Assingment 2: Solving two 1D problems

Juan López-Ríos

20121702

Part 1: Solving a wave problem with sparse matrices

Finite differences method

Solve the following (time-harmonic) wave problem:

$$\begin{aligned}\frac{d^2u}{dx^2} + k^2u = 0 \quad \text{in} \quad (0,1),\\ u = 0 \quad \text{if} \quad x = 0,\\ u = 1 \quad \text{if} \quad x = 1,\end{aligned}$$

with wavenumber $k = \frac{29\pi}{2}$.

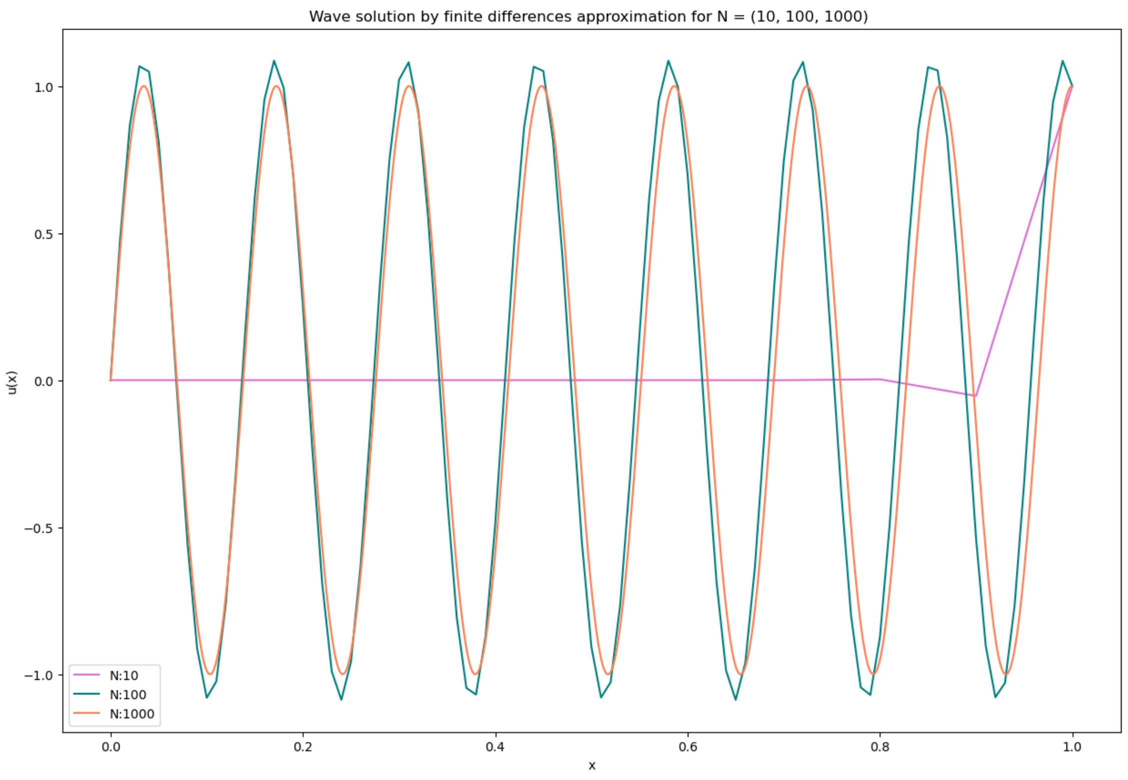
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse import coo_matrix
def wave_system(N):
    '''
    Constructs the sparse matrix A and boundary vector b for the di
scretized
    wave equation.

    Parameters:
    N : Number of discretization points.

    Returns:
    A : Sparse matrix representing the finite difference scheme.
    f : Boundary vector
    '''

    k = 29*np.pi/2
    h=1/N

    # f-vector
    f = np.zeros(N+1)
    for i in range(N+1):
        if i == N:
            f[i]=1.

    # A matrix
    rows = []
    cols = []
    data = []
    for i in range(N+1):
        for j in range(N+1):
            if i == 0 or i == N:
                if i == j:
                    rows.append(i)
                    cols.append(j)
                    data.append(1.)
            else:
                if i == j:
                    rows.append(i)
                    cols.append(j)
                    data.append(2-h**2 * k**2)
                if j == i+1 or j == i-1:
                    rows.append(i)
                    cols.append(j)
                    data.append(-1.)
    A = coo_matrix((data, (rows,cols)),(N+1,N+1))
    return A,f
```

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

u = []
x = []
# N = 10, 100, 1000
N_val = [10,100,1000]
for N in N_val:
    A,f = wave_system(N)
    A = csc_matrix(A) # Avoid warning (not necessary).
    u.append(spsolve(A,f)) # Approximations.
    x.append(np.linspace(0, 1, N+1)) # Evenly spaced values and u 
has been approxiamted for each one them.
```

Plot solutions for each value of N

In [3]:
```python
fig = plt.figure(figsize= (15,10))
# N = 10
plt.plot(x[0],u[0], color='orchid',label = f'N:{N_val[0]}')
# N = 100
plt.plot(x[1],u[1], color='teal',label = f'N:{N_val[1]}')
# N = 1000
plt.plot(x[2],u[2], color='coral',label = f'N:{N_val[2]}')
plt.title(f'Wave solution by finite differences approximation for N 
= {N_val[0], N_val[1], N_val[2]} ')
plt.xlabel('x')
plt.ylabel('u(x)')
plt.legend()
```

