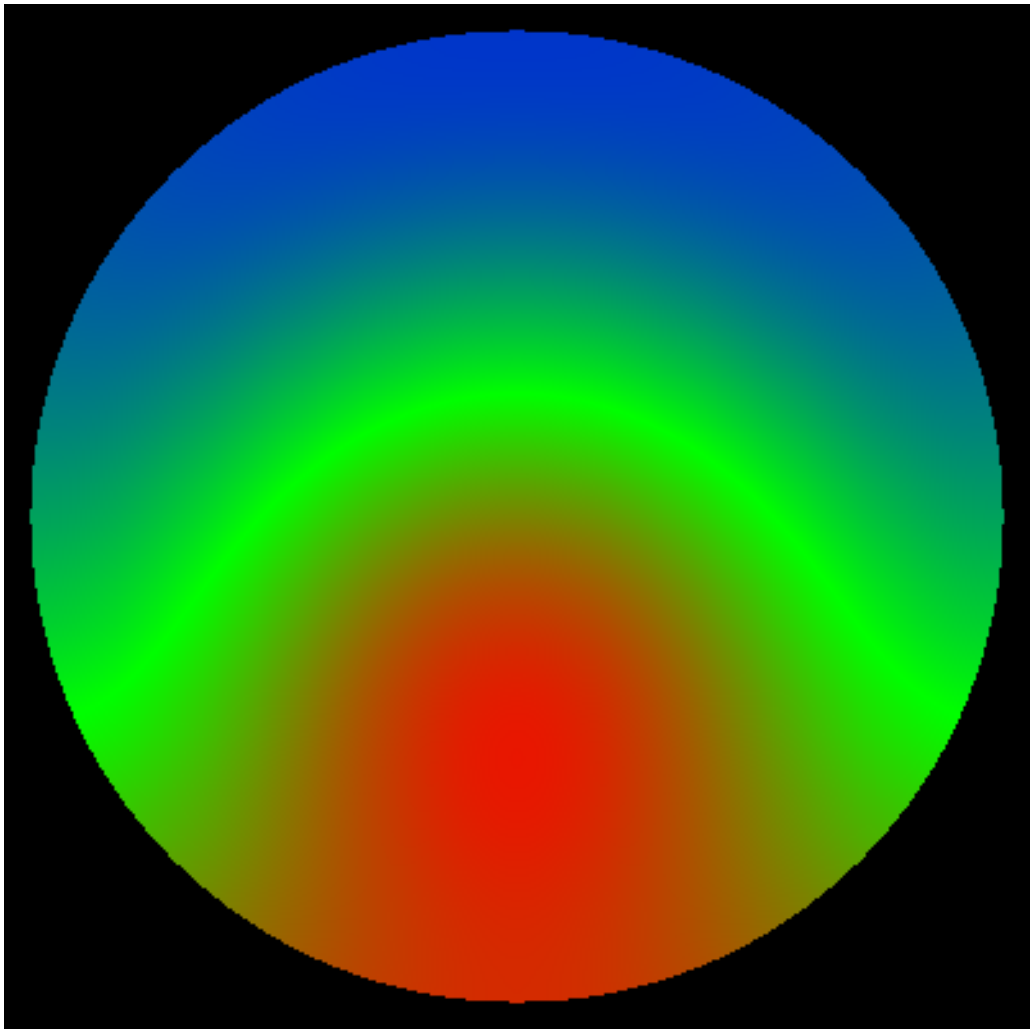
Comparing linear PDE solvers

Computer simulations in physics

Kürti Zoltán

November 20, 2021



Contents

1 Introduction

Based on the feedback about the last project, in this project I will be more focused on explaining my thought processes and motivations. I will also use a more informal tone as it is better suited to describe the journey I took while working with partial differential equations.

1.1 Motivation

The importance of numerically solving partial differential equations can hardly be overstated both in purely academic and engineering fields. At the same time I also like working on both partial and ordinary differential equations, this was one of the reasons my first project involved differential equations too. I dream about obtaining solutions for complicated situations in general relativity, but that's simply not feasible during a couple weeks, in fact I'm not even sure if a single person could get meaningful results in that field. So I have to set realistic goals and the obvious first step is to start with a simple linear problem, like the heat equation.

