On the numerical simulation of the diffusion equation in 1D with explicit and implicit schemes, and the 2D simulation of heat diffusion in the litosphere using Jacobi's iterative method

Eivind Støland, Anders P. Åsbø

(Dated: December 16, 2020)

We solved the general diffusion equation numerically in one and two dimensions. Comparison between numerical and analytical results showed a good level of agreement, and thus we conclude that the solvers work as intended. To solve the one-dimensional diffusion equation three different schemes were used: a forward Euler based explicit scheme, a backward Euler based implicit scheme and the Crank-Nicolson (implicit) scheme. The explicit scheme was clearly better suited for the problem we applied our solvers to, but when simulating systems that require a high resolution in the spatial coordinate, the timestep must be very small for the solution to be stable. Therefore implicit schemes may be a better choice due to computational cost as they converge for any choice of timestep. The two-dimensional solver used Jacobi's iterative method, and our test-simulation showed good correspondance with an analytic solution. A simulation of the temperature distribution in the lithosphere showed results as we expected, with a temperature increase of $25 - 100°$C at a depth of $10 - 30$ km due to the radioactive elements introduced as heat sources in the heat equation. Our results could potentially be compared with measurements in order to ascertain whether the subduction zone that was on the western coast of Norway enriched the upper mantle 1 Gy ago.

CONTENTS

I. INTRODUCTION

The general diffusion equation permeates physics, and is present in many different applications as it is related to continuity equations. It is used to describe events such as transfer of particles, heat transfer, and more. Solutions to a general dimensionless diffusion equation is thus something that can be used to describe a multitude of physical systems.

In this report we present and implement numerical solvers for the one-dimensional dimensionless diffusion equation using an explicit scheme, and two implicit schemes with different truncation errors. The explicit scheme and one of the implicit schemes are based on the Euler method, and the other implicit scheme is the Crank-Nicolson scheme. We also present and implement a solver for the two-dimensional dimensionless diffusion equation. In this case we wish to apply our solver to real-world problem with the heat equation with a heat source term, and so the solver is made to be compatible with such source terms in general.

We judge both the one-dimensional solvers and the two-dimensional solvers by comparison of the numerical results with analytical results. This, factored together with the computational costs will let us compare the one-dimensional schemes, and confirm the validity of the results of both the one-dimensional and two-dimensional solvers. These comparisons are done both visually with plots, and are also quantified with a measure of the error in the numerical solutions.

As for the real-world application of the two-dimensional solver, we seek to the model and simulate the temperature distribution of the lithosphere before and after a further enrichment of the upper mantle. Such an event is theorized to have occured when an active subduction zone on the western coast of Norway led to a refertilization of the upper mantle in the Oslo Graben about 1 Gy ago. We solve this system numerically to develop expectations of what the temperature distribution would look like if this was the case. This can be compared with experimental results, and used to further improve the model in further research.

II. FORMALISM

A. Diffusion equation

By scaling variables, we can write the general diffusion equation as follows:

$$\nabla^2 u(\vec{r},t) = \frac{\partial u(\vec{r},t)}{\partial t}\,, \tag{1}$$

where ∇ is the spatial derivative, $\vec{r}$ a vector containing spatial coordinates, t is time, and what u is depends on which system we are looking it in particular. This equation can be used to model several physical phenomena, such as mixing of particles, and perhaps most famously heat transfer through the heat equation:

$$\frac{k}{c_p\rho}\nabla^2 T(\vec{r},t) = \frac{\partial T(\vec{r},t)}{\partial t}\,, \tag{2}$$

where k is the thermal conductivity, c_p is the specific heat, ρ is the density of the material, and T is the temperature gradient. We show this now as an example of how such equations can be scaled. Even though we will use it later on we will do so in two dimensions with some further constraints that makes it so that we have to scale it differently. Firstly we can define the diffusion constant $D = c_p\rho/k$. This means we can write the heat equation as:

$$\nabla^2 T(\vec{r},t) = D\frac{\partial T(\vec{r},t)}{\partial t}\,.$$

Now we define the spatial coordinate vector $\vec{r} = \alpha\hat{\vec{r}}$, where α is an arbitrary constant. Substituting this affects the spatial derivative such that the equation now reads:

$$\frac{1}{\alpha^2}\nabla^2 T(\hat{\vec{r}},t) = D\frac{\partial T(\hat{\vec{r}},t)}{\partial t}\,.$$

As α is an arbitrary constant we can now choose it such that $D = 1/\alpha^2$. The equation then reads:

$$\nabla^2 T(\hat{\vec{r}},t) = \frac{\partial T(\hat{\vec{r}},t)}{\partial t}\,,$$

which is just the general diffusion equation we have listed earlier, with u exchanged for T. In other words, if we can solve the general diffusion equation, we can also solve the heat equation.

When solving the diffusion equation numerically it is common to scale the spatial coordinates so that $x, y, z \in [0,1]$ if we are using cartesian coordinates. As long as we are looking at a square system, this does not cause any problems for the method outlined earlier, as we can easily model other choices of limits by adding a constant to the spatial coordinate and correctly choosing α. We also assume that the initial state and the boundary conditions are known:

$$\begin{aligned} u(x,y,z,0) &= f(x,y,z) \\ u(0,y,z,t) &= g(y,z,t) \\ u(1,y,z,t) &= h(y,z,t) \\ u(x,0,z,t) &= k(x,z,t) \\ &\vdots \end{aligned}$$

and so on for the other coordinates as well.

In the following sections we will look at solutions to the diffusion equation in 1D and 2D both numerically and analytically.

B. Analytical solution of diffusion equation in one dimension

In order to find a solution we need to make some assumptions based on the initial and boundary conditions. We assume that:

$$u(x,0) = g(x) \quad 0 < x < L$$

and:

$$u(0,t) = u(L,t) = 0 \quad t \geq 0$$

The equation wish to solve is the general diffusion equation in one dimension:

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

In order to proceed we assume separation of variables:

$$u(x,t) = F(x)G(t)$$

The diffusion equation can then be rewritten:

$$\begin{aligned} G\frac{\partial^2 F}{\partial x^2} &= F\frac{\partial G}{\partial t} \\ \frac{F''}{F} &= \frac{G'}{G}\,, \end{aligned}$$

where we have denoted the derivatives with primes. In the equation above the left-hand side is completely independent of t and the right-hand side is completely independent of x. This directly implies that both sides have to be equal to a constant. We define this constant as $-\lambda^2$, which gives us the following differential equations we need to solve:

$$\begin{aligned} F'' + \lambda^2 F &= 0 \\ G' + \lambda^2 G &= 0 \end{aligned}$$

These have solutions:

$$\begin{aligned} F(x) &= A\sin(\lambda x) + B\cos(\lambda x) \\ G(t) &= Ce^{-\lambda^2 t} \end{aligned}$$

The boundary conditions are satisfied if we set $B = 0$ and $\lambda = n\pi/L$ where n is a positive integer (n can be 0 as well, but this solution is just 0 everywhere and is thus wholly uninteresting). A solution can then be written:

$$u_n(x,t) = A_n \sin\Big(\frac{n\pi}{L}x\Big) e^{-\frac{n^2\pi^2}{L^2}t}\,,$$

where we have included the constants C and A in A_n. A general solution will thus be a linear combination of these solutions for all n:

$$u(x,t) = \sum_{n=1}^{\infty} A_n \sin\Big(\frac{n\pi}{L}x\Big) e^{-\frac{n^2\pi^2}{L^2}t}$$

We still need to determine the constants A_n and these are given by the initial condition:

$$\begin{aligned} g(x) &= u(x,0) \\ g(x) &= \sum_{n=1}^{\infty} A_n \sin\Big(\frac{n\pi}{L}x\Big) \end{aligned}$$

This is the expression for a Fourier expansion of $g(x)$, and we can thus determine the coefficients as:

$$A_n = \frac{2}{L}\int_0^L g(x)\sin\Big(\frac{n\pi}{L}x\Big)\,dx$$

We can also adapt our solution for different boundary conditions. In most cases we assume the the boundary values are constants $u(0,t) = a$, $u(L,t) = b$ where a and b are constants. If we assume there to be a steady state solution $f(x)$ fitting these boundary conditions, then we can write $u(x,t) = v(x,t) + f(x)$, where $v(x,t)$ is the solution of the diffusion equation with Dirichlet boundary conditions. The steady state solution is determined by solving the Laplace equation:

$$\frac{\partial^2 f(x)}{\partial x^2} = 0$$

If the boundary conditions are constant this is simply a linear polynomial of first order:

$$f(x) = \frac{b-a}{L}x + a$$

As this is a first order polynomial, it falls away in the diffusion equation, meaning that the solution for a $u(x,t)$ with general constant boundary conditions can be given as the solution with boundary conditions zero plus the steady state solution of the problem. In other words, the solution is given fully as:

$$u(x,t) = f(x) + \sum_{n=1}^{\infty} A_n \sin\Big(\frac{n\pi}{L}x\Big) e^{-\frac{n^2\pi^2}{L^2}t}$$

Finding the coefficients turns out slightly different, as we have:

$$g(x) = u(x,0)$$
$$g(x) = f(x) + \sum_{n=1}^{\infty} A_n \sin\Big(\frac{n\pi}{L}x\Big)$$

This is not a simple Fourier expansion, but by defining $h(x) = g(x) - f(x)$ we can rewrite the equation above as:

$$h(x) = \sum_{n=1}^{\infty} A_n \sin\Big(\frac{n\pi}{L}x\Big)$$

This is a Fourier expansion of $h(x)$, and thus we can find the coefficients:

$$A_n = \frac{2}{L}\int_0^L h(x) \sin\Big(\frac{n\pi}{L}x\Big)\, dx\,,$$

by using $h(x)$ instead of $g(x)$.

C. Numerical solutions of one-dimensional diffusion equation

1. *Euler methods*

First, we write down the diffusion equation in one dimension:

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}\,, \tag{3}$$

where x is the spatial coordinate, and other variables are as previously defined. This can be written in the short-hand notation:

$$u_{xx} = u_t\,,$$

where the subscripts denote a partial derivative with respect to the variable in the subscript. The spatial derivative part of this equation can be approximated:

$$u_{xx} = \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2} + \mathcal{O}(\Delta x^2)\,.$$

We also need to approximate the time derivative part, and we can do this in one of two ways, either using the forward approximation of the derivative:

$$u_t = \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} + \mathcal{O}(\Delta t)\,,$$

or the backwards approximation:

$$u_t = \frac{u(x,t) - u(x,t-\Delta t)}{\Delta t} + \mathcal{O}(\Delta t)\,.$$

We can discretize the equations by first defining a steplength:

$$\Delta x = \frac{1}{n+1}\,,$$

and a timestep Δt, and then defining a set of points for both x and t given by the subscripts i and j respectively:

$$x_i = i\Delta x \quad 0 \le i \le n+1$$
$$t_j = j\Delta t \quad 0 \le j\,,$$

with both i, j being positive integers or zero. We define the following short-hand notation:

$$u_{i,j} = u(x_i, t_j)\,.$$

We can then rewrite the spatial approximation using this notation:

$$u_{xx} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}\,,$$

the forward time approximation:

$$u_t \approx \frac{u_{i,j+1} - u_{i,j}}{\Delta t}\,,$$

and the backwards time approximation:

$$u_t \approx \frac{u_{i,j} - u_{i,j-1}}{\Delta t}\,.$$

Now, we set up the diffusion equation using the forward approximation in time first:

$$
\begin{aligned}
u_t &= u_{xx} \\
\frac{u_{i,j+1}-u_{i,j}}{\Delta t} &= \frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{\Delta x^2} \\
u_{i,j+1} &= u_{i,j} + \frac{\Delta t}{\Delta x^2}\Big(u_{i+1,j}-2u_{i,j}+u_{i-1,j}\Big) \\
u_{i,j+1} &= u_{i,j} + \alpha\Big(u_{i+1,j}-2u_{i,j}+u_{i-1,j}\Big), \qquad (4)
\end{aligned}
$$

where $\alpha = \Delta t/\Delta x^2$ (not to be confused with the constant used for scaling earlier). It is clear that if we know the initial state and boundary conditions of this system that we can move the system ahead in time using the equation above. At this point we make the assumption that the boundary conditions are zero. By the arguments given in Section II.B this can easily be extended to constant boundary conditions by use of the steady state solution of the problem. Implementing a numerical solution for other boundary conditions is not difficult either, as the only requirement we must have is that the solver cannot modify the boundary values outside to anything else than what is specified by the boundary conditions. Assuming Dirichlet boundary conditions simplify the calculaton of the condition for convergence, and so we keep them in this section.

The equation above gives directly the solution in the next timestep, and thus this is an explicit scheme. This can also be rewritten as a matrix vector equation by using $U_j = [u_{1,j} \quad u_{1,j} \quad ... \quad u_{n,j}]^T$. And the matrix:

$$
A_{\mathrm{f}} = \begin{bmatrix}
1-2\alpha & \alpha & 0 & \cdots & 0 \\
\alpha & 1-2\alpha & \alpha & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
0 & \cdots & \alpha & 1-2\alpha & \alpha \\
0 & \cdots & 0 & \alpha & 1-2\alpha
\end{bmatrix},
$$

where the subscript f denotes that this is the scheme based on the forward Euler method. This gives us that the explicit scheme can be described by the matrix-vector equation:

$$
U_{j+1} = A_{\mathrm{f}} U_j \,.
$$

Note that if we know the solution at a previous timestep U_j we can directly determine the solution in the next timestep U_{j+1}, and thus we do not need to solve this matrix-vector equation per se. It is just a practical way of denoting things that we will get back to. The vector U_j does not contain the boundary elements $u_{0,j}$ and $u_{n+1,j}$ as they are generally assumed to be zero.