Out[3]:
```
<matplotlib.legend.Legend at 0x7fbad747ce50>
```



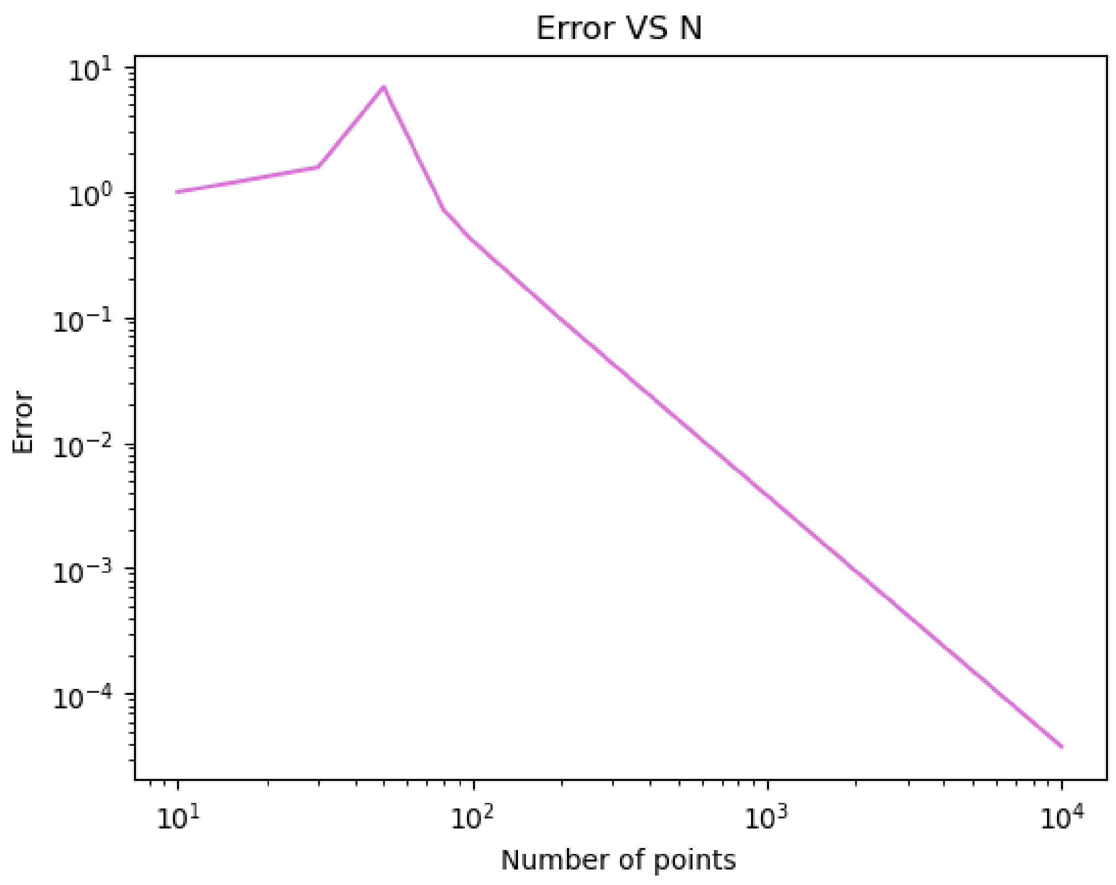
The approximation used for $\frac{d^2u}{dx^2}$ is exact for $N \rightarrow \infty$, which means that the greater the N value is the closest is gonna be to the true solution (better fit to actual solution).

Therefore, while increasing N to numbers larger than 1000 will yield a closer fit to the actual solution, there is a practical limit to how large N can be before computational constraints, such as memory limitations and processing power, prevent further improvements. In extreme cases, attempting to use an excessively large N can lead to a system crash or excessive computational times.

Error computation

In [4]:
```
def u_exact(N):
    '''
    Exact wave equation solution.
    '''
    k = 29*np.pi/2
    x = np.linspace(0,1, N+1)
    return np.sin(k*x)

def u_approx(N):
    '''
    Approximate solution of the wave equation.
    '''
    A,f = wave_system(N)
    return spsolve(csc_matrix(A),f)

def error(N):
    '''
    Error calculation max(|u_i - u_exact(x_i)|)
    '''
    u_e = u_exact(N)
    u =u_approx(N)
    return np.max(np.abs(u-u_e))

errors = []
# Set of N (discrete points)
N_values = [10,30,50,80,100,200,350,600,1000,1500, 5000, 10000]
for i in N_values:
    errors.append(error(i))


# Plot
plt.title('Error VS N')
plt.plot(N_values, errors, c='orchid')
plt.xscale("log")
plt.yscale("log")
plt.xlabel("Number of points")
plt.ylabel("Error")
```

```
Out[4]: Text(0, 0.5, 'Error')
```

