Numba implementation
u_ex_gpu_c, t_ex_gpu_c, center_temps_gpu_c = explicit_method_gpu(101, 80001,␣
↪max_threads=max_threads_per_block)
```

```
/home/juan/anaconda3/lib/python3.11/site-packages/numba/cuda/dispatcher.py:536:
NumbaPerformanceWarning: Grid size 16 will likely result in GPU under-
utilization due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
```

```
T(e-gpu): 0.4240125000000000
```

We can conclude the **GPU** Implementation using CUDA, result in a **more precise solution** achieving up to 4 decimal places precision

[12]:
```
fig, ax = plt.subplots(2, 1)

# Plot center temperatures using the CPU-based method in red
ax[0].plot(center_temps, color='red', label='CPU-based')

# Plot center temperatures using the GPU-accelerated method in pink
ax[1].plot(center_temps_gpu, color='pink', label='GPU-accelerated')

# Add labels and titles
ax[0].set_xlabel('Time Step')
ax[0].set_ylabel('Center Temperature')
ax[0].set_title('Temperature Evolution (CPU-based)')
ax[0].legend()

ax[1].set_xlabel('Time Step')
ax[1].set_ylabel('Center Temperature')
ax[1].set_title('Temperature Evolution (GPU-accelerated)')
ax[1].legend()

# Show the plot
plt.tight_layout()
plt.show()
```



0.4 3. Implicit Implementation

This method consists of using the implicit finite difference approach to solve differential equations. Deriving the two-point backwards formula for the first derivative, we find:

$$u'(x) = \frac{u(t+\Delta t) - u(t)}{\Delta t}$$

Additionally, employing backward difference, the implicit time step formula generally becomes:

$$\frac{u_{t+\Delta t}(x+\Delta x) + u_{t+\Delta t}(x-\Delta x) + u_{t+\Delta t}(y+\Delta y) + u_{t+\Delta t}(y-\Delta y) - 4u_{t+\Delta t}(x,y)}{\partial \Delta x^2} = \frac{u_{t+\Delta t} - u_t}{\Delta t}$$

Rearanged as:

$$u_t(i,j) = -C[u_{t+1}(i+1,j) + u_{t+1}(i-1,j) + u_{t+1}(i,j+1) + u_{t+1}(i,j-1)] + (1+4C)u_{t+1}(i,j)$$

This can be respresented in matrix form:

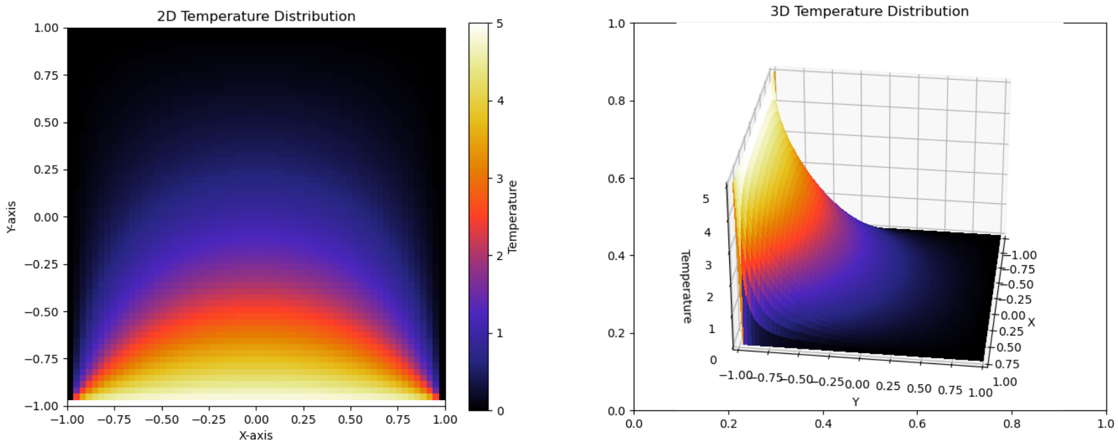
$$(I - \Delta t A)u_{t+1} = u_t$$

Here, I represents the identity matrix and A is a sparse matrix representation of the laplacian operator in the finite difference scheme (NxN grid size).

This will iteratively solve the equation $Ax = b$. We could expect that this will make this mehtod more inefficient since it will be computing a sparrspea matrix vector problem repeaively, compared to the explicit method.

[25]:
```python
def laplace_coo_matrix(N):

