Assignment 2 - Solving two 1D problems

November 4, 2022

1 Assignment 2- Solving two 1D problems

[ ]:
```
# imports
import numpy as np
import math
import time
import matplotlib.pyplot as plt
from scipy.sparse import coo_matrix
from scipy.sparse.linalg import spsolve


plt.rcParams['figure.figsize'] = [8, 6]
```

[ ]:
```
# for cuda debugging

# import os

# # needs to appear before from numba import cuda
# os.environ["NUMBA_ENABLE_CUDASIM"] = "1"
# # set to "1" for more debugging, but slower performance
# os.environ["NUMBA_CUDA_DEBUGINFO"] = "1"

import numba
from numba import cuda

cuda.detect()
```

```
Found 1 CUDA devices
id 0             b'Tesla T4'                              [SUPPORTED]
                      Compute Capability: 7.5
                           PCI Device ID: 4
                              PCI Bus ID: 0
                                    UUID:
GPU-524cd7f0-9a20-a7cc-a34e-bd9e5b2665c8
                                Watchdog: Disabled
             FP32/FP64 Performance Ratio: 32
Summary:
        1/1 devices are supported
```

[ ]: True

1.1 Part 1: Solving a wave problem with sparse matrices

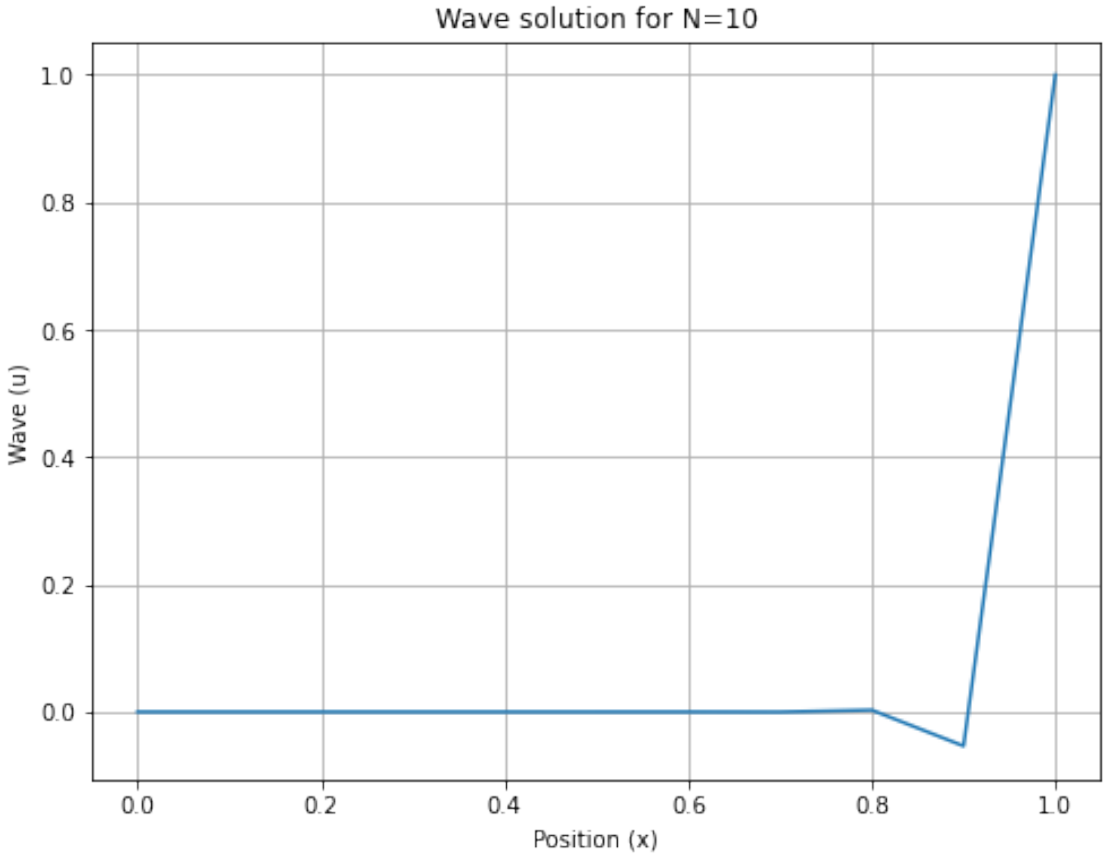
[ ]:
```
# wavenumber
k = 29*math.pi/2
```

1.a Write a Python function that takes N as an input and returns the matrix A and vector f.