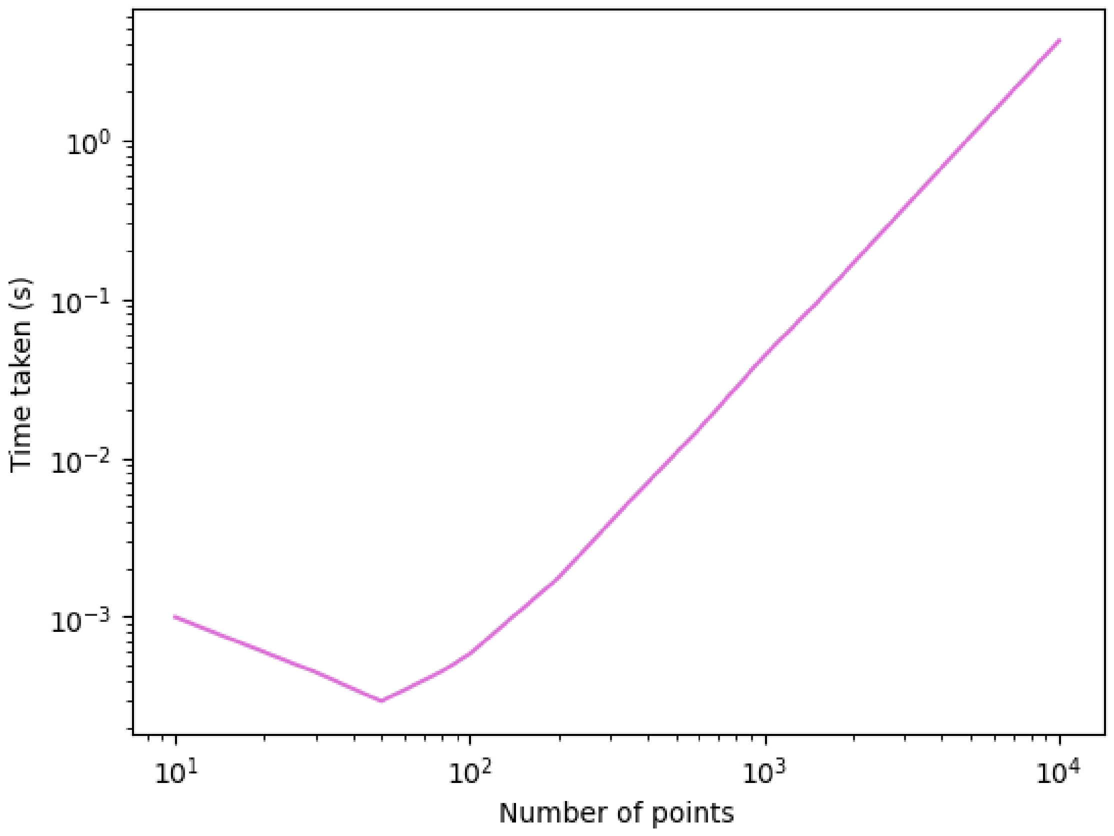


The bump at the beginning in error could be due to discretization error or numerical instabilities at lower values of N.

```
In [5]: from timeit import timeit
        # Time the computation of our solution takes.
        time = []
        for N in N_values:
            t = timeit("u_approx(N)", globals=globals(), number= 1)
            time.append(t)
```

In [6]:
```
plt.plot(N_values,time, c='orchid')
plt.xscale("log")
plt.yscale("log")
plt.xlabel("Number of points")
plt.ylabel("Time taken (s)")
```

Out[6]:
```
Text(0, 0.5, 'Time taken (s)')
```



N calculation for an error of 10e-8, using regression.

In [7]:
```
from scipy.stats import linregress

# Log of both sides.
log_N = np.log10(N_values)
log_y = np.log10(errors)

# Perform the linear regression.
slope, intercept, r_value, p_value, std_err = linregress(log_N, lo
g_y)

# Calculate 'a' from the intercept.
a = 10 ** intercept

# 'slope' is the exponent b.
b = slope


print(f"Estimated a: {a}")
print(f"Estimated b: {b}")
print(f"R-squared: {r_value**2}")

# Calculate N for a target error of 10e-8 using the regression mode
l.
target_error = 10e-8
N_target = 10 ** ((np.log10(target_error) - intercept) / slope)

n_exp=np.log10(N_target)
print(f"N is 10^({np.round(n_exp,2)})")
```

```
Estimated a: 753.1985380938613
Estimated b: -1.7571909005771942
R-squared: 0.9188340923144269
N is 10^(5.62)
```

Regression of the form $y = aN^b$ is performed to determine the coefficient a and exponent b. Once these parameters are obtained, the code calculates the specific value of N that would yield an error of 10^{-8}

In [8]:
```
# Based on the errors plot, a possible N value that could give an e
rror of 10e-8 could be 10e+8
# N needs to be an integer
#t = timeit("u_approx(int(N_target))", globals=globals(), number=
1)
#print(f"Time for wave equation approx of error less than 10e-8 is
{t}")
```

The creation of the matrix/vector, not the solving of the sparse system, is the computational bottleneck, taking over an hour. Since machine precision for numpy's default double-precision floats is about 2.22×10^{-16}, an error of 10^{-8} is achievable. However, `scipy.sparse.linalg.spsolve` might use lower precision, leading to larger errors than expected.

