Project3

December 17 2022

1 Project 3 on Machine Learning

1.1 Solving partial differential equations with neural networks

1.1.1 Oliwia Malinowska & Svein-Magnus Lommerud

December 2021

1.1.2 Abstract:

Partial differential equations include several variables and types of derivatives. If such an equation is very complicated, it is extremely hard to solve it analytically. With the help of machine learning algorithms and the computer power, we can tackle that problem and end up with quite precise results. Here, we took on a diffusion equation that has a relatively simple analytical solution. We implemented and studied two methods, the standard explicit scheme (SES) and the neural network (NN) algorithm, and used them to solve the diffusion equation. Then we compared the results with the analytical solution (AS). After the first time-step the solution from NN overlaps with the AS while the SES has a curve in the opposite direction compared to AS. With time the deviation from the AS shrinks for both the SES and NN, but near the last time-step we see that SES fluctuates much closer to the AS than the NN does. It seems therefore that the standard explicit scheme gives us a better solution for the diffusion equation than the neural network algorithm.

1.2 Introduction

Machine learning algorithms are versatile tools used to effectively solve very complicated problems, that are practically impossible to solve with pen and paper. Most of the partial differential equations are very hard to solve without a computer because of the multiple variables and derivatives of different order in the same equation. There are some analytical ways to solve the least complicated ones, but for the rest of them, we are helpless without machine learning. In this paper we will look at a relatively simple one that has a pretty straight forward analytical solution. Because of that, we will be able to verify our numerical results with the known solution.

In this paper we will use two numerical methods to solve the diffusion equation and compare the results with the analytical solution derived in Appendix. First we introduce the explicit scheme and the equations that need to be implemented. Then we present a brief overview of the neural network set up and explain how we can use it to solve the equation in the Methods section. For the explicit scheme we show our own code, while for the neural network part we explain the algorithm

and use a borrowed code [4] to generate the results. Testing of the algorithms, generated results and the analysis are all in the "Results and Discussion" section. The figures include all three solutions which helps compare them with each other.

1.3 Theory

1.3.1 Diffusion equation

Our physical problem is the temperature gradient $u(x,t)$ in a rod of length L which is the solution to a partial differential equation given below:

$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}$, $t < 0$, $x \in [0, L]$,

where t is time and x is the length og the rod.

Given conditions $u(0,t) = 0$, and $u(x,0) = sin(\pi x)$, we arrive at:

$u(x,t) = sin(\pi x)e^{-\pi^2 t}$ (Details in the Appendix).

1.3.2 Explicit scheme

This part is a summary from the lecture notes in Computational Physics [1]. We need a numerical algorithm to solve equation $\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}$. For that we will use a method where time and length x changes with step length for x ($\Delta x = \frac{1}{n+1}$ and a time step length Δt. Renaming $\frac{\partial u(x,t)}{\partial t}$ as u_t and $\frac{\partial^2 u(x,t)}{\partial x^2}$ as u_{xx}, and approximating the derivatives we arrive at:

$u_t \approx \frac{u(x,t+\Delta t)-u(x,t)}{\Delta t} = \frac{u(x_i,t_j+\Delta t)-u(x_i,t_j)}{\Delta t}$

and

$u_{xx} \approx \frac{u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)}{\Delta x^2}$

or

$u_{xx} \approx \frac{u(x_i+\Delta x,t_j)-2u(x_i,t_j)+u(x_i-\Delta x,t_j)}{\Delta x^2}$,

where the position x_i and time t_j are defined as

$t_j = j\Delta t$, $j \geq 0$ for each time-step j, and

$x_i = i\Delta x$, $0 \leq i \leq n+1$ for each step i.

By further simplification of the equations above we get to a discretized version of the initial diffusion equation:

$\frac{u_{i,j+1}-u_{i,j}}{\Delta t} = \frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{\Delta t^2}$.