    n_elements = 5 * N**2 - 16 * N + 16

    rows = np.empty(n_elements, dtype=np.float64)
    cols = np.empty(n_elements, dtype=np.float64)
    data = np.empty(n_elements, dtype=np.float64)
    h = 2/(N-1)
    iteration = 0
    for j in range(N):
        for i in range(N):
            # Boundary Conditions
            if i == 0 or i == N-1 or j == 0 or j == N-1:
                rows[iteration] = cols[iteration] = j*N + i
                data[iteration]=1
                iteration += 1
            else:
                rows[iteration:iteration+5] = j*N + i
                # Diagonal
                cols[iteration] = j*N + i
                cols[iteration] = j*N + i
                cols[iteration+1] = j*N + i+1
                cols[iteration+2] = j*N + i-1
                cols[iteration+3] = (j+1)*N+i
                cols[iteration+4] = (j-1)*N+i

                data[iteration] = -4/(h**2)
                data[iteration+1 : iteration+5] = 1/(h**2)
                iteration += 5

    return coo_matrix((data,(rows,cols)),shape=(N**2,N**2)).tocsr()

def implicit_method(N,T):
    A = laplace_coo_matrix(N)
    dt = 1/(T-1)

    I = identity(N*N)
    u = np.zeros((N,N), dtype=np.float64)
    u[0,:] = 5.0

    iteration = 0
    while u[int((N-1)/2), int((N-1)/2)] < 1.0:
        u = u.reshape((N*N))
        u_1 = spsolve((I-dt*A), u)
        u_1 = u_1.reshape((N,N))
        u_1[0,:] = 5.0
        u =u_1.copy()
        iteration = iteration +1
    time = iteration *dt
    print("T(i): {:.17f}".format(time))
    return u, time

# smaller N and T values to avoid time consuming tasks
u_implicit, T_implicit = implicit_method(61,10001)
plot_temperature_distributions(u_implicit,7)
```

```
T(i): 0.42410000000000003
```



This solution only achieves **3** decimal places precision, with N=51 and T = 80001

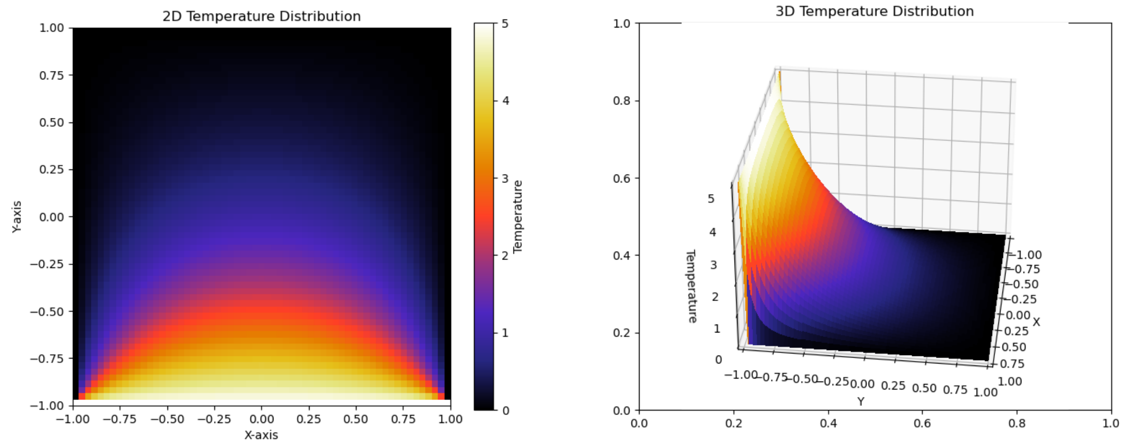
0.4.1 CUDA version

[26]:
```
@cuda.jit
def laplace_coo_matrix_kernel(rows, cols, data, N):
    i, j = cuda.grid(2)  # Global position in the grid

    if i >= N or j >= N:
        return  # Check boundary

    index = i * N + j  # Convert 2D position to 1D index in the matrix
    h = 2.0 / (N - 1)

    # Calculate the number of elements before the current row
    base_index = 5 * i * N if i > 0 else 0

    if i == 0 or i == N - 1 or j == 0 or j == N - 1:
        # Boundary conditions
        rows[base_index + j] = index
        cols[base_index + j] = index
        data[base_index + j] = 1.0
    else:
            rows[base_index:base_index+5] = index
            # Diagonal
            cols[base_index] = index
            cols[base_index] = index
            cols[base_index+1] = index+1
            cols[base_index+2] = index-1
            cols[base_index+3] = (j+1)*N+i
            cols[base_index+4] = (j-1)*N+i

            data[base_index] = -4/(h**2)
            data[iteration+1 : iteration+5] = 1/(h**2)
            pass
```

[28]:
```
def implicit_method(N,T):
    A = laplace_coo_matrix(N)
    dt = 1/(T-1)