Using the backwards time approximation in a similar fashion gives us the following:

$$
\begin{aligned}
u_t &= u_{xx} \\
\frac{u_{i,j}-u_{i,j-1}}{\Delta t} &= \frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{\Delta x^2} \\
u_{i,j-1} &= u_{i,j} - \frac{\Delta t}{\Delta x^2}\Big(u_{i+1,j}-2u_{i,j}+u_{i-1,j}\Big) \\
u_{i,j-1} &= u_{i,j} - \alpha\Big(u_{i+1,j}-2u_{i,j}+u_{i-1,j}\Big). \qquad (5)
\end{aligned}
$$

As we cannot directly determine the solution in the next timestep from this equation, this is an implicit scheme, and we must formulate a way we can solve this equation. We can write down the above equation as a matrix-vector equation by using the same vector U_j as earlier and the matrix:

$$
A_{\mathrm{b}} = \begin{bmatrix}
1+2\alpha & -\alpha & 0 & \cdots & 0 \\
-\alpha & 1+2\alpha & -\alpha & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
0 & \cdots & -\alpha & 1+2\alpha & -\alpha \\
0 & \cdots & 0 & -\alpha & 1+2\alpha
\end{bmatrix},
$$

where the subscript b denotes that this is the matrix based on the backwards Euler method. This gives us the matrix-vector equation:

$$
\begin{aligned}
U_{j-1} &= A_{\mathrm{b}} U_j \\
A_{\mathrm{b}}^{-1} U_{j-1} &= U_j \,.
\end{aligned}
$$

In order to determine the solution in the next timestep j it is thus necessary to solve this matrix-vector equation. This can be done in a multitude of ways, but the simplest way of solving this is by recognizing that this matrix is tridiagonal if we exclude the boundaries, and that we can thus use a specialized solver for such a matrix-vector equation (see Section II.C.4). Solving the equation as such greatly reduces the computational cost of solving the problem compared to a general matrix inversion.

We also wish to examine the convergence of these schemes. By defining the matrix:

$$
B = \begin{bmatrix}
2 & -1 & 0 & \cdots & 0 \\
-1 & 2 & -1 & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
\vdots & \vdots & \ddots & \ddots & -1 \\
0 & 0 & \cdots & -1 & 2
\end{bmatrix}, \qquad (6)
$$

we can rewrite $A_{\mathrm{f}} = I - \alpha B$ and $A_{\mathrm{b}} = I + \alpha B$. First we look at the explicit scheme.

For the solution to converge we need the spectral radius to be less than one. The spectral radius of a matrix is the absolute of the largest eigenvalue of the matrix. Thus we need to find the eigenvalues of A_{f}, λ. Since $A_{\mathrm{f}} = I - \alpha B$

it is obvious that it has eigenvalues $\lambda_k = 1 - \alpha\mu_k$ where μ are the eigenvalues of B and the subscript k denotes which eigenvalue we are looking at. The eigenvalues of B are ([1, p. 307]):

$$\mu_k = 2 - 2\cos\left(\frac{k\pi}{n} + 1\right),$$

where n is as defined earlier. The requirement for the solution to converge is that the spectral radius $\rho(A_\text{f})$:

$$\begin{aligned} \rho(A_\text{f}) &< 1 \\ |\max(\lambda_k)| &< 1 \\ |\max(1 - 2\alpha(1 - \cos\left(\frac{k\pi}{n} + 1\right))| &< 1 \end{aligned}$$

This is only the case if:

$$\begin{aligned} 2\alpha(1 - \cos\left(\frac{k\pi}{n} + 1\right)) &< 2 \\ \alpha &< (1 - \cos\left(\frac{k\pi}{n} + 1\right))^{-1} \end{aligned}$$

The right-hand side can never be smaller than 1/2, which gives us that α has to be less than 1/2 in order for the solution to converge. The condition for convergence with the explicit scheme is thus:

$$\frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}, \tag{7}$$

In order to determine the condition for convergence with the implicit scheme we need that $\rho(A_\text{b}^{-1}) < 1$. This is the case if $\rho(A_\text{b}) > 1$. As we can write $A_\text{b} = I + \alpha B$, we can infer that the eigenvalues of A_b are $\lambda_k = 1 + 2\alpha\mu_k = 1 + 2\alpha(1 - \cos(k\pi/n + 1))$ which is always greater than one. Thus the spectral radius $\rho(A_\text{b})$ is always greater than one, and the numerical solution converges for any choice of α.

2. *Crank-Nicolson scheme*

The two schemes outlined above can be combined into a single scheme using a parameter θ:

$$\frac{\theta}{\Delta x^2}\left(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}\right) + \frac{1-\theta}{\Delta x^2}\left(u_{i+1,j-1} - 2u_{i,j-1} + u_{i-1,j-1}\right) = \frac{1}{\Delta t}\left(u_{i,j} - u_{i,j-1}\right) \tag{8}$$

We recognize that by setting $\theta = 0$ this returns the explicit scheme, and that it returns the implicit scheme when $\theta = 1$. If we set $\theta = 1/2$, however, we get the Crank-Nicolson scheme. By using $\alpha = \Delta t/\Delta x^2$ we can in this case rewrite the equation above as:

$$\begin{aligned} &-\alpha u_{i-1,j} + (2 + 2\alpha)u_{i,j} - \alpha u_{i+1,j} \\ &= \alpha u_{i-1,j-1} + (2 - 2\alpha)u_{i,j-1} + \alpha u_{i+1,j-1} \end{aligned} \tag{9}$$

For the full derivation of this scheme we refer to [1, p. 311]. It is derived from approximating around a midpoint $t' = t + \Delta t/2$, which gives that the error from the time approximation goes as $\mathcal{O}(\Delta t^2)$ instead of $\mathcal{O}(\Delta t)$. The spatial approximation still results in an error $\mathcal{O}(\Delta x^2)$, however.

Equation (9) can be rewritten as a matrix-vector equation using the identity matrix I, the matrix B and the vector U_j as defined in the previous section, and by assuming Dirichlet boundary conditions:

$$\begin{aligned} (2I + \alpha B)U_j &= (2I - \alpha B)U_{j-1} \\ U_j &= (2I + \alpha B)^{-1}(2I - \alpha B)U_{j-1} \end{aligned} \tag{10}$$

In other words a matrix inversion is required in order to find the solution in the next timestep. This means that this scheme is an implicit one. First we need to multiply the previous solution with the matrix $(2I - \alpha B)$, which is straightforward. After that we need to find the inverse of $(2I + \alpha B)$ and apply this. As this matrix is tridiagonal we can use a tridiagonal solver for this operation. Thus the numerical solution of this scheme can be split into two parts, one which functions similar to the solver for the forward Euler based explicit scheme, and a second part that functions similarly to the solver for the backwards Euler based implicit scheme.

The solution with the Crank-Nicolson scheme converges if the spectral radius:

$$\rho\left((2I + \alpha B)^{-1}(2I - \alpha B)\right) < 1.$$

This is the case if:

$$\left|\frac{2 - \alpha\mu_k}{2 + \alpha\mu_k}\right| < 1$$

As $\mu_k = 2 - 2\cos(k\pi/n + 1)$ is always positive this condition is always met, and thus the Crank-Nicolson scheme also converges for any α. Note that the numerical solution using this scheme can also easily be adapted to general boundary conditions in the same fashion as was discussed in the previous section.

3. *Convergence and errors in numerical schemes*

In order to easily refer to the errors and convergence requirements for each scheme later on in the report we list them in Table I, which is nearly the same as a table found on page 312 in [1].

Table I. This table lists the truncation and errors and stability requirements in the three numerical schemes (see Section II.C) used for solving the one-dimensional diffusion equation as they relate to the timestep Δt and the steplength Δx.

Scheme:	Truncation error:	Stability requirements:
Forward Euler	$\mathcal{O}(\Delta t)$ and $\mathcal{O}(\Delta x^2)$	$\Delta t < \frac{1}{2}\Delta x^2$
Backward Euler	$\mathcal{O}(\Delta t)$ and $\mathcal{O}(\Delta x^2)$	None on Δt and Δx
Crank-Nicolson	$\mathcal{O}(\Delta t^2)$ and $\mathcal{O}(\Delta x^2)$	None on Δt and Δx

4. Tridiagonal solver

In the previous sections we have mentioned that we can use a tridiagonal solver to solve the matrix-vector equations we get. We define a tridiagonal matrix where the elements are constant along the diagonal bands:

$$A = \begin{bmatrix} b & c & 0 & \cdots & 0 \\ a & b & c & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & a & b & c \\ 0 & \cdots & \cdots & a & b \end{bmatrix}, \tag{11}$$

and the matrix-vector equation that we need to solve:

$$Au = f\,,$$

where u and f are vectors. Dirichlet boundary conditions are assumed such that $u_0 = u_{n+1} = 0$, and that these components are thus not included in the vector u. Assuming that we know f as well, we can find the components of u by first finding:

$$\begin{aligned} \tilde{b}_i &= b - \frac{ac}{\tilde{b}_{i-1}} \\ \tilde{f}_i &= f_i - \frac{\tilde{f}_{i-1}a}{\tilde{b}_{i-1}}\,, \end{aligned} \tag{12}$$

where the subscript i indicates which component of the vectors we are looking at. The first components need to be handled separately:

$$\begin{aligned} \tilde{b}_1 &= b \\ \tilde{f}_1 &= f_1\,. \end{aligned}$$

We can then find the components of u as:

$$u_i = \frac{\tilde{f}_i - cu_{i+1}}{\tilde{b}_i}\,, \tag{13}$$

where the endpoint is found separately as:

$$u_n = \frac{\tilde{f}_n}{\tilde{b}_n}\,.$$

This can be performed as first a forward iteration to find $\tilde{b}_i$ and $\tilde{f}_i$, and a backwards iteration to find u_i. This iterative solver is discussed in more detail in [2].

This solution assumes Dirichlet boundary conditions, but if they are non-zero a simple adaptation can be made to accommodate this. If we assume that the boundary values u_0 and u_{n+1} are known, non-zero values (which also means we have to exclude these indices from the iterations above), we need to use that:

$$\tilde{f}_1 = f_1 - au_0\,,$$

and that u_n is no longer handled as a special case. By these simple adaptations the solver now works for non-zero boundary conditions.

D. Analytical solution of diffusion equation in two dimensions

The equation we wish to solve is the two-dimensional diffusion equation:

$$\frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2} = \frac{\partial u(x,y,t)}{\partial t} \tag{14}$$

We wish to solve for a square system, with $x, y \in [0, L]$. We assume a general initial state $u(x,y,0) = g(x,y)$ and Dirichlet boundary conditions $u(0,y,t) = u(L,y,t) = u(x,0,t) = u(x,L,t) = 0$. We assume separation of variables:

$$u(x,y,t) = F(x,y)G(t)$$

The diffusion equation then gives:

$$\begin{aligned} G\left(\frac{\partial^2 F}{\partial x^2} + \frac{\partial^2 F}{\partial y^2}\right) &= F\frac{\partial G}{\partial t} \\ GF'' &= FG' \\ \frac{F''}{F} &= \frac{G'}{G}\,, \end{aligned} \tag{15}$$

where:

$$\begin{aligned} F'' &= \frac{\partial^2 F}{\partial x^2} + \frac{\partial^2 F}{\partial y^2} \\ G' &= \frac{\partial G}{\partial t} \end{aligned}$$

In equation (15) the right-hand side and left-hand side do not share any variables, and thus they must be constant. We define this constant as $-\lambda^2$. This gives us the differential equations:

$$\begin{aligned} F'' + \lambda^2 F &= 0 \\ G' + \lambda^2 G &= 0 \end{aligned}$$

The second of these has the familiar solution:

$$G(t) = Ee^{-\lambda^2 t}\,,$$

where E is an integration constant. The first differential equation is not as easy to solve, but we can do it by again assuming separation of variables in F:

$$F(x, y) = X(x)Y(y)$$

Inserting this into the differential equation gives:

$$\begin{aligned} F'' &= -\lambda^2 F \\ Y\frac{\partial^2 X}{\partial x^2} + X\frac{\partial^2 Y}{\partial y^2} &= -\lambda^2 XY \\ \frac{X''}{X} + &= -\frac{Y''}{Y} - \lambda^2\,, \end{aligned}$$

where:

$$\begin{aligned} X'' &= \frac{\partial^2 X}{\partial x^2} \\ Y'' &= \frac{\partial^2 Y}{\partial y^2}\,. \end{aligned}$$

As X is only dependent on x and Y only dependent on y, this means that:

$$\frac{X''}{X} = -\frac{Y''}{Y} - \lambda^2 = \text{constant}\,.$$

We define this constant to be $-\mu^2$. And we also define $\nu^2 = \lambda^2 - \mu^2$. This gives us the following set of differential equation:

$$\begin{aligned} X'' + \mu^2 X &= 0 \\ Y'' + \nu^2 Y &= 0\,, \end{aligned}$$

which have solutions:

$$\begin{aligned} X(x) &= A\sin(\mu x) + B\cos(\mu x) \\ Y(y) &= C\sin(\nu y) + D\cos(\nu y) \end{aligned}$$

The boundary conditions imply that $B = D = 0$, and that $\mu = n\pi/L$ and $\nu = m\pi/L$, where n and m are positive integers. We also have that $\lambda^2 = \nu^2 + \mu^2 = (n^2 + m^2)\pi^2/L^2$. This gives us that a solution of the diffusion equation in two dimensions can be written as:

$$u_{n,m}(x, y, t) = A_{n,m}\sin\Big(\frac{n\pi}{L}x\Big)\sin\Big(\frac{m\pi}{L}y\Big)e^{-\frac{(n^2+m^2)\pi^2}{L^2}t}\,,$$

where the coefficients $A_{n,m}$ include the constants A, C and E. A general solution can be written as a linear combination of these (sum over all n and m):

$$u(x, y, t) = \sum_{n=1}^{\infty}\sum_{m=1}^{\infty} A_{n,m}\sin\Big(\frac{n\pi}{L}x\Big)\sin\Big(\frac{m\pi}{L}y\Big)e^{-\frac{(n^2+m^2)\pi^2}{L^2}t} \tag{16}$$

In order to find an expression for the coefficients $A_{n,m}$ we first write down the initial state of the system:

$$\begin{aligned} g(x, y) &= u(x, y, 0) \\ g(x, y) &= \sum_{n=1}^{\infty}\sum_{m=1}^{\infty} A_{n,m}\sin\Big(\frac{n\pi}{L}x\Big)\sin\Big(\frac{m\pi}{L}y\Big) \end{aligned}$$

Similarly to the one-dimensional case, this is reminiscent of a Fourier expansion. As x and y are independent, we can find the coefficients $A_{n,m}$ as:

$$A_{n,m} = \frac{4}{L^2}\int_0^L\int_0^L g(x, y)\sin\Big(\frac{n\pi}{L}x\Big)\sin\Big(\frac{m\pi}{L}y\Big)\,dx\,dy \tag{17}$$

This lets us determine the analytical solution in two dimensions with Dirichlet boundary conditions for any initial state $g(x, y)$ such that this integral is solvable.

