Production Systems and Information Engineering
Volume 10 (3), pp. 105–114.
https://doi.org/10.32968/psaie.2022.3.9

EXAMINING THE PERFORMANCE OF MATLAB'S MATRIX CAPABILITIES, TESTING ON EULER'S METHOD APPLIED ON THE DIFFUSION EQUATION

DÁNIEL KOICS
University of Miskolc, Hungary
Institute of Automation and Infocommunication
daniel.koics@uni-miskolc.hu

KÁROLY NEHÉZ
University of Miskolc, Hungary
Institute of Informatics
aitnehez@uni-miskolc.hu

ENDRE KOVÁCS
University of Miskolc, Hungary
Institute of Physics and Electrotechnics
fizendre@uni-miskolc.hu

Abstract. When one develops, tests and uses numerical methods to solve a differential equation, the performance of the method depends on the concrete way how the method is implemented and coded. In this tentative work, we solve the linear diffusion equation by the simplest explicit Euler method implemented with for loops as well as the built-in matrix operations of MATLAB. We obtain that the for loop performs better in one space dimension, but the matrix operations are faster in two space dimensions.

Keywords: CPU time; numerical methods; partial differential equations; MATLAB

1. Aims and Scope of the Publication

While most physical and engineering problems are related to ordinary or partial differential equations (ODEs or PDEs), there is no clear consensus about the numerical strategies to solve these equations. Explicit methods are quite simple to implement and one time step is usually performed in a very small time. However, they suffer from a very restrictive time step to satisfy the requirement of numerical stability. For time step sizes larger than the so called CFL limit, the magnitude of any computation error is amplified in each time step, leading to completely unusable results. Implicit schemes have better stability properties, but they are more complicated to use in practice, and the calculation of one timestep can be much longer than with explicit methods. If the number of space dimensions is

larger than one, the number of nodes or cells can be very large, thus the running time becomes a critical issue.

Several novel explicit methods are proposed recently in the scientific literature, both for ODEs [1], and PDEs [3], [5]. In order to develop and test new numerical methods, as well as to explore which one can be proposed to different problems under specific circumstances, one should have information on what the actual running time depends on. When Runge-Kutta type methods are compared in the case of ODEs, it is usual to consider the number of function calls or evaluations as an indicator for the running time [4]. However, for PDEs, the CPU time depends on other factors as well, such as the number of space dimensions and the amount and way of memory usage. For example, remarkably different running times can be obtained if the calculation of the node variables is performed one after another by 'for' loops over the space index or by the built-in matrix operations of *MATLAB*. The goal of our work is to make experiments on this issue.

To achieve our goal, we have implemented Euler's method in two different ways. To describe the characteristics of our implementations, we find useful to plot the CPU-time as a function of the timestep count and the numerical grid-point count, as well as to plot the computation error as a function of the CPU-time. What is more interesting, is to make comprehensive figure between different number of dimensions, as well as different implementations. At the same time, it is important to check whether the different implementations give the same computation error for any given timestep size and spatial step size.

Therefore, in Section 2, we discuss the studied equation, Euler's method and its theoretical background. In 2.6 we give how to measure the error of an implementation and highlight the error measurement way we use to verify our work. In Section 3, we go into some details of our actual implementation. In Section 4, we give the exact data that has been used for the test. In Section 5, we present our measurement results. In case of both implementations (in 5.1 and 5.2), we firstly plot the computation time and make comparison between 1-dimensional and 2-dimensional case. Then we show the error – computation time characteristics of the given implementation. In 5.3, we make a comparison between the two implementations. Finally, in Section 6, we give our conclusions.

2. The Studied Equation and the Applied Algorithms

2.1. Heat Transfer or Diffusion Equation

In physics, one of the most well-known partial differential equations is the *diffusion equation* or alternatively, *Fick's 2nd law*:

$$\frac{\partial n(\boldsymbol{r},t)}{\partial t} = -D \cdot \tilde{\Delta} n(\boldsymbol{r},t) \tag{1}$$

where n is the spatial *concentration* of matter (a quantity varying across both space $\boldsymbol{r}$ and time t), $D\left[\frac{m^2}{s}\right]$ (in practice, rather $\left[\frac{cm^2}{s}\right]$) is the *diffusion coefficient* and $\tilde{\Delta}=\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}+\frac{\partial^2}{\partial z^2}$ is the *Laplacian operator*.

According to this equation, the more different is the concentration at a given spatial point from the concentration at the neighboring points, the faster it changes, as time passes. The direction of concentration change is to reduce the difference in space. Behind this phenomenon, we can find the second law of thermodynamics. The equation in details, indicating dependency on both space and time:

$$\frac{\partial n(x,y,z,t)}{\partial t}=D\cdot\left(\frac{\partial^2 n(x,y,z,t)}{\partial x^2}+\frac{\partial^2 n(x,y,z,t)}{\partial y^2}+\frac{\partial^2 n(x,y,z,t)}{\partial z^2}\right) \tag{2}$$

where $\frac{\partial}{\partial t}$ is the derivation with respect to time, while $\frac{\partial^2}{\partial x^2}$, $\frac{\partial^2}{\partial y^2}$ and $\frac{\partial^2}{\partial z^2}$ denotes to the second derivation (i.e., taking the derivative of the derivative) with respect to the appropriate spatial coordinate.

Heat transfer is based on an analogous principle: We only have to substitute concentration n with temperature T[K] (also a function of space and time), and diffusion coefficient D with thermal diffusivity $\alpha\ \left[\frac{m^2}{s}\right]$:

$$\frac{\partial T(\boldsymbol{r},t)}{\partial t}=-\alpha\cdot\tilde{\Delta}\,T(\boldsymbol{r},t) \tag{3}$$

The thermal diffusivity can be expressed by substantial properties:

$$\alpha=\frac{k}{c\rho} \tag{4}$$

where $k\ \left[\frac{\mathrm{W}}{\mathrm{m^2K}}\right]$, $c\ \left[\frac{\mathrm{J}}{\mathrm{kgK}}\right]$, and $\rho\ \left[\frac{\mathrm{kg}}{\mathrm{m^3}}\right]$ are the *heat conductivity*, *specific heat*, and *mass density*, respectively.

In the rest of this paper we denote both $n(\boldsymbol{r},t)$ and $T(\boldsymbol{r},t)$ as $f(\boldsymbol{r},t)$.

2.2. Spatial and Time-Domain Discretization, FDM

The solution of the equations by a **f**inite **d**ifference **m**ethod consists of the following steps:

- The studied spatial domain is discretized into a $N_x\times N_y\times N_z$ rectangular grid, where N_x, N_y and N_z denotes to the number of nodes along axis x, y and z. The grid-points are:

$$\boldsymbol{r}_{i,j,k}=\begin{bmatrix}X_{init}+(i-1)\cdot\Delta x\\ Y_{init}+(j-1)\cdot\Delta y\\ Z_{init}+(k-1)\cdot\Delta z\end{bmatrix} \tag{5}$$

where:

- $i = 1 \dots N_x$, $j = 1 \dots N_y$ and $k = 1 \dots N_z$ are the possible spatial indexes,
- $X_{init} = x_1$, $Y_{init} = y_1$ and $Z_{init} = z_1$ are the initial values of spatial scales,
- $X_{fin} = x_{N_x}$, $Y_{fin} = y_{N_y}$ and $Z_{fin} = z_{N_z}$ final values of spatial scales and
- $\begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} (X_{fin} - X_{init})/(N_x - 1) \\ (Y_{fin} - Y_{init})/(N_y - 1) \\ (Z_{fin} - Z_{init})/(N_z - 1) \end{bmatrix}$ are the division steps of spatial scales

- The time domain is also discretized, and sampled in N_t timepoints:

$$t_n = T_{init} + n \cdot \Delta t \tag{6}$$

where:

- $n = 0..N_t$ are the possible time indexes,
- $T_{init} = t_0$ is the initial time value,
- $T_{fin} = t_{N_t}$ is the final time value and
- $\Delta t = \frac{T_{fin} - T_{init}}{N}$ is the size of timesteps.