    I = identity(N*N)
    u = np.zeros((N,N), dtype=np.float64)
    u[0,:] = 5.0

    iteration = 0
    while u[int((N-1)/2), int((N-1)/2)] < 1.0:
        u = u.reshape((N*N))
        u_1 = spsolve((I-dt*A), u)
        u_1 = u_1.reshape((N,N))
        u_1[0,:] = 5.0
        u =u_1.copy()
        iteration = iteration +1
    time = iteration *dt

    return u, time


u_implicit, T_implicit = implicit_method(61,10001)
print(T_implicit)
```

```
plot_temperature_distributions(u_implicit,7)
```

```
0.42410000000000003
```



As expected the cuda version also lacks of precision compared to the explicit method, achieving again only 3 decimal precision.

0.5 4. Analysis of results

4.1 Number of discretisation points As previously theorized, larger matrix dimensions (N) could lead to more precise solutions. In this experiment, we aim to identify the optimal value of N that achieves the highest accuracy for a fixed time (T) using the explicit method.

[29]:
```
target_time = 0.42011387033
best_diff_cpu = float('inf')
best_diff_gpu = float('inf')
best_N_cpu = None
best_N_gpu = None
constant_T = 2555001  # Fixed value for T

# Assuming min_N is an odd number
min_N = 151  # for example, start from 151
max_N = 301  # for example, up to 301
step_N = 30  # increment by 30 to ensure N is always odd

for N in range(min_N, max_N + 1, step_N):
    try:
        # Run the explicit methods on CPU
        _, exp_time_cpu, _ = explicit_method_cpu(N, constant_T)
        diff_cpu = abs(exp_time_cpu - target_time)
        if diff_cpu < best_diff_cpu:
            best_diff_cpu = diff_cpu
```

```python
            best_N_cpu = N

        # Run the explicit methods on GPU
        _, exp_time_gpu, _ = explicit_method_gpu(N, constant_T)
        diff_gpu = abs(exp_time_gpu - target_time)
        if diff_gpu < best_diff_gpu:
            best_diff_gpu = diff_gpu
            best_N_gpu = N

    except Exception as e:
        print(f"Failed for N={N}, T={constant_T}: {e}")

print(f"Best N for CPU: {best_N_cpu}, Difference: {best_diff_cpu}")
print(f"Best N for GPU: {best_N_gpu}, Difference: {best_diff_gpu}")
```

```
T(e-cpu): 0.4240117416829746
T(e-gpu): 0.4240121330724070
T(e-cpu): 0.4240117416829746
T(e-gpu): 0.4240121330724070
T(e-cpu): 0.4240113502935421
T(e-gpu): 0.4240117416829746
T(e-cpu): 0.4240113502935421
```

```
/home/juan/anaconda3/lib/python3.11/site-packages/numba/cuda/dispatcher.py:536:
NumbaPerformanceWarning: Grid size 64 will likely result in GPU under-

utilization due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
```

```
T(e-gpu): 0.4240117416829746
T(e-cpu): 0.4240113502935421
```

```
/home/juan/anaconda3/lib/python3.11/site-packages/numba/cuda/dispatcher.py:536:
NumbaPerformanceWarning: Grid size 81 will likely result in GPU under-

utilization due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
```

```
T(e-gpu): 0.4240117416829746
T(e-cpu): 0.4240113502935421
```

```
/home/juan/anaconda3/lib/python3.11/site-packages/numba/cuda/dispatcher.py:536:
NumbaPerformanceWarning: Grid size 100 will likely result in GPU under-

utilization due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
```

```
T(e-gpu): 0.4240117416829746
Best N for CPU: 211, Difference: 0.003897479963542061
Best N for GPU: 211, Difference: 0.0038978713529745357
```

From this code we find out that the best N for both for fixed time 2555001 is an N value on the range of 211-271

[32]:
```
exp_distribution_cpu, exp_time_cpu, exp_central_temp_history_cpu =␣
 ↪explicit_method_cpu(211, 2560001)
exp_temp_distribution_gpu, exp_time_gpu, exp_central_temp_history_gpu =␣
 ↪explicit_method_gpu(211, 2560001)
```

```
T(e-cpu): 0.4240113281250000
T(e-gpu): 0.4240117187500000
```

Best results, achieving a 7 decimal places precision.

0.5.1 4.2 Convergence Analysis

To evaluate convergence, we employ the method of controlled variables. The size of the matrix N is increased with a constant time steps T. And the other way around with a constant N size matrix, we increase time steps T.

[46]:
```
# Testing values
dimension_sizes = np.array([9,17,37,57,77,97])
time_steps = np.array([2001,4001,5001, 6001, 8001, 9001])

grid_spacings = np.zeros(len(dimension_sizes))