Again, similarly to the one-dimensional case, we can extend this solution for general time-independent boundary conditions. If the boundary conditions are independent of time we can simply add the steady-state function of that system to the solution with Dirichlet boundary conditions:

$$u(x, y, t) = v(x, y, t) + f(x, y)\,,$$

where $v(x, y, t)$ is the solution with Dirichlet boundary conditions, and $f(x, y)$ is the steady state solution. The steady state solution must fulfill the Laplace equation:

$$\nabla^2 f(x, y) = 0\,,$$

constrained by the boundary conditions specified. As long as the Laplace equation holds for the steady state solution, it is easy to the see that adding it to $v(x, y, t)$ does not disturb the diffusion equation:

$$\begin{aligned}
\nabla^2 u(x,y,t) &= \frac{\partial u(x,y,t)}{\partial t} \\
\nabla^2 (v(x,y,t) + f(x,y)) &= \frac{\partial v(x,y,t)}{\partial t} + \frac{\partial f(x,y)}{\partial t} \\
\nabla^2 v(x,y,t) &= \frac{\partial v(x,y,t)}{\partial t}\,,
\end{aligned}$$

where we have used that the Laplace equation holds for $f(x,y)$ and that it is independent of time. Finding a general steady state solution is not as easy as in the one-dimensional case, however, as the boundary conditions generally will only be constant with respect to one of the spatial coordinates unless they are all equal to the same constant value. This has to be the case in order for the boundary conditions to be continuous in the corners. Note that it is also possible for the boundaries in one of the spatial dimensions to be constant and unequal to eachother, but this then automatically demands that the boundary counditions in the other spatial dimension cannot be constant with respect to both spatial coordinates.

E. Numerical solution of diffusion equation in two dimensions

We wish to solve the two-dimensional diffusion equation (14) numerically. The scheme we want to find should be implicit, as that will converge for any choice of timestep Δt and steplength h. First we will make some simplifications. We assume Dirichlet boundary conditions, a general initial configuration $u(x,y,0) = g(x,y)$, and $x, y \in [0,1]$. A discretization will also be necessary, and so we define:

$$\begin{aligned}
x_i &= ih \quad 0 \le i \le n+1 \\
y_j &= jh \quad 0 \le j \le n+1 \\
t_l &= l\Delta t \quad 0 \le l\,,
\end{aligned}$$

where $h = 1/(n+1)$, $n+2$ is the number of gridpoints, and Δt is the timestep. We denote a value of the solution u by the indices i, j and l as follows:

$$u^l_{i,j} = u(x_i, y_j, t_l)$$

Using this notation we can approximate the derivatives in the diffusion equation:

$$\begin{aligned}
\frac{\partial^2 u}{\partial x^2} &= \frac{u^l_{i+1,j} - 2u^l_{i,j} + u^l_{i-1,j}}{h^2} + \mathcal{O}(h^2) \\
\frac{\partial^2 u}{\partial y^2} &= \frac{u^l_{i,j+1} - 2u^l_{i,j} + u^l_{i,j-1}}{h^2} + \mathcal{O}(h^2) \\
\frac{\partial u}{\partial t} &= \frac{u^l_{i,j} - u^{l-1}_{i,j}}{\Delta t} + \mathcal{O}(\Delta t)
\end{aligned}$$

The diffusion equation can then be written using these approximations as:

$$\begin{aligned}
\frac{u^l_{i,j} - u^{l-1}_{i,j}}{\Delta t} &= \frac{u^l_{i+1,j} + u^l_{i-1,j} + u^l_{i,j+1} + u^l_{i,j-1} - 4u^l_{i,j}}{h^2} \\
u^l_{i,j} - u^{l-1}_{i,j} &= \alpha(\Delta^l_{i,j} - 4u^l_{i,j})\,,
\end{aligned}$$

where $\alpha = \Delta t/h^2$ and:

$$\Delta^l_{i,j} = u^l_{i+1,j} + u^l_{i-1,j} + u^l_{i,j+1} + u^l_{i,j-1}$$

We can write this in terms of $u^l_{i,j}$:

$$u^l_{i,j} = \frac{1}{1+4\alpha}(\alpha\Delta^l_{i,j} + u^{l-1}_{i,j}) \tag{18}$$

As we have now moved to two dimensions this cannot be represented as a matrix-vector equation as in the one-dimensional case. This makes it difficult to solve this with the methods we have used so far, and so we need to find an alternative way of solving this. For this purpose we choose to use Jacobi's iterative method (this method is outlined in [1, p .189-190]). With a standard matrix-vector equation:

$$Ax = b\,,$$

this method can be formulated as:

$$x^{k+1} = D^{-1}(b - (L+U)x^k)\,, \tag{19}$$

where $A = D + U + L$, D is a diagonal matrix, U an upper triangular matrix, L a lower triangular matrix, and k denotes the iteration. This is iterated until x^k is sufficiently close to x^{k+1}, which can be said to be the case when the absolute difference between all of their elements is less than a specified tolerance limit.

We can rewrite this for a component i in terms of the elements of A, $a_{i,j}$, as follows:

$$x^{k+1}_i = \frac{1}{a_{ii}}(b_i - \sum_i \sum_{j\neq i} a_{i,j} x^k_j)$$

This can be adapted to a two-dimensional case by adding more indices to a:

$$x^{k+1}_{i,j} = \frac{1}{a_{i,j,i,j}}(b_{i,j} - \sum_i \sum_j \sum_{m\neq i} \sum_{n\neq j} a_{i,j,m,n} x^k_{m,n})\,,$$

where $a_{i,j,i,j}$ equates to the "diagonal elements" in this case. We wish to adopt this iterative scheme to solve the 2D diffusion equation as it is given in (18). This equation is already nearly on the form of what we need for the iterative scheme if we set:

$$
\begin{aligned}
x_{i,j} &= u^l_{i,j} \\
b_{i,j} &= u^{l-1}_{i,j} \\
\alpha\Delta^l_{i,j} &= -\sum_m \sum_n a_{i,j,m,n} u^l_{m,n}
\end{aligned}
$$

We can then also add the iteration index k, in order to obtain the equation that we iterate:

$$
u^{l,k+1}_{i,j} = \frac{1}{1+4\alpha}(\alpha\Delta^{l,k}_{i,j} + u^{l-1}_{i,j})\,, \tag{20}
$$

where $u^l_{i,j}$ is left as a constant during the iteration. If the solution has converged so that $u^{l,k+1}_{i,j}$ is sufficiently close to $u^{l,k}_{i,j}$ for all i and j, we stop the iteration. In the one-dimensional case, this solution converges if the spectral radius $\rho(D^{-1}(L+U)) < 1$. This is always the case if the matrix A is diagonally dominant. In the two-dimensional case, this equates to:

$$
\begin{aligned}
|a_{i,j,i,j}| &> \sum_{m\neq i} \sum_{n\neq j} |a_{i,j,m,n}| \\
|1+4\alpha| &> 4|-\alpha|\,,
\end{aligned}
$$

which is always the case as α is a positive number. Thus this method converges to the correct answer for any choice of timestep Δt and steplength h.

Assuming boundary conditions that differ from the Dirichlet conditions is not difficult, as we only need to set them manually and make sure that the solver does not edit them.

F. Temperature distribution in the lithosphere

In order to show the functionality of the two-dimensional solver that we implement we wish to apply it to a real-world problem. This problem is originally outlined in [3, p. 4-5], due to this there may be similarities between what is written here and that which is written there.

Based on geological evidence found at the surface, it has been proposed that there was an active subduction zone on the western coast of Norway about 1 Gy ago. This causes a refertilization of the mantle wedge, because the subducting oceanic crust releases water and other chemical components that may be trapped in it into the mantle above as it melts in the lower mantle. The concentration of radioactive elements in these chemical components is higher than what is regularly found in the mantle, and therefore it is expected that the mantle will be warmer than it would be if the refertilization hadn't occured. By simulating this model numerically in two dimensions we can give an estimate to what kind of temperature difference this might cause, which can later be compared with experimental results. In order to do this we need to solve the heat equation in two dimensions with an added source term:

$$
\frac{\partial^2 T(x,y,t)}{\partial x^2} + \frac{\partial^2 T(x,y,t)}{\partial y^2} + \frac{Q(x,y,t)}{k} = \frac{\rho c_p}{k}\frac{\partial T(x,y,t)}{\partial t}\,, \tag{21}
$$

where Q is the source term and all other variables are as defined in equation (2). It is assumed that the boundary conditions are constant in time, and that the temperature at the surface is 8° C, and at the bottom of the upper mantle is 1300° C. We divide the lithosphere into three parts, the upper crust from 0 to 20 km depth, the lower crust from 20 to 40 km depth, and the upper mantle from 40 to 120 km depth. For the entire lithosphere we assume a constant density $\rho = 3.5 \times 10^3$ kg/m^3, thermal conductivity $k = 2.5$ W/m/K and specific heat $c_p = 1000$ J/kg/K.

Before the fertilization some radioactive materials are already present, and they cause a (assumed constant) heat production of about 1.4 μW/m^3 in the upper crust, 0.35 μW/m^3 in the lower crust and 0.05 μW/m^3 in the mantle. After the fertilization a 150 km wide area above the subducting slab is enriched with Uranium, Thorium and Potassium such that an additional 0.5 μW/m^3 is present in the mantle (not the crust). These elements have halflives of 4.47 Gy, 14.0 Gy and 1.25 Gy respectively.

As for the dimensionality of the model we decide to model the depth as the y-direction with then $y \in [0, 120]$ km. The x-direction needs to be wider than the area that is enriched, and so we choose $x \in [0, 300]$ km, with the enriched area of the mantle being $x \in [75, 225]$ km.

1. Modeling of the heat source

Before the enrichment the heat source comes purely from the radioactive elements already present. The added heat source on a function form can be written as:

$$
Q_{\text{before}}(x,y,t) = \begin{cases} 1.4\ \mu\text{W/m}^3 & \text{upper crust} \\ 0.35\ \mu\text{W/m}^3 & \text{lower crust} \\ 0.05\ \mu\text{W/m}^3 & \text{mantle} \end{cases}, \tag{22}
$$

where the upper crust corresponds to $y < 20$ km, the lower crust to 20 km $< y <$ 40 km and the mantle corresponds to $y > 40$ km. When the mantle is refertilized a term is added to the heat generated in the mantle. This term is 0.5 μW/m^3 initially at $t = 0$, but decays with the decay of the radioactive elements. The rate of decay can be expressed as an exponential:

$$
D(t) = e^{-\frac{\ln(2)}{T_{1/2}}t}\,,
$$

where $T_{1/2}$ is the halflife of the radioactive element. We assume that the composition of the radioactive elements is 40 % Thorium, 40 % Uranium, and 20 % Potassium. We define the additional term as:

$$Q_{\text{add}}(t) = (0.4e^{-t(\ln(2)/(4.47\text{ Gy}))} + 0.4e^{-t(\ln(2)/(14.0\text{ Gy}))} + 0.2e^{-t(\ln(2)/(1.25\text{ Gy}))}) \cdot 0.5\ \mu\text{W/m}^3 \tag{23}$$

This means that the heat source can be written on function form as:

$$Q_{\text{after}}(x,y,t) = \begin{cases} 1.4\ \mu\text{W/m}^3 & \text{upper crust} \\ 0.35\ \mu\text{W/m}^3 & \text{lower crust} \\ 0.05\ \mu\text{W/m}^3 + Q_{\text{add}}(t) & \text{enriched mantle} \\ 0.05\ \mu\text{W/m}^3 & \text{regular mantle} \end{cases}, \tag{24}$$

where the enriched mantle corresponds to $y > 40$ km and 75 km $< x < 225$ km, and the regular mantle corresponds to $y > 40$ km, $x < 75$ km and $x > 225$ km.

2. *Numerical solution*

We need to formulate a numerical solution to the heat equation with an added source term (21). First we note that we wish to have $x \in [0, 300]$ km and $y \in [0, 120]$ km, or in other words a non-square grid. We can define a set of parameters α, a_x and a_y such that:

$$x = \alpha a_x \hat{x}$$
$$y = \alpha a_y \hat{y}\,,$$

where $\hat{x}, \hat{y} \in [0, 1]$ are dimensionless parameters. This implies that αa_x and αa_y have units of length, and that $\alpha a_x = 300$ km, and $\alpha a_y = 120$ km. The heat equation in two dimensions can then be written as:

$$\frac{1}{\alpha^2 a_x^2}\frac{\partial^2 T}{\partial \hat{x}^2} + \frac{1}{\alpha^2 a_y^2}\frac{\partial^2 T}{\partial \hat{y}^2} = D\frac{\partial T}{\partial t} - \frac{Q}{k}\,,$$

where $D = \rho c_p/k$. We can multiply by α^2 on both sides:

$$\frac{1}{a_x^2}\frac{\partial^2 T}{\partial \hat{x}^2} + \frac{1}{a_y^2}\frac{\partial^2 T}{\partial \hat{y}^2} = \alpha^2 D\frac{\partial T}{\partial t} - \frac{Q}{k}\alpha^2$$

We now wish to set α such that $\alpha^2 D = 1$, which means that $\alpha^2 = 1/D$. Inserting this gives:

$$\frac{1}{a_x^2}\frac{\partial^2 T}{\partial \hat{x}^2} + \frac{1}{a_y^2}\frac{\partial^2 T}{\partial \hat{y}^2} = \frac{\partial T}{\partial t} - \frac{Q}{\rho c_p}$$

Here we note that by this choice of α^2 that α^2 has the units as the constant $1/D$, which has units of m^2/s (if we use the values given earlier for the parameters that make up D). As we wish that αa_x and αa_y has units of length, this implies that a_x and a_y have units of s$^{1/2}$. We can then determine a_x and a_y:

$$a_x = \frac{300 \times 10^3\text{ m}}{\alpha} = 300 \times 10^3\text{ m} \cdot \sqrt{D} \approx 3.55 \times 10^8\text{ s}^{1/2}$$
$$a_y = \frac{120 \times 10^3\text{ m}}{\alpha} = 120 \times 10^3\text{ m} \cdot \sqrt{D} \approx 1.42 \times 10^8\text{ s}^{1/2}$$