The updated functions $u_{i,j+1}$ are given by the explicit scheme

$u_{i,j+1} = \alpha u_{i-1,j} + (1-2\alpha)u_{i,j} + \alpha u_{i+1,j}$,

where $\alpha = \Delta t/\Delta x^2$.

Since $u_{i,0} = sin(\pi x_i)$, after one time-step ($j = 0$) we get:

$u_{i,1} = \alpha u_{i-1,0} + (1 - 2\alpha)sin(\pi x_i) + \alpha u_{i+1,0}$.

To calculate u at the next time-step we use $u_{i,1}$ and the boundary conditions.

Since in our case the conditions are

$u_{0,j} = u_{n+1,j} = 0$, we can write our initial partial differential equation as a set of multiplications between the vector V_j corresponding to time t_j and a matrix $\hat{A}$ dependent on the α:

$V_{j+1} = \hat{A}V_j = ... = \hat{A}^{j+1}V_0$,

where V_0 is defined by the $u(x, 0) = sin(\pi x)$.

Stability limit The stability condition for the explicit scheme is given by [1]:

$\Delta t/\Delta x^2 \leq 1/2$.

1.3.3 Neural networks

A neural network is an algorithm that consists of layers, an input layer, an output layer and layers in between, called the hidden layers. In a neural network the information travels forward from the input layer, through the hidden layers and into the output layer. The hidden layers consist of nodes and connections between them called edges. In the nodes the algorithm assigns weights to determine later on how important the specific variables are for the output layer. The input data in each of these nodes goes through an activation function and if the result is higher than a threshold (bias) value, the data is sent to the node in the next hidden layer and after traversing all hidden layers it ends up in the output layer and makes up a fitted model for the new observations.

The neural network algorithm has to be trained to increase the accuracy of our model. We can change the accuracy of the model by adjusting the weights and bias values using backward propagation. To calculate if the model is more accurate, we use a cost function that is minimal for very accurate models. The aim of the training is therefore to minimize the cost function. More details on that can be found in our previous project on machine learning [2].

1.4 Methods

1.4.1 Explicit scheme

The explicit scheme algorithm for updating our function $u(x, t)$ can be implemented in the following way: (The code is loosely based on the code from the lecture notes in Computational Physics [1]).

[4]:
```
import numpy as np
import matplotlib.pyplot as plt

def u(x, t):

    delta_t = t[1]-t[0]
    delta_x = x[1]-x[0]
    alpha = delta_t/(delta_x**2)
```

```
    u = np.zeros((len(x), len(t)))

    #Including condition u(0, t) = 0:
    u[0, :] = 0
    #Including condition u(L, t) = 0
    u[len(x)-1, :] = 0

    #First time-step:
    for i in range(1, len(x)-1):
        u[i, 0] = alpha * np.sin(np.pi*x[i-1]) + (1-2*alpha) * np.sin(np.
↪pi*x[i])+alpha*np.sin(np.pi*x[i+1])

    #The rest of the steps:
    for j in range(len(t)-1):
        for i in range(1, len(x)-1):
            u[i, j+1] = alpha * u[i-1, j] + (1-2*alpha) * u[i,j] + alpha *␣
↪u[i+1, j]

    return u
```

1.4.2 Solving partial differential equations using neural networks

A partial differential equation (PDE) is expressed in the following way:

$$f\left(x_1, ..., x_N, \frac{\partial g(x_1,...,x_N)}{\partial x_1}, ..., \frac{\partial g(x_1,...,x_N)}{\partial x_N}, \frac{\partial g(x_1,...,x_N)}{\partial x_1 \partial x_2}, ..., \frac{\partial^n g(x_1,...,x_N)}{\partial x_N^n}\right) = 0,$$

where N is number of variables and f is an expression that includes all types of derivatives of $g(x_1, ..., x_N)$ up to order n.