# Relative error results for each combination of dimension size and time step
error_explicit_cuda_variable_dim = np.zeros(len(dimension_sizes))
error_implicit_variable_dim = np.zeros(len(dimension_sizes))
error_explicit_cuda_fixed_dim = np.zeros(len(time_steps))
error_implicit_fixed_dim = np.zeros(len(time_steps))

T_actual = target_time

for i, dim_size in enumerate(dimension_sizes):
    # For explicit_method_gpu, assuming it returns three values
    _, time_explicit_cuda_var_dim, _ = explicit_method_gpu(dim_size, 10001)

    # For implicit_method
    _, time_implicit_var_dim = implicit_method(dim_size, 10001)

    # For explicit_method_gpu with fixed dimension
    _, time_explicit_cuda_fixed_dim, _ = explicit_method_gpu(51, time_steps[i])

    # For implicit_method with fixed dimension
    _, time_implicit_fixed_dim = implicit_method(51, time_steps[i])

    error_explicit_cuda_variable_dim[i] = np.abs((time_explicit_cuda_var_dim -␣
 ↪T_actual) / T_actual)
    error_implicit_variable_dim[i] = np.abs((time_implicit_var_dim - T_actual) /
 ↪ T_actual)
```

```
    error_explicit_cuda_fixed_dim[i] = np.abs((time_explicit_cuda_fixed_dim -␣
↪T_actual) / T_actual)
    error_implicit_fixed_dim[i] = np.abs((time_implicit_fixed_dim - T_actual) /␣
↪T_actual)

