Consider a square plate with sides $[-1, 1] \times [-1, 1]$. At time t = 0 we are heating the plate up such that the temperature is $u = 5$ on one side and $u = 0$ on the other sides. The temperature evolves according to:

$$u_t = \Delta u.$$

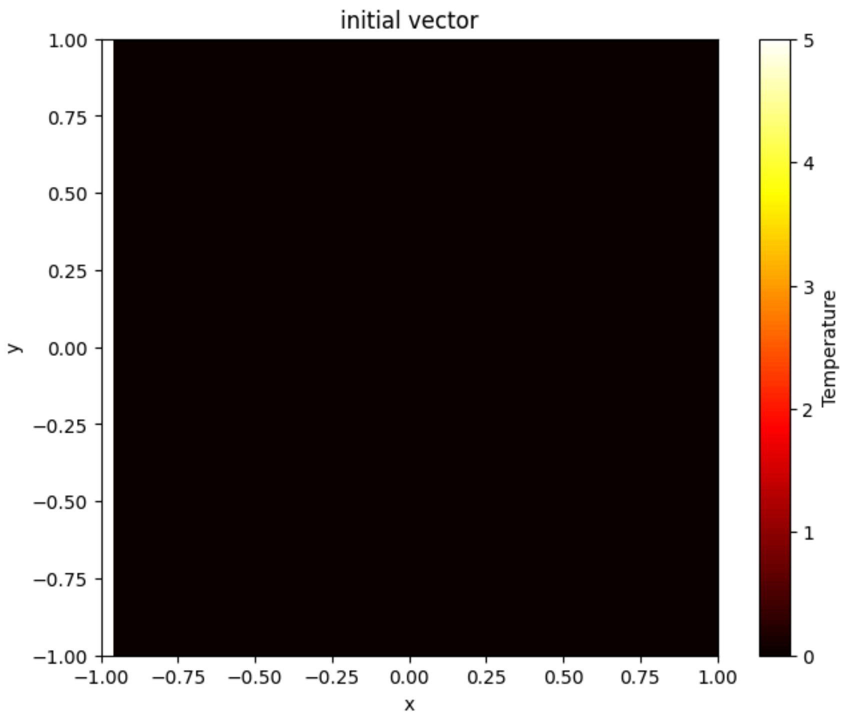
At what time t^* does the plate reach $u = 1$ at the center of the plate?
Implement a finite difference scheme and try with explicit and implicit time-stepping. Numerically investigate the stability of your schemes.

By increasing the number of discretisation points demonstrate how many correct digits you can achieve. Also, plot the convergence of your computed time t^* against the actual time. To 12 digits the wanted solution is $t^* = 0.424011387033$.

A GPU implementation of the explicit time-stepping scheme is not necessary but would be expected for a very high mark beyond 80%.

1) Explicit time-stepping

```
import numpy as np
import matplotlib.pyplot as plt
from numba import cuda


def initial_conditions(N):

    # square plate with sides [-1,1]×[-1,1], L=1-(-1)=2
    x = np.linspace(-1, 1, N)
    y = np.linspace(-1, 1, N)
    h = x[1] - x[0]

    u = np.zeros((N, N), dtype=np.float64)

    u[:, 0] = 5.0    # Left boundary
    u[:, -1] = 0.0   # Right boundary
    u[0, :] = 0.0    # Top boundary
    u[-1, :] = 0.0   # Bottom boundary


    u[0, 0] = 5.0    # Top-left corner
    u[-1, 0] = 5.0   # Bottom-left corner


    return u, h

# checking the initial conditions by plotting it
N = 50
u, h = initial_conditions(N)

plt.figure(figsize=(8, 6))
plt.imshow(u, extent=[-1, 1, -1, 1], origin='lower', cmap='hot', vmin=0, vmax=5)
plt.colorbar(label='Temperature')
plt.title('initial vector')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

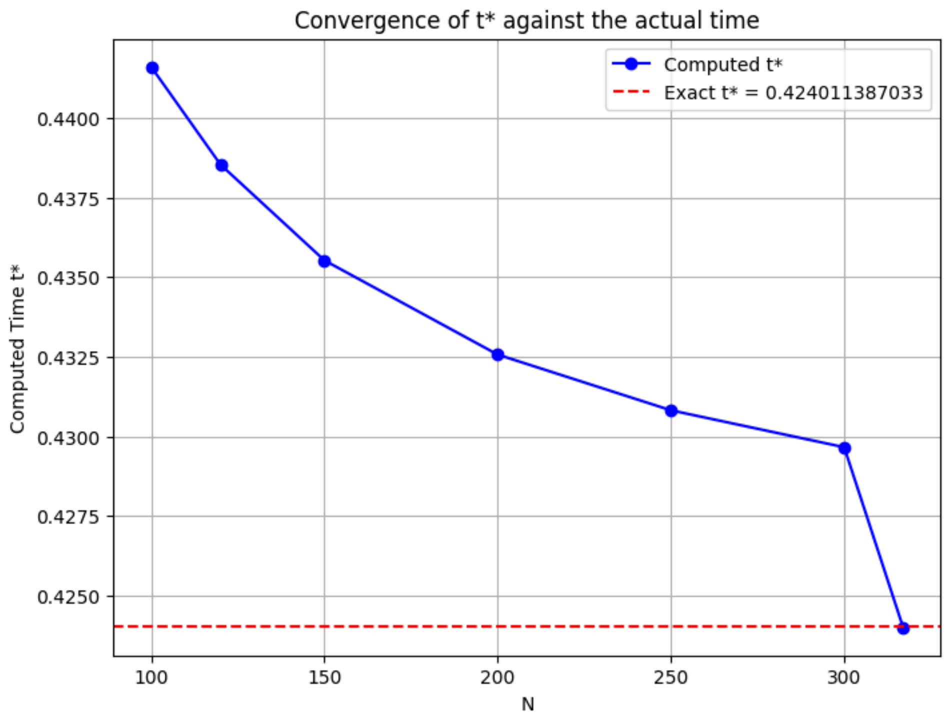