Part 2: Solving the heat equation with GPU acceleration

Heat equation:

$$\begin{aligned} \frac{\partial u}{\partial t} &= \frac{1}{1000}\frac{\partial^2 u}{\partial x^2}, && \text{for } x \in (0,1), \\ u(x,0) &= 0, && \text{if } x \neq 0 \text{ and } x \neq 1, \\ u(0,t) &= 10, \\ u(1,t) &= 10. \end{aligned}$$

- Represents a rod that starts at 0 temperature which is heated to a temperature of 10 at both ends

In [9]:
```
from numba import cuda
cuda.detect()
```

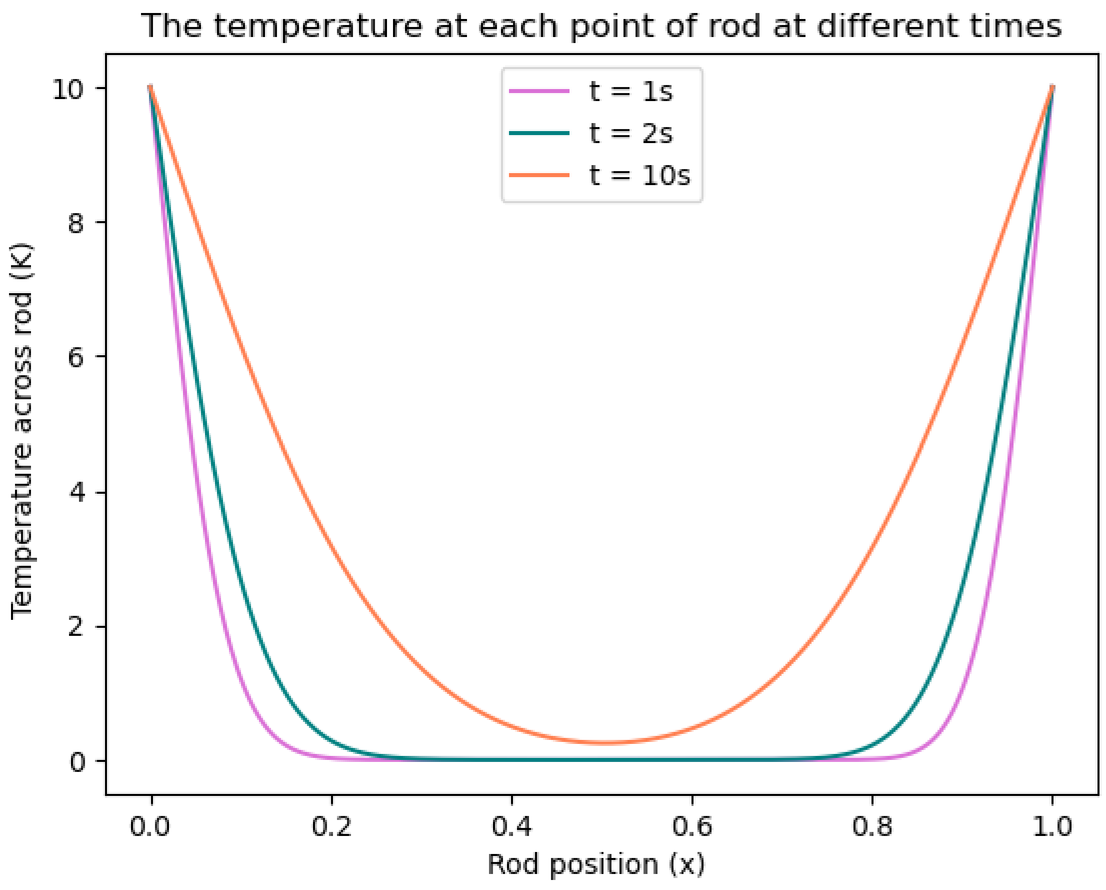
```
Found 1 CUDA devices
id 0    b'NVIDIA GeForce RTX 4070 Laptop GPU'
[SUPPORTED]
                      Compute Capability: 8.9
                           PCI Device ID: 0
                              PCI Bus ID: 1
                                    UUID: GPU-c31ae7e6-b548-cf71-b6
e6-452aad49be0c
                                Watchdog: Enabled
             FP32/FP64 Performance Ratio: 64
Summary:
        1/1 devices are supported
```

Out[9]:
```
True
```

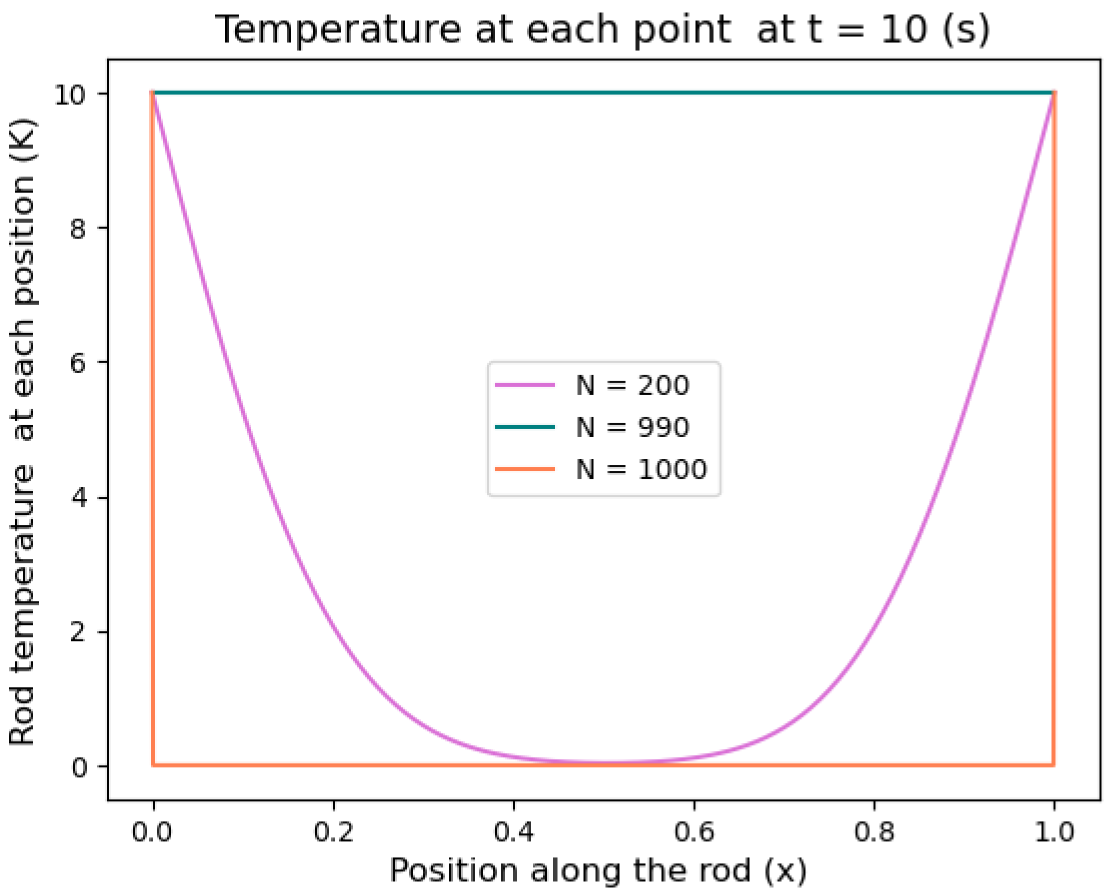
Heat equation (CPU)