In order to simplify the notation we define:

$$A_x = \frac{1}{a_x^2}$$
$$A_y = \frac{1}{a_y^2}$$
$$Q' = \frac{Q}{\rho c_p}$$

The equation can then be written as:

$$A_x\frac{\partial^2 T}{\partial \hat{x}^2} + A_y\frac{\partial^2 T}{\partial \hat{y}^2} = \frac{\partial T}{\partial t} - Q'$$

To proceed we need to define a discretization of the coordinates. To this end we set:

$$\hat{x}_i = ih \quad 0 \le i \le n$$
$$\hat{y} = jh \quad 0 \le j \le n$$
$$t_l = l\Delta t \quad 0 \le l\,,$$

where i, j and l are integers, $h = 1/n$ and Δt is a timestep that we can choose. We use the notation:

$$T(\hat{x}, \hat{y}, t_l) = T^l_{i,j}$$

We can approximate the spatial derivatives in the heat equation as we have earlier:

$$\frac{\partial^2 T}{\partial \hat{x}^2} = \frac{T^l_{i+1,j} - 2T^l_{i,j} + T^l_{i-1,j}}{h^2} + \mathcal{O}(h^2)$$
$$\frac{\partial^2 T}{\partial \hat{y}^2} = \frac{T^l_{i,j+1} - 2T^l_{i,j} + T^l_{i,j-1}}{h^2} + \mathcal{O}(h^2)\,,$$

and the time derivative using the backwards Euler approximation:

$$\frac{\partial T}{\partial t} = \frac{T^l_{i,j} - T^{l-1}_{i,j}}{\Delta t} + \mathcal{O}(\Delta t)$$

Note that the added source term Q' is also a function of $\hat{x}$, $\hat{y}$ and t, and so it too needs to be discretized. As there is no approximation involved, however, we decide to not include indices on Q' and generally it is assumed that:

$$Q' = Q'^{l}_{i,j} = Q'(\hat{x}_i, \hat{y}_j, t_l)\,.$$

By defining:

$$\beta = \frac{\Delta t}{h^2}\,,$$

and:

$$\Delta^l_{i,j} = A_x(T^l_{i+1} + T^l_{i-1,j}) + A_y(T_{i,j+1} + T_{i,j-1})\,,$$

we can write the discretized heat equation as:

$$\begin{aligned} \frac{T^l_{i,j} - T^{l-1}_{i,j}}{\Delta t} &= \frac{\Delta^l_{i,j} - 2(A_x + A_y)T^l_{i,j}}{h^2} + Q' \\ T^l_{i,j} - T^{l-1}_{i,j} &= \beta(\Delta^l_{i,j} - 2(A_x + A_y)T^l_{i,j}) + \Delta t Q' \\ T^l_{i,j} &= \frac{1}{1 + 2\beta(A_x + A_y)}(\beta\Delta^l_{i,j} + T^{l-1}_{i,j} + \Delta t Q') \end{aligned} \tag{25}$$

This is similar to what we found in equation (18), and can be solved in the same fashion (see Section II.E) by setting the components of the matrix b to be:

$$b_{i,j} = T^{l-1}_{i,j} + \Delta t Q'\,,$$

the "diagonal" elements to be $1+2\beta(A_x+A_y)$ and the "off-diagonal" elements to be either $-A_x\beta$ or $-A_y\beta$ (two elements of each value for each diagonal element). The solution still converges, as the absolute of the "diagonal" elements are still larger than the absolute of the "off-diagonal" elements added together:

$$|1 + 2\beta(A_x + A_y)| > |-2A_x\beta| + |-2A_y\beta|$$

This equality always holds true, as A_x, A_y and β are all positive numbers. This means that we can use the same iterative solver as we did earlier, assuming that it is sufficiently generalized.

3. *Analytical discussion*

First we wish to find the steady-state solution without considering the effects of the radioactive elements. In order to do this, we need to solve Laplace's equation:

$$\frac{\partial^2 T_s(x,y)}{\partial x^2} + \frac{\partial^2 T_s(x,y)}{\partial y^2} = 0\,,$$

where the subscript s is used to mark this as the steady-state solution. We know only the boundary conditions in the y-direction:

$$\begin{aligned} T_s(x, 0\text{ km}) &= 8\ ^\circ\text{C} \\ T_s(x, 120\text{ km}) &= 1300\ ^\circ\text{C}\,, \end{aligned} \tag{26}$$

and not the boundaries in the x-direction. That does not mean we cannot progress beyond this point, however, as we expect the temperature to depend only on depth, as there are no other sources of heat at this point. This means that we expect the temperature gradient to be only dependent on y ($T_s(x,y) \to T_s(y)$). Laplace's equation is then reduced to only one dimension:

$$\frac{\partial^2 T_s(y)}{\partial y^2} = 0\,,$$

which results in the steady-state being a first-order polynomial of y, depending on the boundary conditions:

$$T_s(y) = \frac{1300\ ^\circ\text{C}\ - 8\ ^\circ\text{C}}{120\text{ km}}y + 8\ ^\circ\text{C}\,. \tag{27}$$

We will also use this result as the boundary conditions in the x-direction, so that:

$$T(0\text{ km}, y) = T(300\text{ km}, y) = T_s(y)\,. \tag{28}$$

Now, when we introduce the radioactive elements we expect the steady-state solution to differ from this, as we would need to add the source term when solving for the steady state as well. The steady state is defined as when the system reaches an equillibrium where it is constant in time. Before the enrichment the additional source term is independent of time and the x-coordinate, meaning that the steady-state solution can be found by solving Poisson's equation:

$$\frac{\partial^2 T_{s,\text{before}}}{\partial y^2} = -\frac{Q_{\text{before}}(y)}{k}\,,$$

This is not an equation that is easily solved, and thus we will not do so. However, this allows us to say something about how we expect the numerically calculated steady-state solution to look like. Generally we expect the added heat to result in an increase in the temperature for all depths, except in the limits where the boundary values still reign. This can be seen from the fact that the right-hand side in the above equation is negative, meaning that the change in temperature as a function of depth is larger in the crust than in the mantle. Thus we expect a departure from the pure linear behaviour that is expected if there was no radioactive materials, with the temperatures at all points being higher than in the case when there are no additional heat sources.

After the mantle is enriched further we expect some of the same differences. It is quite easy to see that the additional heat $Q_{\text{add}}(y,t)$ (see equation (23)) tends to zero as $t \to \infty$. This means that the steady state solution remains the same as before the further enrichment. It is still interesting however, to discuss how we expect the temperature distribution to look like after an amount of time such that the additional heat term has not disappeared. The source term is still positive for all depths, and so the temperatures should be higher compared to the steady state with no radioactive materials. The temperatures should also be higher than before the further enrichment, as the only change is that there is more heat being produced in the upper mantle. The temperature will in this case also not be linear in terms of the depth, but as an added complication it will also not be constant along the x-axis, as only a portion of the upper mantle is further enriched in radioactive materials. As we have positioned this region in the middle of our x-axis we expect the temperature to vary along the x-axis, reaching a peak around the middle of the x-axis. The temperatures should gradually increase as we move from one of the edges of the system towards the middle along the x-axis.

4. *Choices of units and parameter values*

We need to decide on which units that are to be used for the parameters in the simulation. In general we use SI units, but as we are interested in looking at the results on a geological timescale, we decide to use gigayears as the unit of time. This has some implications. Particularly, the values calculated for the parameters a_x and a_y in Section II F 2 are instead:

$$a_x \approx 3.55 \times 10^8 \text{ s}^{1/2} \approx 2.00 \text{ Gy}^{1/2}$$
$$a_y \approx 1.42 \times 10^8 \text{ s}^{1/2} \approx 0.80 \text{ Gy}^{1/2} \tag{29}$$

We also need the value that we should multiply the heat source terms Q with in order to get Q' in units of K/Gy. The heat source terms Q are in units μW/m^3, where W is J/s. We have that:

$$\frac{1}{\text{s}} \approx 3.15 \times 10^{16} \frac{1}{\text{Gy}} .$$

We also need to find the value of $1/(\rho c_p)$ as $Q' = Q/(\rho c_p)$. This value is:

$$\frac{1}{\rho c_p} = \frac{1}{3500 \frac{\text{kg}}{\text{m}^3} \cdot 1000 \frac{\text{J}}{\text{kg K}}} = 2.86 \times 10^{-7} \frac{\text{K m}^3}{\text{J}}$$

Lastly we also need a factor 10^{-6} as the added heat has units of μW/m^3 and not W/m^3. Tallying up all the factors gives us:

$$Q' \approx Q \cdot 3.15 \times 10^{16} \cdot 2.86 \times 10^{-7} \frac{\text{K m}^3}{\text{J}} \cdot 10^{-6}$$
$$Q' \approx Q \cdot 9000 \frac{\text{K m}^3}{\text{J}} , \tag{30}$$

which makes it so that Q' is in units of K/Gy as wanted.

III. METHOD

A. One-dimensional solver

We have implemented an object-oriented solver in C++ for the one-dimensional diffusion equation (3) using the schemes outlined in Section II.C. For the full code used, see Appendix A.

The forward Euler based explicit scheme is implemented as follows:

```
// Iterate over timesteps
for (int j = 1; j <= m_M; j++){
  // Matrix multiplication with tridiagonal matrix
  for (int i = 1; i < m_N; i++){
    m_y(i) = m_coeff*m_u(i) + m_alpha*(m_u(i+1) + m_u(i-1));
  }

  // Update previous solution
  m_u = m_y;
  }
}
```

This uses a double for loop. The first outer loop iterates over the amount of timesteps `m_M`, and the inner loop iterates over all the components of the Armadillo [4] vector `m_y`. This is the solution in the next timestep, and thus to continue the iteration, the solution in the previous timestep `m_u` (also an Armadillo vector) is overwritten so that the iteration continues. This expression inside the inner for loop is a fairly simple implementation of equation (4), with `m_alpha` being α, and `m_coeff` being $1-2\alpha$. The variable `m_N` is the amount of elements in the solution vector minus one. This implementation takes $4(N-1)M \approx 4NM$ FLOPs (floating point operations), where N is the amount of spatial points (`m_N` in the code snippet) minus one, and M is the amount of timesteps (`m_M` in the code snippet).

In order to discuss the implementation of the two other schemes, we need to first discuss the implementation of the tridiagonal solver:

```
// Precalculate factors to reduce necessary FLOPs
double ac = m_a*m_c;

// Initialize temporary vector for modified RHS of equation
arma::vec f_tilde = m_y;
f_tilde(1) -= m_a*m_u(0);

// Initialize temporary vector for reduced diagonal elements
arma::vec d_tilde = arma::zeros(m_N);
d_tilde(1) = m_b;

// Update RHS elements
for (int i = 2; i < m_N; i++){
  d_tilde(i) = m_b - ac/d_tilde(i-1);
  f_tilde(i) -= f_tilde(i-1)*m_a/d_tilde(i-1);
}

// Find m_u (boundaries m_u(N) and m_u(0) are set manually outside
// of the tridiag function)
for (int i = m_N; i >= 2; i--){
  m_u(i-1) = (f_tilde(i-1) - m_c*m_u(i))/d_tilde(i-1);
}
```

This an implementation of the method discussed in Section II.C.4. The vector f in that section equates to the vector `m_y` in the code snippet, and similarly for $\tilde{b}$ and `d_tilde`, and $\tilde{f}$ and `f_tilde`. The elements of the matrix A are stored in the doubles `m_a`, `m_b` and `m_c`, and correspond to a, b, and c in equation (11). The first components of `d_tilde` and `f_tilde` have to be set manually. We then proceed with a for loop calculating the rest of the components of `d_tilde` and `f_tilde`. This is the forward part of the iteration as we generate the next values of the vectors based on the previous ones starting with the first components. Following this we perform a loop calculating the components of the solution vector `m_u`. This is the backwards part of the iteration as we use the next element in the solution vector to find the previous element, starting with the last element. The equations used in the forward and backwards part of the iteration are equations (12) and (13). This solver takes about $3+5(N-2)+3(N-1) \approx 8N$ FLOPs to complete.

The backwards Euler based implicit scheme is implemented as follows:

```
// Iterate over timesteps
for (int j = 1; j <= m_M; j++){
  // Use tridiagonal solver to move one step
  tridiag();

  // Set boundary conditions
  m_u(0) = m_lb;
  m_u(m_N) = m_ub;

  // Update previous solution
  m_y = m_u;
  }
}
```

As all the calculations happen within the tridiagonal solver function (`tridiag()`) there is only a loop over the amount of timesteps. As this is a set of object-oriented code, all the variables used in the tridiagonal solver are stored as class variables, meaning that no arguments are passed. The solver uses the solution in the previous timestep `m_y` to find the solution in the current timestep `m_u`. After this the solution in the previous timestep is overwritten and the iteration continues. As for the variables used in the tridiagonal solver, and relating them to the constants in equation (5), `m_a` and `m_c` are set to $-\alpha$ and `m_b` is set to $1+2\alpha$. The boundary conditions are also set manually each iteration to make sure they are not changed somehow, using the the variables `m_lb` (lower boundary) and `m_ub` (upper boundary). They should be the first and last elements of the solution vector respectively. This solver calls the tridiagonal solver in every timestep, and thus takes about $8NM$ FLOPs to complete.

The Crank-Nicolson (implicit) scheme is implemented as follows:

```
// Iterate over timesteps
for (int j = 1; j <= m_M; j++){
  // Set correct y
  for (int i = 1; i < m_N; i++){
    m_y(i) = m_coeff*m_u(i) + m_alpha*(m_u(i-1) + m_u(i+1));
  }

  // Set boundary conditions
  m_y(0) = m_lb;
  m_y(m_N) = m_ub;

  // Use tridiagonal solver to move one step
  tridiag();
  // Set boundary conditions
  m_u(0) = m_lb;
  m_u(m_N) = m_ub;
}
```

This is an amalgamation of the methods used to solve with the forward Euler based and the backward Euler based schemes. Finding the solution in the next timestep using the Crank-Nicolson scheme (see Section II.C.2) is a two-part process. First we multiply the solution vector in the previous timestep U_{j-1} with a matrix $2I-\alpha B$ (see equation (10)), which is done in the first for loop. The result is stored in the vector `m_y`. Following this we need to perform a matrix inversion with the tridiagonal matrix $2I+\alpha B$ in order to proceed. As this matrix is tridiagonal, we use the tridiagonal solver discussed earlier, where we now solve for the solution in the next timestep `m_u` using the right-hand side vector `m_y`. In terms of the tridiagonal solver, `m_a` and `m_c` are set to $-\alpha$ and `m_b` is set to $2+2\alpha$, which is as dictated by the definition of the matrix as $2I+\alpha B$ where B is defined in equation (6) and I is the identity matrix. The boundary conditions are also set manually each iteration similarly as in the previous code snippet.

