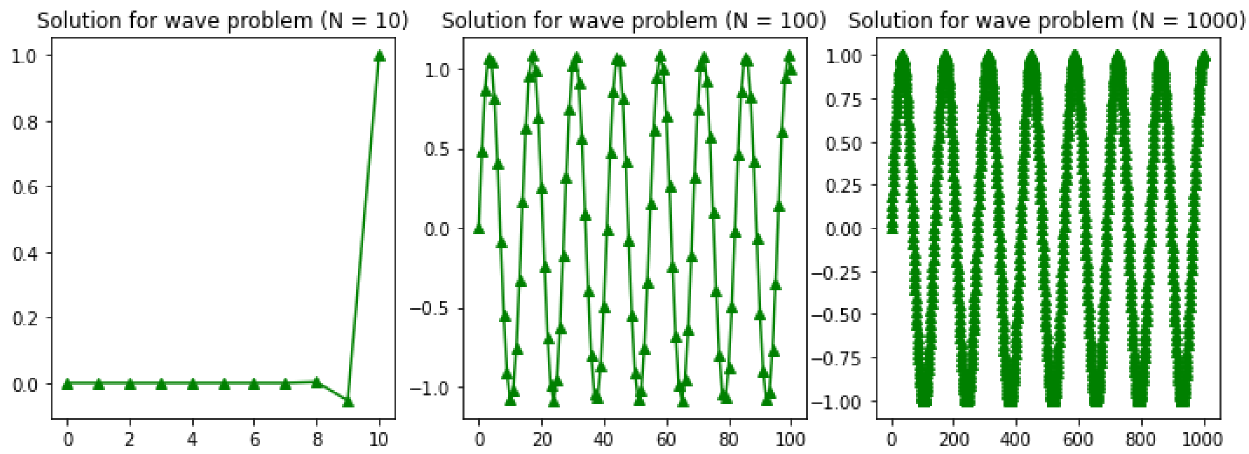
Part 1

```
import math
import numpy as np
import matplotlib.pylab as plt
from scipy.sparse import coo_matrix, linalg
from timeit import timeit
```

```
def wave_problem(N):
    h = 1 / N
    k = 29 * math.pi / 2

    f = np.zeros((N + 1), dtype=np.float64)
    f[N] = 1.0

    row = [0, N]
    col = [0, N]
    data = [1, 1]

    for i in range(1, N):
        row += [i, i, i]
        col += [i, i + 1, i - 1]
        data += [2 - (h ** 2) * (k ** 2), -1, -1]

    row = np.array(row)
    col = np.array(col)
    data = np.array(data)

    A = coo_matrix((data, (row, col)), (N+1, N+1)).tocsr()

    return A, f
```

```
n = [10, 100, 1000]
U = []

for i in n:
    A, f = wave_problem(i)
    u = linalg.spsolve(A, f)
    U.append(u)
```

```
plt.figure(figsize=(12,4))

for i in range(len(n)):
    ax = plt.subplot(1, 3, i + 1)
    ax.plot(range(n[i] + 1), U[i], "g^-")
    ax.set_title("Solution for wave problem (N = %d)" %n[i])
```