In [10]:
```
def heat_equation(N, t):
    h = 1 / N
    k = 1/(1000*h)
    # Initial conditions temperature distribution
    # Ends of rod at 10 (K?) and the rest at 0 (K?)
    u = [0] * (N+1)
    u[0] = u[N] = 10.0

    for j in range(N*t):
        for i in range(1,N):
            u[i] = u[i] + k*(u[i-1]-2*u[i]+u[i+1])
    return u

N = 500
t1=1
t2=2
t3=10
x = np.linspace(0,1,N+1)
y1 = heat_equation(N,t1)
y2 = heat_equation(N,t2)
y3 = heat_equation(N,t3)
plt.plot(x,y1, color = "orchid")
plt.plot(x,y2, color = "teal")
plt.plot(x,y3, color = "coral")
plt.legend([f"t = {t1}s", f"t = {t2}s", f"t = {t3}s"])
plt.ylabel("Temperature across rod (K) ")
plt.xlabel("Rod position (x) ")
plt.title("The temperature at each point of rod at different time
s")
```

Out[10]:
```
Text(0.5, 1.0, 'The temperature at each point of rod at different t
imes')
```



In [11]:
```
N1 = 200
N2 = 990
N3 = 1000
x1 = np.linspace(0,1,N1+1)
x2 = np.linspace(0,1,N2+1)
x3 = np.linspace(0,1,N3+1)
plt.plot(x1, heat_equation(N1,10), color = "orchid")
plt.plot(x2, heat_equation(N2,10), color = "teal")
plt.plot(x3, heat_equation(N3,10), color = "coral")
plt.legend(["N = 200", "N = 990","N = 1000",])
plt.ylabel("Rod temperature  at each position (K)", size = "12")
plt.xlabel("Position along the rod (x) ", size = "12")
plt.title("Temperature at each point  at t = 10 (s)", size = "14")
plt.show()
```