The inner for loop takes $4(N-1) \approx 4N$ FLOPs to complete, and the tridiagonal solver is called in every timestep, which adds another $8N$ FLOPs. In total this solver takes about $12NM$ FLOPs to complete. For ease of access, the amount of FLOPs needed to complete a simulation with every solver is listed in Table II.

Table II. This table lists the amount of FLOPs (floating point operations) needed for the three one-dimensional schemes we have adopted to complete. N is the amount of spatial points minus one, and M is the amount of timesteps. The schemes can be read about in Section II.C and their implementations can be read about in III.A.

Scheme:	FLOPs
Forward Euler	$4NM$
Backward Euler	$8NM$
Crank-Nicolson	$12NM$

B. Two-dimensional solver

We have implemented an object-oriented solver in C++ for the numerical solution of the two-dimensional dimensionless diffusion equation (14). An object of the solver class is generated with a constructor, and the simulation is run with the `solve()` instance method. The code in its entirety can be found in Appendix A.

This implementation is based on the methods outlined in Section II.E. The scheme we have set up is an implicit one, and we have decided to use the Jacobi iterative method in order to solve it. This method is implemented as follows:

```
while (!converged && k < m_maxiter){
  // Define elements used for extraction
  double u10;
  double u20;
  double u01;
  double u02;
  double q;

  for (i = 1; i < m_N-1; i++){
    for (j = 1; j < m_N-1; j++){
      // Extract elements
      u10 = old(i+1,j);
      u20 = old(i-1,j);
      u01 = old(i,j+1);
      u02 = old(i,j-1);
      q = m_q(i,j);

      // Find next approximation to solution
      m_u(i,j) = m_diag_element*(m_alpha*(m_Ax*(u10 + u20)
               + m_Ay*(u02 + u01)) + q);
    }
  }
  // Check convergence
  converged = arma::approx_equal(m_u, old, "absdiff", m_abstol);
  old = m_u;
  k++;
}
```

In this case the Armadillo matrix `m_q` contains the solution in the previous timestep, and the elements of the matrix `m_u` is updated iteratively until they have converged. The outer while loop runs until either the maximum number of iterations has been reached, or the solution has converged, specified by the bool called `convergence`. The double for loop iterates over and calculates all the elements of the solution matrix `m_u` using equation (20). Related to the parameters in this equation, the variable `m_diag_element` is set to $1/(1+4\alpha)$ (the inverse of the diagonal elements) in this case and `m_alpha` is set to α. The variables `m_Ax` and `m_Ay` are in this case set to one, but as we will cover in the Section III.B.1 they can be changed so that the solution fits a rectangular grid.

In order to check the convergence of the solution we used a built-in Armadillo function `approx_equal()` that performs an element-wise comparison of two matrices and checks whether the elements are closer than the tolerance specified. If this is the case for all of the elements then the function returns `true`, and otherwise it returns `false`. This is assigned to the bool `converged` which is present in the while loop statement so that the iteration is interrupted if the solution has converged.

We chose to implement parallelization of the for-loops that perform the Jacobi iterations, using OpenMP directives. The solver automatically parallelizes the loop if the dimension of the matrix is $N > 100$. Furthermore, the number of threads used is chosen to be the minimum of either 3/4 of the available threads on the machine, or $3(N+1)/100$ as this seemed to give the best load balancing without overwhelming our machine. The elements in the matrix `old` are read atomically to avoid race conditions where the indices $i-1$, $i+1$, $j-1$ and $j+1$ overlap for different threads, while all constants are kept private to each thread.

Thus we can implement the full solution as such:

```
// Time iteration
for (m_t = 1; m_t <= m_M; m_t++){
  // Set source term
  set_source_term();

  // Move one step in time
  jacobi();
}
```

The for loop performs `m_M` iterations, corresponding to the set amount of timesteps that we should simulate for. Each iteration the `set_source_term()` and `jacobi()` functions are called. The first of these functions sets the matrix `m_q` to the solution in the previous timestep `m_u`, which is then used by the Jacobi iterative solver (the latter function) to find the solution in the next timestep. As all variables are stored as member variables, no arguments are passed in these functions, and they both instead access the variables directly.

Lastly, we need to cover the amount of FLOPs needed. This will depend on the amount of iterations needed by the Jacobi solver for the solution to converge, so we will not be able to give definitive numbers. For each iteration using the Jacobi iteration, a total of $8(N-2)^2 \approx 8N^2$ FLOPs is needed for the first double for loop to find the elements of the solution in the next iteration. After this we need to check the convergence with a second double for loop, which requires $2N^2$ FLOPs to complete. In total, we thus write that the Jacobi solver needs $\mathcal{O}(10N^2)$ FLOPs to complete, and that when doing this for M timesteps that the total amount of FLOPs needed is $\mathcal{O}(10MN^2)$.

1. *Solver for the heat equation with a source term*

We wished to solve the heat equation with an added source term Q as outlined in Section II.F.1. This can be done in two dimensions as we have outlined in Section II.F.2, in a fashion that is similar to that which was implemented in the last section. By changing the values of some variables we can use the exact same solver, so that is what we do. In terms of the variables specified in the last section referred to, the variables `m_Ax` and `m_Ay` from the code snippet of the Jacobi iterative method are set to A_x and A_y respectively. The variable `m_diag_element` is also set to $1/(1+2\alpha(A_x+A_y))$ (the inverse of the diagonal elements). The final modification is to `m_q` which now needs to contain the $\Delta t Q'$ as well. This is implemented as follows:

```
for (int i = 0; i < m_N; i++){
  for (int j = 0; j < m_N; j++){
    m_q(i,j) = m_dt*m_source_term(i*m_h,j*m_h,m_t*m_dt) + m_u(i,j);
  }
}
```

We iterate over the elements of `m_q` setting them equal to the elements of the solution in the previous timestep `m_u` plus the timestep `m_dt` multiplied with a function that corresponds to Q' (`m_source_term()`) which is specified upon instantiating an object of the solver class.

As for the FLOPs needed, it is clear that we need an additional $5N^2$ FLOPs per timestep in order to calculate the new source term and include it in `m_q`, plus any additional FLOPs k necessary in order to compute the result of the `m_source_term()` function. This means that in this case the total amount of FLOPs needed is $\mathcal{O}(10MN^2) + (5+k)MN^2$. The second term will generally be much less than the first one (assuming that `m_source_term()` does not require a lot of FLOPs), and so we assume that this specialization of the solver will not use an amount of FLOPs that is significantly larger than it would otherwise.

C. Source of errors and error calculation

We need a way to quantify the errors produced by our solvers. For this purpose we adopt the following expression:

$$\epsilon(t) = \frac{\sqrt{\sum_{i=0}^{N}(u_n(x_i,t)-u_a(x_i,t))^2}}{\sqrt{\sum_{i=0}^{N}u_a(x_i,t)^2}}, \tag{31}$$

where u_n is the numerically calculated solution and u_a is the analytically calculated solution. As we have a finite discretization of the spatial points x_i a sum is used (and not a integral), where it is assumed that there are $N+1$ such points. The expression above is the root-mean-square of the difference between the analytic and numerical solution divided by the root-mean-square of the analytical solution, and as such represents a kind of relative error in the solution. Henceforth, we refer to this expression as the relative RMS error.

We can also extend this to the two-dimensional case:

$$\epsilon(t) = \frac{\sqrt{\sum_{i=0}^{N}\sum_{j=0}^{N}(u_n(x_i,y_j,t)-u_a(x_i,t))^2}}{\sqrt{\sum_{i=0}^{N}\sum_{j=0}^{N}u_a(x_i,t)^2}}, \tag{32}$$

where we have changed the single sums to double sums such that we sum over all the spatial points (x_i, y_j) of the solutions. It is assumed that there are $N+1$ points y_j as well in this case.

Generally we expect that the errors in the solutions are the products of truncation error from the schemes used, and of machine precision. In the two-dimensional case there is an added error from the fact that we are using an iterative method to move the system a step in time. As this is only an approximation to the solution of the scheme itself, this can cause an additional increase of the error. Note that we do check that this has converged properly, and the program outputs an error if this is not the case, and so we assume that this will not be a main contributor to the errors in the results.

There are several floating point operations performed with each solver, and thus it is not unexpected that this might cause a sizeable contribution. The multiplications and divisions performed do not cause a sizeable contribution, but there are also several additions and subtractions performed that might cause larger errors. Generally we expect that this effect will make itself known if we set the timestep or steplength to be too short. For the one-dimensional schemes we know that the implicit schemes require more FLOPs than the explicit scheme (see Table II), which might cause the error to be larger for these than with the explicit scheme.

D. Notes on one-dimensional simulations

When testing the one-dimensional solvers we have implemented we use the relative RMS error as defined in equation (31) as a basis for comparison along with visual comparisons. The initial state we set is given by a sinusoidal plus a linear term:

$$g(x) = \sin(2\pi x) + x\,,$$

where we have set the boundary conditions to be $u(0,t) = 0$ and $u(1,t) = 1$. With this set of boundary conditions the steady-state solution is x, which means that the analytical solution is:

$$u(x,t) = \sin(2\pi x)e^{-4\pi^2 t} + x\,, \tag{33}$$

according to the results we found in Section II.B.

E. Notes on 2D simulation

When testing the two-dimensional solver we have implemented and used the relative RMS error as defined in equation (32) as a basis for comparison along with visual comparisons. The initial state is given by a product of sinusoidals plus a linear term:

$$g(x,y) = \sin(2\pi x)\sin(2\pi y) + y\,,$$

where we set $u(0,y,t) = u(1,y,t) = y$, $u(x,0,t) = 0$ and $u(x,1,t) = 1$ as the boundary conditions. In this case, according to the results in Sections II.D and Section II.B, the analytical solution is:

$$u(x,y,t) = \sin(2\pi x)\sin(2\pi y)e^{-8\pi^2 t} + y\,, \tag{34}$$

which we will compare with the numerical result using equation (32). The steady-state solution, y, is found by reducing the problem to a one-dimensional one.

F. Notes on lithosphere temperature simulations

We did not find the steady-state solution when there is an additional heat source (before further enrichment) (22) in Section II.F.3. It is, however, something which we wish to estimate and we can do so numerically. First we simulate the system with boundary conditions and initial state dictated by the steady-state solution when there are no additional sources (27). We can then fit a polynomial $p(y)$ to the middle of the system (in the numerical solution after it has stabilized) in the x-direction, and use this as the boundary conditions $T(0\text{ km}, y) = T(300\text{ km}, y) = p(y)$. This is something that can be done iteratively until the polynomial estimated is similar to the one that is currently being used. At that point we can assume that we have found a better set of boundary conditions. Thus, when simulating the system after the further enrichment with radioactive materials we can first simulate the system before further enrichment with this set of boundary conditions until it has reached a steady state, and then use that solution as the initial state when we simulate the system with the heat source as it is after further enrichment (24). This is how we choose to perform our simulations.

We estimate the polynomial $p(y)$ as a second order polynomial. As the added heat source before further enrichment is constant for a given region, Poisson's equation (used to find the steady-state solution) is reduced to a double derivative of T with respect to y being equal to a constant. This results in a second order polynomial of y in each region:

$$T(y) = ay^2 + by + c\,,$$

where the constants are unknown, and must be such that the solution in each region fits the solutions in the other regions as well. These constants will differ for each region, but as it behaves as a second order polyonomial everywhere, we approximate the entire solution with a second order polynomial. This is not entirely accurate, but it should improve the results compared to if we used the boundary conditions as dictated by the steady-state solution with no added heat sources (27). The polynomial is fitted using NumPy's [5] `polyfit()` function.

G. Unit testing

To test our implementation of the 1D solver methods and the 2D solver, we wrote unit tests using the CATCH2 C++ unit testing framework. The tests can all be found in `test_functions.cpp` (A.7). There are five tests, the first two checks that an output data file is created when the constructor of either solver is called. The remaining three tests checks that the solver methods behave as expected for a single time step. The forward Euler test function compares the forward Euler 1D solver with the expected results for 3 points, with $h = 0.5$, $\Delta t = 0.98 \cdot h^2/2$, and boundary conditions $u(1,t) = 1$ and $u(0,t) = 0$, as well as initial condition $u(x,0) = 0$ for $x \neq 1$. The tests for the remaining methods, simply check if the simulated system remains zero everywhere when the initial conditions and boundary conditions are zero.

IV. RESULTS

A. Benchmark

We performed a benchmark of the three methods of the 1D solver, as well as the 2D solver. For the benchmarks we used a spatial resolution of $h = 1/N$ where $N + 1 = 101$ is the amount of steps taken in space, as well as $M = 1000$ time steps and a timestep $\Delta t = 2.5 \times 10^{-5}$. We used the initial state and boundary conditions outlined in III.D for 1D and III.E for 2D. In the 2D case, we use $N + 1 = 101$ spatial steps along each axis. The results are averaged over k simulations, and are listed in table III.

Table III. This table lists the mean times spent in seconds on simulating systems with initial and boundary conditions as outlined in Sections III.D for 1D and III.E for 2D. The results are averaged over $k = 100$ simulations per method, with $N + 1 = 101$ spatial steps along each axis, a steplength $h = 1/N$, $M = 1000$ time steps and $\Delta t = 2.5 \times 10^{-5}$ (dimensionless).