I also like to implement everything on my own, and I also like to come up with my own ideas even if I'm not the first to do so. This method takes a long time and generally isn't optimized for obtaining results fast and with the least amount of energy. However I get a deep understanding and lot of practice in this process. During evaluating a lab measurement to write the report or during a BSc. thesis the focus is on getting results in physics, not programming, and time is critical. In these situations usually people can't afford to begin the project by writing for loops, multiplications and additions in C, I certainly couldn't. This course however does focus on computer programming and writing low level algorithms. All in all this is the perfect opportunity for me to spend time on solving the heat equation "fro scratch".

2 Solution outline

The first problem I thought about was calculating the Laplacian on a discretized grid. This seemed to be an important and frequent operation while solving linear partial differential equations. While working on this problem I created the `coefficients.py` file. This file contains functions that help finding coefficients to approximate the Laplacian

at a point based on neighboring points up to different orders and with different conditions on the error terms. The functions generally work in N dimensions, although the time for computation grows exponentially with the number of dimensions. Some of the methods work for irregular grids, which would be very useful in case the discretization of a problem was matched to the geometry of the problem. I did not investigate this aspect, but it is a high priority extension I would investigate if I continue this project. With these functions it is possible to reproduce all the stencils listed in [1], but my approach also applies to irregular and higher dimensional cases.

The next step was to test some of these stencils I got from `coefficients.py`. I focused on one and two dimensional cases and compared single and double precision calculations too. Solving partial differential equations isn't just about choosing the right algorithms. A huge part of the process is finding optimal hyperparameters for the algorithms, like the resolution of discretization, the time step, relaxation parameters, precision of the floating point operations and so on. The choice of these parameters makes or breaks the success of the calculation. My impression so far is that there is no explicit general rule for finding an optimal value for a parameter. Experimentation is usually needed, and this experimentation process can be sped up dramatically with good intuition. This part of the project will certainly help me make more informed guesses during these types of experimentation in the future.

The last component to solve the first concrete problem is treating boundary conditions. The goal I set was to be able to handle arbitrary geometries both with Dirichlet and Neumann type boundary conditions. I did meat this goal in two dimensions and my solution generalizes to higher dimensions easily (adding one or two more nested for loops depending on the function in question). The basic idea applies to irregular discretization, although in that case additional data structures would be needed to look up nearby points. To demonstrate this capability I solved the heat equation in a circle using a square grid for discretization with different combinations of Dirichlet and Neumann boundary conditions. [?, 3]

3 Floating point precision

The speed of algorithms is critical for these problems. Especially as the dimension of the problem grows, discretization can quickly lead to millions of points, which need to be updated each time step. in a naiiv implementation calculating the Laplacian on a grid

is cache constrained. This is because each neighboring points only used a couple times (depending on the size of the stencil) before the calculation moves on and the element is replaced in the cache with other elements. Later in the next row this original value will have to be loaded again, if the rows are long enough. In my computer the L1 cache is $32kiB$. This is enough to hold 4096 double precision floating point number. Using a 5x5 stencil for the Laplacian to calculate each row the program has to load 5 rows of the original data, and the results of the calculation go to the L1 cache first too. Depending on the cache update policy even with a couple hundred elements long row elements may need to be loaded multiple times.

This can be reduced by changing the order in which the Laplacian is calculated. Instead of calculating it row by row, for large grids it may be more efficient to calculate the result in square chunks, designed to align with cache lanes ($64B$ on my computer) while maximizing the area of it with the constraint that it fits into the cache. This method would increase the number of uses a single value after each load and so would decrease the total amount of cache operations.

Another great improvement could be the usage of single precision numbers. The discretization of the differential equation already introduces an error that can easily be bigger than single precision. Therefore to store the results of a calculation it ia probably enough to use single precision floating point numbers. The problem comes when the change of the solution is slow compared to the grid resolution (which is a desirable property if we want to minimize the error caused by discretization). The difference of two close floating point numbers will have a large error as the significant digits cancel out. This works against minimizing the discretization error. Increasing the precision of the number representation or using higher order approximations for the Laplacian with smaller resolution may be two options to improve this situation.