[ ]:
```
def make_matrix_sparse(N):
    """
    Returns the (sparse) matrix and the vector needed to solve
    the wave problem using linear algebra
    """
    # constants
    h = 1/N
    diag_el = 2 - (h*k)**2

    # Initializing A and f
    rows = []
    cols = []
    data = []
    f = np.zeros((N+1))

    # Filling vector f
    f[-1] = 1.0


    # Filling sparse matrix A
    rows.append(0)
    cols.append(0)
    data.append(1.0)

    for i in range(1, N):
        rows += [i, i, i]
        cols += [i, i+1, i-1]
        data += [diag_el, -1.0, -1.0]

    rows.append(N)
    cols.append(N)
    data.append(1.0)


    # creating sparse matrix in CSR format
    A = coo_matrix((data, (rows, cols)), (N+1, N+1)).tocsr()

    return A, f
```

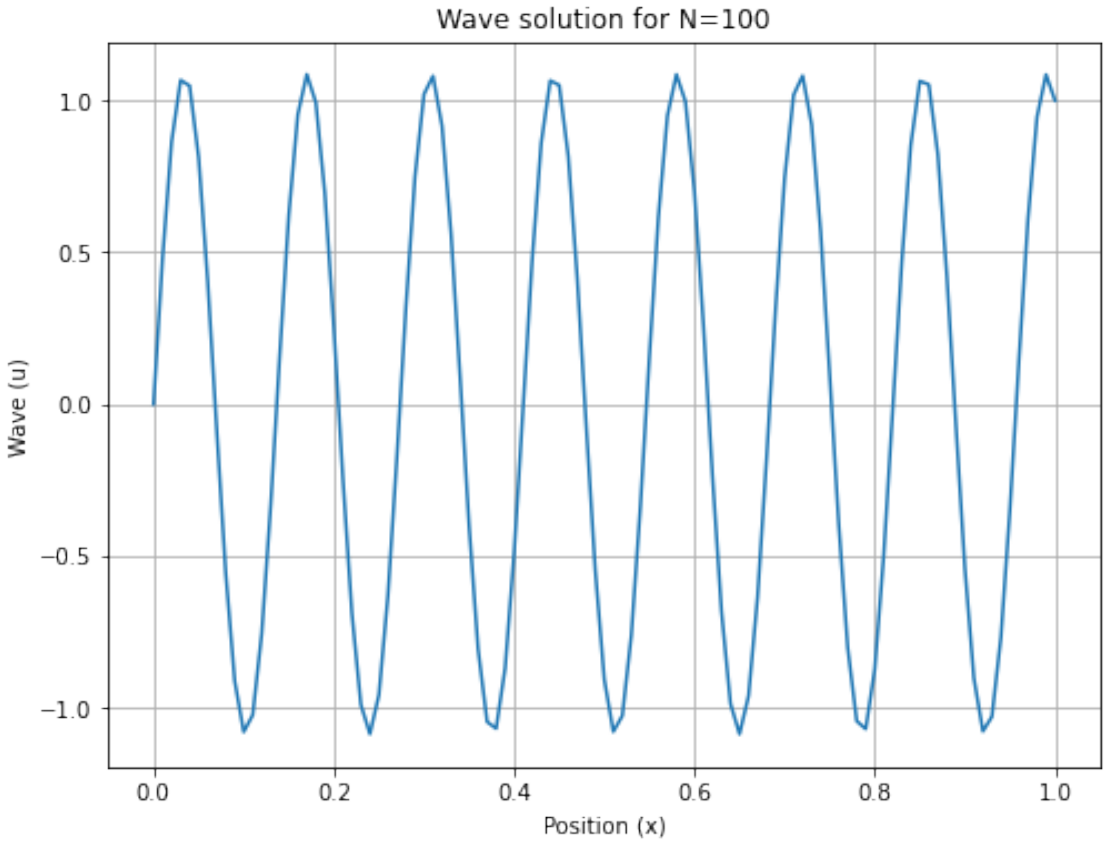
1.b Compute the approximate solution for your problem for N = 10, 100 and 1000. Plot the solutions for these three values of N

[ ]:
```
N = 10
A, f = make_matrix_sparse(N)
sol = spsolve(A, f)

x = np.linspace(0, 1, N+1)

plt.plot(x, sol)
plt.grid()
plt.xlabel("Position (x)")
plt.ylabel("Wave (u)")
plt.title("Wave solution for N="+str(N))
plt.show()
```