Method	Mean time [s]	Standard deviation [s]
ForwardEuler	3.446×10^{-4}	2.002×10^{-5}
BackwardEuler	2.113×10^{-3}	4.144×10^{-5}
CrankNicolson	2.212×10^{-3}	2.122×10^{-4}
2D Solver	7.232×10^{-1}	2.928×10^{-2}

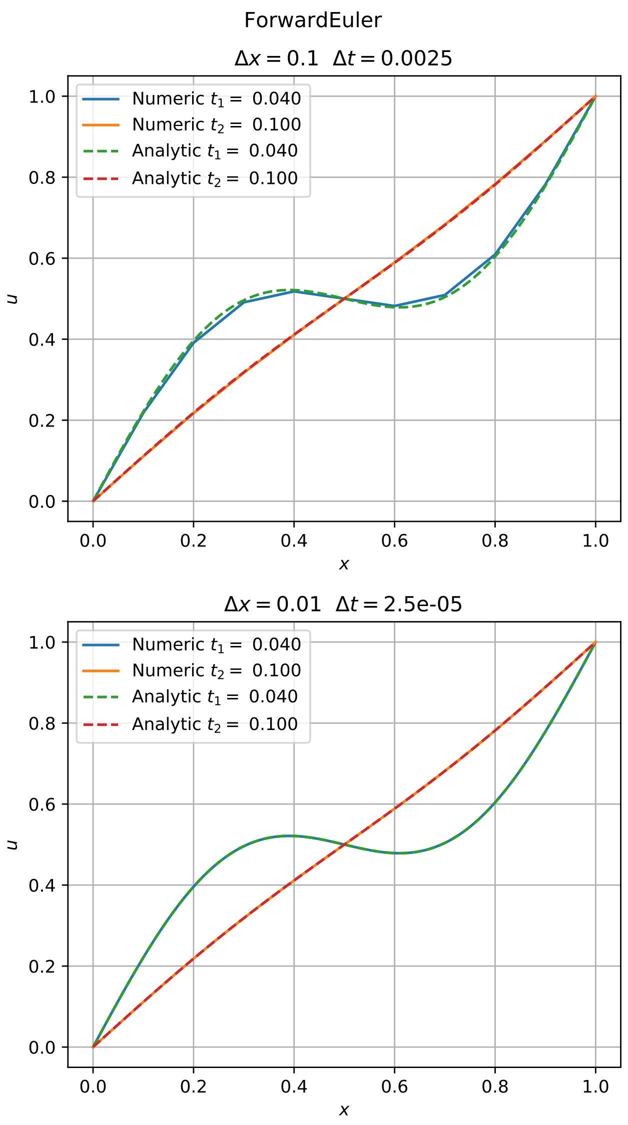

Figure 1. This figure contains plots of solutions to the one-dimensional diffusion equation, both analytical and numerical with the forward Euler based explicit scheme, for two sets of timestep Δt and steplength Δx. For each set of timestep and steplength a separate plot is provided, and the solutions at two points in time are plotted, as indicated by the legend text. All quantities are dimensionless.

B. One-dimensional simulations

We solved the one-dimensional dimensionless diffusion equation (3) numerically with the three schemes outlined in Section II.C and compared the results with an analytical solution. The initial state, boundary conditions and analytical solution (equation (33)) are as specified in Section III.D. Plots of the solutions at two different times, for two different sets of timestep Δt and steplength Δx can be seen in Figures 1, 2 and 3. Each of these figures correspond to one of the schemes used. The relative RMS error (see equation (31)) was plotted as a function of time against the amount of timesteps for two different choices of timestep and steplength in Figure 4, with the results from the three schemes being in the same figure this time.

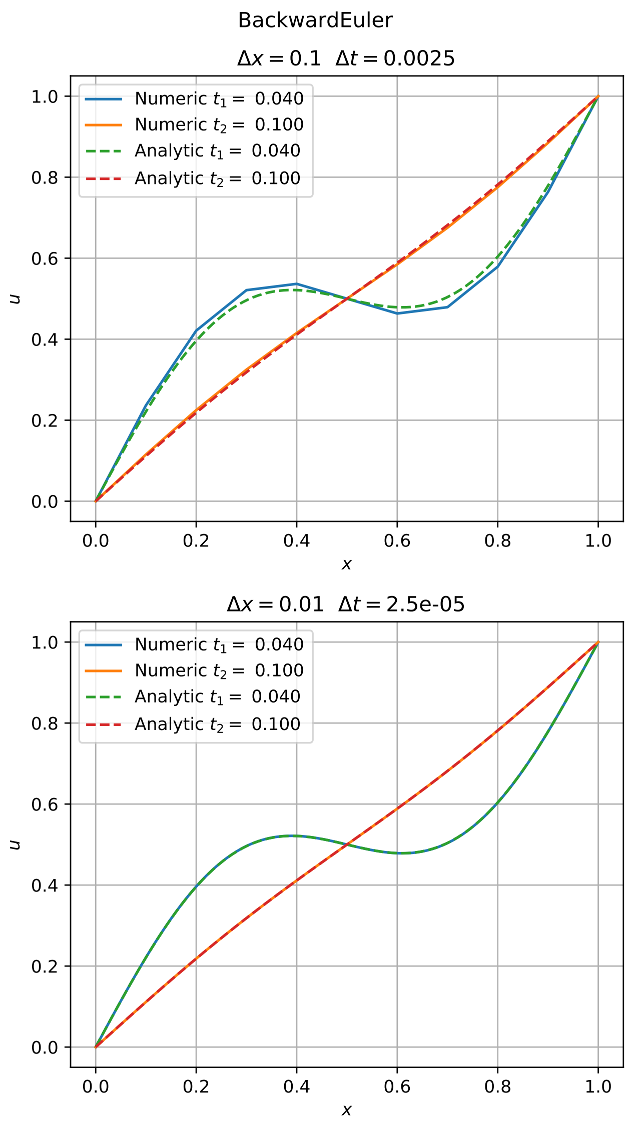

Figure 2. This figure contains plots of solutions to the one-dimensional diffusion equation, both analytical and numerical with the backward Euler based implicit scheme, for two sets of timestep Δt and steplength Δx. For each set of timestep and steplength a separate plot is provided, and the solutions at two points in time are plotted, as indicated by the legend text. All quantities are dimensionless.

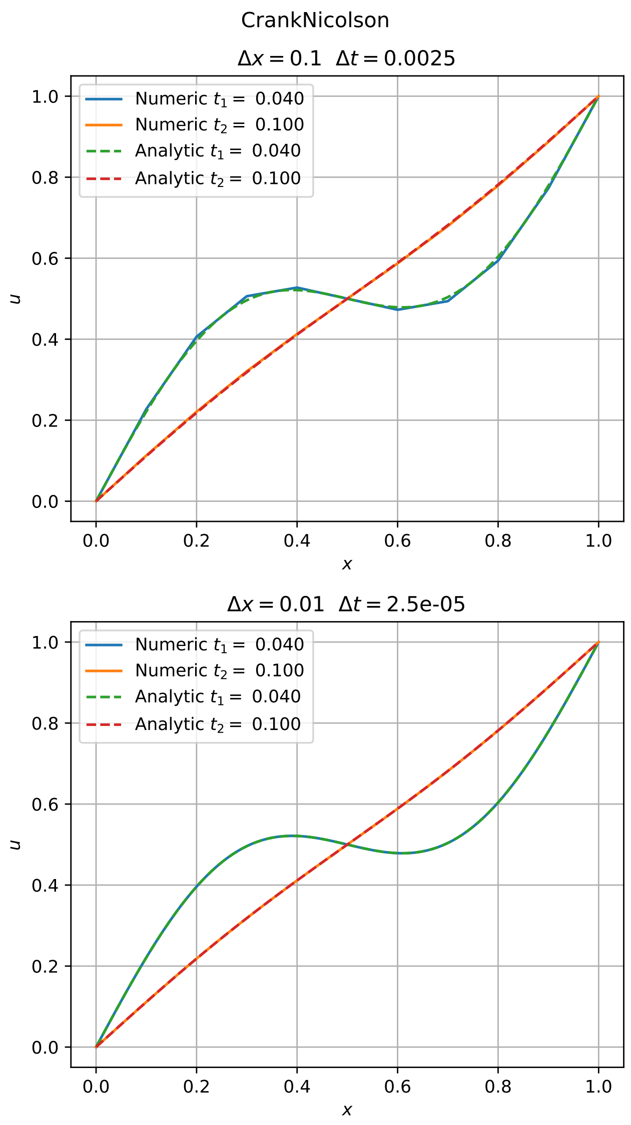

Figure 3. This figure contains plots of solutions to the one-dimensional diffusion equation, both analytical and numerical with the Crank-Nicolson scheme, for two sets of timestep Δt and steplength Δx. For each set of timestep and steplength a separate plot is provided, and the solutions at two points in time are plotted, as indicated by the legend text. All quantities are dimensionless.

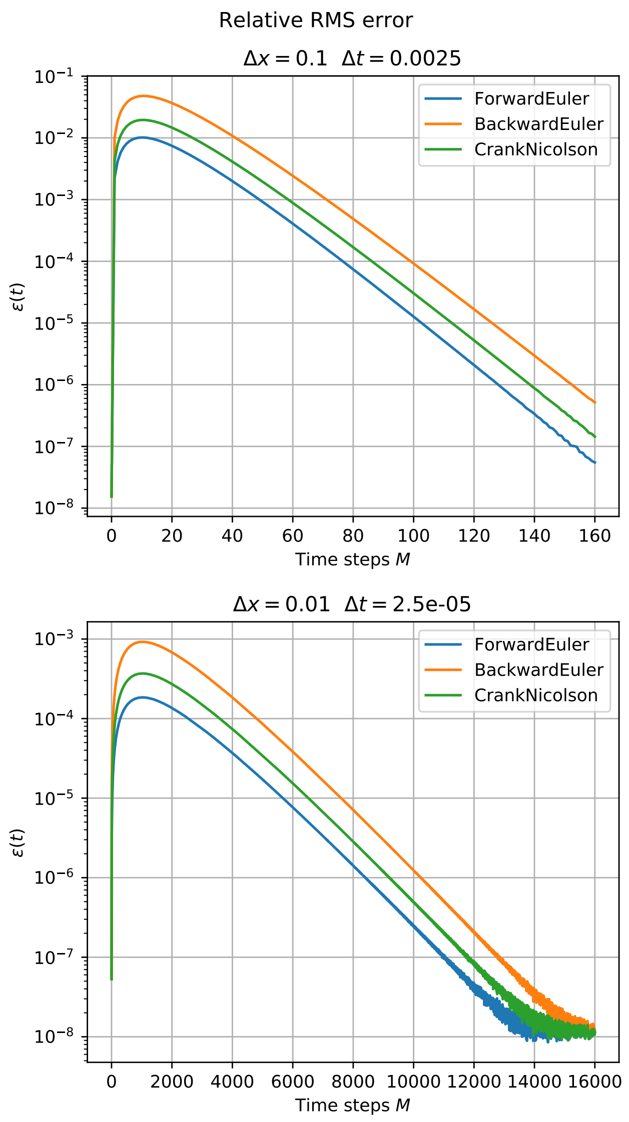

Figure 4. This figure contains plots of the relative RMS error as a function of time against the amount of timesteps for two sets of timestep Δt and steplength Δx as indicated by the plot titles. The relative RMS error is calculated as in equation (31). Each plot contains curves for each of the three numerical schemes outlined in Section II.C, as indicated by the legend text.

C. Two-dimensional simulation

We solved the two-dimensional dimensionless diffusion equation (14) numerically, and compared our results with the analytical solution. The intial state, boundary conditions, and analytical solution (equation (34)) are all given in Section III.E. We have plotted the numerical solution at $t = 0$, and $t = 5$ in Figure 5. The timestep in this simulation was set to $\Delta t = 1 \times 10^{-4}$, and the steplength (both in x- and y-directions) was set to $h = 0.005$ resulting in 201×201 grid-points. Note that all quantities are dimensionless. The relative RMS error (equation (32)) as a function of time was also plotted against the amount of timesteps. This plot is shown in Figure 6.

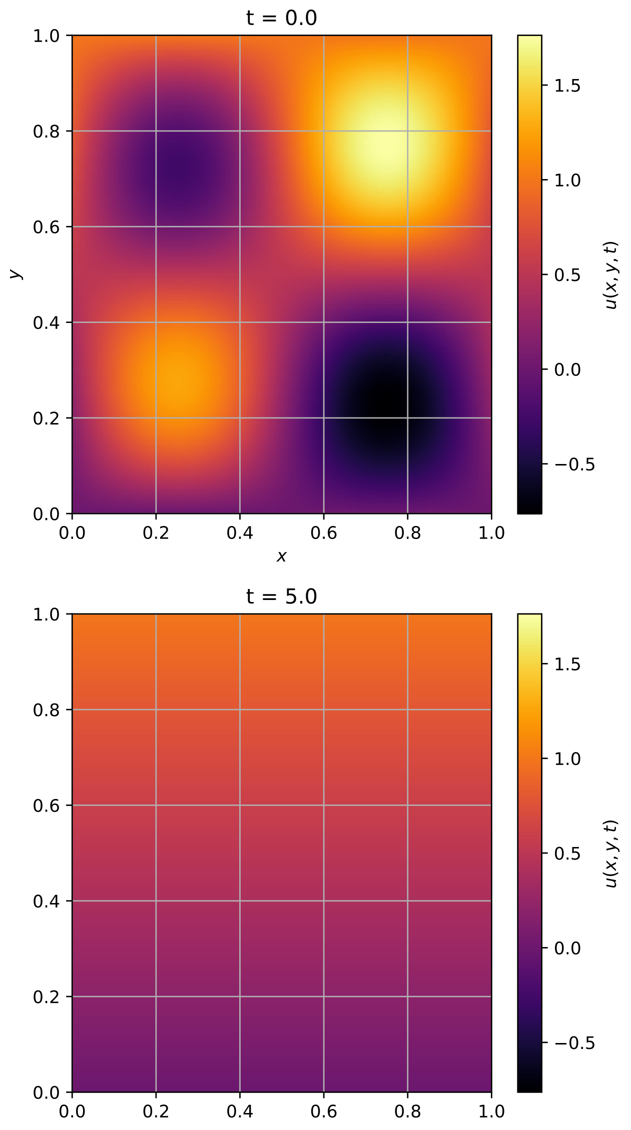

Figure 5. This figure contains plots of the initial state in the two-dimensional simulation and numerical solution after a certain amount of time has passed. The first plot shows the initial state, and the second plot shows the numerical solution after a time t specified in the plot title. The timestep in this simulation was $\Delta t = 1 \times 10^{-4}$ and the steplength was $h = 0.005$. All quantities in the plots are dimensionless.