- The continuous function $f(\boldsymbol{r}, t)$ can be approximated by a $f_{i,j,k}^{(n)}$ data matrix:

$$f(\boldsymbol{r}, t) \cong f(\boldsymbol{r}_{i,j,k}, t_n) =: f_{i,j,k}^{(n)} \tag{7}$$

- Since our differential operators are linear, each (spatial) differential operator $\widetilde{D}$ can be replaced by a multi-index matrix $[\widetilde{D}]_{i,j,k}^{p,q,r}$, for which:

$$\widetilde{D} f(\boldsymbol{r}) \cong [\widetilde{D} f]_{i,j,k} = \sum_{p=1,q=1,r=1}^{N_x,N_y,N_z} [\widetilde{D}]_{i,j,k}^{p,q,r} f_{p,q,r} \tag{8}$$

  where *p*, *q* and *r*, as well as *i*, *j* and *k* are indexes running over the 3 spatial dimensions.

- As a result, instead of differential equations, a system of algebraic equations are sufficient to be solved.

We obtain the matrix of the *Laplacian* operator by the central difference formula:

$$f(r_{i,j,k}) = [\tilde{\Delta} f]_{i,j,k} \cong$$
$$\cong \frac{f_{i+1,j,k} - 2 \cdot f_{i,j,k} + f_{i-1,j,k}}{\Delta x^2} + \frac{f_{i,j+1,k} - 2 \cdot f_{i,j,k} + f_{i,j-1,k}}{\Delta y^2} + \frac{f_{i,j,k+1} - 2 \cdot f_{i,j,k} + f_{i,j,k-1}}{\Delta z^2} \tag{9}$$

To introduce the $\tilde{\Delta} f(r_{i,j,k}) \cong \sum_{p,q,r} [\tilde{\Delta}]_{i,j,k}^{p,q,r} f_{p,q.r}$ formalism – as (8) suggests –, the matrix of *Laplacian* operator must be:

$$\left[\tilde{\Delta}\right]_{i,j,k}^{p,q,r} = \begin{Bmatrix} -\frac{2}{\Delta x^2} - \frac{2}{\Delta y^2} - \frac{2}{\Delta z^2} & if\ i = p, j = q\ and\ k = r \\ \frac{1}{\Delta x^2} & if\ i = p \pm 1, j = q\ and\ k = r \\ \frac{1}{\Delta y^2} & if\ i = p, j = q \pm 1\ and\ k = r \\ \frac{1}{\Delta z^2} & if\ i = p, j = q\ and\ k = r \pm 1 \\ 0 & else \end{Bmatrix} \tag{10}$$

In our paper, we consider only 1-dimensional and 2-dimensional cases.

2.3. Euler's Method

It is well-known that the simplest and most famous explicit method is Euler's method. This is a first order method, which means that the local error (i.e. error per timestep) is decreasing proportionally to the second power of the size of timesteps, while the global error only with first power.

Let the differential equation to be solved be:

$$\frac{\partial f(\boldsymbol{r},t)}{\partial t} = \widetilde{D} f(\boldsymbol{r}, t) \tag{11}$$

where $\widetilde{D}$ is a differential operator containing only spatial derivation. According to Euler:

$$f_{i,j,k}^{(n)} \cong f_{i,j,k}^{(n-1)} + \left[\widetilde{D} f\right]_{i,j,k}^{(n-1)} \cdot \Delta t \tag{12}$$

where $\widetilde{D} f_{i,j,k}^{(n-1)}$ is the approximated value of the derivative around the spatial point $\boldsymbol{r}_{i,j,k}$ at timepoint t_{n-1}. In our case of 2-dimensional diffusion with $\boldsymbol{r}_{i,j}$ spatial points, one can substitute:

$$\widetilde{D} = D \cdot \tilde{\Delta} = D \cdot \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) \tag{13}$$

where D is either the diffusion coefficient or the thermal diffusivity.

Based on the above derivation, *Euler*'s time-stepper formula (12) is the following:

$$f_{i,j}^{(n)} \cong f_{i,j}^{(n-1)} + \\ + D \cdot \Delta t \cdot \left(\frac{f_{i+1,j}^{(n-1)} - 2 \cdot f_{i,j}^{(n-1)} + f_{i-1,j}^{(n-1)}}{\Delta x^2} + \frac{f_{i,j+1}^{(n-1)} - 2 \cdot f_{i,j}^{(n-1)} + f_{i,j-1}^{(n-1)}}{\Delta y^2}\right) \tag{14}$$

Regarding performance and optimization aspects, we would like to make two remarks:

- It is sufficient to only store two data grids, for two sequential timesteps.
- Datapoints of the new timestep can be calculated independently from each other, which gives us an opportunity to parallelization.

2.4. Handling Boundary Conditions

The above formula has one incompletion: It can only be used in the inner domain – where $1 < i < N_x$ and $1 < j < N_y$. Only these datapoints have the neighbors referenced by the formula. The boundaries – where $i = 1$, $i = N_x$, $j = 1$ or $j = N_y$, – require further considerations. In our work, we deal with the fixed boundary condition: We only perform calculation in the inner domain, and the values on the boundaries remain unchanged. In case of heat transfer it means, that we are heating or cooling the walls of the domain so that its temperature does not change. (This kind of boundary condition belongs to the *Dirichlet*'s type of boundary conditions, which is the name when a solution has to obtain predefined values at the boundaries. In this paper, we simply refer to our fixed boundary condition as *Dirichlet*'s boundary condition.)

This boundary condition is implemented such that the values at the boundary are not refreshed during the calculation. It implies that the (1D) spatial differential operator matrix must start and end with a zero row-vector:

$$\left[\tilde{D}\right]_i^p = \left[D\tilde{\Delta}\right]_i^p = \frac{D}{\Delta x^2} \cdot \begin{bmatrix} 0 & 0 & 0 & & & & & & \\ 1 & -2 & 1 & & & & & \bar{\bar{0}} & \\ 0 & 1 & \ddots & \ddots & & & & & \\ & & \ddots & \ddots & \ddots & & & & \\ & & & \ddots & \ddots & \ddots & & & \\ & & & & \ddots & \ddots & \ddots & & \\ & & & & & \ddots & \ddots & 1 & 0 \\ & \bar{\bar{0}} & & & & & 1 & -2 & 1 \\ & & & & & & 0 & 0 & 0 \end{bmatrix} \tag{15}$$

where D is either the diffusion coefficient or the thermal diffusivity.

