SIMPLE method for the incompressible Navier-Stokes with the FTCS scheme for the viscous and thermal fluxes implemented in the Julia programming language

Appoloni Alberto

September 2, 2024

Abstract

This document presents the implementation of the SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) method for solving the incompressible Navier-Stokes equations. The implementation utilises the FTCS (Forward-Time Central-Space) scheme to handle the viscous and thermal fluxes and is carried out using the Julia programming language.
The paper presents a discussion in terms of computation time performance between Julia and Matlab fixing the grid size and varying the simulation time stop(t_{end}), and fixing t_{end} and varying the grid size. Two versions of Julia are performed: one faithfully converting the code from Matlab to Julia: "Julia base", the other implementing some improvements to Julia's code: "Julia improved". In both cases, Julia "improved" performs better compared to Matlab, in the first case at $t_{end} = 4000s$ "Julia improved" takes $\approx 27.1\%$ of Matlab computation time, while in the second case at 500^2 grid size, "Julia improved" takes 19.6% of Matlab computational time.
Julia "base", instead, takes more time compared to Matlab, fixing t_{end} Julia base has an exponential behaviour which rises faster than Matlab while fixing the grid size "Julia base" has a quadratic behaviour.

1 Introduction

The SIMPLE (Semi-Implicit Method for Pressure Linked Equations) method is a numerical technique widely used in computational fluid dynamics (CFD) to solve the Navier-Stokes equations, particularly for problems involving incompressible flows. This method, developed by Suhas Patankar and Spalding in 1972, provides a way to couple velocity and pressure in a flow field, allowing for a stable and accurate solution of fluid motion equations.
The SIMPLE method is based on iteratively correcting the pressure field to ensure that the velocity field satisfies the continuity equation, representing mass conservation in incompressible flows.
The method is based on the following procedure:

- Initialization: start with an initial guess for the new pressure field, for example: $P_h^* = 0$ or $P_h^* = P_h^n$, where P_h^n is the discrete pressure field at time t^n.
- Solving momentum equations: compute an auxiliary velocity $\mathbf{v}_h^*$ at time step t^{n+1} that accounts for the hyperbolic nonlinear advection terms, parabolic diffusive terms and is based on the initial pressure field P_h^*:

$$\mathbf{v}_h^* = \mathbf{v}_h^n - \Delta t \mathbf{v}_h^n \cdot \nabla_h \mathbf{v}_h^n - \Delta t \nabla_h P_h^* + \Delta t \nu \nabla_h^2 \mathbf{v}_h^n \tag{1}$$

  where $\nu = \frac{\mu}{\rho}$ is the kinematic shear viscosity, μ is the dynamic shear viscosity while ρ is the mass density.
- Correcting pressure and velocity fields: $\mathbf{v}_h^{n+1} = \mathbf{v}_h^* + \mathbf{v}_h^{'}$ and $P_h^{n+1} = P_h^* + P_h^{'}$, where $\mathbf{v}_h^{'}$ and $P_h^{'}$ are a velocity and pressure correction respectively.
  The new pressure and velocity fields must satisfy the discrete momentum conservation law:

$$\mathbf{v}_h^{n+1} = \mathbf{v}_h^n - \Delta t \mathbf{v}_h^n \cdot \nabla_h \mathbf{v}_h^n - \Delta t \nabla_h P_h^{n+1} + \Delta \nu \nabla_h^2 \mathbf{v}_h^n \tag{2}$$

  So subtracting 1 from 2 we get and using the div-free condition on v_h^{n+1} ($\nabla \mathbf{v}^{n+1} = 0$) we get a Poisson equation on $P_h^{'}$:

$$\nabla^2 P_h^{'} = \frac{1}{\Delta t} \nabla_h \cdot \mathbf{v}_h^* \tag{3}$$

- Update the velocity field as:

$$\mathbf{v}_h^{n+1} = \mathbf{v}_h^* - \Delta t \nabla_h P_h^{'} \tag{4}$$

- Iteration: Repeat the process. The corrected pressure field is used to solve the momentum equations again, obtaining a new provisional velocity field. Another pressure correction is computed, and so on, until the solution converges, meaning the pressure and velocity corrections become negligible.