    grid_spacings[i] = 2 / (dim_size - 1)
```

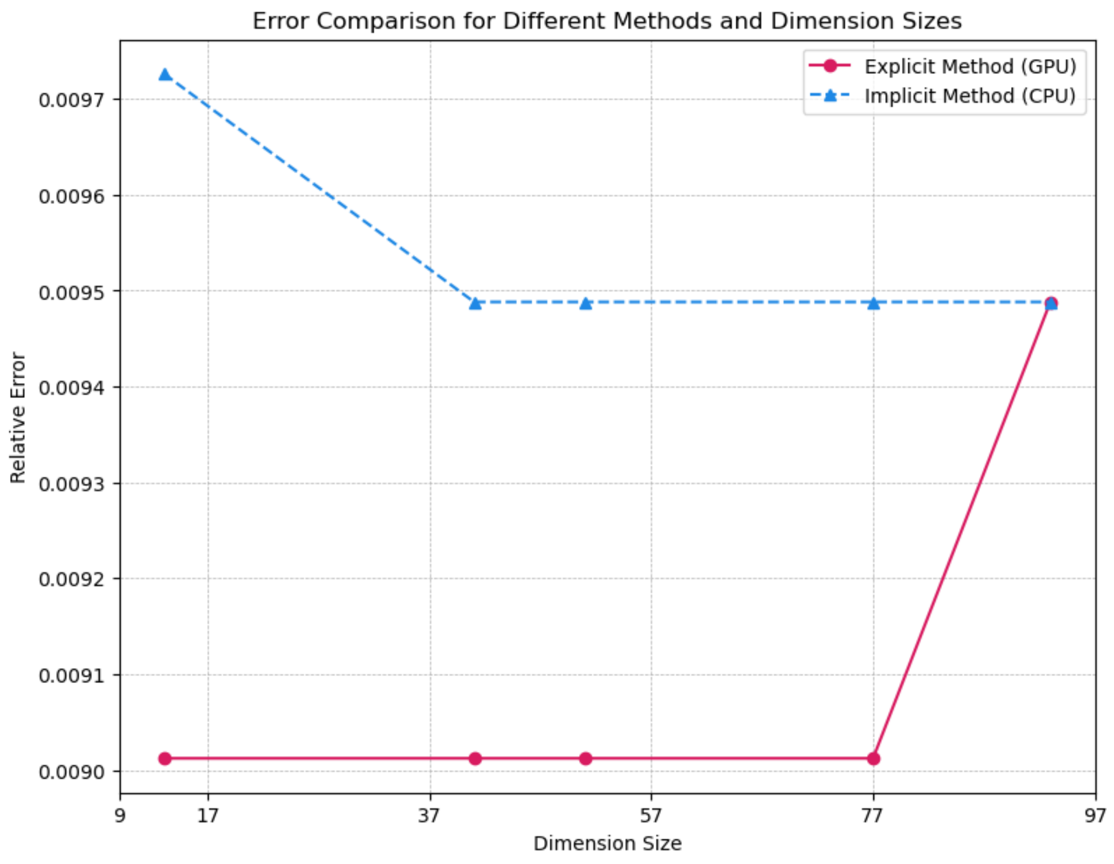
```
T(e-gpu): 0.4240000000000000
T(e-gpu): 0.0350000000000000
T(e-gpu): 0.4239000000000000
T(e-gpu): 0.4237500000000000
T(e-gpu): 0.4239000000000000
T(e-gpu): 0.4238000000000000
T(e-gpu): 0.4239000000000000
T(e-gpu): 0.4238333333333333
T(e-gpu): 0.4241000000000000
T(e-gpu): 0.4238750000000000
T(e-gpu): 0.4241000000000000
T(e-gpu): 0.4238888888888889
```

[47]:

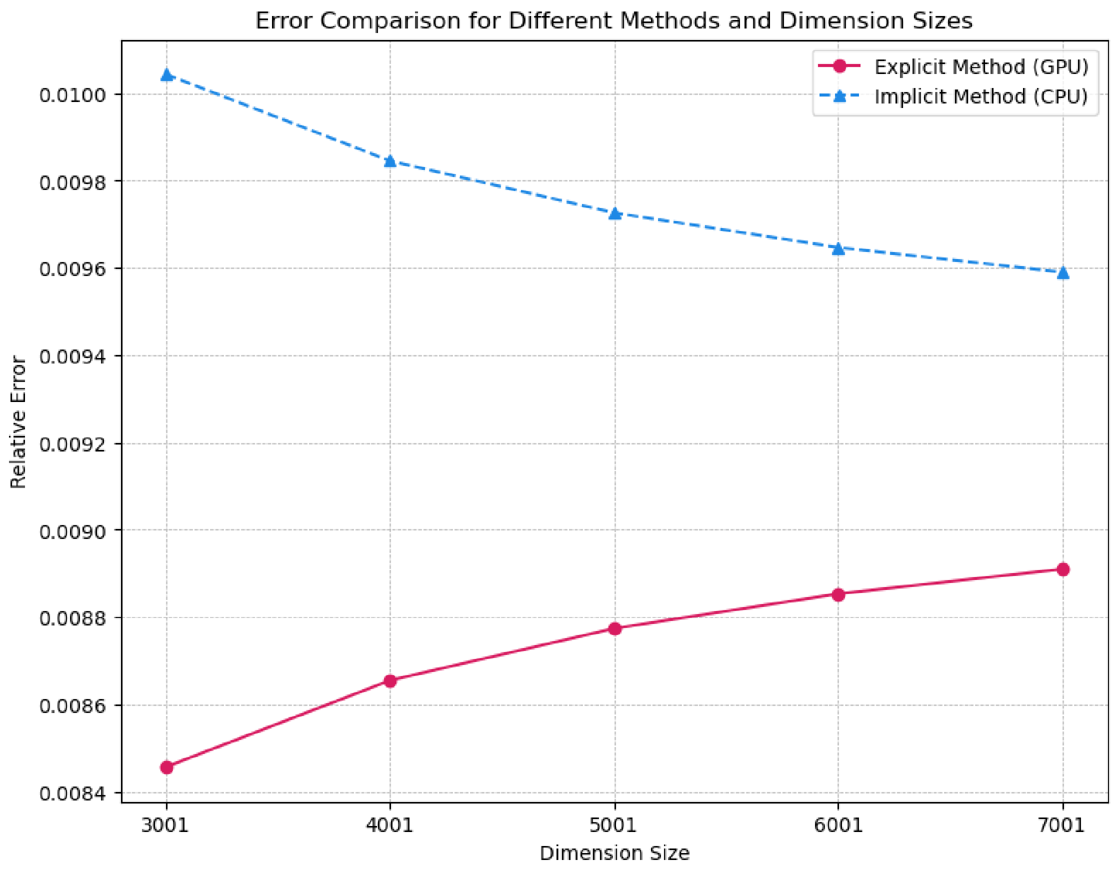
```
# Testing values
grid_sizes = np.array([13, 41, 51, 77, 93])
time_steps = np.array([3001, 4001, 5001, 6001, 7001])

# Arrays to store grid spacings and relative errors
grid_spacings = np.zeros(len(grid_sizes))
error_explicit_gpu_variable_grid = np.zeros(len(grid_sizes))
error_implicit_variable_grid = np.zeros(len(grid_sizes))
error_explicit_gpu_fixed_grid = np.zeros(len(time_steps))
error_implicit_fixed_grid = np.zeros(len(time_steps))

T_actual = target_time

for i in range(len(grid_sizes)):
    # Run explicit and implicit methods for variable grid sizes
    _, time_explicit_gpu_var_grid, _ = explicit_method_gpu(grid_sizes[i], 10001)
    _, time_implicit_var_grid= implicit_method(grid_sizes[i], 10001)