Function $g(x_1, ..., x_N)$ is the function we try to find and using neural network we first need a trial function $g_t(x_1, ..., x_N)$ that we will adjust to fit the analytical solution in the end. The trial function is given by:

$$g_t(x_1, ..., x_N) = h_1(x_1, ..., x_N) + h_2(x_1, ..., x_N, N(x_1, ..., x_N, P)),$$

where $h_1(x_1, ..., x_N)$ is a function that makes $g_t(x_1, ..., x_N)$ satisfy specified conditions of the function, $N(x_1, ..., x_N, P)$ is the neural network including weights and biases in the P. [3]

To increase the accuracy of the model we need to minimize the cost function, which in this case is the mean squared error given by:

$$C(x_1, ..., x_N, P) = \left(f\left(x_1, ..., x_N, \frac{\partial g(x_1,...,x_N)}{\partial x_1}, ..., \frac{\partial g(x_1,...,x_N)}{\partial x_N}, \frac{\partial g(x_1,...,x_N)}{\partial x_1 \partial x_2}, ..., \frac{\partial^n g(x_1,...,x_N)}{\partial x_N^n}\right)\right)^2.$$

The neural network needs to adjust parameters P to minimize the cost function.

1.5 Results and discussion

1.5.1 Explicit scheme

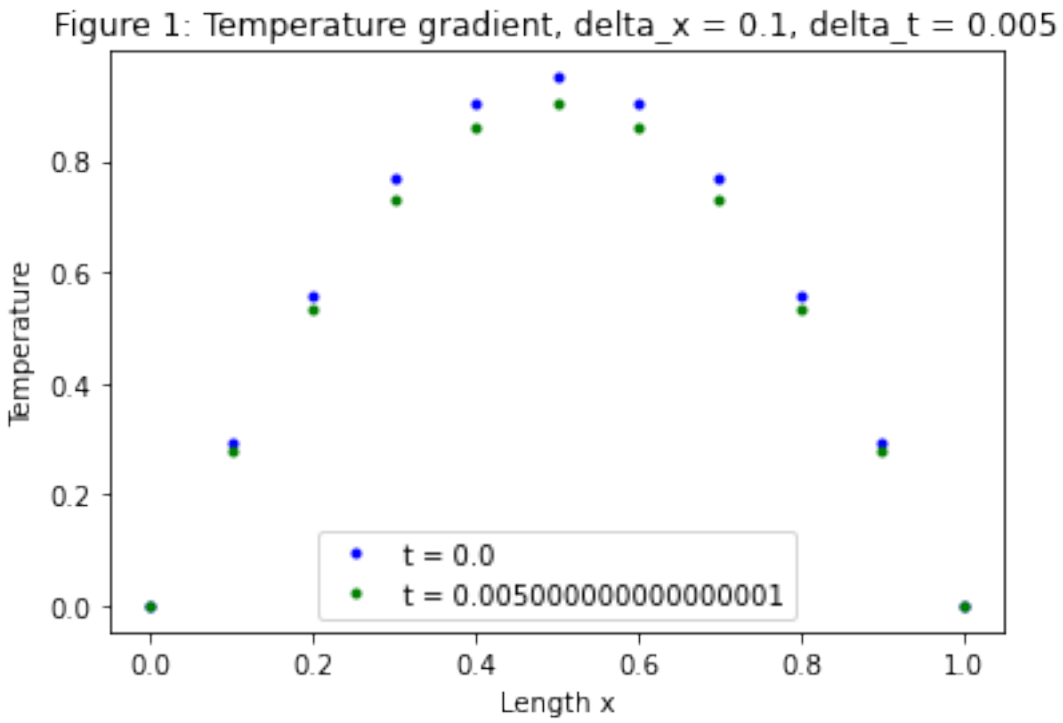
When implementing the explicit scheme, we included the stability condition mentioned earlier. Here we plotted the temperature gradient as a function of length x for different time points. First we tested our algorithm by changing the Δx and therefore corresponding Δt according to the stability condition.