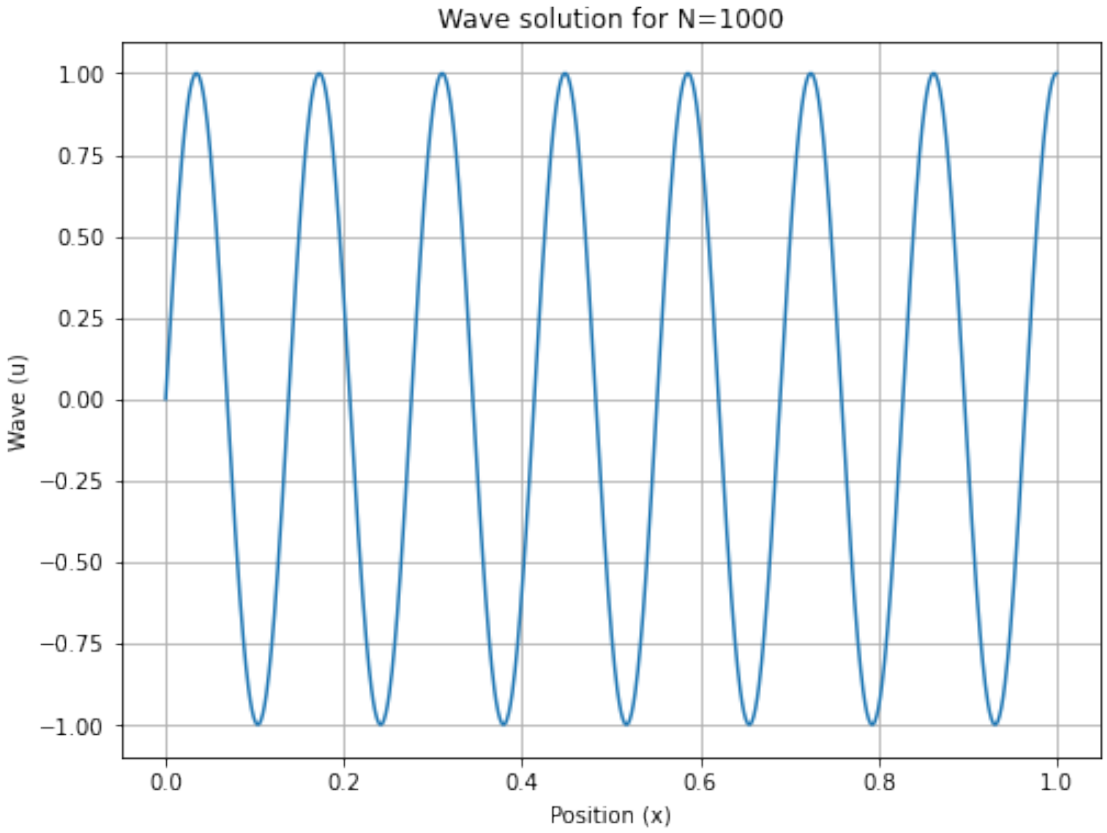
[ ]:
```
N = 100
A, f = make_matrix_sparse(N)
sol = spsolve(A, f)
```

```
x = np.linspace(0, 1, N+1)

plt.plot(x, sol)
plt.grid()
plt.xlabel("Position (x)")
plt.ylabel("Wave (u)")
plt.title("Wave solution for N="+str(N))
plt.show()
```



[ ]:
```
N = 1000
A, f = make_matrix_sparse(N)
sol = spsolve(A, f)

x = np.linspace(0, 1, N+1)

plt.plot(x, sol)
plt.grid()
plt.xlabel("Position (x)")
plt.ylabel("Wave (u)")
plt.title("Wave solution for N="+str(N))
```

```
plt.show()
```



1.c Briefly (1-2 sentences) comment on your plots

The approximation used for the second derivative of u with respect to x is exact for $N \to \infty$, which means that approximations with the largest value for N will be closest to the true solution. Hence, we expect N=1000 to be closest to the true solution, while N=10 is furthest (we can see from the plot that N=10 is so small that the resulting solution does not look like a wave a.k.a. the approximation is not valid for such small N).

1.d Compute this error for a range of values for N of your choice, for the method you wrote above. Plot N against the error in your solution.

[ ]:
```
def get_exact_sol(x):
    """
    Returns the true solution to the wave problem
    """
    return np.sin(k*x)

def measure_err(appr, true):
    """
    Computes an measure of the error in the approximation
```

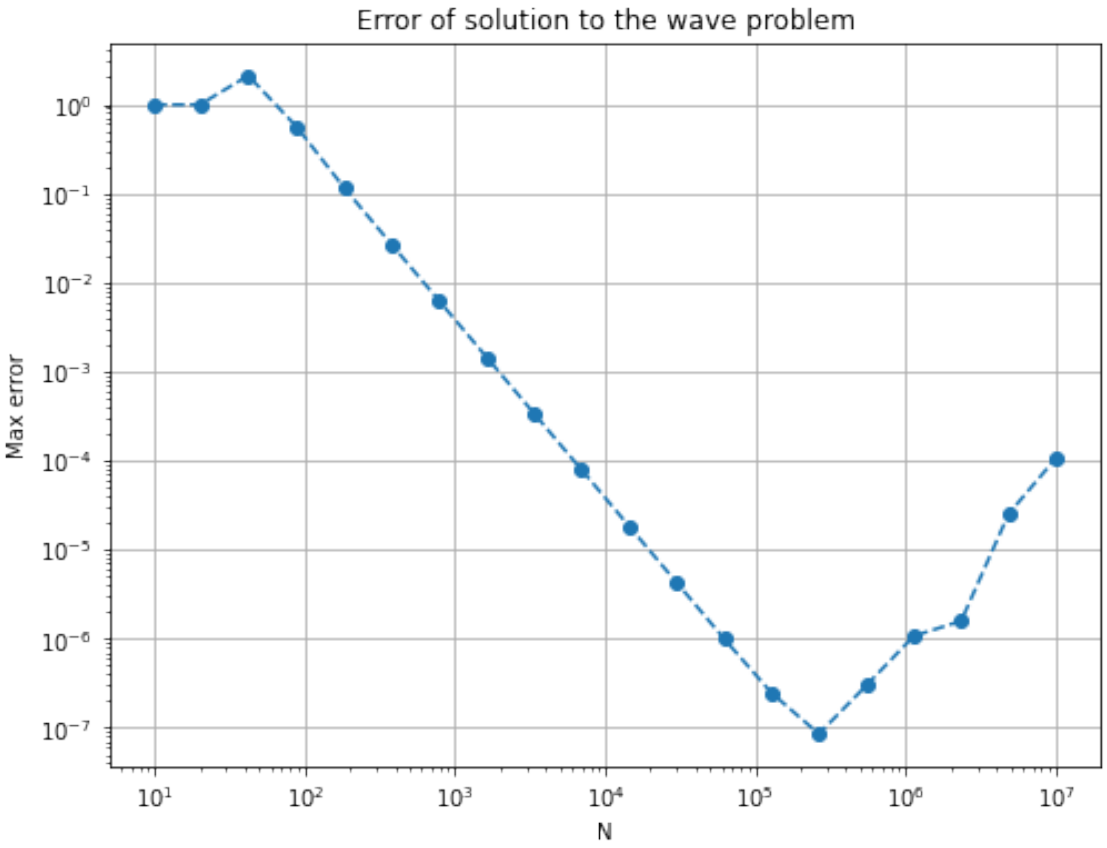
```
    of a function as the maximum difference between the
    approximation and the truth at each point in space.
    """
    return np.max(appr-true)
```