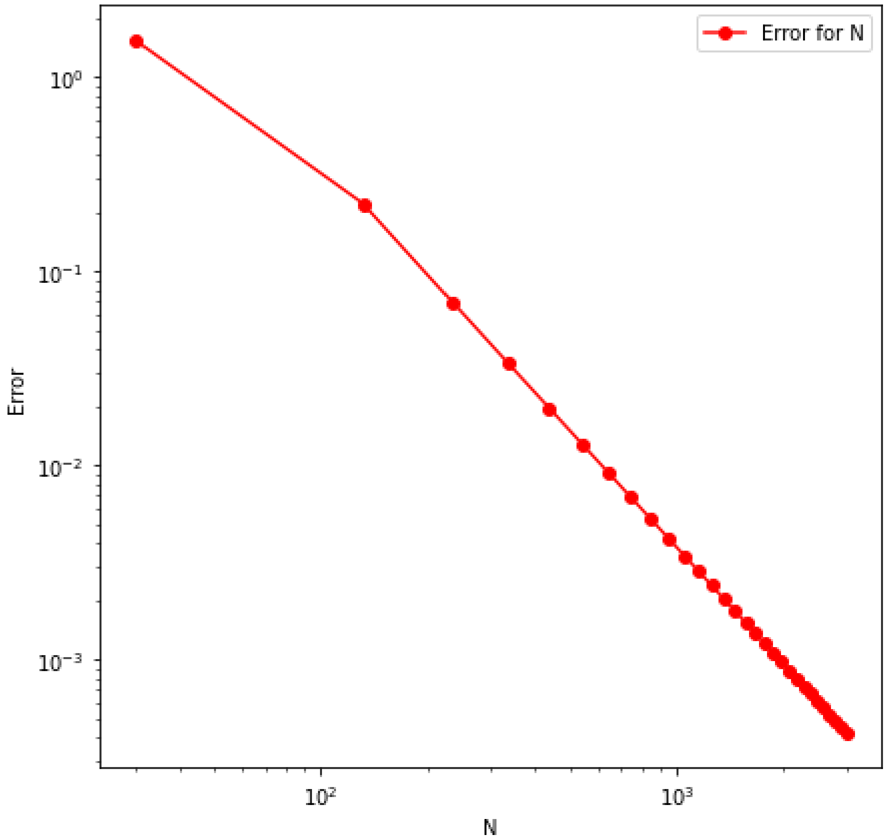
Obviously, when $N = 10$, there are only a few points in the plot 1, and the shape of the wave cannot be seen. However, with N increases, there are more points in the plot, the shape of the wave becomes clearer, and the result becomes more accurate.
The last solution ($N = 1000$) is the closest solution to the actual solution of the wave problem.

```
def error(N):
    k = 29 * math.pi / 2
    x = np.linspace(0, 1, N + 1)

    A, f = wave_problem(N)
    u = linalg.spsolve(A, f)

    err = np.abs(u - np.sin(k * x))

    return np.max(err)
```

```
N = np.linspace(30, 3000, 30, dtype=np.int32)
solution = []

for i in N:
    solution.append(error(i))
```

```
plt.figure(figsize=(7, 7))
plt.plot(N, solution, "ro-")
plt.xscale("log")
plt.yscale("log")
plt.xlabel("N")
plt.ylabel("Error")
plt.legend(["Error for N"])
```

```
<matplotlib.legend.Legend at 0x7f97f3fb9e50>
```

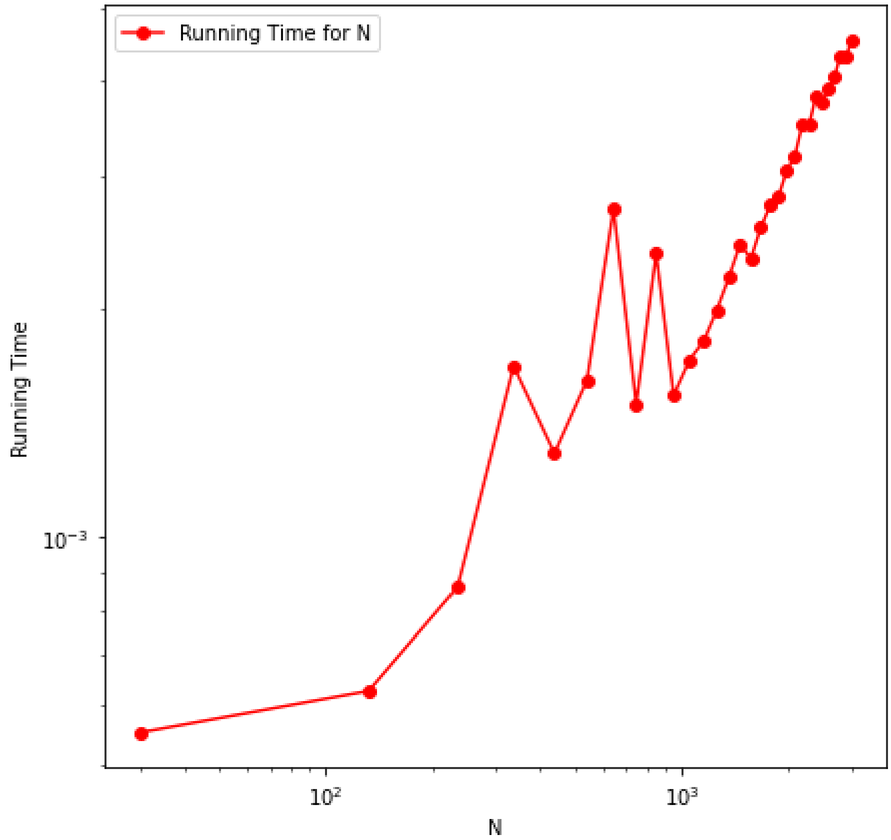



```
def time_counter(N):
        wave_problem(N)
```

```
time = []

for i in N:
    time.append(timeit(lambda: time_counter(i), number=1))
```

```
plt.figure(figsize=(7, 7))
plt.plot(N, time, "ro-")
plt.xscale("log")
plt.yscale("log")
plt.xlabel("N")
plt.ylabel("Running Time")
plt.legend(["Running Time for N"])
```

```
<matplotlib.legend.Legend at 0x7f97f3ce4f10>
```




```
e_A = np.mat([[N[2], 1], [N[-2], 1]])
e_b = np.array([-1.8, -3.3]).T
(a, b) = np.linalg.solve(e_A, e_b)
(a, b)
```

```
(-0.0005632745024408561, -1.6681937664288398)
```

```
(-9 - b) / a
```

```
13016.400000000001
```