3.1 Precision, step size and error order

In one dimension the laplacion of f can be calculated as

$$\triangle f_i = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} + \mathcal{O}(h^4) \tag{1}$$

and as

$$\triangle f_i = \frac{-\frac{1}{12}f_{i+2} + \frac{4}{3}f_{i+1} - \frac{5}{2}f_i + \frac{4}{3}f_{i-1} - \frac{1}{12}f_{i-2}}{h^2} + \mathcal{O}(h^6). \tag{2}$$

These expressions can be obtained from the Taylor expansion of f around x_i, this method will be discussed later more generally. In the first test program I calculated the Laplacian of the sine function for different resolutions using equations 1. and 2 both with single and double precision. The results are summarized in figure 1, the plot was made using the `1d.py` file and the data to be plotted was generated with the `1d64-32Laplace` executable. The error on the figure is the biggest difference between the discretized Laplacian and the analytic formula. The region on which the calculation was carried out is 2π long, and N represents the number of points of the discretized grid.

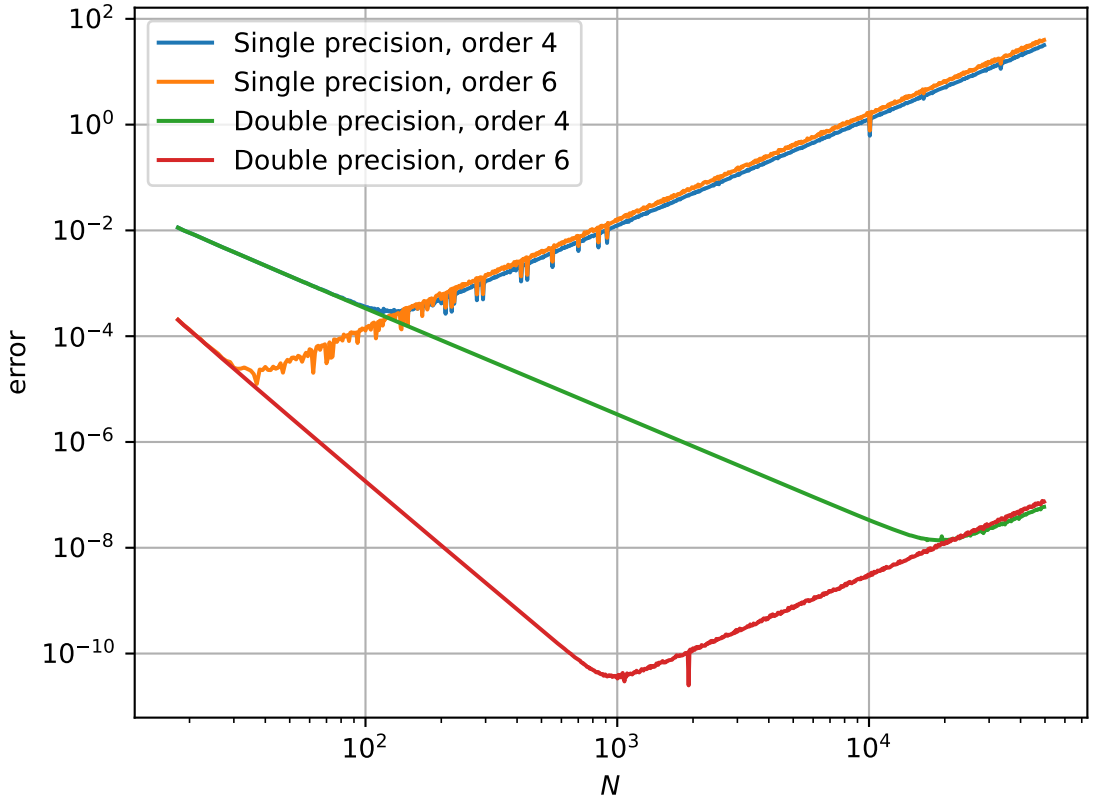


Figure 1: Error values of the discrete Laplace for approximation with 4th and 6th order errors and single and double precision.