[ ]:
```
N_vals = [10**i for i in range(1, 8)]
N_vals = np.power(10, np.linspace(1, 7, 20)).astype("int")

print(N_vals)
```

```
[      10       20       42       88      183      379      784     1623
     3359     6951    14384    29763    61584   127427   263665   545559
  1128837  2335721  4832930 10000000]
```

[ ]:
```
errors = np.zeros((len(N_vals)))

for i in range(len(N_vals)):
    print("Round",i+1,"/",len(N_vals), end="\r")

    N = N_vals[i]

    # Finding approximate solution
    A, f = make_matrix_sparse(N)
    approx_sol = spsolve(A, f)

    # Computing exact solution
    x = np.linspace(0, 1, N+1)
    exact_sol = get_exact_sol(x)

    # Calculating and storing error
    errors[i] = measure_err(approx_sol, exact_sol)
```

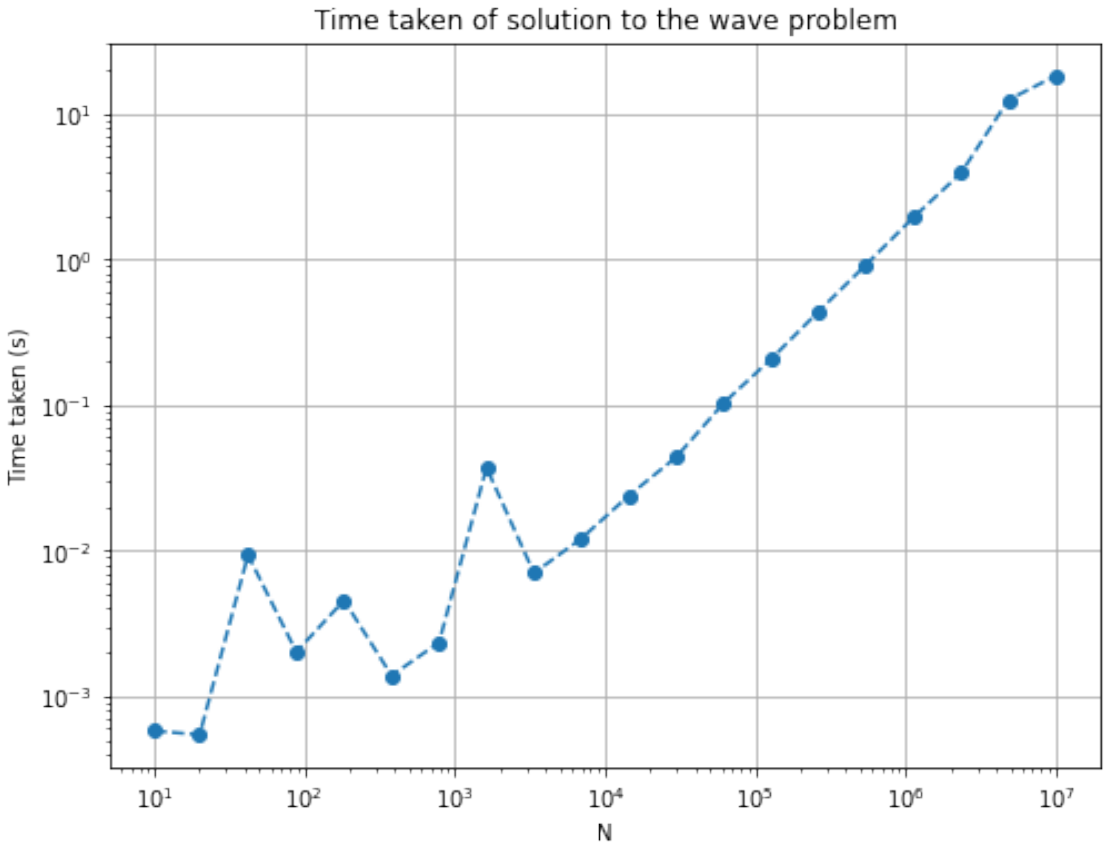
[ ]:
```
plt.plot(N_vals, errors, 'o--')
plt.xscale("log")
plt.yscale("log")
plt.grid()
plt.xlabel("N")
plt.ylabel("Max error")
plt.title("Error of solution to the wave problem")
plt.show()
```