According to the plot of N against the error, we can see that when N is from 10^2 to 10^3, error is linearly reduced from 10^{-1} to 10^{-3}. Based on the changing trend of this plot, I assume that the relationship between error and N obeys the function $error = aN + b$, and calculated parameters a and b by taking the second and penultimate values of N (see above), and finally calculated that the error will less than 10^{-8} when $N = 11329.83$. Thus, I predict that when $N = 10^4$, the error will less than 10^{-8}, and I chose $N = 30000$ because of the possible errors.

According to the plot of N against the time taken to compute, we can see that when N is from 10^2 to 10^3, running time is linearly increased from 10^{-3} to 6×10^{-3}. Therefore, when $N = 10^4$, the running time may reach at 8×10^{-3}.

```
pred = 30000
pred_err = error(pred)
pred_time = timeit(lambda: time_counter(pred), number=1)
print("Error of prediction N", pred_err)
print("Running time of prediction N", pred_time)
```

```
Error of prediction N 4.226854057858692e-06
Running time of prediction N 0.06867381999973077
```

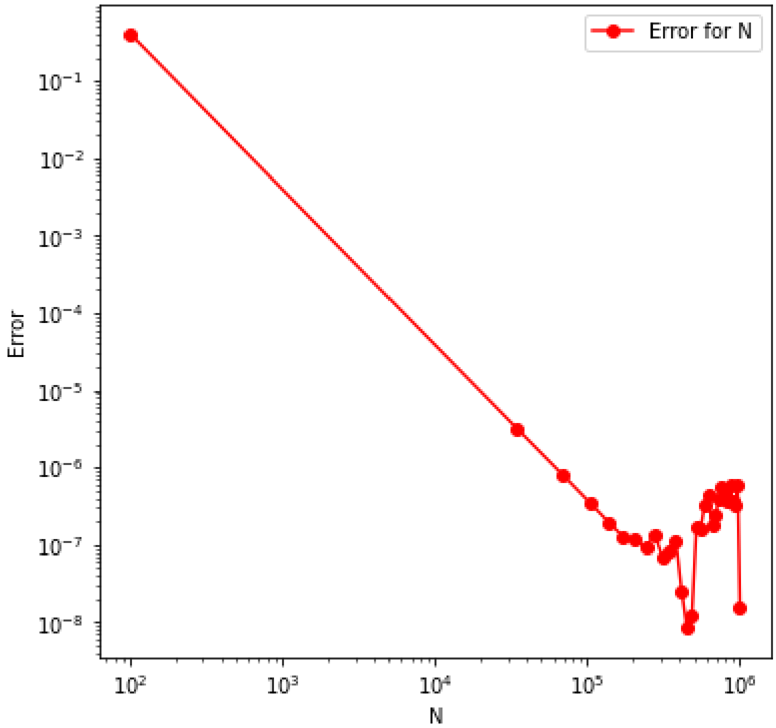
```
print(pred_err - 1e-08)
```

```
4.216854057858692e-06
```

```
N = np.linspace(100, 1000000, 30, dtype=np.int32)
solution = []

for i in N:
    solution.append(error(i))
```

```
plt.figure(figsize=(6, 6))
plt.plot(N, solution, "ro-")
plt.xscale("log")
plt.yscale("log")
plt.xlabel("N")
plt.ylabel("Error")
plt.legend(["Error for N"])
```

```
<matplotlib.legend.Legend at 0x7f683de72f10>
```