[5]:
```
delta_x = 0.1
#Including the stability condition:
delta_t = 0.5 * (delta_x**2)
N = 100/delta_x
T = N * delta_t

t = np.arange(0, T + delta_t, delta_t)
x = np.arange(0, 1 + delta_x, delta_x)

u1 = u(x, t)

plt.title("Figure 1: Temperature gradient, delta_x = 0.1, delta_t = 0.005")
plt.xlabel("Length x")
plt.ylabel("Temperature")
plt.plot(x, u1[:, 0], 'bo', markersize = 3, label =  "t = %s" %(t[0]))
plt.plot(x, u1[:, 1], 'go', markersize = 3, label =  "t = %s" %(t[1]))
plt.legend()
plt.show()
```

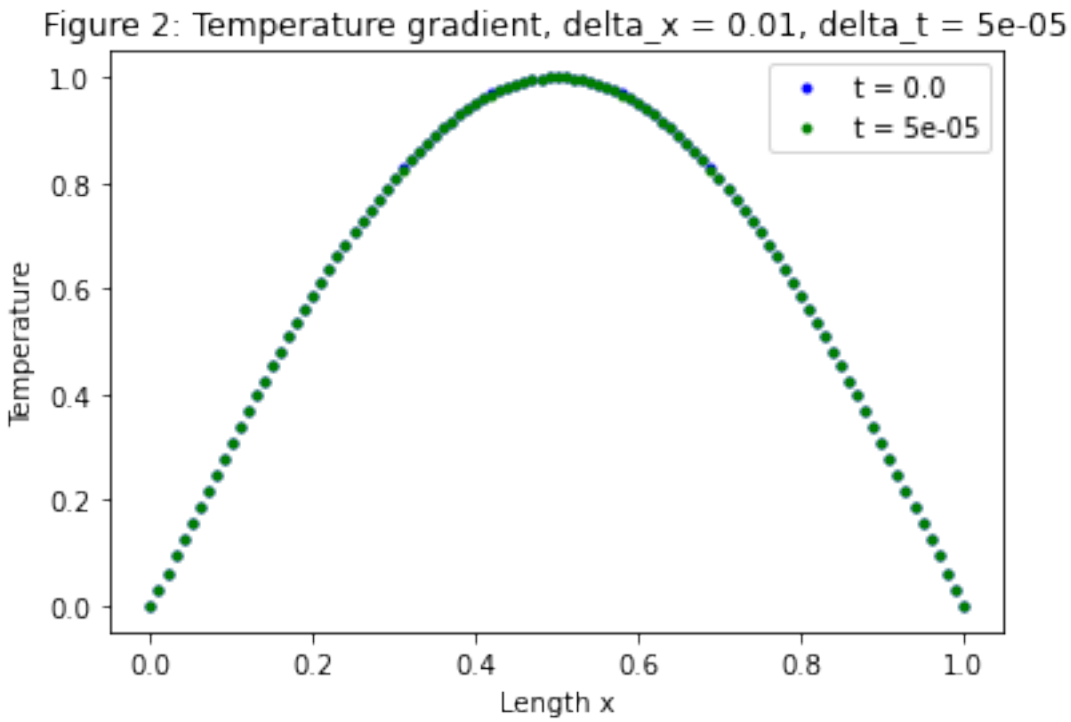


[6]:
```
delta_x = 0.01
#Including the stability condition:
delta_t = 0.5 * (delta_x**2)
N = 100/delta_x
T = N * delta_t

t = np.arange(0, T + delta_t, delta_t)
x = np.arange(0, 1 + delta_x, delta_x)

u2 = u(x, t)

plt.title("Figure 2: Temperature gradient, delta_x = 0.01, delta_t = 5e-05")
plt.xlabel("Length x")
plt.ylabel("Temperature")
plt.plot(x, u2[:, 0], 'bo', markersize = 3, label =  "t = %s" %(t[0]))
plt.plot(x, u2[:, 1], 'go', markersize = 3, label =  "t = %s" %(t[1]))
plt.legend()
plt.show()
```



We get the expected results in figure 2, which are that the smaller Δx the smaller Δt and larger number of points N (which gives us a smoother graph). We see the smaller Δt by comparing the two lines in the two plots (figure 1 and 2). The distance between the blue and green line (that correspond to the first and the second time step) in the second plot (where Δx is smaller) is smaller, indicating smaller Δt.