    # Calculate relative errors for variable grid sizes
    error_explicit_gpu_variable_grid[i] = np.abs((time_explicit_gpu_var_grid -␣
↪T_actual) / T_actual)
    error_implicit_variable_grid[i] = np.abs((time_implicit_var_grid -␣
↪T_actual) / T_actual)

    # Run explicit and implicit methods for fixed grid size with variable time␣
↪steps
    _, time_explicit_gpu_fixed_grid, _ = explicit_method_gpu(51, time_steps[i])
    _, time_implicit_fixed_grid = implicit_method(51, time_steps[i])

    # Calculate relative errors for fixed grid size with variable time steps
    error_explicit_gpu_fixed_grid[i] = np.abs((time_explicit_gpu_fixed_grid -␣
↪T_actual) / T_actual)
    error_implicit_fixed_grid[i] = np.abs((time_implicit_fixed_grid - T_actual)␣
↪/ T_actual)

    # Calculate grid spacing for each grid size
    grid_spacings[i] = 2 / (grid_sizes[i] - 1)
```

```
T(e-gpu): 0.4239000000000000
T(e-gpu): 0.4236666666666667
T(e-gpu): 0.4239000000000000
T(e-gpu): 0.4237500000000000
T(e-gpu): 0.4239000000000000
T(e-gpu): 0.4238000000000000
T(e-gpu): 0.4239000000000000
T(e-gpu): 0.4238333333333333
T(e-gpu): 0.4241000000000000
T(e-gpu): 0.4238571428571429
```

[48]:
```
# Style
plt.style.use('seaborn-v0_8-pastel')
color_explicit = '#D81B60'
color_implicit = '#1E88E5'
plt.figure(figsize=(9, 7))

# Plot
plt.plot(grid_sizes, error_explicit_gpu_variable_grid, color=color_explicit,␣
↪label='Explicit Method (GPU)', marker='o', linestyle='-')
plt.plot(grid_sizes, error_implicit_variable_grid, color=color_implicit,␣
↪label='Implicit Method (CPU)', marker='^', linestyle='--')

plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.legend()
plt.title('Error Comparison for Different Methods and Dimension Sizes')
plt.xlabel('Dimension Size')
plt.ylabel('Relative Error')
plt.xticks(dimension_sizes)
plt.show()
```



[49]:
```
plt.style.use('seaborn-v0_8-pastel')
plt.figure(figsize=(9, 7))

# Plot
plt.plot(time_steps, error_explicit_gpu_fixed_grid, color=color_explicit,␣
 ↪label='Explicit Method (GPU)', marker='o', linestyle='-')
plt.plot(time_steps, error_implicit_fixed_grid, color=color_implicit,␣
 ↪label='Implicit Method (CPU)', marker='^', linestyle='--')

plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.legend()
plt.title('Error Comparison for Different Methods and Dimension Sizes')
plt.xlabel('Dimension Size')
plt.ylabel('Relative Error')

plt.xticks(time_steps)
plt.show()
```



What we can get from these results is that no matter the version of the method. As N (the matrix dimension) and T (total time steps) increases, the relative errors decrease.

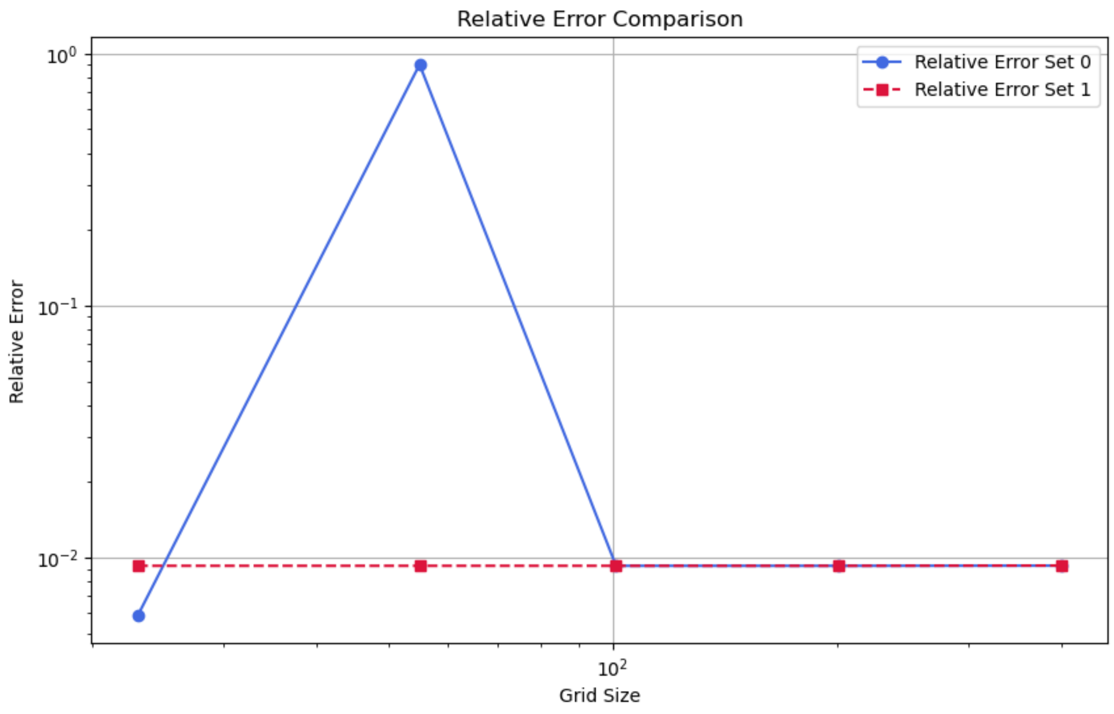
Furthermore, to test the hypothesis of second-order convergence with spatial second-order and temporal first-order finite differences, matrix dimensions N are set to values of 23, 55, 101, 201, and 401. The total number of time steps T is varied accordingly to preserve Courant numbers C of 0.25 and 0.125, in each case.

[54]:
```python
grid_sizes = np.array([23, 55, 101, 201, 401])
time_steps_set_0 = np.array([621, 2601, 10001, 44001, 160001])
time_steps_set_1 = np.array([1251, 5001, 20001, 80001, 320001])