The result shows that its error higher than $N = 10^{-8}$, and the difference is $4.216e - 6$.
The reason why my prediction is not correct is that: for larger N, error and N not maintain the

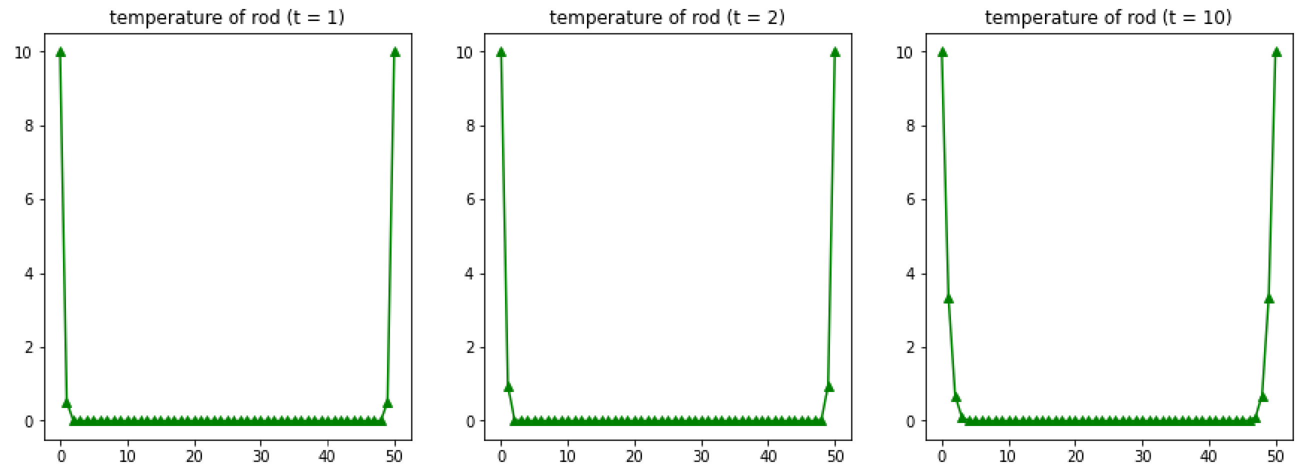
functional relationship that I was found.

Part 2

```
def heat_equation(N, j):
    h = 1 / N

    u = []
    for i in range(N + 1):
        if i == 0 or i == N:
            u.append(10.0)
        else:
            u.append(0.0)

    u = np.array(u)

    for iter in range(j):
        temp_u = np.copy(u)
        for i in range(N + 1):
            if i == 0 or i == N:
                u[i] = 10.0
            else:
                u[i] = temp_u[i] + (temp_u[i-1] - 2*temp_u[i] + temp_u[i+1]) / (1000*h)

    return u
```

```
t = [1, 2, 10]
U = []

for i in t:
    u = heat_equation(50, i)
    U.append(u)
```

```
plt.figure(figsize=(15,5))

for i in range(len(t)):
    ax = plt.subplot(1, 3, i + 1)
    ax.plot(range(len(U[i])), U[i], "g^-")
    ax.set_title("temperature of rod (t = %d)" %t[i])
```



I think that a large N will simulate the heating process of the rod more precisely, but it will require more iterations and cause more time and costs. Therefore, I chose $N = 50$, and this allows program to accurately simulate the points on rod without having to iterate too many times.

```
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-c3555ba6-1e0f-e5a4-1eea-b6d99bb69b25
                                Watchdog: Disabled
             FP32/FP64 Performance Ratio: 32
Summary:
        1/1 devices are supported
True
```

```
@cuda.jit
def cuda_heat_equation(u, j):
    N = u.shape[0] - 1
    h = 1 / N

    p = cuda.grid(1)

    cuda.syncthreads()
    if p == 0 or p == N:
        u[p] = 10
    else:
        u[p] = 0

    cuda.syncthreads()
    size = u.shape[0]
    for iter in range(j):
        if p == 0 or p == N:
            u[p] = 10
```

```
        else:
            u[p] = u[p-size] + (u[p-size-1] - 2*u[p-size] + u[p-size+1]) / (1000*h)

        cuda.syncthreads()
```

```
%timeit u = heat_equation(50, 10000)
```

```
850 ms ± 97.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
threadsperblock = (32, 32)
blockspergrid = (256, 256)

u = np.zeros(51, dtype=np.float64)
u = cuda.to_device(u)
```