Continuing with $\Delta x = 0.01$ we inspect how the shape of the curve will change for a more drastic change of the time variable.

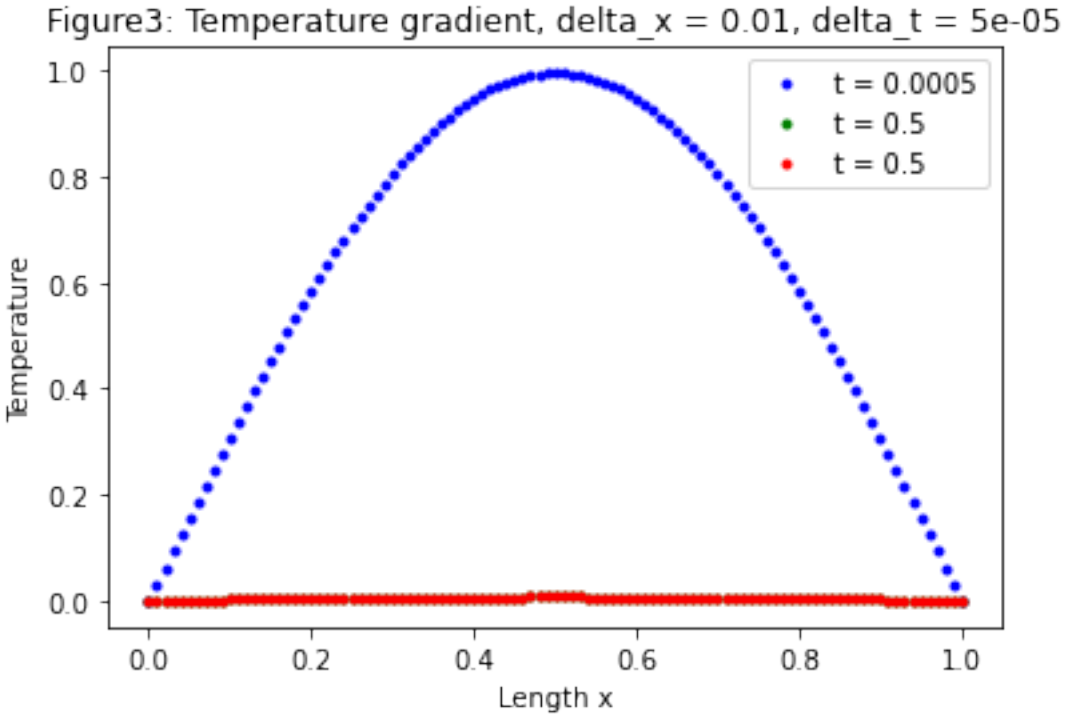
[7]:
```
delta_x = 0.01
#Including the stability condition:
delta_t = 0.5 * (delta_x**2)
N = 100/delta_x
T = N * delta_t

t = np.arange(0, T + delta_t, delta_t)
x = np.arange(0, 1 + delta_x, delta_x)

u3 = u(x, t)

plt.title("Figure3: Temperature gradient, delta_x = 0.01, delta_t = 5e-05")
plt.xlabel("Length x")
plt.ylabel("Temperature")
plt.plot(x, u3[:, 10], 'bo', markersize = 3, label =  "t = %s" %(t[10]))
```

```
plt.plot(x, u3[:, 10000], 'go', markersize = 3, label =  "t = %s" %(t[10000]))
plt.plot(x, u3[:, -1], 'ro', markersize = 3, label =  "t = %s" %(t[-1]))
plt.legend()
plt.show()
print(T)
```