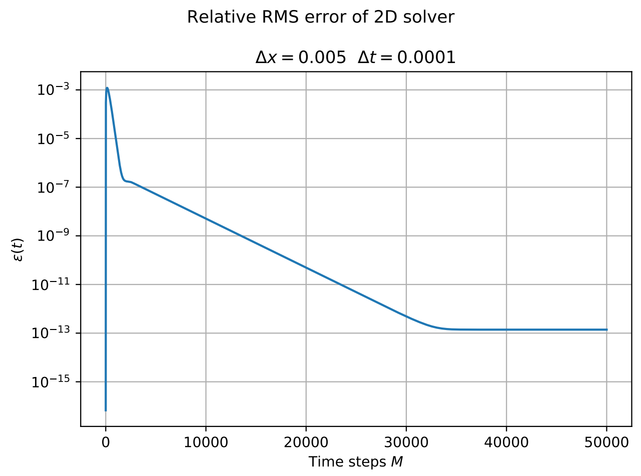

Figure 6. This figure contains a plot of the relative RMS error (see equation (32)) as a function of time plotted against the amount of timesteps in the two-dimensional simulation. The timestep Δt and steplength h used is indicated in the title of the plot. All quantities in the plot are dimensionless.

D. Temperature gradient of lithosphere

We simulated the temperature gradient of the lithosphere using our two-dimensional solver. This equates to solving the heat equation with a source term (21). In this section we assume that the reader has read Section II.F.1. All simulations were run with $N+1 = 301$ grid points in the spatial dimensions and a timestep of $\Delta t = 1\times10^{-4}$ Gy.

We estimated the second-order polynomial $p(y)$ as discussed in Section III.F, and used this as the boundary conditions for further simulations. We found:

$$p(y) = -\frac{439.46}{120^2}\,\frac{^\circ\mathrm{C}}{\mathrm{km}^2}\cdot y^2 + \frac{1634.21}{120}\,\frac{^\circ\mathrm{C}}{\mathrm{km}}\cdot y + 81.40\,^\circ\mathrm{C}\,, \tag{35}$$

and in order to showcase the fact that the numerical solution is not stable in x with the boundary conditions as dictated by the steady-state solution with no heat sources (27), we have included a plot of the first "iteration" of these simulations in Figure 7.

We then ran simulations with the boundary conditions being $T(0,y) = T(1,y) = p(y)$, and the other boundary conditions being the same as before. First a simulation was ran with these boundary conditions with the initial state being the steady state with no heat sources (27) until the numerical solution had stabilized. We then used the final result from this simulation as the initial state when the heat sources include the further enrichment from the melting sediments (24), and simulated this for 1 Gy. The initial and final states of these two simulations can be seen in Figures 8 and 9 respectively. We also plotted the difference in temperature between the initial and final states of the simulation where the mantle is further enriched. This plot is shown in Figure 10.

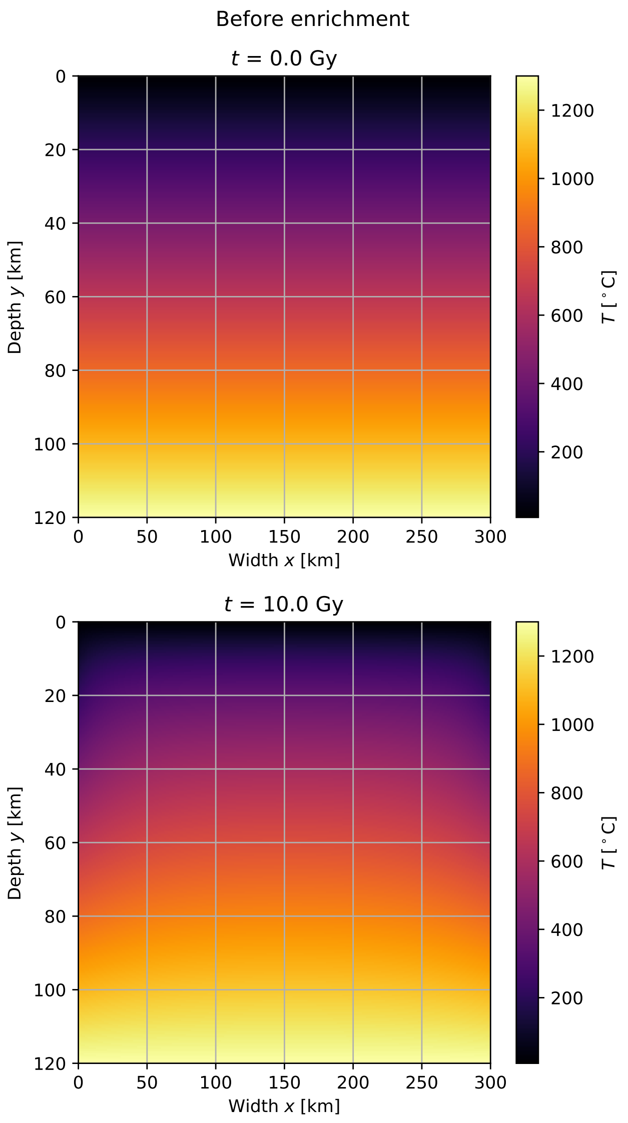

Figure 7. This figure shows the simulated temperature distribution of the lithosphere before further radioactive enrichment from the melting of subducting oceanic crust. This simulation uses the boundary conditions derived from the steady-state solution with no added heat. Further details on this system can be found in Sections II.F.1 and III.F. The temperature is marked by the color of the plot as indicated by the colorbar. Units are as shown in the plots. The first plot shows the initial state of the system, and the second plot shows the system after a time of 10 Gy has passed.

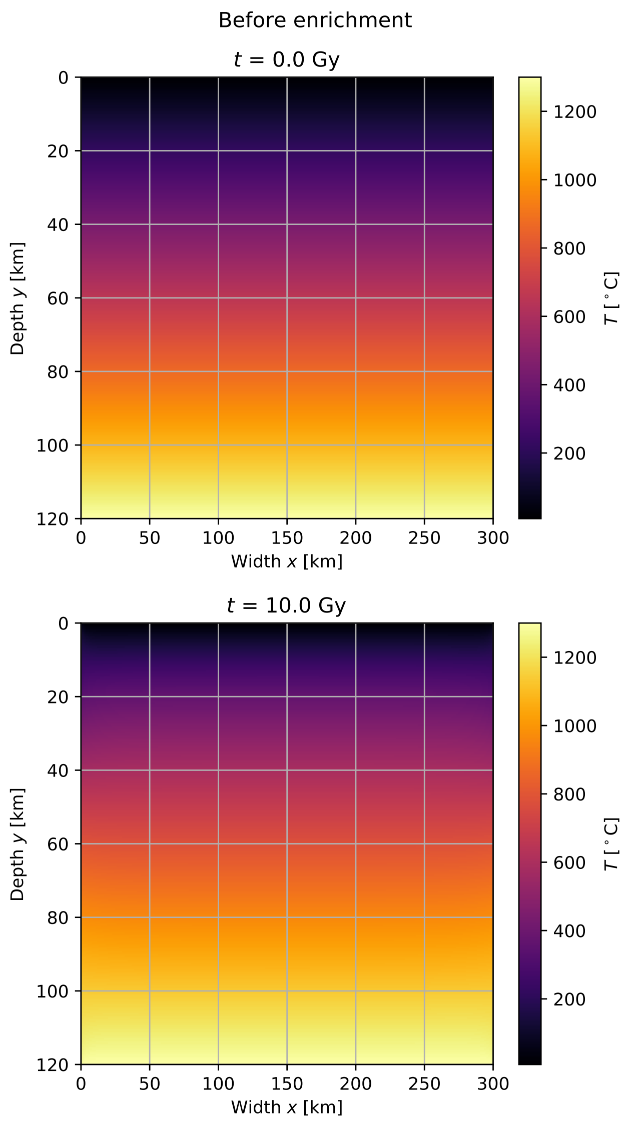

Figure 8. This figure shows the simulated temperature distribution of the lithosphere before further radioactive enrichment from the melting of subducting oceanic crust. This simulation uses the boundary conditions that were found from estimating the steady state solution with added heat from radioactive materials. Further details on this system can be found in Sections II.F and III.F. The temperature is marked by the color of the plot as indicated by the colorbar. Units are as shown in the plots. The first plot shows the initial state of the system, and the second plot shows the system after a time of 10 Gy has passed.

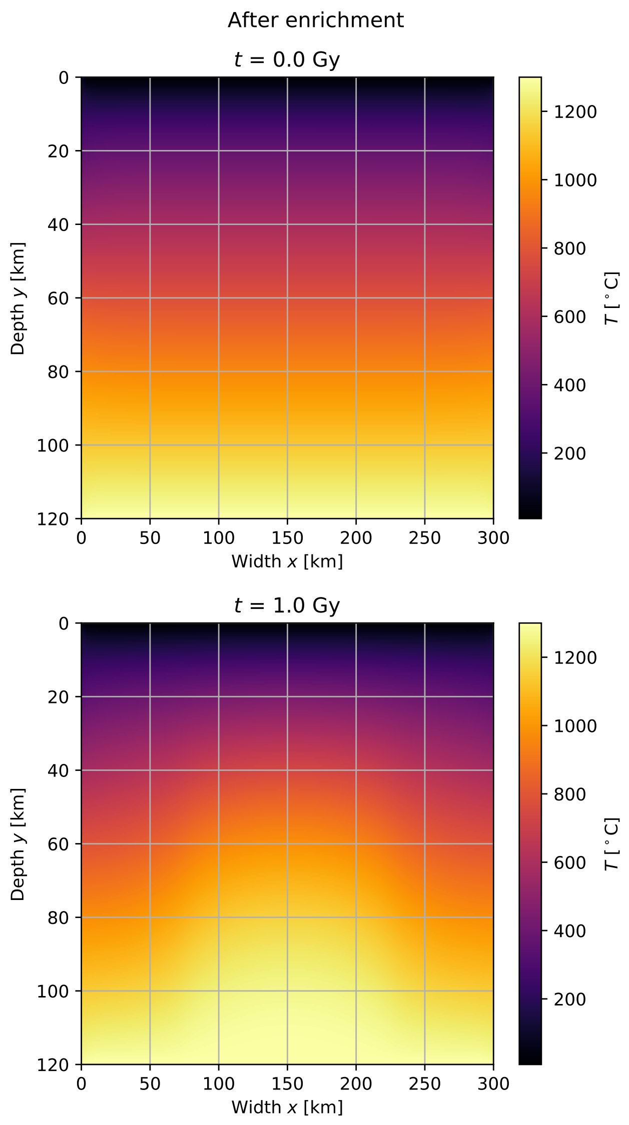

Figure 9. This figure shows the simulated temperature distribution of the lithosphere after further radioactive enrichment from the melting of subducting oceanic crust. Further details on this system can be found in Sections II.F and III.F. The temperature is marked by the color of the plot as indicated by the colorbar. Units are as shown in the plots. The first plot shows the initial state of the system, and the second plot shows the system after a time of 1 Gy has passed.

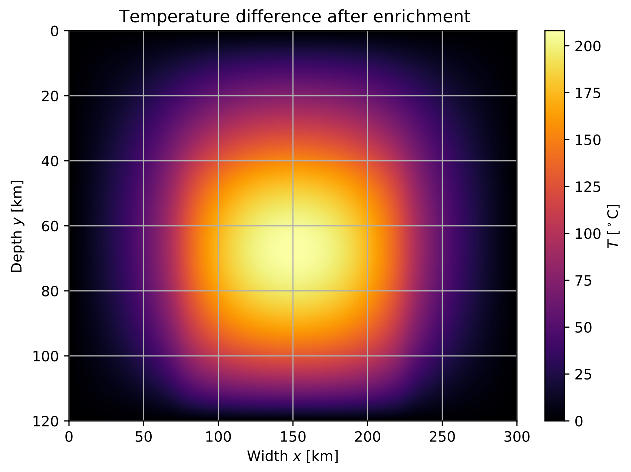

Figure 10. This figure shows the difference in temperature caused by the further enrichment from the melting of the subducting slab. See Section II.F for further details on this system. The difference in temperature is calculated as the difference in temperatures in the initial state and final states of the simulation (after 1 Gy in simulation time). The temperature difference is marked by the color of the plot as indicated by the colorbar. Units are as shown in the plot.

V. DISCUSSION

A. Benchmark

We see from the benchmark results listed in Table III that the forward Euler based scheme is the fastest of the 1D solver schemes. By comparison, the backward Euler and Crank-Nicolson schemes are approximately one order of magnitude slower, with the Crank-Nicolson method being the slowest of the 1D solver methods. This is what we expected to see given the FLOP counts listed in Table II.

From Table III we also see that the 2D solver is slower than the 1D solvers by at least two orders of magnitude, something which is expected given the amount of FLOPs needed goes as $\mathcal{O}(MN^2)$ (see Section III.B), where N is the number of steps minus one along each axis, and M is the number of steps in time.

B. One-dimensional simulations

We ran simulations with our one-dimensional solver with the same initial state and boundary conditions and two sets of steplength Δx and timestep Δt for all three of the schemes outlined in Section II.C. The results can be seen in Figure 1 for the forward Euler based scheme, in Figure 2 for the backward Euler based scheme and in Figure 3 for the Crank-Nicolson scheme. The steplength was chosen so as to be $\Delta x = 0.1$ and $\Delta x = 0.01$ (dimensionless), and the timestep was chosen to be $\Delta t = \Delta x^2/4$ (also dimensionless), which is within the stability limit of the forward Euler based scheme (see Table I). The initial state, boundary conditions and the accompanying analytical solution are as stated in Section III.D.

The plots in these figures contain solutions at two points in time: after a short amount of time and after a time such that the solution is starting to resemble the steady-state solution while not yet being perfectly linear, with the analytic solution in these timepoints plotted as well. In all these plots we see the behaviour that we expect for the solvers. All the numerical solutions seem closer to the analytic solution when the resolution (in both time and space) is better, and they also converge similarly with the analytic solution to the steady-state solution. This seems to indicate that the solvers work as they should. We also note that the solution from using the explicit scheme is ahead of the analytical solution, while both the implicit schemes have solutions that lag behind the analytical solution. This might be a coincidence, or a feature of explicit and implicit schemes. In order to determine which is the case, more conclusive evidence or mathematical proofs would be necessary, and thus we leave this to further research.