It can be seen that there are two sources of error. One is the discretization error that decreases as $\frac{1}{N^4}$ or $\frac{1}{N^6}$. The other is the error coming from rounding error when two almost equal floating point numbers are subtracted. The key takeaway is that there is

an optimum in precision, neither too big nor too small grid distances work. I observed similar behavior for two dimensional stencils as will be discussed later, and I expect the same for higher dimensions.

4 Higher order in many dimensions

Based on the above simple one dimensional case study, and as can be expected without any experiments, the most precise way to obtain the Laplacian on a grid is using high precision floating point numbers and a finite difference approximation with higher order error terms. Obtaining the coefficients of these stencils becomes a non trivial task, and in the following section I will discuss my solution to this problem.

4.1 Designing the kernels

The code I will discuss can be found in the `coefficients.py` file. My strategy was the following. First, collect the displacements of the stencil points from the center. Since I only applied this method to stencils over a regular grid, this task was not difficult. Second, calculate the Taylor polynomial corresponding to each point of the stencil. Third I searched for stencil coefficients with which multiplying the Taylor expansion of grid points and summing the up gives the Laplace operator up to a given order in derivatives. Finally, in case there are multiple such stencils, I further narrowed the stencil coefficients so that the leading order error term is a multiple of the power of the Laplace operator. These stencils are called isotropic stencils. Their error due to discretization (in leading order) does not depend on the orientation of the grid. This provides additional stability in solving differential equations, especially in cases where the difference between neighboring grid points is considerable. I would like to describe some of the key elements of the algorithms laid out above.

The first important structure is the Taylor polynomial, used in many places in this code. It is represented by a list, each element of this list corresponds to a term in the Taylor polynomial. Each term of the Taylor polynomial is represented by a list, who's first element is the coefficient of the derivative, and the second term contains N integer numbers corresponding to the order of derivative in the N different directions of the N dimensional grid. The Taylor polynomial corresponding to a stencil point is built up using N repeated one dimensional Taylor expansions. To do this I had to implement a

polynomial multiplier function.

Another important element of this file is the assumptions I made about the form of the stencils. First I assumed that they are symmetric to mirroring along each coordinate. This assumption works out because the Laplacian operator has this property. This assumption cancels out all the terms from the final result with odd derivatives. My second assumption was that it's symmetric under coordinate exchanges. This is a valid assumption, since exchanging coordinates is just a special case of a rotation or a rotation and a mirroring depending on the parity of the dimension.

Solving for all possible finite difference stencils that approximate the Laplacian is a matter of solving a set of linear equations,

$$Ax_0 = b, . \tag{3}$$

A is the matrix that produces the coefficients of the Taylor polynomial from the coefficients of the stencil. b contain the Taylor coefficients corresponding to the Laplacian, which is just the sum of the second derivatives with respect to the coordinates. x contains the independent coefficients of the stencil. With a particular solution and a matrix C who's columns span the null space of A all of the stencil coefficients that give the Laplace operator up to the specified order are of the form

$$x = x_0 + Ct, \tag{4}$$

where t is a free parameter vector. The dimension of the t vector is equal to the number of linearly independent stencils that approximate the Laplacian. In this program for a $(2n+1) \times (2n+1) \times \ldots (2n+1)$ stencil I required the stencil to be precise up to terms containing $2n+1$ derivatives. I choose this limit because in the one dimensional case this is the best that can be achieved. From this I expect that in any dimensions this is the best result.

The last calculation this file does is from this group of stencils, parameterized by t, choose the ones that are isotropic. These solutions are determined by the equation

$$A(x_0 + Ct) - b = \alpha b^{(n+1)}. \tag{5}$$

Here the A matrix includes $2n+2$ order terms in order to contain the leading order error terms. The left hand side of the equation will result in the leading order error of the stencil. $b^{(n+1)}$ are the Taylor coefficients of the Laplace operator to the $n+1$th power. I

calculate this earlier in the code by repeated multiplication. This equation can be solved by multiplying the equation by a $P = I - b^{(n+1)} \circ b^{(n+1)}/b^{(n+1)} \cdot b^{(n+1)}$ projection matrix. This matrix projects to the hyperplane orthogonal to $b^{(n+1)}$ This gives

$$PCt = -PAx_0 + Pb. \tag{6}$$

The solution for t from this equation is in general an affine space, similar to the situation in equation 3. These solutions determine the isotropic finite difference approximations of the Laplace operator.