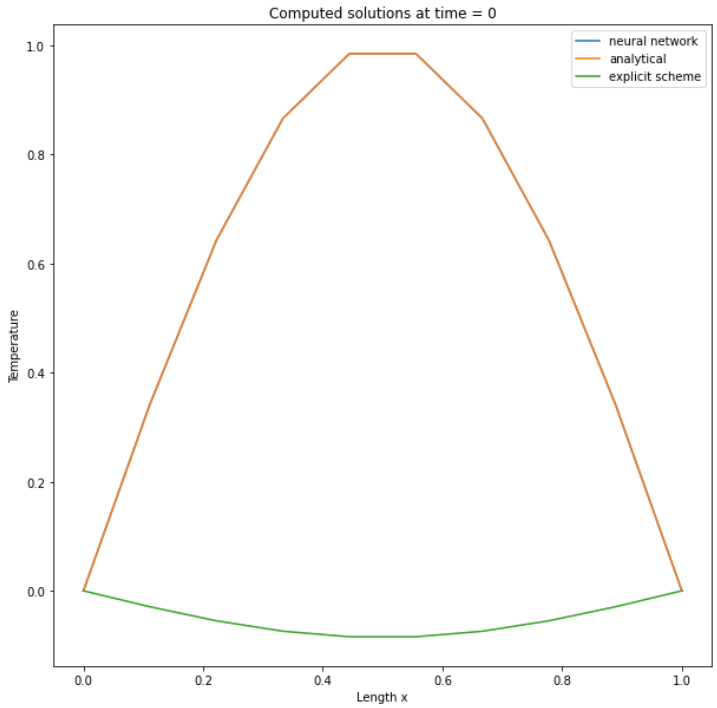

```
0.5
```

We see in figure (3) that when the time point approaches the total time the line appears to get almost linear. That is the case only when we plot it together with a function corresponding to a much smaller time t. Thereby, $u(x, t_2)$ is almost linear compared to $u(x, t_1)$ when t_2 is very high and when the difference between t_2 and t_1 is big.

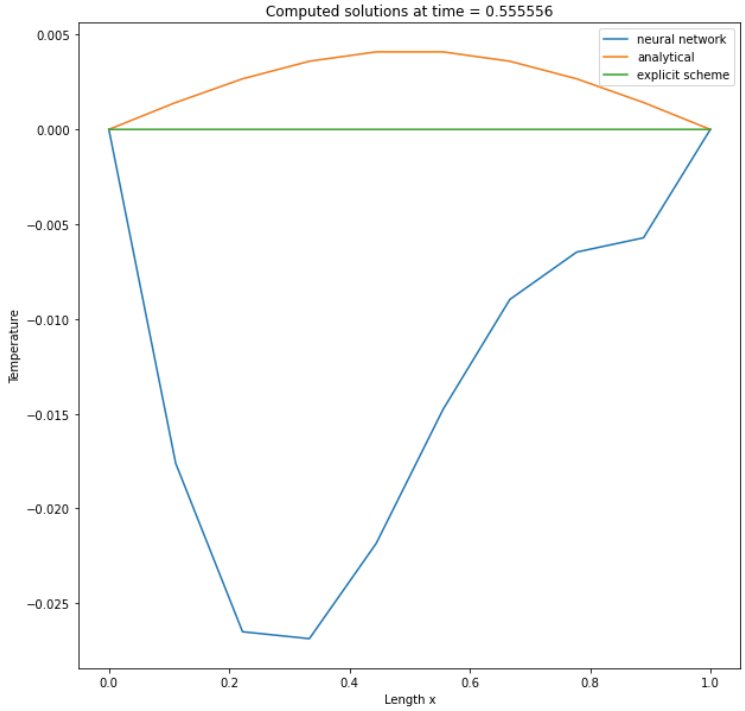
1.5.2 Solving partial differential equations using neural networks

To generate and present results for the neural network solution, we borrowed the code from the lecture notes [4] and implemented slight changes to be able to plot all three solutions in one figure (explicit scheme, neural network and the analytical solution.