1.g Measure the time taken to compute your approximation for your function. Plot N against the time taken to compute a solution

[ ]:
```
times = np.zeros((len(N_vals)))

for i in range(len(N_vals)):
    print("Round",i+1,"/",len(N_vals), end="\r")
    N = N_vals[i]

    # Timing and storing
    start = time.time()

    A, f = make_matrix_sparse(N)
    approx_sol = spsolve(A, f)

    end = time.time()
    time_taken = end - start
    times[i] = time_taken
```

[ ]:
```
plt.plot(N_vals, times, 'o--')
plt.xscale("log")
plt.yscale("log")
plt.grid()
plt.xlabel("N")
plt.ylabel("Time taken (s)")
plt.title("Time taken of solution to the wave problem")
plt.show()
```



1.i Pick a value of N that you would expect to give error of 10^-8 or less. Briefly (1-2 sentences) explain how you picked your value of and predict how long the computation will take.

We see that the max error comes to a minimum around $N = 10^{5.5}$ before increasing. If we extrapolate, we'd expect the increasing trend to continue, such that we are most likely to find an error of 10^{-8} or less around the minimum. Hence, we pick $N = 10^{5.5}$. This corresponds to a time taken of about 0.5 seconds.

1.j Compute the approximate solution with your value of N

[ ]:
```
N = int(10**5.5)
```

```
# Finding approximate solution and measuring time taken
start = time.time()
A, f = make_matrix_sparse(N)
approx_sol = spsolve(A, f)
end = time.time()

# Computing exact solution
x = np.linspace(0, 1, N+1)
exact_sol = get_exact_sol(x)

# Calculating error and time taken
error = measure_err(approx_sol, exact_sol)
time_taken = end - start
```

[ ]:
```
print("Computing a solution for $N={0}$ took {1} seconds and yielded an error␣
↪of {2}".format(N, time_taken, error))
```

```
Computing a solution for $N=316227$ took 0.5820028781890869 seconds and yielded
an error of 1.086765301187842e-07
```

1.k Briefly (1-2 sentences) comment on how these compare to your predictions.

Since $N = 10^{5.5}$ was already plotted, the prediction for time taken was spot on (0.3 seconds predicted vs 0.6 seconds obtained). The goal of an error of 10^{-8} or less was not reached, as we obtained ~10^{-7}, which seems to be the best we can obtain given our interpretive extrapolation of the error graph.

The increase in error past $N = 10^{5.5}$ (and hence the reason why there aren't any errors of 10^{-8} for higher N) may be due to an ensuring decrease in the accuracy of $\frac{d^2u_i}{dx^2}$ due to the finite precision of floats (whose end digits matter more and more as the spacing between adjacent space points $h = 1/N$ decreases).

1.2 Part 2: Solving the heat equation with GPU acceleration

2.a Implement this iterative scheme in Python. You should implement this as a function that takes N as an input.

[ ]:
```
def next_timestep(rod, N, k):
    """
    Gives the temperature state of the rod
    at the following time step.

    k = 1 / (h * 1000)
    """
    # note: k is passed as input so that it doesn't
    # need to be re-computed at each timestep in the "iterate_rod" function

    new_rod = np.zeros(rod.shape)
```