2.5. Deriving 2-dimensional from 1-dimensional case

Let $\widetilde{\mathrm{A}}\{\mathrm{f}(x)\}$ be an operator dealing with functions with variable marked as x. Also let $\widetilde{\mathrm{B}}\{\mathrm{g}(y)\}$ be another operator dealing with functions with variable denoted to as y. According to the mathematicians' definition [6],[7], the *tensor product* of these operators is an $\widetilde{\mathrm{A} \otimes B}\{\mathrm{h}(x,y)\}$ operator dealing with 2-variable functions in such a manner, that for any functions $\mathrm{f}(x)$ and $\mathrm{g}(y)$ it satisfies the equation:

$$\widetilde{\mathrm{A} \otimes B}\{\mathrm{f}(x) \cdot \mathrm{g}(y)\} = \widetilde{\mathrm{A}}\{\mathrm{f}(x)\} \cdot \widetilde{\mathrm{B}}\{\mathrm{g}(y)\} \tag{16}$$

When we approximate continuous functions by discrete data vectors, operators has to be replaced by matrices. When dealing with higher dimensions and multiple variables, data vectors become matrices, while operator matrices turn into multi-dimensional hypermatrices. To simplify formalism, we have to flatten the data matrix to a single vector, containing the rows or columns of the matrix in an ordered way. Let operator $\tilde{A}$ have an $m \times n$ sized matrix $\bar{\bar{A}}$, whilst let the matrix of $\tilde{B}$ be a $p \times q$ matrix, marked as $\bar{\bar{B}}$. In this case, when taking the above tensor

product of the two operators, we have to take the so-called *Kronecker's product* of the two matrices:

$$\left[\bar{\bar{A}}\right]_{m\times n} \otimes \left[\bar{\bar{B}}\right]_{p\times q} = \begin{bmatrix} a_{1,1}\cdot\bar{\bar{B}} & \cdots & a_{1,n}\cdot\bar{\bar{B}} \\ \vdots & \ddots & \vdots \\ a_{m,1}\cdot\bar{\bar{B}} & \cdots & a_{m,n}\cdot\bar{\bar{B}} \end{bmatrix}_{(m\cdot n)\times(p\cdot q)} \tag{17}$$

We've displayed the sizes of the matrices to highlight that the sizes along with the corresponding dimensions are multiplicated.

In case of two dimensions, operator $\widetilde{\mathrm{D}}\{\mathrm{f}(x,y)\} = \frac{\partial^2 \mathrm{f}(x,y)}{\partial x^2}$ can be written as the tensor product of the one-dimensional second derivation $\widetilde{\mathrm{A}}\{\mathrm{f}(x)\} = \frac{d^2 \mathrm{f}(x)}{dx^2}$ and an identity operator $\widetilde{\mathrm{B}}\{\mathrm{f}(y)\} = \mathrm{f}(y)$. As the two-dimensional *Laplacian* operator contains two such a member summed up, it can be considered as the sum of two such tensor products. Having a $N_x \times N_y$ data grid, its matrix has a size of $(N_x \cdot N_y) \times (N_x \cdot N_y)$ and can be written as:

$$\left[\overline{\overline{\Delta_{N_x\times N_y}}}\right] = \left[\overline{\overline{\Delta_{N_x}}}\right] \otimes \left[\overline{\overline{I_{N_x}}}\right] + \left[\overline{\overline{I_{N_x}}}\right] \otimes \left[\overline{\overline{\Delta_{N_y}}}\right] \tag{18}$$

where:

- $\overline{\overline{\Delta_{N_x}}}$ and $\overline{\overline{\Delta_{N_y}}}$ are $N_x \times N_x$ and $N_y \times N_y$ one-dimensional *Laplacian* operator matrices,
- $\overline{\overline{I_{N_x}}}$ and $\overline{\overline{I_{N_y}}}$ are $N_x \times N_x$ and $N_y \times N_y$ unity matrices,
- N_x and N_y are the number of datapoints along the x and y axes, respectively.

When applying this to the differential operator $\widetilde{D}$ and its finite matrix in equation (12), one has to take further considerations regarding the boundary points. Namely, we have to replace with zero the first and last element of the unity matrix in case of *Dirichlet*'s condition:

$$\left[\overline{\overline{\Delta^{Dirichlet}_{N_x\times N_y}}}\right] = \\ = \left[\overline{\overline{\Delta^{Dirichlet}_{N_x}}}\right] \otimes \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \overline{\overline{I_{N_y-1}}} & \vdots \\ 0 & \cdots & 0 \end{bmatrix} + \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \overline{\overline{I_{N_x-1}}} & \vdots \\ 0 & \cdots & 0 \end{bmatrix} \otimes \left[\overline{\overline{\Delta^{Dirichlet}_{N_x}}}\right] \tag{19}$$

(Otherwise, simulation will enable heat transfer/diffusion at the boundary, in a direction parallel to the boundary, and only the corner points will be truly fixed.)

MATLAB has a built-in function to calculate *Kronecker*'s product, and based on it, we have created a *MATLAB* function to generate this matrix. See **APPENDIX 2**.2.

2.6. Reference Solutions, Measuring Error, Verification

If we wish to measure the accuracy of the computation and make statistics, we need a reference solution to deal with. In our paper, we only use initial problems with well-known analytical solutions, hence we can use the analytical solution, as a reference. As we restrict our examination to *Dirichlet*'s case only, it is easy to choose this option.

Once we have the reference solution, we have several ways to measure the distance between the output of our algorithm and the reference. The 2 most used error definitions (with one index for both dimension) are the following:

- Absolute maximum of difference:

$$Err_{max} = \frac{\max_{1 \le p \le N_x \cdot N_y} \left| f_p - f_p^{(ref)} \right|}{N_x \cdot N_y} \tag{20}$$

- RMS value of difference:

$$Err_{RMS} = \sqrt{\frac{\sum_{p=1}^{N_x \cdot N_y} \left(f_p - f_p^{(ref)} \right)^2}{N_x \cdot N_y}} \tag{21}$$

In our work, we use the absolute maximum value formula (20). Our test-framework we used to test the implementations can be found at [10].

3. Implementing the Algorithms

The repository, where we implemented the algorithms can be found at [11].

3.1. Using Simple For-Loop in *MATLAB*

When using *MATLAB*, we would like to compare the computation time of single `for`-loops and matrix-based implementation. We have to start with the simple, loop-based version of Euler's method. The implementation in *1D* and *2D* can be seen at APPENDIX 1.