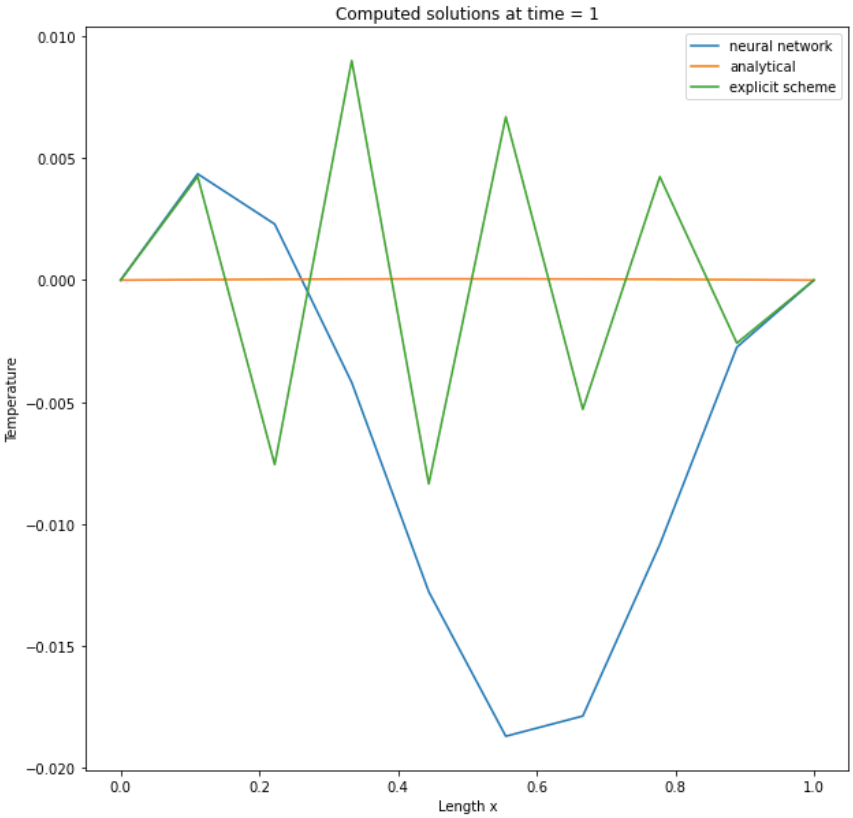
The three figures (4), (5) and (6) show the computed solutions for the temperature gradient for time $t = 0$, $t = 0.556$ and $t = 1$ calculated in the three ways, with use of neural network algorithm, explicit scheme algorithm and lastly, the analytical solution for comparison. $t = 0.556$ is approximately half of the total time and $t = 1$ corresponds to the last time-step.



Figur 3 Temperature gradient at time t=0. There is a significant difference between the explicit scheme graph and the analytical graph.



Figur 4 Temperature gradient at time t=0.556. The neural network graph deviates from the analytical solution more than the explicit scheme does.



Figur 5 Temperature gradient at time t=1. The explicit scheme graph fluctuates a little bit along the analytical solution graph. The neural network graph deviates a little bit more from the analytical solution.

In the figure (4) (after the first time step) we can see a significant difference between the explicit scheme solution and the analytical solution, while the neural network solution is completely overlapping with the analytical one. With time (figure (5)) the neural network solution starts to deviate significantly from the analytical solution, while the difference between the analytical solution and the explicit scheme solution shrinks. Deviation from the analytical solution is smaller after the last time-step for both the neural network solution and the explicit scheme solution (figure (6)). In the same figure we also see a fluctuation of the explicit scheme solution along the analytical solution. It seems therefore that in comparison to the neural network solution the explicit scheme gives us a much better model. We think that the shape of the explicit scheme solution comes from the broken stability condition. Figures (4), (5) and (6) include plots for all three methods and in the case of the explicit scheme, we forgot to include the stability condition. In our calculations $\Delta t = \Delta x = 0.1$, while the correct Δt corresponding to $\Delta x = 0.1$ is below or equal to 0.005. Unfortunately after adjusting the code the run time was simply too long to generate any results in time.