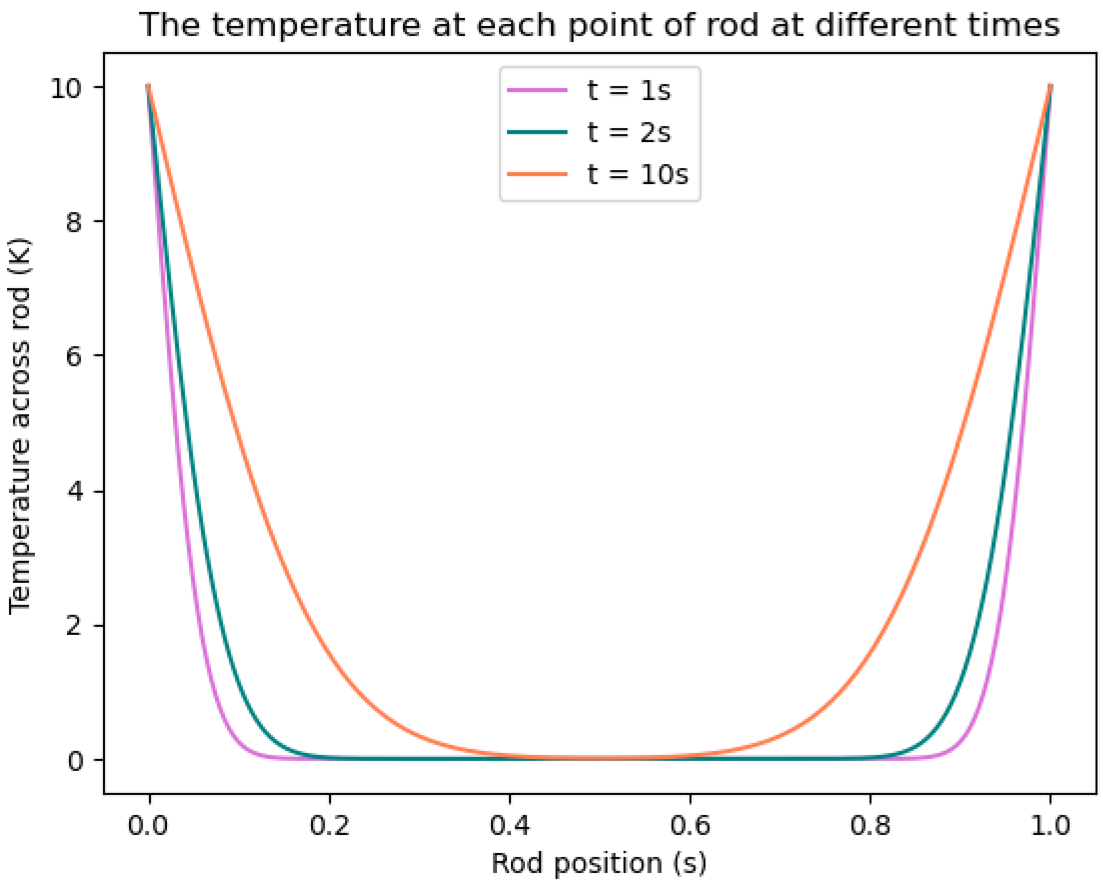
Numba CUDA implementation

In [12]:
```
import numba

# Initialize variables
N = 200
t = 600
h = 1/N

TPB = N+1 # Threads per block, one for each spatial division plus b
oundary
BPG = 1 # All threads in a single block

@cuda.jit
def heat_system(u):
    index = cuda.grid(1) # Thread's unique index in a 1D grid (for
parallel)
     # Shared array
    sh_array = cuda.shared.array(TPB, numba.float32)

    for j in range(t*N):
        sh_array[index] = u[j,index]
        cuda.syncthreads() # Barrier synchronization

        # Heat calculation
        # Initial conditions (boundaries)
        if index == 0:
            u[j+1, index] = numba.float32(10)
        elif index == N:
            u[j+1, index] = numba.float32(10)
        else:
            u[j+1,index] = sh_array[index] + (1/(1000*h))*(sh_arra
y[index-1] - 2*sh_array[index] + sh_array[index+1])
        cuda.syncthreads()


u = np.zeros(((N*t)+1, N+1))
u[0, 0] = 10
u[0, N] = 10

result = cuda.to_device(u.astype('float32'))
heat_system[BPG, TPB](result)

u_sol1 = result.copy_to_host()
```

```
/home/juan/anaconda3/envs/env/lib/python3.11/site-packages/numba/cu
da/dispatcher.py:536: NumbaPerformanceWarning: Grid size 1 will lik
ely result in GPU under-utilization due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
```

In [13]:
```
t1=1
t2=2
t3=10
x_rod = np.linspace(0,1,N+1)
plt.plot(x_rod,u_sol1[t1*N], color = "orchid")
plt.plot(x_rod,u_sol1[t2*N], color = "teal")
plt.plot(x_rod,u_sol1[t3*N], color = "coral")
plt.legend([f"t = {t1}s", f"t = {t2}s", f"t = {t3}s"])
plt.ylabel("Temperature across rod (K)")
plt.xlabel("Rod position (s)")
plt.title("The temperature at each point of rod at different time
s")
```

Out[13]:
```
Text(0.5, 1.0, 'The temperature at each point of rod at different t
imes')
```



The chosen value of N (200) serves a double purpose: firstly, it is sufficiently large to ensure that the temperature distribution appears continuous when graphed, its a good fit to the heat equation solution, facilitating a comparative analysis of the temperature variation over time. Secondly, it is constrained to avoid excessive computational load for merely plotting purposes. Note that above N=1000, the results begin to diverge as the incremental steps become very small.

It is important to notice that the constant k determines the staibility if the equation, for N = 1000, the equation becomes unstable, becuase k become sunity, and closer values of k to unity are more unstable whereas smaller values K=200/1000, are more stable as it can be appreciated in teh plots.