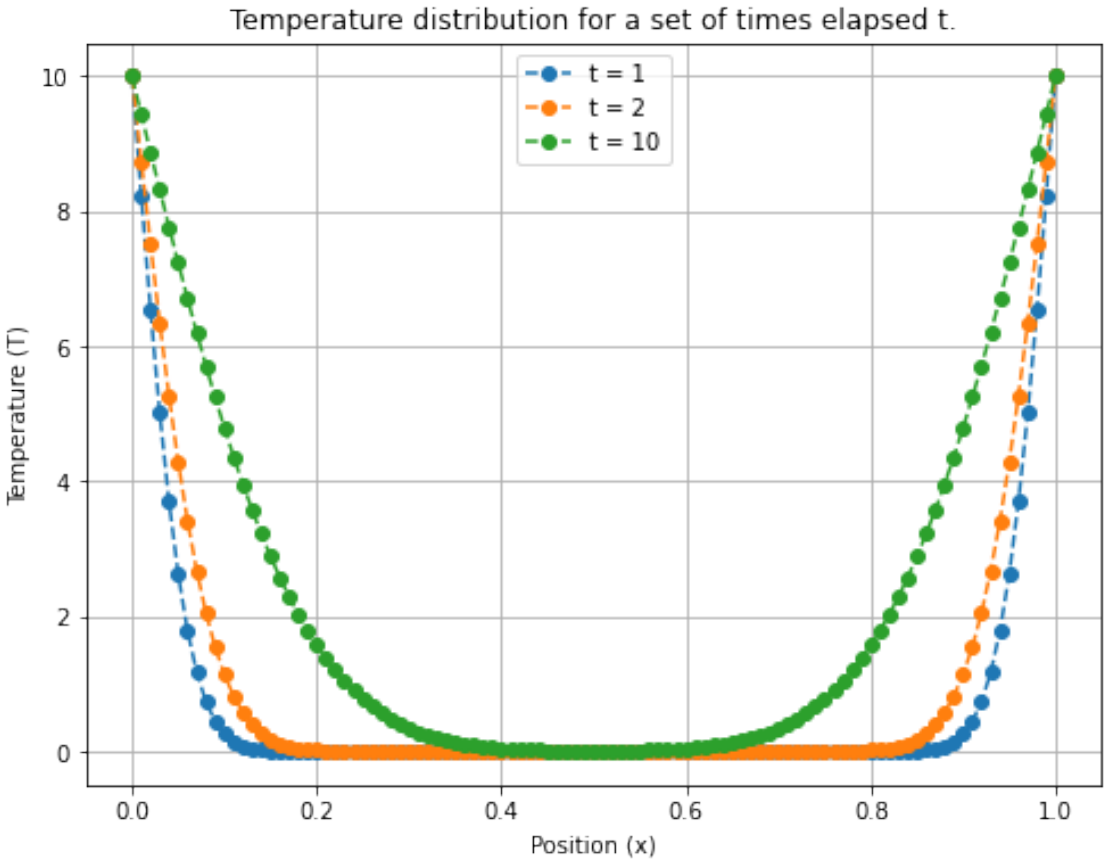
```
    for i in range(N+1):
        if i==0 or i==N:
            new_rod[i] = 10

        else:
            new_rod[i] = rod[i] + k * (rod[i-1] - 2*rod[i] + rod[i+1])

    return new_rod


def iterate_rod(N, t_end):
    """
    Runs a simulation of heat flowing
    through rod (1D solid) according to the heat equation
    for a discretisation of N+1 points and up to time t=t_end
    """
    h = 1/N
    k = 1/(1000*h)
    iters = int(t_end / h)

    rod_state = np.zeros((N+1))

    for t in range(iters):
        rod_state = next_timestep(rod_state, N, k)

    return rod_state
```

2.b Using a sensible value of N, plot the temperature of the rod at t=1, 2, 10. Briefly (1-2 sentences) comment on how you picked a value for N.

[ ]:
```
# Running the temperature in the rod until t = 1, 2, 10

N = 100
t_ends = [1, 2, 10]

rods = np.zeros((len(t_ends), N+1))

for i in range(len(t_ends)):
    t_end = t_ends[i]

    rods[i] = iterate_rod(N, t_end)
```

[ ]:
```
# Ploting the rod at states t = 1, 2, 10
x = np.linspace(0, 1, N+1)

for i in range(len(t_ends)):
    plt.plot(x, rods[i], 'o--', label="t = {0}".format(t_ends[i]))
```

```
plt.xlabel("Position (x)")
plt.ylabel("Temperature (T)")
plt.title("Temperature distribution for a set of times elapsed t.")
plt.grid()
plt.legend()
plt.show()
```



The value for N was picked so that: 1. It is just large enough for the temperature distribution to look continuous when plotted on a graph; larger than that would mean an unecessary amount of computational power used for a simple comparative plot of the temperature distribution across times elaplsed. 2. For larger N, the result starts to diverge as $1/1000h$ stops being small (and actually goes passed 1 at $N = 1000$).

2.c Use numba.cuda to parallelise your implementation on a GPU. You should think carefully about when data needs to be copied, and be careful not to copy data to/from the GPU when not needed.

We just saw that N cannot exceed 1000 without diverging, and since blocks can have up to 1024 threads, it means that one block is sufficient if each of its threads takes care of updating one of the $N + 1$ cells.

[62]:
```
# Independent params
N = 100
t_end = 15

# Dependent params
h = 1/N
k = 1/(h*1000)
timesteps = int(t_end/h)

# Initialising arrays
init_rod = np.zeros(N+1)
result_rod = np.zeros(N+1)
d_init_rod = cuda.to_device(init_rod)
d_result_rod = cuda.to_device(result_rod)

# Checking the N+1 cells fit in one block (blocks have a max of 1024 threads)
assert N+1 <= 1024

threadsperblock = 16 * (int(N/16) + 1) # Picking minimum number that's a␣
 ↪multiple of 16
blockspergrid = 1


@cuda.jit
def cuda_iterate_rod(init_rod, result_rod, timesteps, k):
    """
    Runs a simulation of heat flowing
    through rod (1D solid) according to the heat equation
    for a discretisation of N+1 points and up to time t=t_end
    """
    pos = cuda.grid(1)
    rod_length = len(init_rod)

    # Initializing fixed size for shared array as their size
    # must be set at compile time so can't hold "rod_length"
    # Using two arrays to:
    # - cater for racing conditions,
    # - store the current rod state and the next rod state
    rod_1 = cuda.shared.array((threadsperblock), numba.float32)
    rod_2 = cuda.shared.array((threadsperblock), numba.float32)

    # copying rod values to shared memory (for faster access)
    rod_1[pos] = init_rod[pos]
    rod_2[pos] = result_rod[pos]
    cuda.syncthreads()
```