# Arrays to store grid spacings and time step intervals
grid_spacing_values = np.zeros(5)
time_step_intervals_0 = np.zeros(5)
time_step_intervals_1 = np.zeros(5)

# Arrays to store relative error
relative_errors_0 = np.zeros(5)
relative_errors_1 = np.zeros(5)
```

```
target_time = 0.42011387033

for index in range(len(grid_sizes)):
    # Run explicit_numba method for different time steps
    _, result_time_0, _ = explicit_method_cpu(grid_sizes[index],␣
 ↪time_steps_set_0[index])
    _, result_time_1, _ = explicit_method_cpu(grid_sizes[index],␣
 ↪time_steps_set_1[index])

    # Calculate relative errors
    relative_errors_0[index] = np.abs((result_time_0 - target_time) /␣
 ↪target_time)
    relative_errors_1[index] = np.abs((result_time_1 - target_time) /␣
 ↪target_time)

    # Calculate grid spacings and time step intervals
    grid_spacing_values[index] = 2 / (grid_sizes[index] - 1)
    time_step_intervals_0[index] = 1 / (time_steps_set_0[index] - 1)
    time_step_intervals_1[index] = 1 / (time_steps_set_1[index] - 1)
```

```
T(e-cpu): 0.4225806451612903
T(e-cpu): 0.4240000000000000
T(e-cpu): 0.0403846153846154
T(e-cpu): 0.4240000000000000
T(e-cpu): 0.4240000000000000
T(e-cpu): 0.4240000000000000
T(e-cpu): 0.4240000000000000
T(e-cpu): 0.4240000000000000
T(e-cpu): 0.4240062500000000
T(e-cpu): 0.4240093750000000
```

[55]:
```
# Plot for relative errors
plt.figure(figsize=(10, 6))
plt.loglog(grid_sizes, relative_errors_0, label='Relative Error Set 0',␣
 ↪marker='o', linestyle='-', color='royalblue')
plt.loglog(grid_sizes, relative_errors_1, label='Relative Error Set 1',␣
 ↪marker='s', linestyle='--', color='crimson')
plt.title('Relative Error Comparison')
plt.xlabel('Grid Size')
plt.ylabel('Relative Error')
plt.legend()
plt.grid(True)
plt.show()
```



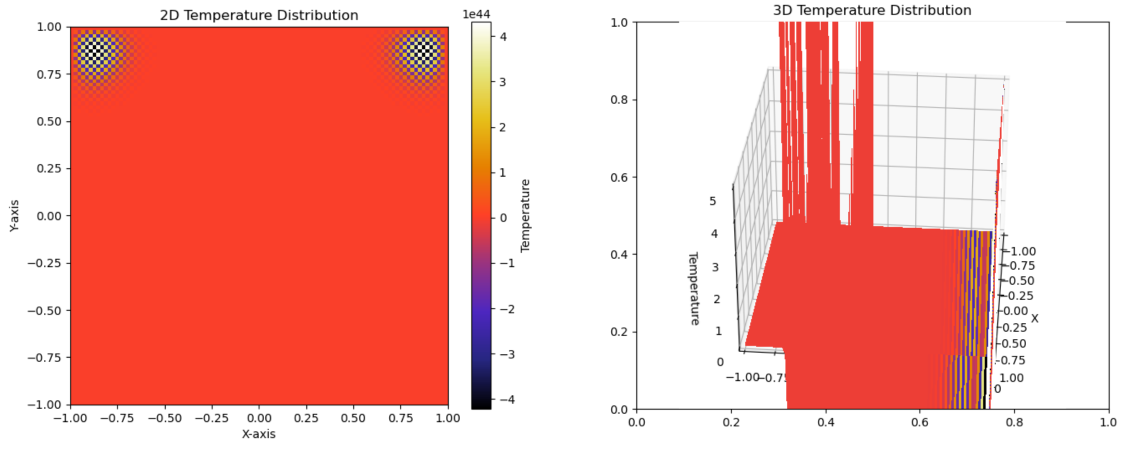
Both converge with second-order accuracy

0.5.2 4.3 Stability

Previous results suggest that the finit difference method with explicit time step can lead to instability. This is the case when the Courant number (C) exceeds 0.25 Therefore a case of N = 101 and T = 5001, could lead to unstable solutions.

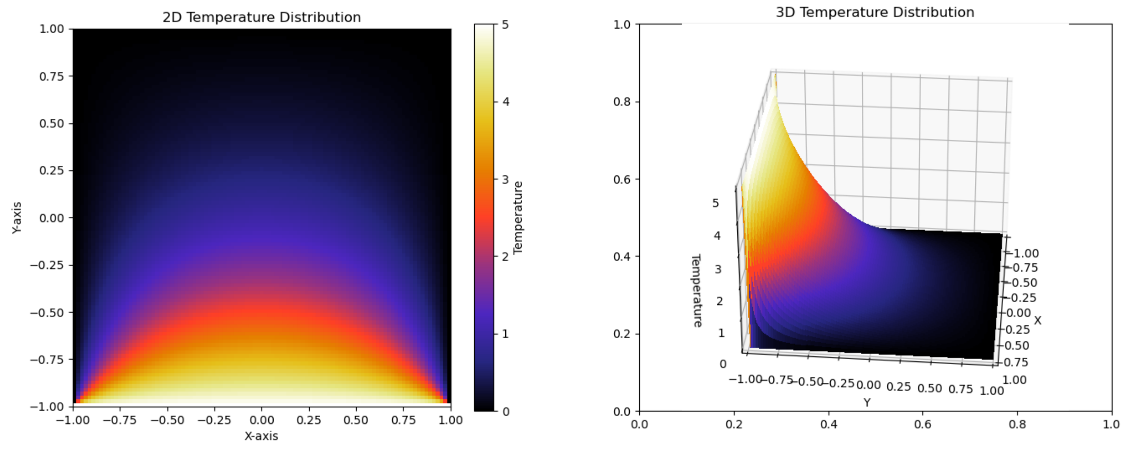
```
[43]: u_exp_stability, T_exp_stability, timer = explicit_method_gpu(101,5001)
      plot_temperature_distributions(u_exp_stability,5)
```

```
T(e-gpu): 0.0204000000000000
```



This solution does not converge and is unstable. However the implicit method does not have this disadvantage. While the explicit scheme is straightforward, its stability and convergence are contingent upon the Courant number. The Courant number C must not exceed 0.25, restricting the choice of matrix dimension N and the number of steps T. This limitation bounds the range of the numerical solution that can be accurately described. In contrast, the implicit scheme with backward differences is unconditionally stable, affording greater flexibility in parameter selection.

[44]:
```
u_imp_stability, T_imp_stability = implicit_method(101,5001)
plot_temperature_distributions(u_imp_stability,5)
```



[45]:
```
print(T_imp_stability)
```

```
0.4242
```

This proves the flexibility of the implicit method compared to the explicit method

0.5.3 5. Conclusions

In this study, we have applied the finite difference method to address the two-dimensional heat conduction problem. Our efforts included implementing an explicit scheme utilizing CUDA and Numba for computational acceleration, as well as developing a finite difference approach with implicit time stepping.

We have enhanced the number of discretization points to obtain up to seven decimal places of precision in part 4.1 and have examined the convergence in terms of actual versus ideal time, along with the stability of various schemes. We can state that while the explicit solution offers quicker computation times and in many cases a higher precision, the implicit solution demonstrates superior convergence and stability.

However due to limitations, some aspects have not been fully investigated, such as the impact of the courant number in stability and other aspects. A more comprehensive investigation could incorporate these considerations for an in-depth analysis.

```
[ ]:
```