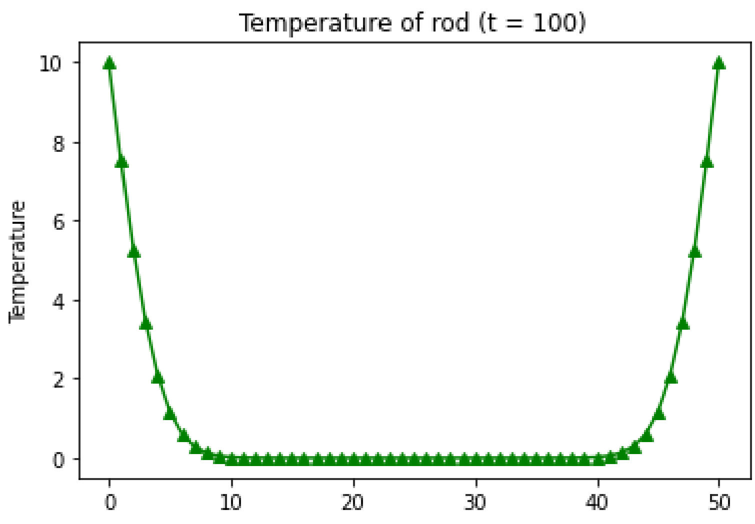
```
%timeit cuda_heat_equation[blockspergrid, threadsperblock](u, 10000)
```

```
The slowest run took 4.98 times longer than the fastest. This could mean that an intermediate
76.5 µs ± 60.4 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
threadsperblock = (32, 32)
blockspergrid = (256, 256)

iter = 100
u = np.zeros(51, dtype=np.float64)
u = cuda.to_device(u)
cuda_heat_equation[blockspergrid, threadsperblock](u, iter)
```

```
result_100 = u.copy_to_host()

plt.plot(range(len(result_100)), result_100, "g^-")
plt.title("Temperature of rod (t = %d)" %iter)
plt.ylabel("Temperature")
```

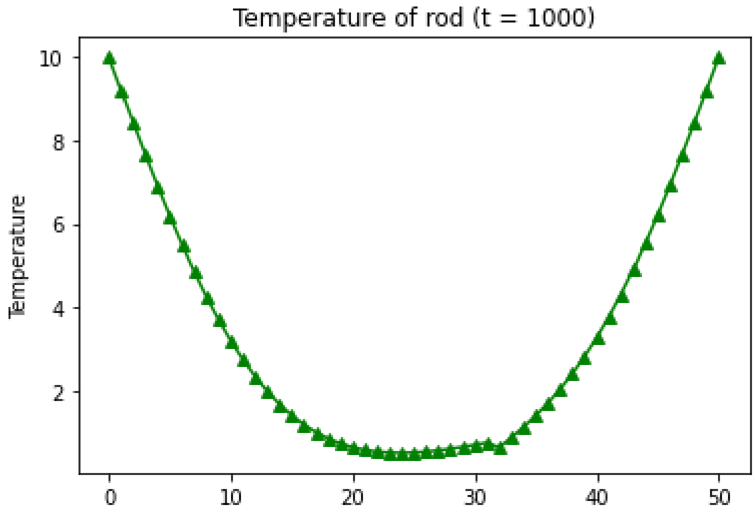
```
Text(0, 0.5, 'Temperature')
```



```
iter = 1000
u = np.zeros(51, dtype=np.float64)
u = cuda.to_device(u)
cuda_heat_equation[blockspergrid, threadsperblock](u, iter)
```

```
result_1000 = u.copy_to_host()

plt.plot(range(len(result_1000)), result_1000, "g^-")
plt.title("Temperature of rod (t = %d)" %iter)
plt.ylabel("Temperature")
```