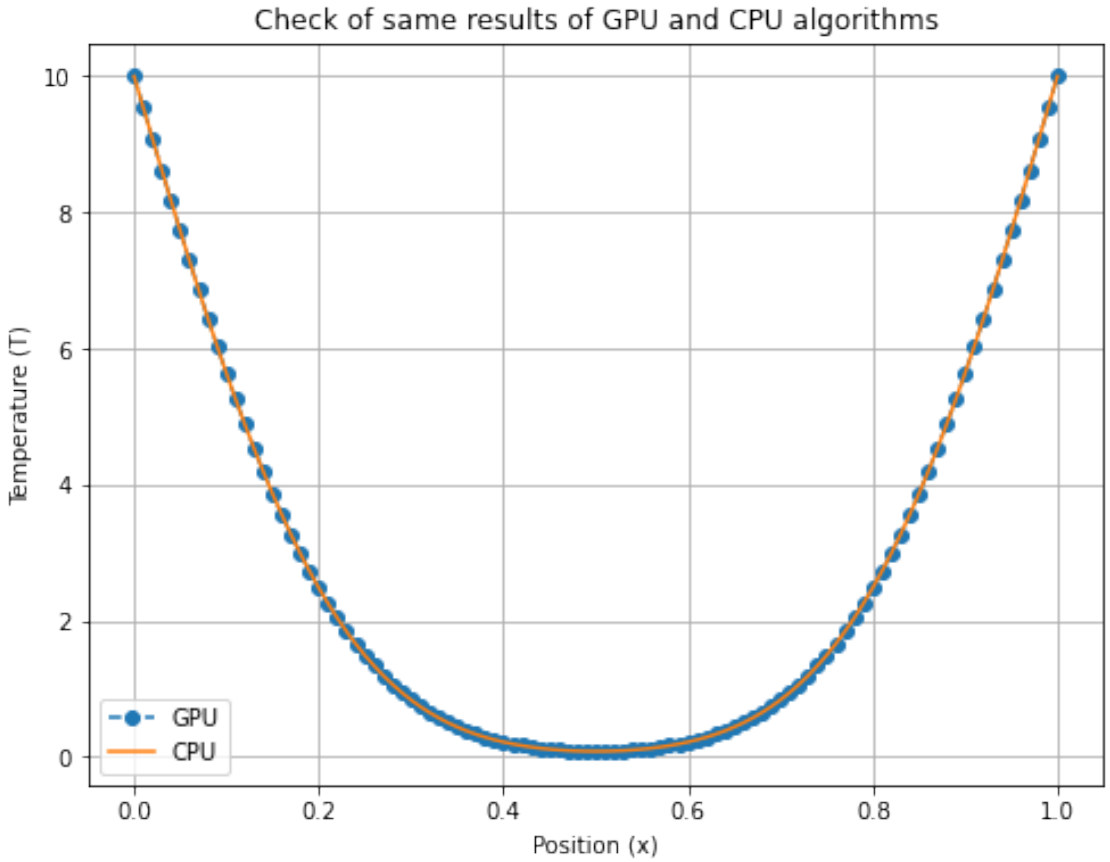
```
    # Stopping unecessary threads
    if pos >= rod_length:
        return

    for t in range(timesteps):
        # Swapping array references after every iteration
        if (t % 2) == 0:
            curr_rod = rod_1
            next_rod = rod_2
        else:
            curr_rod = rod_2
            next_rod = rod_1

        # Computing next rod
        if pos == 0 or pos == rod_length - 1:
            next_temp = 10
        else:
            next_temp = curr_rod[pos] + k * (curr_rod[pos-1] - 2*curr_rod[pos]␣
 ↪+ curr_rod[pos+1])

        # Writing new value and syncing threads
        next_rod[pos] = next_temp
        cuda.syncthreads()


    # Copying result from shared to global memory
    if ((timesteps - 1) % 2) == 0:
      result_rod[pos] = rod_2[pos]
    else:
      result_rod[pos] = rod_1[pos]
```

[64]:
```
# Running CUDA iterate function
start = time.time()
cuda_iterate_rod[blockspergrid, threadsperblock](d_init_rod, d_result_rod,␣
 ↪timesteps, k)
end = time.time()

print("timesteps:", timesteps)
print("blockspergrid:", blockspergrid)
print("GPU time taken is ", end-start)
```

```
timesteps: 1500
blockspergrid: 1
GPU time taken is  0.0009987354278564453
```

[65]:
```
# Comparison graph with CPU result
x = np.linspace(0, 1, N+1)
```

```
plt.plot(x, d_result_rod, 'o--', label="GPU")
plt.plot(x, iterate_rod(N, t_end), label="CPU")
plt.xlabel("Position (x)")
plt.ylabel("Temperature (T)")
plt.title("Check of same results of GPU and CPU algorithms")
plt.grid()
plt.legend()
plt.show()
```



2.d Use your code to estimate the time at which the temperature of the midpoint of the rod first exceeds a temperature of 9.8. Briefly (2-3 sentences) describe how you estimated this time. You may choose to use a plot or diagram to aid your description, but it is not essential to include a plot.