3.2. Taking Advantage of *MATLAB*'s Sparse Matrices

As *MATLAB* is based on a matrix-oriented framework with lots of built-in optimizations, we suppose that it takes less computation time to use matrix-algebra instead of using loops directly. Namely, we use matrix (15) and its 2-dimensional version. In this case, the kernel of the main loop of *Euler's method* (12) becomes:

$$f_{i,j,k}^{(n)} \cong f_{i,j}^{(n-1)} + \sum_{p=1,q=1}^{N_x,N_y} \left[\widetilde{D}\right]_{i,j}^{p,q} f_{p,q}^{(n-1)} \cdot \Delta t \tag{22}$$

or if we have flattened the data matrix to a single-column vector:

$$f_i^{(n)} \cong f_i^{(n-1)} + \sum_{p=1}^{N_x \cdot N_y} \left[\widetilde{D}\right]_i^p f_p^{(n-1)} \cdot \Delta t \tag{23}$$

In a matrix form:

$$f^{(n)} \cong f^{(n-1)} + \overline{\overline{D}} \cdot f^{(n-1)} \cdot \Delta t \tag{24}$$

As mentioned earlier, we created a dedicated function in *MATLAB* to produce kernel matrix $\overline{\overline{D}}$, which can be seen at **APPENDIX 2**.2. Note that in case of a $N_x \times N_y$ spatial grid, this matrix has $\left(N_x \cdot N_y\right)^2$ elements, which can easily deplete the computer's memory for large grids (let alone 3-dimensional cases of future development), even without the presence of actual data. To make the situation better, these functions use *MATLAB*'s sparse matrix architecture, which means only the non-zero elements of the matrix allocate memory. The main loop of *Euler*'s method in its simplified, matrix-based version can be seen at **APPENDIX 2.1**.

4. Sample Data

4.1. Scalings

4.1.1. Timescales

We always use 11 different timescales, namely (neglecting physical dimensions):

Table 1. The timescales used in this paper

N_t	T_{init}	T_{fin}	Δt
500	0	0.04	8.000e-5
900	0	0.04	4.444e-5
1,500	0	0.04	2.667e-5
3,000	0	0.04	1.333e-5
5,000	0	0.04	8.000e-6
9,000	0	0.04	4.444e-6
15,000	0	0.04	2.667e-6
30,000	0	0.04	1.333e-6
50,000	0	0.04	8.000e-7
90,000	0	0.04	4.444e-7
150,000	0	0.04	2.667e-7

4.1.2. 1D Spatial Grids

In case of 1 dimension, we used 7 different spatial grids:

Table 2. The spatial grids used in 1-dimensional case

N_x	X_{init}	X_{fin}	Δx
50	0	10	2.041e-1
100	0	10	1.010e-1
200	0	10	5.025e-2
400	0	10	2.506e-2
800	0	10	1.252e-2
1,200	0	10	8.340e-3
2,000	0	10	5.003e-3

4.1.3. 2D Spatial Grids

In 2D case, we examine the same timescales, as in 1D. On the other hand, the number of spatial grids has increased:

Table 3. The spatial grids used in the 2-dimensional case

N_x	N_y	N_xN_y	$\Delta x\Delta y$
25	25	625	1.736e-3
25	50	1,250	8.503e-4
50	50	2,500	4.165e-4
50	75	3,750	2.758e-4
75	75	5,625	1.826e-4
75	100	7,500	8.365e-4
100	100	10,000	5.020e-4

We have reduced the examined spatial intervals to have the same number of stable datapoints as in 1D. Hence, the starting and ending grid-points are:

$$\begin{bmatrix} X_{init} \\ Y_{init} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} X_{fin} \\ Y_{fin} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}. \tag{25}$$

4.2. Sample Data

4.2.1. 1-dimension

To test our implementations in 1 dimension, we use:

$$\mathrm{f}(x, t = T_{init}) = 0.8 \cdot \sin\left(\pi \frac{x - X_{init}}{X_{fin} - X_{init}}\right) + 1 \tag{26}$$

which has the analytic solution:

$$\mathrm{f}(x,t) = 0.8 \cdot \exp\left\{-\pi^2 \frac{D\cdot(t-T_{init})}{(X_{fin}-X_{init})^2}\right\} \cdot \sin\left(\pi \frac{x-X_{init}}{X_{fin}-X_{init}}\right) + 1 \qquad (27)$$

In Figure 1, this solution is shown for $T_{init} = 0$ and $T_{fin} = 0.5$. Note that we used $T_{fin} = 0.04$ for the actual computation.

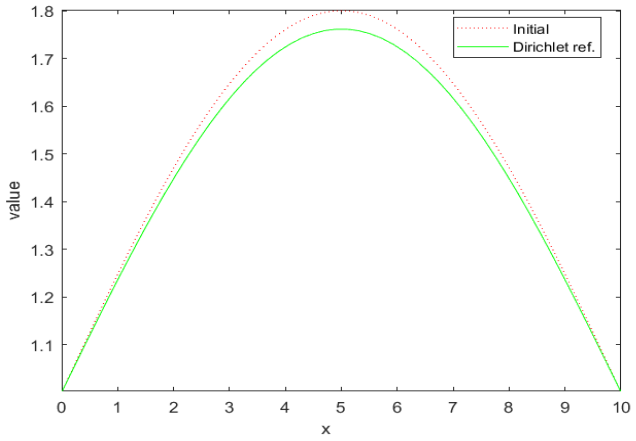

Figure 1. Example initial function (red dotted line) and the analytical solution in 1-dimensional case with *Dirichlet*'s boundary condition.

4.2.2. 2-dimensions

In 2 dimensions, we take the product of 2 sine terms:

$$\mathrm{f}(x,y,t = T_{init}) = 0.8 \cdot \sin\left(\pi \frac{x-X_{init}}{X_{fin}-X_{init}}\right) \cdot \sin\left(\pi \frac{y-Y_{init}}{Y_{fin}-Y_{init}}\right) + 1 \qquad (28)$$

On a colormap ($T_{init} = 0$; $T_{fin} = 0.04$):

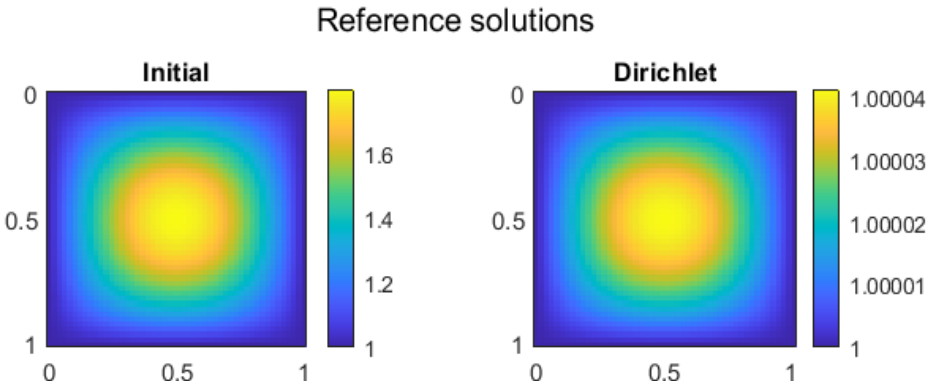

Figure 2. Example initial function and its computed solution in 2-dimensional *Dirichlet*'s case

This has the following analytic solution:

$$\mathrm{f}(x,y,t) = 0.8 \cdot \exp\left\{-\pi^2 DT \cdot \left(\frac{1}{X^2}+\frac{1}{Y^2}\right)\right\} \cdot \\ \cdot \sin\left(\pi \frac{x-X_{init}}{X}\right) \cdot \sin\left(\pi \frac{y-Y_{init}}{Y}\right) + 1 \quad (29)$$

where:

$$\mathrm{T}(t) = t - T_{init} \\ X = X_{fin} - X_{init} \\ Y = Y_{fin} - Y_{init} \quad (30)$$

5. Measurement Results

Before stating the results, we have to notice, that we used a computer with *Intel Core i7-9700*, *3GHz* CPU, *16GB* RAM and *64-bit Windows 10 Pro*.