```
@cuda.jit(fastmath=True)
def build_laplacian_matrix_explicit(u, laplacian, h):
    i, j = cuda.grid(2)
    row, col = u.shape
```

```
        if 1 <= i < row-1 and 1 <= j < col-1:
            # Δu ≈
            laplacian[i, j] = ( u[i, j+1] + u[i, j-1] + u[i+1, j] + u[i-1, j] - 4 * u[i, j]) / (h**2)



    @cuda.jit(fastmath=True)
    def explicit_timestep(u, u_next, laplacian, dt):
        i, j = cuda.grid(2)
        row, col = u.shape

        if 1 <= i < row-1 and 1 <= j < col-1:
            u_next[i, j] = u[i, j] + dt * laplacian[i, j]



    def check_stability(h, alpha, dt):

        courant_number = 2 * alpha * dt / (h * h)

        if courant_number > 1:
            print("The scheme may be unstable.")




    def solve_explicit(N, T, dt, alpha):

        u, h = initial_conditions(N)
        u_next = np.copy(u)

        check_stability(h, alpha, dt)


        u_device = cuda.to_device(u)
        u_next_device = cuda.to_device(u_next)
        laplacian_device = cuda.device_array((N, N), dtype=np.float64)

        threads_per_block = (32, 32)
        blocks_per_grid_x = (N + threads_per_block[0] - 1) // threads_per_block[0]
        blocks_per_grid_y = (N + threads_per_block[1] - 1) // threads_per_block[1]
        blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)

        center = N // 2
        times = []

        num_steps = int(T / dt)
        for step in range(num_steps):
            build_laplacian_matrix_explicit[blocks_per_grid, threads_per_block](u_device, laplacian_device, h)
            explicit_timestep[blocks_per_grid, threads_per_block](u_device, u_next_device, laplacian_device, dt)

            u_device, u_next_device = u_next_device, u_device

            u_host = u_device.copy_to_host()
            times.append(step * dt)

            if u_host[center, center] >= 1.0:
                return times[-1]

        return None




    T = 2.0  # let t=2
    dt = 0.00001
    alpha = 0.01
    points = [100, 120, 150, 200, 250, 300, 317]
    computed_t_stars = []

    for N in points:
        computed_t_star = solve_explicit(N, T, dt, alpha)
        computed_t_stars.append(computed_t_star)

    for i, N in enumerate(points):
        t_star = computed_t_stars[i]
        if t_star is not None:
            print(f"N = {N}: Center temperature reached u = 1 at t* = {t_star:.12f}")


    exact_t_star = 0.424011387033

    plt.figure(figsize=(8, 6))
    plt.plot(points, computed_t_stars, marker='o', linestyle='-', color='blue', label='Computed t*')
```

```
plt.axhline(y=exact_t_star, color='red', linestyle='--', label=f'Exact t* = {exact_t_star:.12f}')
plt.xlabel('N')
plt.ylabel('Computed Time t*')
plt.title('Convergence of t* against the actual time')
plt.legend()
plt.grid(True)
plt.show()
```