[57]:
```
# Defining some small helper functions for code clarity

def midpoint(arr):
  """
  Returns the midpoint value of a Numpy 1-d array
  """
```

```
  return arr[len(arr) // 2]

def isEqual(val1, val2, precision):
  """
  returns whether two values are equal for a certain degree of precision
  """
  return np.abs(val1-val2) < precision
```

[58]:
```
N = 50
goal_temp = 9.8

# FIRST STEP: go up in t_end value until the temperature at the midpoint goes␣
 ↪above the temperature wanted.

t_end = 0
incr = 50
midpoint_temp = 0.0

counter = 0
while midpoint_temp < goal_temp and counter <= 20:
    counter += 1 # to prevent infinite loop
    t_end += incr
    midpoint_temp = midpoint(iterate_rod(N, t_end))
    print("Temperature at midpoint: {0}".format(midpoint_temp))


# SECOND STEP: we use a search algorithm which uses a search window, whose␣
 ↪bound get
# redefined at every iteration depending on whether the true answer is found to␣
 ↪lie
# below or above the midpoint of the window.

# Setting a desired precision for the closeness of the temperature obtained to␣
 ↪that wanted
precision = 0.001

# Initializing the upper and lower search bounds
upper_t_end = t_end
lower_t_end = t_end - incr

print("")
print("Upper and lower bound for the time elapsed are {0} and {1}".
 ↪format(upper_t_end, lower_t_end))
print("")

# Keep searching while t_end gives the desired answer
# to the desired degree of precision
```

```
counter = 0
while not isEqual(midpoint_temp, goal_temp, precision) and counter <= 20:
    counter += 1 # to prevent infinite loop

    # midpoint bewteen the lower and upper bounds of the search
    mid_t_end = lower_t_end + (upper_t_end-lower_t_end)/2

    # Computing the temperature at the middle of the rod
    # at the midpoint of the search window for t_end
    rod = iterate_rod(N, mid_t_end)
    midpoint_temp = midpoint(rod)

    print("Time elapsed of {0} gave a temperature of {1}".format(mid_t_end,␣
 ↪midpoint_temp))

    # If the temperature is higher than the goal, then
    # it means we overshot, so the t_end corresponding to the
    # goal temperature must lie between the lower and middle bounds,
    # so we re-define the upper bound as the current mid
    if midpoint_temp > goal_temp:
        upper_t_end = mid_t_end
    # otherwise, we redefine the lower bound as the current mid
    else:
        lower_t_end = mid_t_end


# taking t_end corresponding to T = 9.8 at the rod's midpoint
# to be the middle t_end.
goal_t_end = mid_t_end
```

```
Temperature at midpoint: 2.277425882274717
Temperature at midpoint: 5.254682032543217
Temperature at midpoint: 7.102304754303583
Temperature at midpoint: 8.230757450175243
Temperature at midpoint: 8.919758105830049
Temperature at midpoint: 9.3404395025865
Temperature at midpoint: 9.597293854546628
Temperature at midpoint: 9.754120751289602
Temperature at midpoint: 9.849874143643115

Upper and lower bound for the time elapsed are 450 and 400

Time elapsed of 425.0 gave a temperature of 9.807872873406588
Time elapsed of 412.5 gave a temperature of 9.782652183025291
Time elapsed of 418.75 gave a temperature of 9.795631085432134
Time elapsed of 421.875 gave a temperature of 9.801826939611047
Time elapsed of 420.3125 gave a temperature of 9.798752855304027
Time elapsed of 421.09375 gave a temperature of 9.800295812365043
```

[59]:
```
rod = iterate_rod(N, goal_t_end)
midpoint_temp = midpoint(rod)

print("Final answer: a time elapsed of t_end={0} gives a temperature at the␣
 ↪rod's midpoint of T={1} for a precision of {2}".format(goal_t_end,␣
 ↪midpoint_temp, precision))
```

```
Final answer: a time elapsed of t_end=421.09375 gives a temperature at the rod's
midpoint of T=9.800295812365043 for a precision of 0.001
```

To obtain the time elapsed required for a temperature of 9.8 at the rod's midpoint, I designed an iterative search algorithm. First, the initial search window is determined by increasing incrementally *t_end* until the goal temperature $goal\_temp = 9.8$ is exeeded; then, the search window is iteratively refined by checking the temperature at its midpoint, and subsequently redefining the lower or upper bound depending on whether the temperature found lies above or below 9.8. The iteration is stopped when the midpoint temperature obtained lies within a pre-defined interval of 9.8, and the corresponding *t_end* is recorded as the final answer.

[ ]:
```

```