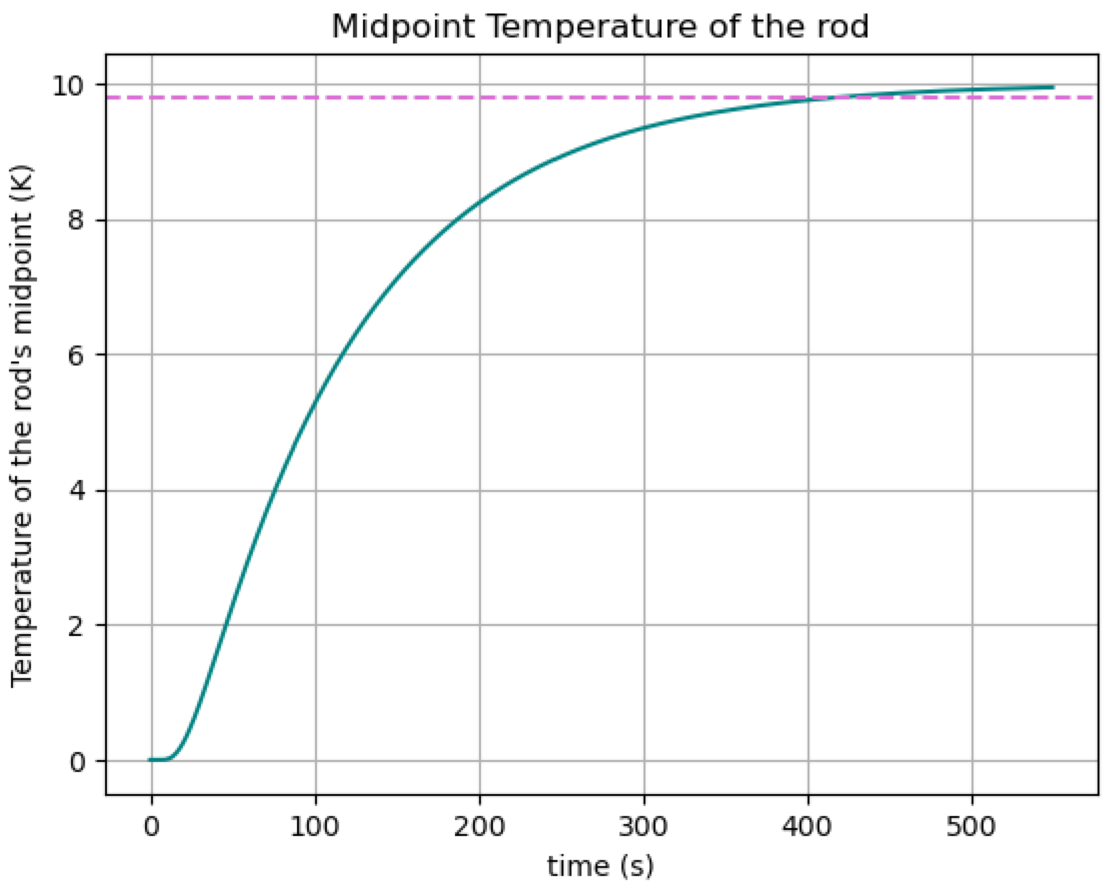
In the context of GPU parallelization with numba.cuda, it's important to strategize the transfer of data between the CPU and GPU memory. Transfers should be minimized to only when absolutely necessary to optimize performance. Given that N must remain below 1000 to prevent divergence and considering the maximum thread limit of a single block is 1024, a single block can efficiently handle the update of each discretized cell, as one thread can be assigned to each of the N+1 cells.

The code's kernel function `heat_system` computes the heat distribution, maintaining fixed boundary conditions, with final results copied back to the CPU. This method ensures that GPU resources are effectively utilized for the simulation.

In [14]:
```
# Midpoint temperature calculation.
midpoint = int(N/2)
times = np.arange(0,550,1)
mid_temp = []

for i in times:
    mid_temp.append(u_sol1[N*i][midpoint])
```

In [15]:
```python
plt.plot(times, mid_temp, color='teal')
plt.xlabel("time (s)")
plt.ylabel("Temperature of the rod's midpoint (K)")
plt.axhline(y=9.8, color='orchid', linestyle='--')
plt.grid()
plt.title("Midpoint Temperature of the rod")
```

Out[15]:
```
Text(0.5, 1.0, 'Midpoint Temperature of the rod')
```



In [16]:
```
timer = np.arange(390, 430, 1)
for t in timer:
    temp = u_sol1[N*t][midpoint]
    if temp > 9.8:
        print(f"Time is {t} and temperature is {temp}")
        print("Achieved! at time {}".format(t))
        break
    else:
        print(f"Time is {t} and temperature is {temp}")
```