5.1. Single *FOR*-loop

5.1.1. Computation time

In case of the single for loop implementation, we have got the following CPU-time values:

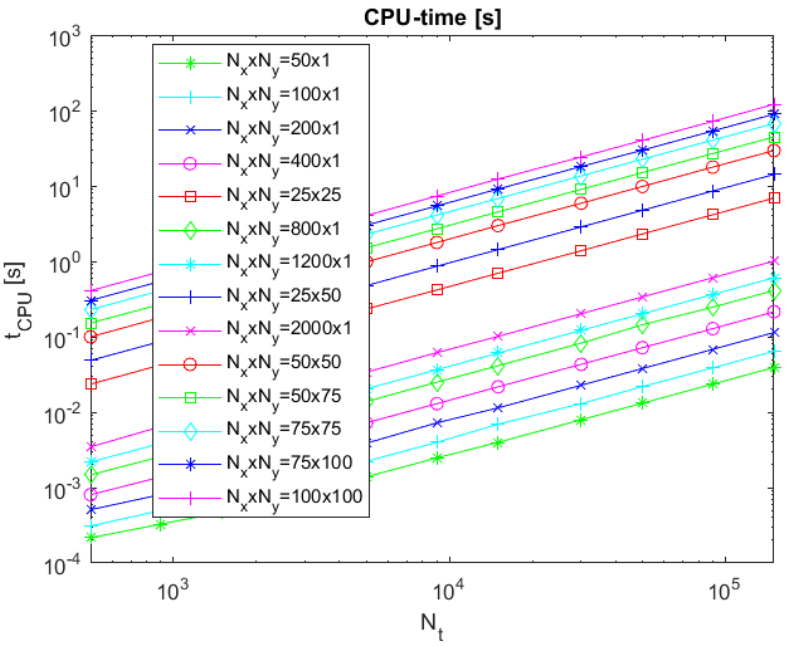

Figure 3. CPU-time of the loop-based implementation, as a function of timestep count, for different spatial grids

As a colormap:

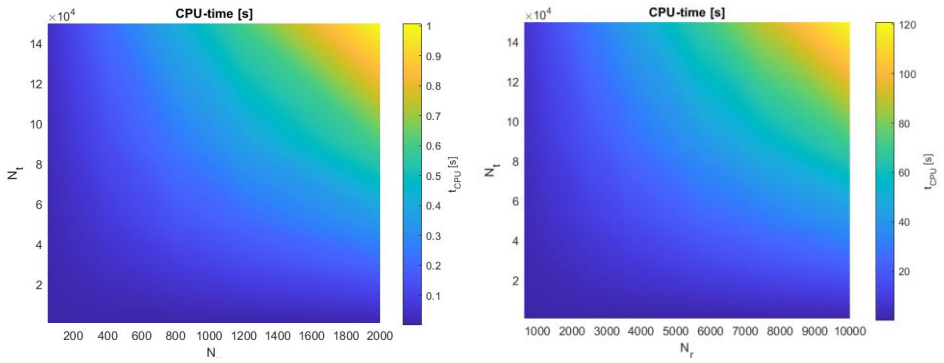

Figure 4. CPU-time of the loop-based implementation, as a function of timestep and spatial data-point count, for 1D (left) and 2D (right)

To compare 1- and 2-dimensional case in a more spectacular way, we can rearrange Figure 3 to let the number of cells be on the horizontal axis, and filter out some timescales:

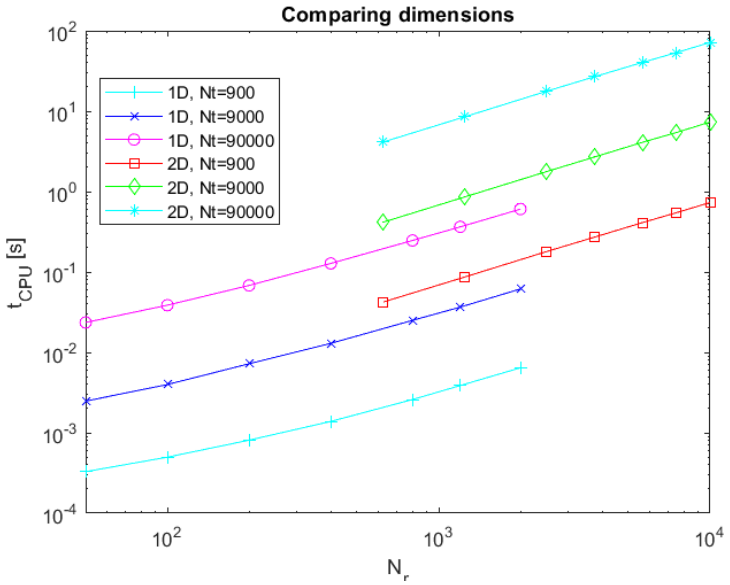

Figure 5. CPU-time of the loop-based implementation, as a function of the number of spatial nodes, for different dimensions and timestep counts

The linear connection between the computation time and the number of steps and grid-points can be seen, especially for higher (>200) number of cells. According to **Figure 5**, the CPU-time in 2-dimensional case is about 20 times greater than in 1

dimension, if the number of spatial datapoints and the number of timesteps are equal.

5.1.2. Error of 1D computation

If we display the error of the 1-dimensional computation:

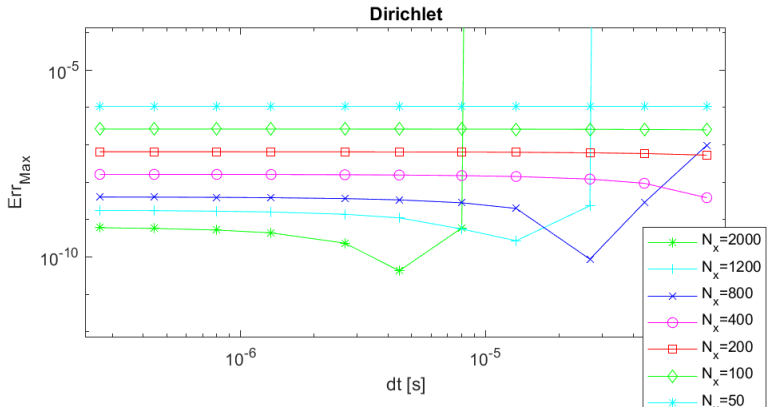

Figure 6. Computation error of the loop-based implementation, as a function of timestep size, for the 1D case

One can see that finer meshes yield more accurate solutions, but only if the time step size is sufficiently small. The CFL limit is lower if the mesh is finer. Above this limit, the algorithm is unstable and the error is extremely large. If one decrease the time step size, the error suddenly drops to a very small value at a special point, where the leading error terms of the space and time discretization cancels each other. If the time step size is decreased further, the error tends to a constant value, which is due to the space discretization only and its leading term is $-\frac{D}{12} f_{(4x)} \Delta x^2$, where $f_{(4x)}$ is the forth derivative of the function with respect to x. (See [9])

To describe the performance better, we can plot the error as a function of the computation time:

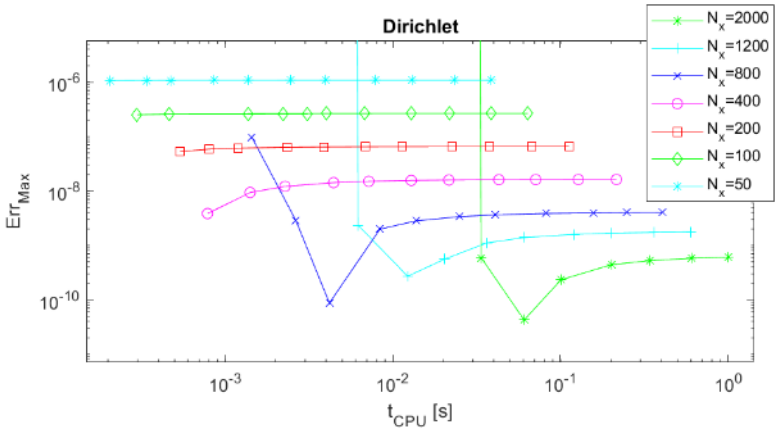

Figure 7. Computation error of the loop-based implementation as a function of CPU-time, for the 1D case

5.1.3. Error of the 2D computation

The error of the 2-dimensional computation as a function of the time step size and the computation time is presented in Figure 8 and Figure 9, respectively.

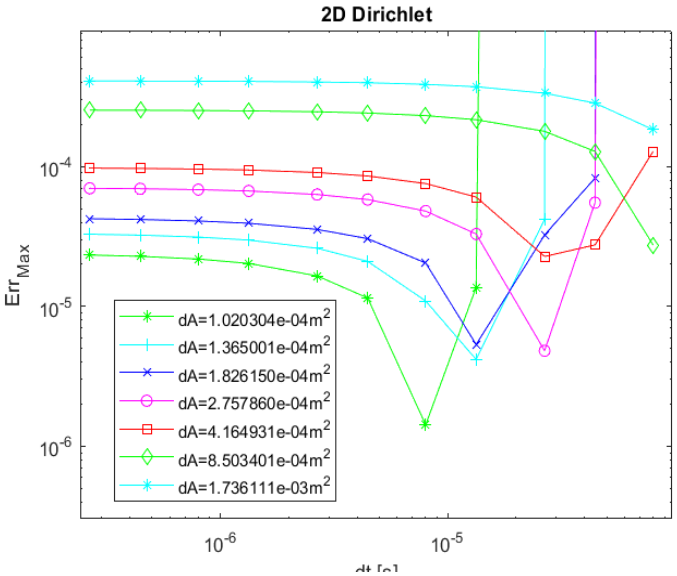

Figure 8. Computation error of the loop-based implementation, as a function of timestep size, for the 2D case

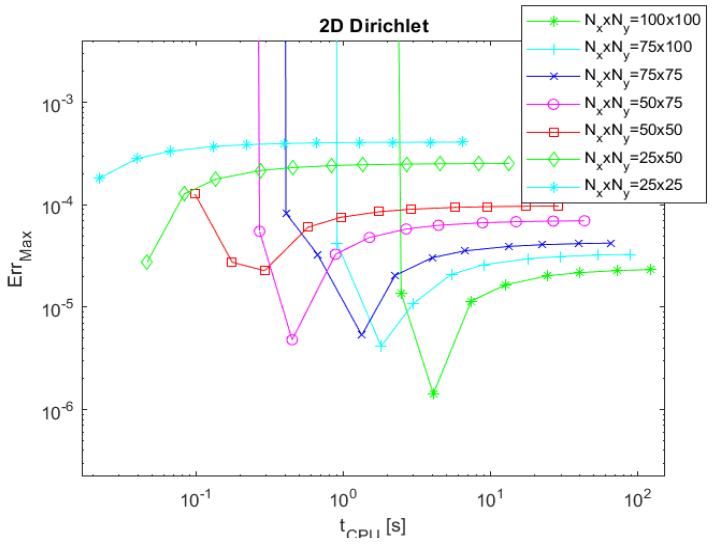

Figure 9. Computation error of the loop-based implementation, as a function of CPU-time, for the 2D case

5.2. Sparse matrix-based computation

5.2.1. Computation time

The computation time of the sparse matrix-based implementation, as a function of the number of time and space points can be seen in Figure 10 and Figure 11, respectively.

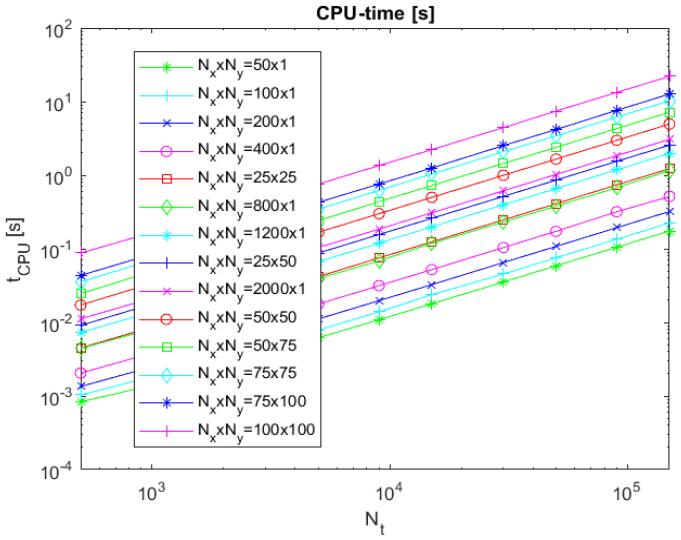

Figure 10. Computation time of the matrix-based implementation, as a function of timestep count, for different spatial grids

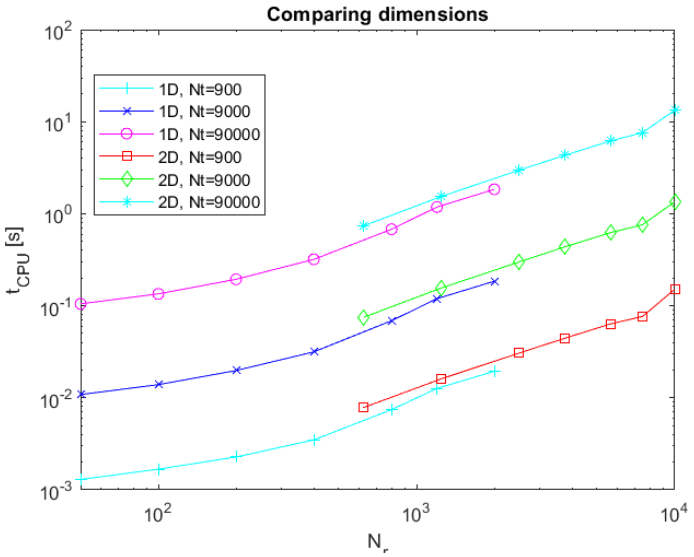

Figure 11. Computation time of the matrix-based implementation, as a function of data-point count, for different dimension and timestep counts

The linear connection between the computation time and the number of timesteps still holds, but in case of the number of cells, the proportional connection is a bit less obvious, than in case of the loop-based implementation. The reason is, that *MATLAB* does some optimization in the background, dynamically changing some implementtation details from experiment to experiment. At the end of the day, the difference between the 1-dimensional and 2-dimensional computations is largely reduced.