We also calculated the relative RMS error (31) in all the three numerical solutions compared to the analytical solution. These results are shown in Figure 4. Here we see for both choices of timestep and steplength that the error is largest with the backward Euler based scheme, less with the Crank-Nicolson scheme and the least with the forward Euler based scheme. The initial state used was chosen such that the analytical solution is simple. This can clearly be seen in the plots as the error is very small initially as we do not need to approximate the analytical solution, and then starts growing as we start making approximations in order to move the numerical solution ahead in time. For all three schemes we can see that they reach a maximum after roughly the same amount of time, before they dip back and start to tend towards smaller values again. We expect the truncation errors from the approximations and the FLOPs to add up over time when the change is large. However, as both the numerical and analytical solutions approach the steady-state solution, we also expect that the differences between these will decrease, as they should converge towards the same final state. Thus this behaviour is wholly as expected.

The error also seems to decay faster when the resolution is better, which makes sense. The trunction errors from the approximations will add up and reach a peak as we discussed earlier, and if the resolution is worse, the truncation error will also be worse as they scale with each other (see Table I for the truncation errors from all schemes). In the plot with better resolution we can also see that the relative RMS error seems to plateau at a certain point, which probably indicates that the error has moved beyond the precision of the numbers recorded. In these simulations the relative error was calculated in a Python script after the simulation was performed. The numerical solutions generated in C++ are written to file with 8 digits of resolution, and the fact that the error plateaus at around 10^{-8} is thus probably only an indication of the precision of the numbers being reached. The noise in the graph can also be attributed to the precision of the numbers, and so it seems very likely that this is what is encountered. This will happen in general, as numbers can never be exactly represented on a computer, but if we chose to write the numbers with a precision pertaining to that of the data-type used (doubles in this case, with between 15 and 18 significant digits) we would probably see the error plateauing at a much smaller value.

The most interesting results that can be obtained from these plots is the fact that the error seems to be the smallest with the forward Euler based scheme, instead of the Crank-Nicolson scheme which was found to have a better truncation error (see Table I). While this may seem weird at first glance, it is not really something that is wholly unexpected. We also counted the amount of FLOPs necessary to solve the system with each numerical scheme (see Table II), and there we found that the Crank-Nicolson scheme requires about three times the amount of FLOPs as the forward Euler based scheme. Thus this increase in error can most likely be attributed to the increased amount of FLOPs when compared to the forward Euler based scheme. We cannot, however, completely rule out the possibility that the error might be a product of a fault in our implementation. The error in the backward Euler based scheme is also significantly larger than the one in the forward Euler based scheme while their truncation errors are the same. Similarly to the Crank-Nicolson scheme, the backward Euler based scheme requires more FLOPs than the forward Euler based scheme, and so this might be the leading cause for difference in the errors. However, as both the Crank-Nicolson scheme and the backward Euler based scheme have significantly larger errors in their numerical solutions, this might also indicate an error in our implementation of the tridiagonal solver. We cannot see any error in the implementation, and finding out whether this is the case would probably require comparisons between these methods (and possibly other methods as well) using other methods of solving, which is something that we leave for future research.

All in all, in terms of the error produced, the explicit scheme fares better than the implicit schemes. The implicit schemes naturally require more FLOPs to solve, and we generally attribute the increase in error this as we do not see any error in our implementations. Assuming that these results are correct, this indicates that the explicit scheme is the better one to choose. This conclusion is a bit naive, however. As the simulations here are not particularly compuationally expensive, the explicit scheme is clearly better. In certain cases when the spatial resolution needs to be very high, however, the stability condition for the explicit scheme might impose that we need an inordinate amount of timesteps in order to simulate for any significant timescale. In these cases we can instead choose to use an implicit scheme, despite it producing more errors, as they do not impose any constraints on the choice of timestep. This can significantly reduce the computational cost, and if the errors produced are not significant enough to matter, the implicit schemes are clearly the better choice. Thus, we conclude that when choosing which scheme to use it is necessary to factor in the computational cost, and not just the truncation error.

C. Two-dimensional simulation

We ran a simulation using our two-dimensional solver with the boundary and initial conditions specified in Section III.E with $\Delta x = \Delta y = h = 1/200$, 201×201 gridpoints, and $M = 50000$ time steps with $\Delta t = 10^{-4}$. The results can be found in Section IV.C. From Figure 5 we see that the initial state is correct, and that the final state has approached the expected steady state of $u(x, y, t = \infty) = y$ suggesting that our solver is operating correctly.

From Figure 6 we can see that the relative RMS error behaves similarly to the one-dimensional case discussed in the previous section. It starts off small as one would expect, given the analytic solution and the initial condition are equal for $t = 0$. The error then quickly jumps to a maximum as the solver moves forward in time. The relative RMS error then begins to decrease as expected, since the difference between the numerical and analytic solution decreases as they both approach the steady state. However, once the error reaches 10^{-7} it flattens out slightly before continuing to decrease at a slower rate than previously. This bend in the error did not occur when using a tolerance of 10^{-12} for the Jacobi iterations, but it appeared when we increased the tolerance to 10^{-10} to save time. We have not verified the cause of this slight change in behavior. It could be an indication that there is a problem with our implementation, though we have not ruled out the possibility of this being a property of the example problem we have chosen. The error eventually stabilizes at 10^{-13} suggesting we have hit the smallest error the precision of our numbers will allow.

D. Temperature gradient of lithosphere

We simulated the temperature gradient of the lithosphere in several settings. First we simulated the system with the regular amount of radioactive materials present, with the initial state being that of the steady-state solution with no radioactive elements (27). The results can be seen in Figure 7. A series of simulations starting with this one was used to estimate the boundary conditions in the x-direction as outlined in Section III.F. The estimated boundary conditions are given in (35).

This result somewhat fits our expectations as they were outlined in Section II.F.3. As the coefficient in front of y^2 is negative and the coefficient in front of y is larger than in the steady-state with no radioactive materials, it is clear that this polynomial fits the expectation (and visually confirmable result) of the temperature being higher for all depths. Note that the original boundary conditions are not preserved by the polynomial, and as such there is now a small discontinuity in the boundary conditions used (in the corners of the system). As we are solving numerically this does not cause any problems, so we continue using this polynomial as the boundaries in the x-direction, even while the boundaries are now discontinuous in the corners. If we tried to "fix" the polynomial this would lead to the other values being wrong as well, so we leave it as is, and remember that it might be a source of problems later down the line.

We then used this estimate of the boundary conditions and simulated the steady-state of the system, as seen in Figure 8. This simulation lasted a total of 10 Gy in simulation time, so we could be certain the the state we had reached was stable. We then used the final state of this system as the initial state in a second simulation where we used the heat source term as it is when the mantle has been refertilized (24). The results for this simulation can be seen in Figure 9. We also included a plot of the difference in temperature between the initial state and the final state, as this will better let us determine the effects of the added radioactive materials directly. This plot can be seen in Figure 10.

We can clearly see a marked difference in the temperature gradient in the part of the mantle that is further enriched in radioactive materials ($x \in [75, 225]$ km). As expected it also seems to increase the temperatures around it. In terms of the physical problem at hand, this means that if such a refertilization of the mantle occured, we would expect to see that the lithosphere is warmer than it would be otherwise. The temperature difference is the largest (slightly more than 200 °C) at about ~ 70 km depth in the middle of the system in the x-direction, with fairly similar decay when moving in all directions from that point, which is as expected because of the fixed boundary conditions.

We see a clear difference in the temperature for depths ranging from $10-30$ km depth of about $25-100$ °C, such that the difference should be measurable. These results indicate that similar measurements in an experimental setup could very likely indicate that such an event has occured in the past, and is something that is of interest for further research. By first making experimental measurements and comparing them with numerical results, it can also be ascertained whether or not the assumptions made on the levels of enrichment that occurs are correct, or approximately correct. We also need to remember that the model we have used is heavily simplified, particularly by assuming constant heat conductance, density, and heat capacity throughout the lithosphere. Thus, measured results might differ heavily from those we have obtained. Such a comparison would still allow us to correct and improve the model, and thus this topic is of interest for further research both in terms of experimental and theoretical aspects. Reaching the depth needed for a clear difference in temperature as predicted by this model is difficult and expensive at this time, such that making such measurements might be improbable. However, while it may not be within reasonable means at this moment, it may very well be in the future.

VI. CONCLUSION

We derived or outlined, implemented, and tested numerical solutions of the general diffusion equation (1) in both one and two dimensions. In general we found that all the solvers behave as expected. All the numerical solutions seemed to converge as they should, and when compared with analytical results by way of the relative RMS error ((31) or (32)) the agreement was found to be satisfactory.

Of the three schemes implemented to solve the one-dimensional diffusion equation, the forward Euler based explicit scheme performed the best, both in terms of error and time consumption. That this scheme used the least amount of time is as expected from the amount of floating point operations required (see Table II). The error, however, was slightly unexpected, as the Crank-Nicolson scheme has a better truncation error (see Table I). The backward Euler based scheme performed worse than the forward Euler based scheme in both measures, and compared to the Crank-Nicolson scheme it spent less time, but the error was worse. This can be explained by the fact that there are more floating points operations being performed by the two implicit schemes, and we thus conclude that this is the most likely cause of the discrepancy that we see. We cannot fully exclude the possibility of an error in our implementation, but we are unable to locate any. While this shows that the forward Euler based method is better in this case, it does not mean that it will always be the better choice. The solution will only converge if the timestep $\Delta t \leq \Delta x^2/2$, where Δx is the steplength. In very stiff systems a small steplength is required in order to get good results, and in that case this stability condition might require an inordinately small timestep. The implicit schemes impose no such restrictions on the timestep, and can thus use longer timesteps as long as the error is still small enough. This means that the implicit schemes are favored in terms of computational cost when the system modelled is sufficiently stiff.

The two-dimensional solver showed no cause for concern in the general simulation, as it replicated the analytical results with satisfying accuracy. When using the two-dimensional solver to model the temperature gradient of the lithosphere, we found an increase in temperature for all depths as expected. The steady-state before the further enrichment from the melting of the subducting slab also seemed to fit well to a second order polynomial of the depth, as can be evidenced by the fact that we were able to find a stable polynomial expressing the temperature gradient. This was then used as the boundary conditions of the system in the x-direction, which better allowed us to model the system when it was further enriched in radioactive materials. In this case we also saw results as expected. The region of the upper mantle that is further enriched is clearly heated up, and this heat spreads to the other parts of the system as well, as evidenced by the temperature being increased all across the system. This indicates that measuring similar temperatures might indicate that such an enrichment of the mantle did in fact occur 1 Gy ago. By comparison with measurements, this model can be further improved, and we can perhaps obtain a better picture of how such processes occur, but this is something we leave for future research, as such measurements are outside of the scope and purposes of this report. They are also likely not feasible or too expensive at this point in time.

[1] M. Hjorth-Jensen, *Computational Physics Lecture Notes Fall 2015* (Department of Physics, University of Oslo, Oslo, 2015).
[2] A. P. Åsbø and E. Støland, *Project 1*, Tech. Rep. (2020).
[3] M. Hjorth-Jensen, *Project 5, deadline December 16, 2020*, Tech. Rep. (Department of Physics, University of Oslo, Norway, Oslo, 2020).
[4] C. Sanderson and R. Curtin, Journal of Open Source Software **1**, 26 (2016).
[5] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, Nature **585**, 357 (2020).

Appendix A: Source code

All code for this report was written in C++ and Python 3.6, and the complete set of files can be found at:
https://github.com/eivinsto/FYS3150_Project_5.git.

1. `diffusion_equation_solver_1D.hpp`

Header file containing definitions for `DiffusionEquationSolver1D` class, which is used to solve the one-dimensional diffusion equation. The file can be found at:
https://github.com/eivinsto/FYS3150_Project_5/blob/main/src/diffusion_equation_solver_1D.hpp

2. `diffusion_equation_solver_1D.cpp`

Source file containing the methods and constructors belonging to the `DiffusionEquationSolver1D` class, which is used to solve the one-dimensional diffusion equation. The file can be found at:
https://github.com/eivinsto/FYS3150_Project_5/blob/main/src/diffusion_equation_solver_1D.cpp

3. `diffusion_equation_solver_2D.hpp`

Header file containing definitions for `DiffusionEquationSolver2D` class, which is used to solve the two-dimensional diffusion equation. The file can be found at:
https://github.com/eivinsto/FYS3150_Project_5/blob/main/src/diffusion_equation_solver_2D.hpp

4. `diffusion_equation_solver_2D.cpp`

Source file containing the methods and constructors belonging to the `DiffusionEquationSolver2D` class, which is used to solve the two-dimensional diffusion equation. The file can be found at:
https://github.com/eivinsto/FYS3150_Project_5/blob/main/src/diffusion_equation_solver_2D.cpp

5. `main.cpp`

Main program used to perform the simulations in the report with input controlled by command line arguments, with implementation of necessary functions passed to the solver classes.
https://github.com/eivinsto/FYS3150_Project_5/blob/main/src/main.cpp

6. `project.py`

Python script used to automate simulations and plotting of results. Some small calculations are also performed in this script using the data that the solvers output.
https://github.com/eivinsto/FYS3150_Project_5/blob/main/project.py

7. Unit-tests

A few unit-tests were implemented using Catch2 in C++, and they can be found in the following files:
https://github.com/eivinsto/FYS3150_Project_5/blob/main/src/test_functions.cpp
and:
https://github.com/eivinsto/FYS3150_Project_5/blob/main/src/test_main.cpp

Appendix B: Selected results

Here is a folder of selected results from running our code.

https://github.com/eivinsto/FYS3150_Project_5/tree/master/data

Appendix C: System specifications

All results included in this report were achieved by running the implementation on the following system:

- CPU: AMD Ryzen 9 3900X
  - 12 cores, 24 threads.
  - 3.8 GHz base clock.
  - 4.2 GHz all core boost clock.
  - 4.6 GHz single core boost clock.

- RAM: 2 × 8 GB Corsair Vengeance LPX DDR4 3200 MHz