1.6 Conclusion

We can conclude that the neural network method didn't give us very accurate results. The reason for that can be number of the data points that unfortunately was limited due to a very slow program. When comparing the neural network solution and the analytical solution with the explicit scheme solution we forgot to include the stability condition and this has certainly altered our results. Assuming that it didn't have that huge impact, we can conclude that the explicit scheme fits our equation better than the neural network does, based on the figures (4), (5) and (6). We therefore hugely regret that we didn't focus on implementing the neural network method using Tensorflow functionalities that would have resulted in a much faster code.

1.7 Appendix

1.7.1 Analytical solution to the diffusion function:

$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}$, $t > 0$, $x \in [0, L]$

Assuming the solution is a product of two functions, one dependent on x and the other on t:

$u(x,t) = f(x)g(t)$.

Breaking the PDE (partial differential equation) into a pair of ODEs (ordinary differential equations):

$\frac{\partial^2 f(x)g(t)}{\partial x^2} = \frac{\partial f(x)g(t)}{\partial t}$,

$g(t)\frac{\partial^2 f(x)}{\partial x^2} = f(x)\frac{\partial g(t)}{\partial t}$,

$\frac{1}{f(x)}\frac{\partial^2 f(x)}{\partial x^2} = \frac{1}{g(t)}\frac{\partial g(t)}{\partial t}$.

Both sides are a constant and it is useful that the constant is negative and squared:

$\frac{1}{f(x)}\frac{\partial^2 f(x)}{\partial x^2} = \frac{1}{g(t)}\frac{\partial g(t)}{\partial t} = -b^2$,

or separately:

$\frac{1}{f(x)}\frac{\partial^2 f(x)}{\partial x^2} = -b^2$,

and

$\frac{1}{g(t)}\frac{\partial g(t)}{\partial t} = -b^2$.

This gives us:

$f(x) = Acos(bx) + Bsin(bx)$,

and

$g(t) = Ce^{-b^2t}$.

Joining them together we get:

$u(x,t) = f(x)g(t) = (Acos(bx) + Bsin(bx))(g(t) = Ce^{-b^2t}) = ACcos(bx)e^{-b^2t} + BCsin(bx)e^{-b^2t}$.

Simplifying the constants, we arrive at:

$u(x,t) = Dcos(bx)e^{-b^2t} + Esin(bx)e^{-b^2t}$.

Using condition $u(0,t) = 0$:

$u(0,t) = De^{-b^2t} = 0$

gives us $D = 0$ and therefore:

$u(x,t) = Esin(bx)e^{-b^2t}$.

Next, using condition $u(x,0) = sin(\pi x)$:

$u(x,0) = Esin(bx) = sin(\pi x)$

gives us $b = \pi$ and $E = 1$.

Therefore, our final solution describing temperature gradient in a rod is:

$u(x,t) = sin(\pi x)e^{-\pi^2 t}$.

1.8 References

1. M. Hjorth-Jensen. "Computational Physics. Lecture Notes Fall 2015." Department of Physics. Univerity of Oslo. 2015. p. 304-308.
2. O. Malinowska, S-M. Lommerud. "Project 2 on Machine Learning". Accessed from https://github.com/Lommerud/Project1.git Date of access: 09.12.2021
3. M. Hjorth-Jensen Department of Physics. Univerity of Oslo. 'Overview of course material: Data Analysis and Machine Learning'. Week 42: Solving differential equations and Convolutional (CNN). Slide nr 37. Accessed from https://compphysics.github.io/MachineLearning/doc/web/course.html Date of access: 14.12.2021
4. M. Hjorth-Jensen Department of Physics. Univerity of Oslo. 'Overview of course material: Data Analysis and Machine Learning'. Week 42: Solving differential equations and Convolutional (CNN). Slide nr 45. Accessed

from https://compphysics.github.io/MachineLearning/doc/web/course.html Date of access: 16.12.2021