```
N = 100: Center temperature reached u = 1 at t* = 0.441610000000
N = 120: Center temperature reached u = 1 at t* = 0.438540000000
N = 150: Center temperature reached u = 1 at t* = 0.435530000000
N = 200: Center temperature reached u = 1 at t* = 0.432570000000
N = 250: Center temperature reached u = 1 at t* = 0.430820000000
N = 300: Center temperature reached u = 1 at t* = 0.429660000000
N = 317: Center temperature reached u = 1 at t* = 0.424000000000
```



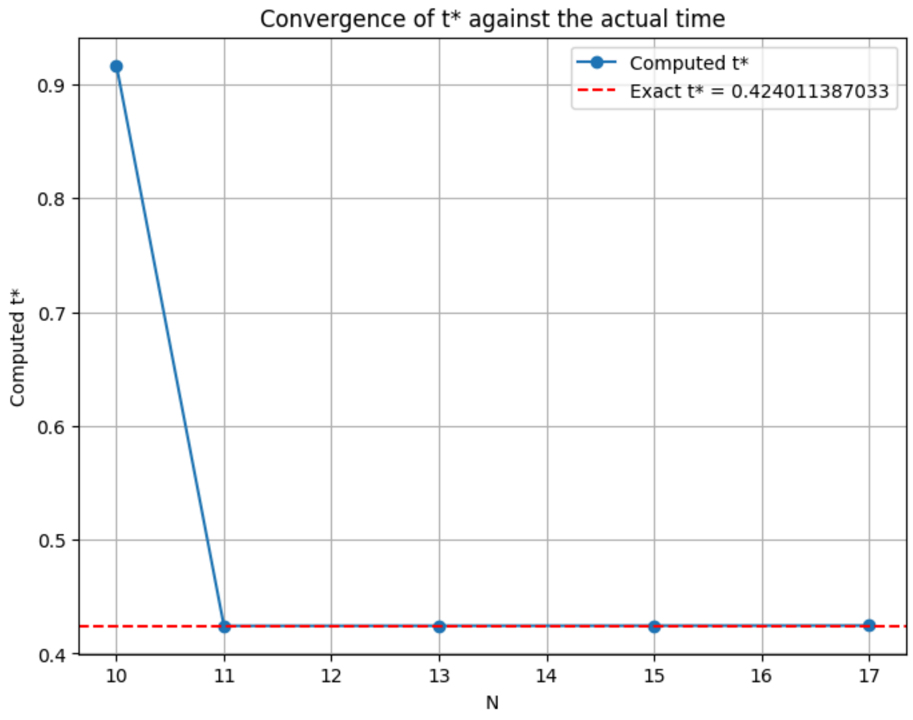
We observed that when N=317, our t* = 0.424000000000, which is the closest value to the exact time. We noticed also, that as we increase discretisation point the computed t* gets closer to the exact solution.

2) Implicit time-stepping

```
import numpy as np
from scipy.sparse import diags, identity
from scipy.sparse.linalg import spsolve
import matplotlib.pyplot as plt

def build_laplacian_matrix_implicit(n, h):
    N = n * n
    diagonals = []

    diagonals.append(-4 * np.ones(N))
    diagonals.append(np.ones(N - 1))
    diagonals.append(np.ones(N - 1))
    diagonals.append(np.ones(N - n))
    diagonals.append(np.ones(N - n))

    positions = [0, -1, 1, -n, n]

    for i in range(1, n):
        diagonals[1][i * n - 1] = 0
        diagonals[2][i * n - 1] = 0

    L = diags(diagonals, positions, shape=(N, N), format='csr') / (h ** 2)
    return L

# h = 2 / (10 + 1)
# A = build_laplacian_matrix(10,h)
# A.toarray()
```

```
def implicit_timestep(L, u, dt):
    I = identity(L.shape[0])
    A = I - dt * L
    return spsolve(A, u)


def solve_implicit(n, dt, T):
    u, h = initial_conditions(n)
    steps = int(T / dt)

    L = build_laplacian_matrix_implicit(n, h)

    I = identity(n * n, format='csr')
    M = I - dt * L

    center = (n // 2, n // 2)
    time = 0

    for _ in range(steps):
        u = spsolve(M, u.ravel()).reshape((n, n))

        u[:, 0] = 5.0         # We reapply boundary conditions after each time step to ensure that the fixed boundary values remain cons
        u[:, -1] = 0.0
        u[0, :] = 0.0
        u[-1, :] = 0.0
        u[0, 0] = 5.0
        u[-1, 0] = 5.0

        time += dt

        if u[center] >= 1.0:
            return time

    return time



T = 2.0  # let t=2
dt = 0.00001
alpha = 0.01

points = [ 10, 11, 13, 15, 17]
computed_t_stars = []

for n in points:
    t_star = solve_implicit(n, dt, T)
    computed_t_stars.append(t_star)
    print(f"N = {n}: Center temperature reached u = 1 at t* = {t_star:.12f}")

exact_t_star = 0.424011387033

plt.figure(figsize=(8, 6))
plt.plot(points, computed_t_stars, 'o-', label='Computed t*')
plt.axhline(y=exact_t_star, color='r', linestyle='--', label=f'Exact t* = {exact_t_star}')
plt.xlabel('N')
plt.ylabel('Computed t*')
plt.title('Convergence of t* against the actual time')
plt.legend()
plt.grid(True)
plt.show()
```

```
N = 10: Center temperature reached u = 1 at t* = 0.916569999998
N = 11: Center temperature reached u = 1 at t* = 0.424350000000
N = 13: Center temperature reached u = 1 at t* = 0.424430000000
N = 15: Center temperature reached u = 1 at t* = 0.424520000000
N = 17: Center temperature reached u = 1 at t* = 0.424640000000
```



we notice that the closest value we could reach to the exact solution is: t* = 0.424350000000. Moreover, the Backward Euler method (implicit) is unconditionally stable for the heat equation, that's why I didn't investigate the stability of it unlike before in (explixit time stepping)

Start coding or generate with AI.