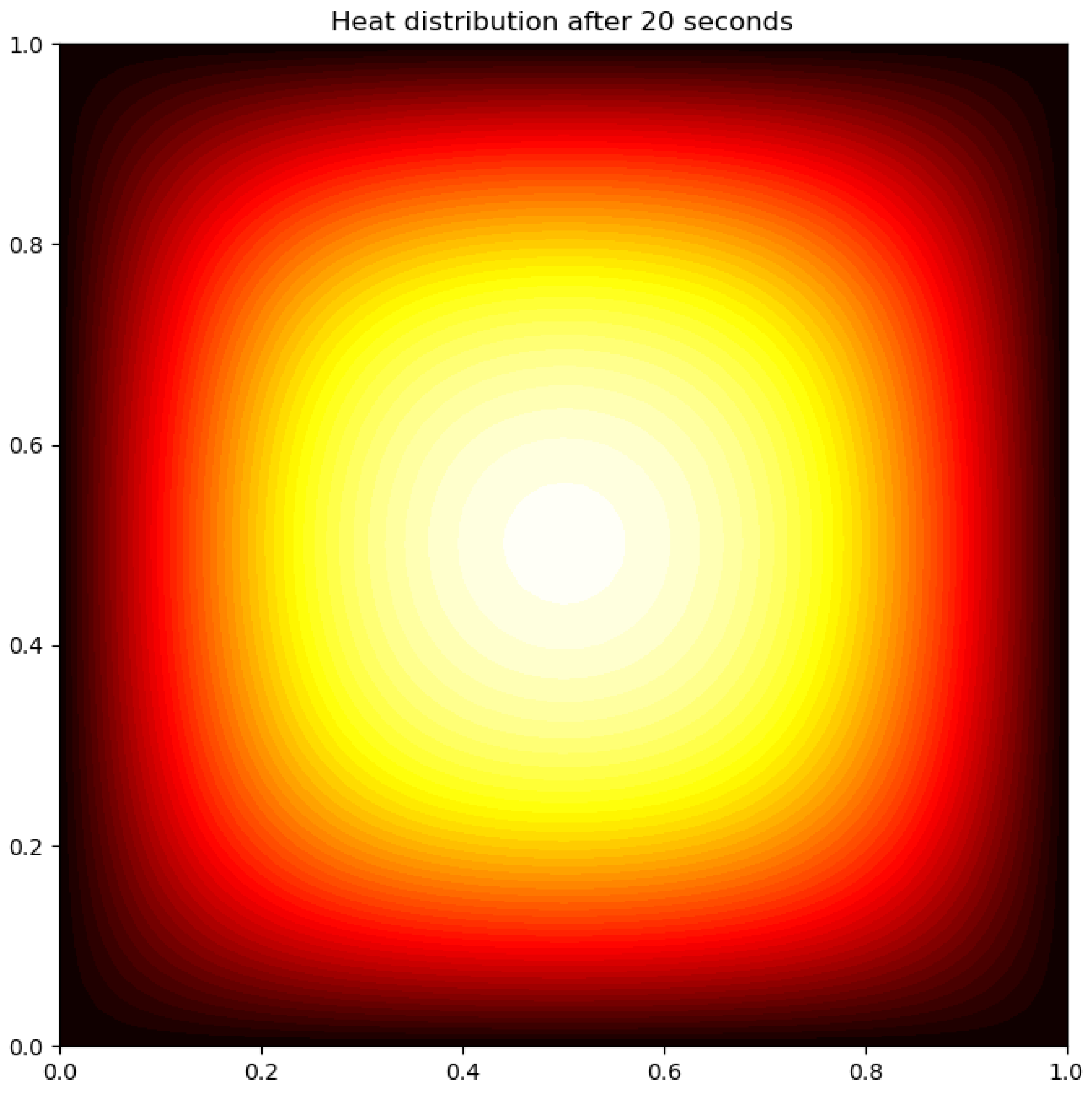
```
Time is 390 and temperature is 9.727570533752441
Time is 391 and temperature is 9.730240821838379
Time is 392 and temperature is 9.732911109924316
Time is 393 and temperature is 9.735581398010254
Time is 394 and temperature is 9.738175392150879
Time is 395 and temperature is 9.740713119506836
Time is 396 and temperature is 9.743192672729492
Time is 397 and temperature is 9.745672225952148
Time is 398 and temperature is 9.748151779174805
Time is 399 and temperature is 9.750631332397461
Time is 400 and temperature is 9.753110885620117
Time is 401 and temperature is 9.755576133728027
Time is 402 and temperature is 9.757960319519043
Time is 403 and temperature is 9.760289192199707
Time is 404 and temperature is 9.762578010559082
Time is 405 and temperature is 9.764866828918457
Time is 406 and temperature is 9.767155647277832
Time is 407 and temperature is 9.769444465637207
Time is 408 and temperature is 9.771733283996582
Time is 409 and temperature is 9.774022102355957
Time is 410 and temperature is 9.7762451171875
Time is 411 and temperature is 9.778438568115234
Time is 412 and temperature is 9.780536651611328
Time is 413 and temperature is 9.782634735107422
Time is 414 and temperature is 9.784732818603516
Time is 415 and temperature is 9.78683090209961
Time is 416 and temperature is 9.788928985595703
Time is 417 and temperature is 9.791027069091797
Time is 418 and temperature is 9.79312515258789
Time is 419 and temperature is 9.795161247253418
Time is 420 and temperature is 9.797163963317871
Time is 421 and temperature is 9.799091339111328
Time is 422 and temperature is 9.80099868774414
Achieved! at time 422
```

As it can be appreciated based on the computational experiment, the midpoint of the rod will achieve a temperature of 9.8 K (assuming we are dealing with Kelvin) at the 422 timestep (s).

Extensions

2D animation implementation of heat equation

In [17]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

def heat_equation_2d(N, time_steps):
    h = 1 / N
    k = 1/ (1000 * h)
    # Similar initial conditions that in Part 2, buta square bounda
ry at 10 K.
    u = np.zeros((N+1, N+1))
    u[0, :] = u[:, 0] = u[N, :] = u[:, N] = 10  # Set boundary con
ditions to 10 Kelvin.

    # Time dimension.
    for _ in range(time_steps*N):
        # Space dimension.
        for j in range(1, N):
            for i in range(1, N):
                u[j, i] = u[j, i] + k * (u[j-1, i] + u[j+1, i] + u
[j, i-1] + u[j, i+1] - 4*u[j, i])
    return u

N = 50
total_time = 20  # Total simulation time in seconds.

# Meshgrid for plotting.
x = np.linspace(0, 1, N+1)
y = np.linspace(0, 1, N+1)
X, Y = np.meshgrid(x, y)


fig, ax = plt.subplots(figsize=(8, 8))

# Function to update each frame in the animation.
def update(frame):
    ax.clear()  # Clear the previous frame.
    u = heat_equation_2d(N, frame)  # Calculate the heat distributi
on at the current frame.
    contour = ax.contourf(X, Y, u, levels=50, cmap='hot_r') # Inve
rse hot colormap.
    ax.set_title(f'Heat distribution after {frame} seconds')
    return contour

# Animation using FuncAnimation.
ani = FuncAnimation(fig, update, frames=total_time + 1, interval=1
00)

# Save GIF
ani.save('heat_distribution.gif', writer='pillow', fps=1)

plt.show()
```


Heat distribution after 20 seconds
0.0
0.2
0.4
0.6
0.8
1.0