4.2 Testing the stencils

I picked three small stencils from the results of the previous section, and repeated a similar test to 3.1. The difference was that this time I used a Gaussian function as a test function, although I found that the results are not changed significantly as long as the test function has non zero higher order derivatives.The result is shown in figure 2.

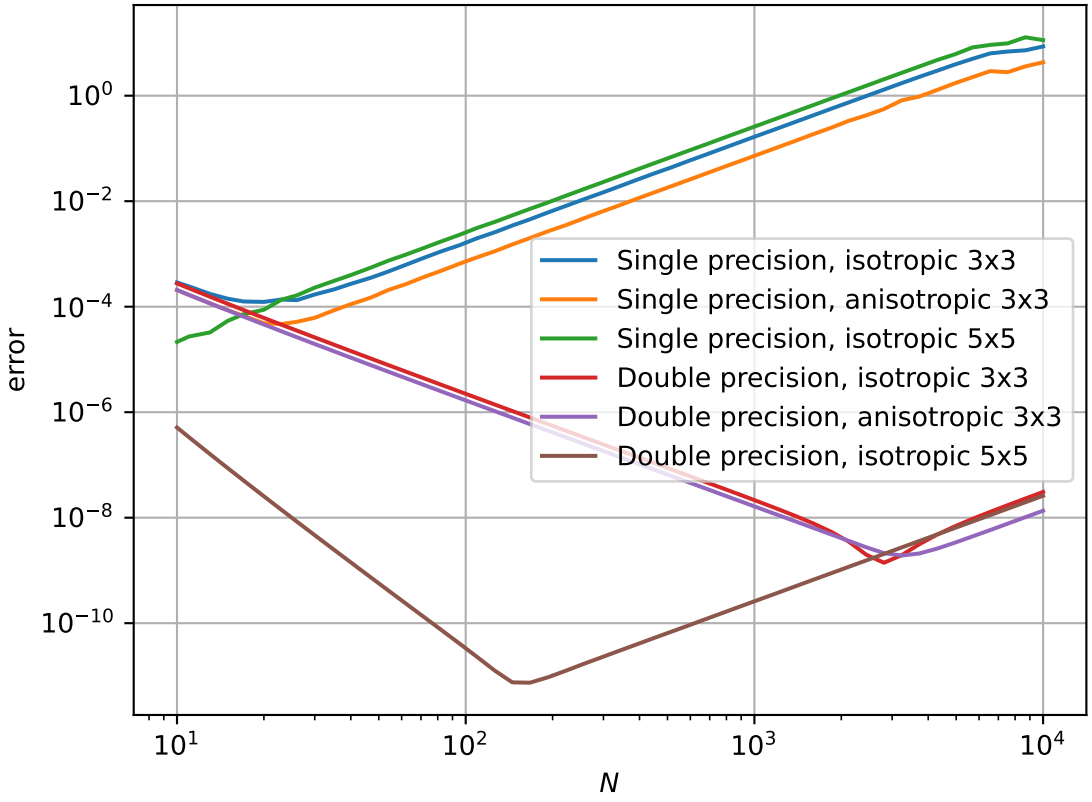


Figure 2: The error of different Laplace stencils as a function of the discretization.

The stencils I choose were

$$
\begin{aligned}
S_3^{iso} &= \begin{bmatrix} \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \\ \frac{2}{3} & -\frac{5}{2} & \frac{2}{3} \\ \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{bmatrix}, \\
S_3^{ani} &= \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \\
S_5^{iso} &= \frac{1}{60} \begin{bmatrix} 0 & -2 & -1 & -2 & 0 \\ -2 & 16 & 52 & 16 & -2 \\ -1 & 52 & -252 & 52 & -1 \\ -2 & 16 & 52 & 16 & -2 \\ 0 & -2 & -1 & -2 & 0 \end{bmatrix}.
\end{aligned}
\tag{7}
$$

The plot was generated with the `2d.py` file, and the data for the plot was generated with the `2dLaplacetest` executable.