1.1 Forward time-centered space (FTCS) and Backward Time Centered Space(BTCS)

FTCS (Forward Time Centered Space) and BTCS (Backward Time Centered Space) are numerical schemes commonly used to solve partial differential equations (PDEs), particularly for problems involving time-dependent phenomena such as heat conduction or diffusion. These schemes are

applied to discretise and solve the heat equation or similar parabolic PDEs.
The FTCS scheme approximates the time derivative using a forward difference:

$$q_t(t^n, x_i) \approx \frac{q_i^{n+1} - q_i^n}{\Delta t}$$

which means that the time derivative term in the equation is discretised using a forward difference approximation. To calculate the value of a function at a future time t_{n+1}, the known values at the current time t_n are used.
The space derivative using a central difference:

$$q_x(t^n, x_i) \approx \frac{q_{i+1}^n - q_{i-1}^n}{2\Delta x}$$

which means approximating the spatial derivative using a centred finite difference. The value of the spatial derivative is calculated by considering the points around a given position.
q is a generic function, while i and n are the indexes for space and time respectively.
The FTCS scheme is explicit: the value at the next time step q_i^{n+1} is calculated directly from the current values q_{i-1}^n, q_i^n, q_{i+1}^n without solving a system of simultaneous equations and it is conditionally stable.
The BTCS scheme is an implicit finite difference method. It approximates the time derivative using a backward difference:

$$q_t(t^n, x_i) \approx \frac{q_i^n - q_i^{n-1}}{\Delta t}$$

and the space derivative using a centred difference. The method is implicit, so the future value q_i^{n+1} depends on other future values, requiring the solution of a system of equations at each time step and it is unconditionally stable: it is stable for any time and spatial step size Δt and Δx.

1.2 Discretization of the operators

The discrete divergence of the discrete velocity is:

$$(\nabla_h \cdot \mathbf{v}_h^n)_{i,j} := \frac{u_{i+1/2,j}^n - u_{i-1/2,j}^n}{\Delta x} + \frac{v_{i+1/2,j}^n - v_{i-1/2,j}^n}{\Delta y} \tag{5}$$

The pressure gradient along x and y is defined at $(x_{i+1/2}, y_j)$ and $(x_i, y_{j+1/2})$ and reads:

$$(\partial_x P_h^n)_{i+1/2,j} := \frac{P_{i+1,j}^n - P_{i,j}^n}{\Delta x}, \quad (\partial_y P_h^n)_{i,j+1/2} := \frac{P_{i,j+1}^n - P_{i,j}^n}{\Delta y} \tag{6}$$

The eq.3 becomes:

$$\frac{P'_{i+1,j} - 2P'_{i.j} + P'_{i-1,j}}{\Delta x^2} + \frac{P'_{i,j+1} - 2P'_{i.j} + P'_{i,j-1}}{\Delta y^2} = \frac{1}{\Delta t}\left(\frac{u^*_{i+1/2,j} - u^*_{i-1/2,j}}{\Delta x} + \frac{v^*_{i,j+1/2} - v^*_{i,j-1/2}}{\Delta y}\right) \tag{7}$$

The velocity field in eq.4:

$$\begin{aligned} u_{i+1/2,j}^{n+1} &= u^*_{i+1/2,j} - \frac{\Delta t}{\Delta x}(P'_{i+1,j} - P'_{i,j}) \\ v_{i,j+1/2}^{n+1} &= v^*_{i,j+1/2} - \frac{\Delta t}{\Delta y}(P'_{i,j+1} - P'_{i,j}) \end{aligned} \tag{8}$$