5.2.2. Error of the 1D computation

The error of the matrix-based implementation in 1-dimensional case:

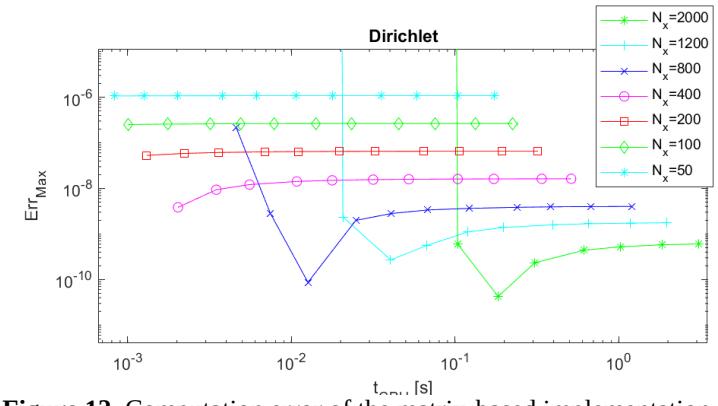

Figure 12. Computation error of the matrix-based implementation, as a function of CPU-time, for the 1D case

This implementation gives the same results as the previous one, but the running times are different.

5.2.3. Error of 2D computation

In 2-dimensional case:

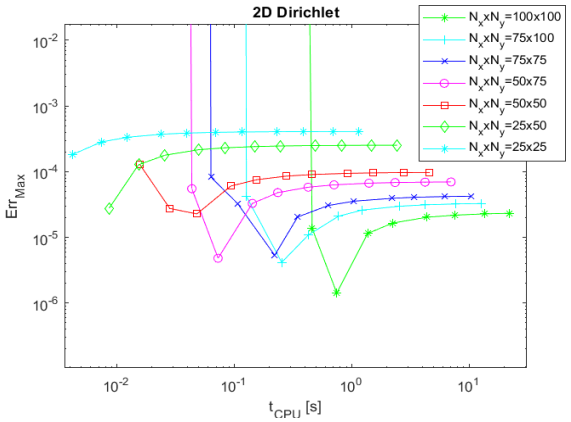

Figure 13. Computation error of the matrix-based implementation, as a function of CPU-time, for the 2D case

5.3. Comparing loop-based and matrix-based solution

We can compare the different implementations by plotting similar figures like Figure 5 or Figure 11. The running time as a function of the spatial nodes are presented in Figure 14, Figure 15 and Figure 16 for the 900, 9000 and 90000-timestep case, respectively.

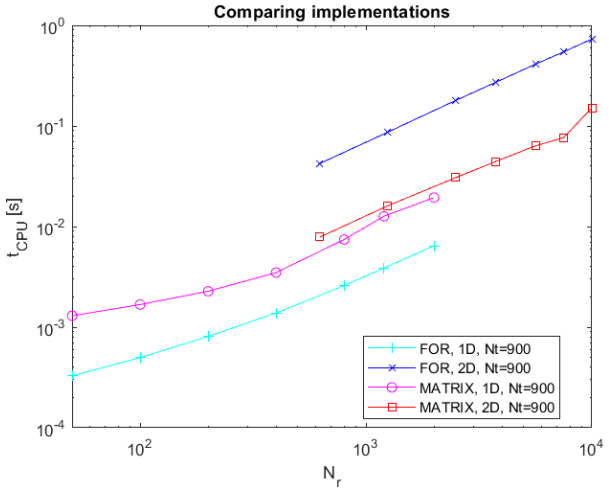

Figure 14. Comparison of the loop-based and the matrix-based implementation, using 900-timestep measurement data

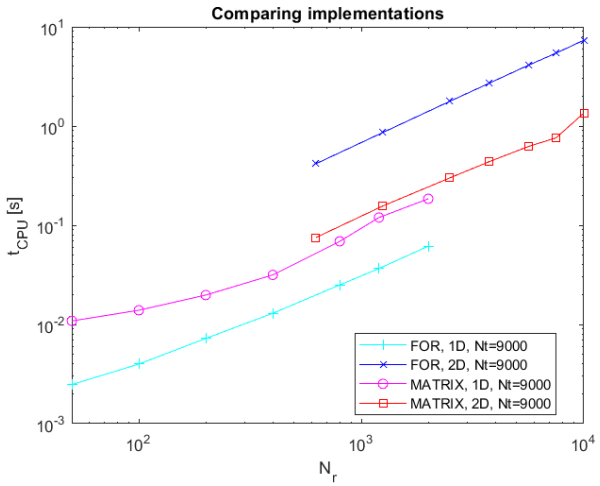

Figure 15. Comparison of the loop-based and the matrix-based implementation, using 9000-timestep measurement data

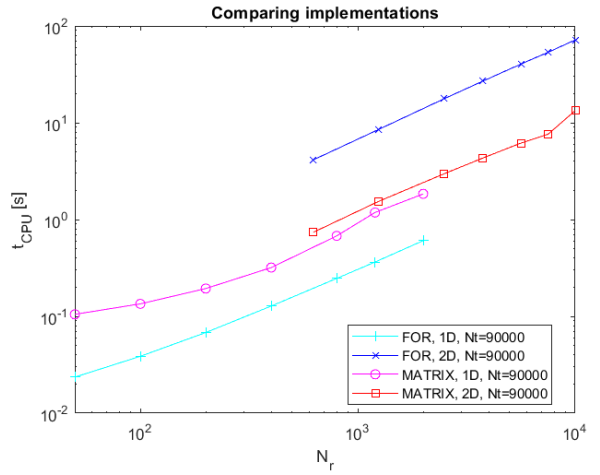

Figure 16. Comparison of the loop-based and the matrix-based implementation, using 90000-imestep measurement data

Based on the above figures – as well as comparing Figure 6 and Figure 11 –, one can say that the 1-dimensional computation time has increased by a factor of 5, while the 2-dimensional time values has decreased to a fraction of 4-5. One can conclude, that only in case of multiple dimensions does the matrix-framework accelerate the computation.

6. Conclusion

In this work, we solved the linear diffusion equation by the explicit Euler (FTCS) method implemented in two different ways: with for loops as well as the built-in matrix operations of *MATLAB*. We obtained that the matrix-based implementation is worth using only if the number of space dimensions is two, regardless of the time step size. In the case of 1 dimension, the overhead of the framework used by *MATLAB* makes things worse and we suggest using the traditional for-loop implementation. We consider this work as only a preliminary investigation, based on which the performance of the *MATLAB* implementations can be compared to other programming languages such as *C++* in the case of more complicated problems and more sophisticated numerical methods as well. After that, we are going to parallelize the calculations on GPUs to achieve a serious increase in speed, which will enable us to solve large-scale engineering problems.

References

[1] Dang, Q. A., Hoang, M. T.: Positive and elementary stable explicit nonstandard Runge-Kutta methods for a class of autonomous dynamical systems. *International Journal Computer Mathematics*, vol. 97, no. 10, pp. 2036–2054. https://doi.org/10.1080/00207160.2019.1677895

[2] Beuken, L., Cheffert, O., Tutueva, A., Butusov, D., Legat, V.: Numerical Stability and Performance of Semi-Explicit and Semi-Implicit Predictor–Corrector Methods. *Mathematics*, vol. 10, no. 12, 2022, https://doi.org/10.3390/math10122015.