5 Boundary conditions, where numeric methods shine

With simple linear differential equations, like the heat equation, the only case when the equations may not have simple analytic solutions is if they have complicated boundary conditions. In this scenario numerical methods may be the only choice. Also there is no particular solution of a differential equation without boundary conditions so it is a central topic in solving partial differential equations. In practice boundary conditions are often complicated, not just their geometry, but also their conditions. They may not be neither Dirichlet nor Neumann. In case thermal radiation is the most important effect, the normal component of the gradient at the boundary will be proportional to the 4th power of the temperature. It is worth spending time working on boundary conditions.

As I was searching for how others treat boundary conditions, I got my inspiration from this website. [4] In the problem they used they had a straight boundary with Neumann condition. They satisfied the boundary condition by mirroring the solution to the boundary, and using the mirrored elements while calculating the finite differences at the boundary. This method can't be easily generalized to curved or even straight but not aligned boundaries. However while reading this page I realized that maybe I can simply change the values at the boundary, and only update the inner points of the region according to the finite difference version of the differential equation.

To sum it up, my general strategy to solve the heat equation is that I update the inner points of the region based on the differential equation and I calculate the values at the boundary so that the boundary conditions are satisfied with respect to the updated inner points. For Dirichlet type boundary conditions this means just setting the value at the boundary to the boundary condition, for Neumann type boundary condition (if the derivative is 0) this means setting the boundary point to the average of the nearby inner points. This method can be easily generalized to implement more complicated boundary conditions.

To make this happen I needed a way to tell outer, inner and boundary points apart. The outer points can be defined by the user with a function: if the function provided by the user returns a positive value for the coordinates of the point, it is an outer point. Otherwise a point is at the boundary if one of it's neighboring points (in N dimensions

$3^N - 1$ neighbors) is an outer point. The rest of the points don't have outer points as neighbors, meaning they can be updated using a 3x3 or maybe a bigger Laplace stencil. They are the inner points. In case a bigger stencil is used, some of the inner points still need to be updated with a smaller stencil. The code that implements these ideas is the `allocateBoundary` function in the `timeEvolution.cpp` file. It fills up an array made of 8 bit values. Each point has a corresponding 8 bit element in this array. Outer points are marked with 0, boundary points with 1 and inner points with 2 or higher values, depending on how big the stencils are that can be used with them as a center point.

5.1 Examples

Two concrete examples are simulated with this code. Both of them have a circle as their boundary, and both of them use the heat equation. The thermal conductivity is assumed to be 1, so the equation is

$$\frac{\partial T}{\partial t} = \triangle T. \tag{8}$$

The first example has Neumann boundary condition with 0 normal derivative along the entire boundary, physically this means perfect insulation. The starting temperature distribution is an off centered Gaussian distribution. The integral of the initial temperature distribution is chosen such that the temperature of the $t \lim \infty$ equilibrium state is determined by the `Tfinal` variable. (If the integral of the initial temperature distribution outside the circle is small, that is. I used the analytic expression for the integral over the whole plane.) The temperature of the center of the circle over time is plotted in figure 3. Using a more dense grid for the simulation does improve the accuracy of the final average temperature. The small additional heat theta seems to remain comes from applying the Neumann boundary condition: in this example after updating the inner points according to the Laplacian the temperature near the boundary will increase after each step. As the boundary points are set to the average of the nearby inner points, a small amount of additional heat is introduced to the system as error. As the resolution of the grid is improved, the area of the boundary points decreases, which proportionally decreases the additional heat introduced by applying the boundary condition. In this simulation I choose the final temperature to be 10, which is well approximated by the simulation. The video of the simulation is uploaded to YouTube, and all the parameters are left unchanged in the source code. [?]