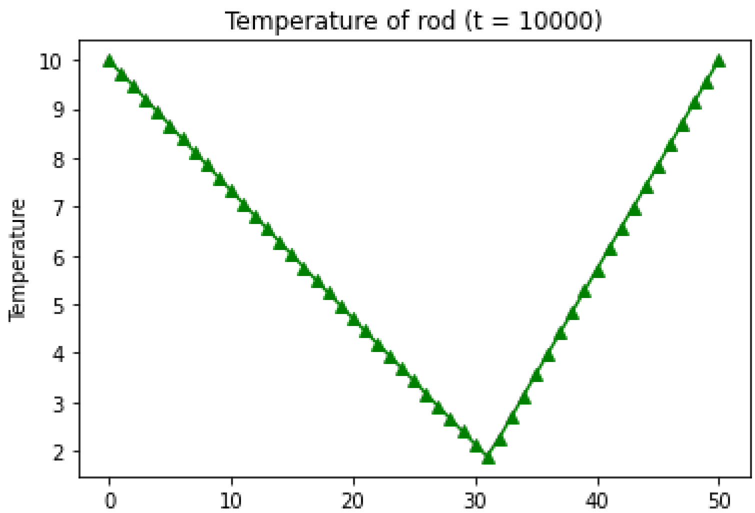
Text(0, 0.5, 'Temperature')




```
iter = 10000
u = np.zeros(51, dtype=np.float64)
u = cuda.to_device(u)
cuda_heat_equation[blockspergrid, threadsperblock](u, iter)
```

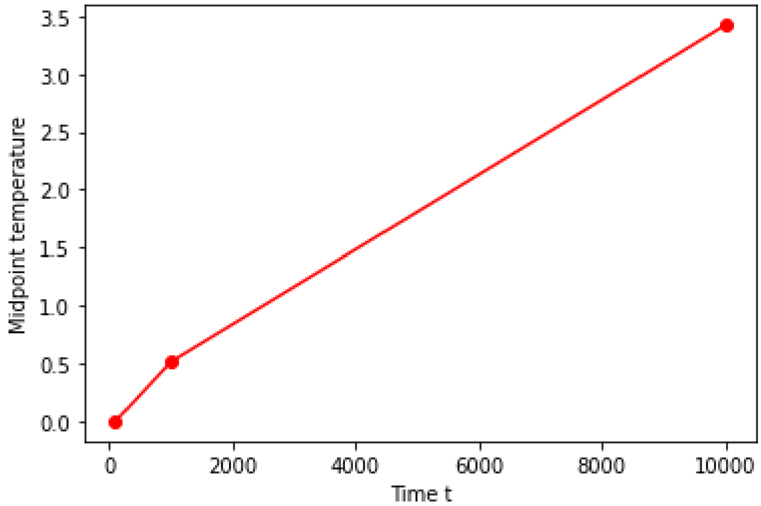
```
result_10000 = u.copy_to_host()

plt.plot(range(len(result_10000)), result_10000, "g^-")
plt.title("Temperature of rod (t = %d)" %iter)
plt.ylabel("Temperature")
```

Text(0, 0.5, 'Temperature')



```
midpoint = len(result_100) // 2
t = [100, 1000, 10000]
temper = [result_100[midpoint], result_1000[midpoint], result_10000[midpoint]]
plt.plot(t, temper, "ro-")
plt.xlabel("Time t")
plt.ylabel("Midpoint temperature")
```

```
Text(0, 0.5, 'Midpoint temperature')
```




```
h_A = np.mat([[t[1], 1], [t[-1], 1]])
h_b = np.array([temper[1], temper[-1]]).T
(a, b) = np.linalg.solve(h_A, h_b)
(a, b)
```

```
(0.000323024728538052, 0.19278074527656625)
```

```
(9.8 - b) / a
```

```
29741.43589007525
```

To estimate the time at which the temperature of the midpoint of the rod exceeds 9.8, I calculated the temperature of midpoint when time is 100, 1000, and 10000 respectively. According to those data, I constructed a function to predict the time value when midpoint temperature is 9.8, and the result is 29741.435 (shown as above). Therefore, I predict that the midpoint temperature will first exceeds 9.8 when time $t = 30000$.