[3] Pourghanbar, S., Manafian, J., Ranjbar, M., Aliyeva, A., Gasimov, Y. S.: An efficient alternating direction explicit method for solving a nonlinear partial differential equation. *Mathematical Problems in Engineering*, vol. 2020, no. November, pp. 1–12, 2020, https://doi.org/10.1155/2020/9647416.

[4] Mazzia, F., Y., Sergeyev, Y. D., Iavernaro, F., Amodio, P., Mukhametzhanov, M. S.: Numerical methods for solving ODEs on the Infinity Computer. *AIP Conference Proceedings*, vol. 1776, no. 1, 2016, p. 090033, https://doi.org/10.1063/1.4965397.

[5] Saleh, M., Kovács, E., Barna, I. F., Mátyás, L.: New Analytical Results and Comparison of 14 Numerical Schemes for the Diffusion Equation with Space-Dependent Diffusion Coefficient. *Mathematics*, vol. 10, no. 15, Aug. 2022, p. 2813. https://doi.org/10.3390/math10152813

[6] Bourbaki, N.: *Elements of Mathematics*. Vol. 1, chap. 2.3. Berlin, Springer-Verlag, 1989.

[7] Petz D.: *Lineáris analízis*. Chap. 1.2. Budapest, Műegyetemi Kiadó, 2001, id. 05057.

[8] Kumar, V., Chandan, K., Nagaraja, K. V., Reddy, M. V.: Heat Conduction with Krylov Subspace Method Using FEniCSx. *Energies*, vol. 15, no. 21, Oct. 2022, p. 8077, https://doi.org/10.3390/en15218077

[9] Nagy, Á., Majár, J., Kovács, E.: Consistency and Convergence Properties of 20 Recent and Old Numerical Schemes for the Diffusion Equation. *Algorithms*, vol. 15, no. 11, Nov. 2022, p. 425, https://doi.org/10.3390/a15110425.

[10] https://bitbucket.org/koicsd/test-tool/.

[11] https://bitbucket.org/koicsd/diffusion/.

APPENDIX

In this appendix, we present some important code snippets from our repository [11], extended with some comments for higher clarity.

1. Loop-based implementation of *Euler*'s method

1.1. 1-dimensional case

```
dt=(Tfin-Tinit)/Nt; % timestep size
Nx=numel(data); % number of spatial grid-points
dx=(Xfin-Xinit)/(Nx-1); % size of spatial step

... % preconditioning of periodic case comes here

COEFF_ = dt * coeff / dx / dx;
temp=zeros(1, Nx); % temporary data-vector for iterations
for n = 1 : Nt
    temp(1) = ... % first point of grid
    for i = 2 : Nx - 1
        % inner datapoints
        temp(i) = data(i) + COEFF_ * ...
            (data(i-1) + data(i+1) - 2 * data(i));
    end
    temp(Nx) = ... % last point of grid;
    data = temp;
end
```

1.2. 2-dimensional case

```
dt=(Tfin-Tinit)/Nt; % timestep size
Nr=size(data); % = [Nx, Ny] -- number of spatial grid-points
dr=(Rfin-Rinit)./(Nr-1); % = [dx, dy] -- size of spatial step
COEFF_ = dt * coeff ./ dr ./ dr;

% preconditioner loops of periodic case come here

temp = zeros(Nx, Ny);
for n = 1 : Nt
    temp(1,1) = ... % top left point
    % temp(1,Ny) = ... % top right point
    % temp(Nx,1) = ... % bottom left point
    % temp(Nx,Ny) = ... % bottom right point
    for i = 2 : Nx-1
        temp(i,1) = ... % top edge of grid
        temp(i,Nx) = ... % bottom edge of grid
    end
    for j = 2 : Ny-1
        temp(1,j) = ... % left edge of grid
        temp(Nx,j) = ... % right edge of grid
    end
    for i = 2 : Nx-1
        for j = 2 : Ny-1
```

```
            % inner point
            temp(i,j) = data(i,j) + COEFF_ * [
                data(i-1,j) + data(i+1,j) - 2 * data(i,j);
                data(i,j-1) + data(i,j+1) - 2 * data(i,j)
            ];
        end
    end
    data = temp;
end
```

2. Matrix-based implementation of *Euler*'s method

2.1. Main loop

Please, substitute `KerMat_Diffusion_FixedBC` from 2.2 to `kerfun`.

```
dt=(Tfin-Tinit)/Nt;  %% time step
Nr = size(data_init);  %% number of datapoints along different axes
as vector
Nflat = prod(Nr);  % numel(Nr)  %% total number of datapoints

if isvector(data_init)
    %% creating sparse matrices
    [kernel, precond] = kerfun(Rinit, Rfin, Nflat, varargin{:});
else
    %% creating sparse matrices
    [kernel, precond] = kerfun(Rinit, Rfin, Nr, varargin{:});
end

%% flattening and preprocessing
flatdata = precond * reshape(data_init, Nflat, 1);

%% processing flattened data (iteration over time)
for i = 1 : Nt
    flatdata = flatdata + dt * kernel * flatdata;
end
data_fin = reshape(flatdata, Nr);
```

2.2. Sparse matrix creator function

To be used with 2.1.

```
function [kernel, precond] = KerMat_Diffusion_FixedBC(...
        Rinit, Rfin, Nr, coeff)
% Rinit, Rfin and Nr must be a vector (or a scalar for 1D calc.)!
% Rinit, Rfin and Nr must have the same array-size!

dim = numel(Nr);
dr = (Rfin - Rinit) ./ (Nr - 1);

switch dim
    case 1
        Nx = Nr;
```

```
        dx = dr;
        kernel = coeff * spdiags([  % Laplace's tridiag. truncated
                %  1 above diag, last element 0:
                ones(1,Nx-2)    0                   0;
                %  -2 in diag, first and last element 0:
                0               repmat(-2, 1,Nx-2)  0
                %  1 below diag, first element 0:
                0               0                   ones(1,Nx-2)
        ]', [-1 0 1], Nx, Nx) / dx / dx;
        precond = speye(Nx);

    case 2
        Ny = Nr(1);
        Nx = Nr(2);
        Yinit = Rinit(1);
        Xinit = Rinit(2);
        Yfin = Rfin(1);
        Xfin = Rfin(2);
        % dy = dr(1);
        % dx = dr(2);
        [kernelY, precondY] = KerMat_Diffusion_FixedBC(...
                Yinit, Yfin, Ny, coeff);  % 1D trunc. Laplacian
        [kernelX, precondX] = KerMat_Diffusion_FixedBC(...
                Xinit, Xfin, Nx, coeff);  % 1D trunc. Laplacian
        kernel = ...
            % 2nd partial with rspt. to x, as
            % 1D Laplacian by identity matrix (all truncated):
            kron(kernelX, ...
                spdiags([0; ones(Ny-2,1); 0], 0, Ny, Ny))...
            + ...
            % 2nd partial with respect to y, as
            % identity by 1D Laplacian matrix (all truncated):
            kron(spdiags([0; ones(Nx-2,1); 0], 0, Nx, Nx),...
                kernelY);
        precond = kron(precondX, precondY);

    otherwise
        error('Unsupported number of dimensions!')
end
end
```