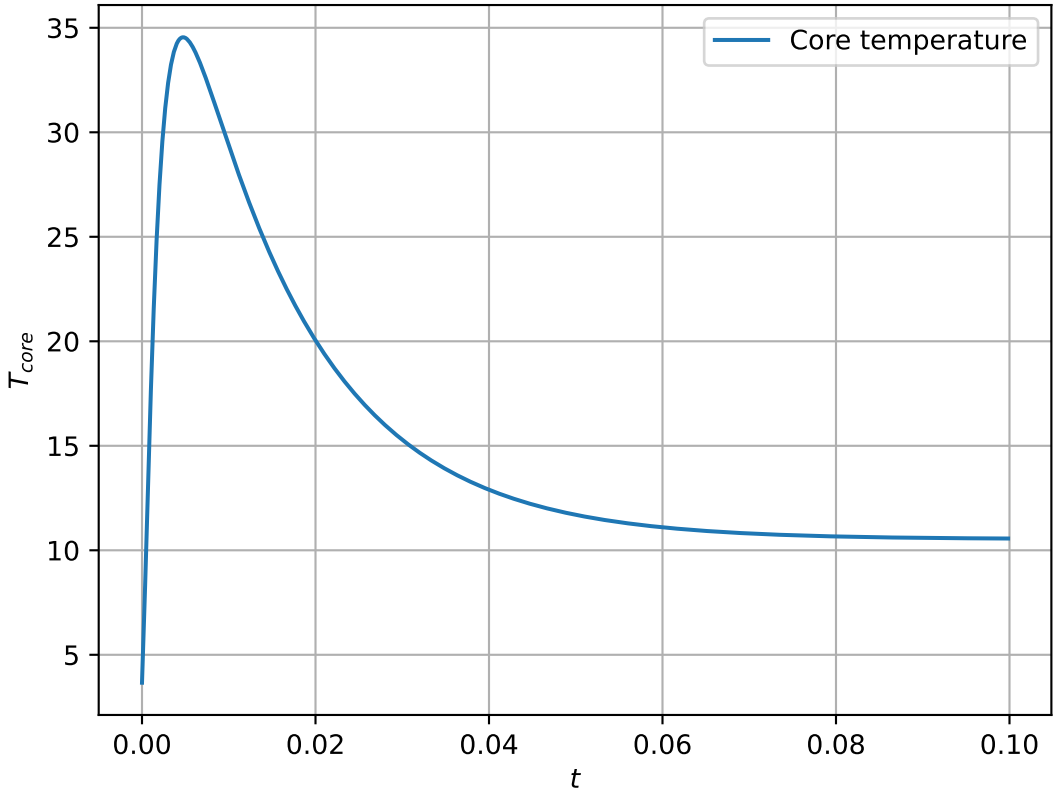


Figure 3: The time dependence of the middle point.

The initial rise in temperature is due to the initial high temperature spot widening. The exponential decrease happens as the heat slowly equalizes over the whole region.

The other example was in a circle too, with mostly Neumann boundary condition oer its perimeter. However, two symmetric sections of the boundary are changed to Dirichlet boundary conditions, one with temperature 20, the other with temperature 0. The starting temperature distribution was constant 0. The animation of this simulation can also be viewed on YouTube, and the parameters of the simulations can be found in the same source file. [3]

6 Conclusion

There are many ways in which this project could be continued. Probably my first choice would be to experiment with making the calculation of the Laplacian more cash

friendly, seeing improvements in runtime is always satisfying. Another big improvement would be to port the stencil calculation code to C++, and use it for irregular discretizations, calculating the matrix corresponding to the Laplacian as part of the initialization of the program. This would allow for experimenting with how the distribution of points affect the simulation, which is a question that interests me.

References

[1] Michael Patra and Mikko Karttunen. *Stencils with isotropic discretization error for differential operators*. Numerical Methods for Partial Differential Equations, 22(4):936–953, 2006. https://onlinelibrary.wiley.com/doi/pdf/10.1002/num.20129

[2] Zoltán Kürti. *Heat conduction in an insulated circle*, 2021. https://youtu.be/GF2Py_qA2BU

[3] Zoltán Kürti. *Heat transfer through a circle*, 2021. https://youtu.be/khKOVJwojZ0

[4] Leif Rune Hellevik. *Numerical methods for engineers - lecture notes*, 2020. https://folk.ntnu.no/leifh/teaching/tkt4140/._main056.html