So the discrete momentum equation(2) in x-direction for $u^*_{i+1/2,j}$ becomes:

$$\begin{aligned}
u^*_{i+1/2,j} = u^n_{i+1/2,j} &- \frac{\Delta t}{\Delta x}\left(u^-_{i+1,j}\left[u^n_{i+3/2,j} - u^n_{i+1/2,j}\right] + u^+_{i+1,j}\left[u^n_{i+1/2,j} - u^n_{i-1/2,j}\right]\right) \\
&- \frac{\Delta t}{\Delta y}\left(v^-_{i+1/2,j+1/2}\left[u^n_{i+1/2,j+1} - u^n_{i+1/2,j}\right] + v^+_{i+1/2,j-1/2}\left[u^n_{i+1/2,j} - u^n_{i+1/2,j-1}\right]\right) \\
&+ \nu\frac{\Delta t}{\Delta x}\left(\frac{u^n_{i+3/2,j} - u^n_{i+1/2,j}}{\Delta x} - \frac{u^n_{i+1/2,j} - u^n_{i-1/2,j}}{\Delta x}\right) \\
&+ \nu\frac{\Delta t}{\Delta y}\left(\frac{u^n_{i+1/2,j+1} - u^n_{i+1/2,j}}{\Delta y} - \frac{u^n_{i+1/2,j} - u^n_{i+1/2,j-1}}{\Delta y}\right) - \frac{\Delta t}{\Delta x}(P^*_{i+1,j} - P^*_{i,j})
\end{aligned}$$

where the average barycenter and corner velocities are:

$$u^n_{i+1,j} = \frac{1}{2}\left(u^n_{i+1/2,j} + u^n_{i+3/2,j}\right), \quad v^n_{i+1/2,j+1/2} = \frac{1}{2}\left(v^n_{i+1,j+1/2} + v^n_{i,j+1/2}\right)$$

and the velocities needed for upwinding:

$$u^\pm_{i,j} = \left(u^n_{i,j} \pm |u^n_{i,j}|\right), \quad v^\pm_{i+1/2,j+1/2} = \frac{1}{2}\left(v^n_{i+1/2,j+1/2} \pm |v^n_{i+1/2,j+1/2}|\right)$$

While for y direction:

$$\begin{aligned}
v^*_{i,j+1/2} = v^n_{i,j+1/2} &- \frac{\Delta t}{\Delta y}\left(v^-_{i,j+1}\left[v^n_{i,j+3/2} - v^n_{i,j+1/2}\right] + v^+_{i,j}\left[v^n_{i,j+1/2} - v^n_{i,j-1/2}\right]\right) \\
&- \frac{\Delta t}{\Delta x}\left(u^-_{i+1/2,j+1/2}\left[v^n_{i+1,j+1/2} - v^n_{i,j+1/2}\right] + u^+_{i-1/2,j+1/2}\left[v^n_{i,j+1/2} - v^n_{i-1,j+1/2}\right]\right) \\
&+ \nu\frac{\Delta t}{\Delta y}\left(\frac{v^n_{i,j+3/2} - v^n_{i,j+1/2}}{\Delta y} - \frac{v^n_{i,j+1/2} - v^n_{i,j-1/2}}{\Delta y}\right) \\
&+ \nu\frac{\Delta t}{\Delta x}\left(\frac{v^n_{i+1,j+1/2} - v^n_{i,j+1/2}}{\Delta x} - \frac{v^n_{i,j+1/2} - v^n_{i-1,j+1/2}}{\Delta x}\right) - \frac{\Delta t}{\Delta y}(P^*_{i,j+1} - P^*_{i,j})
\end{aligned}$$

Where the average barycenter and corner velocities are:

$$v^n_{i,j} = \frac{1}{2}\left(v^n_{i,j+1/2} + v^n_{i,j-1/2}\right), \quad u^n_{i+1/2,j+1/2} = \frac{1}{2}\left(u^n_{i+1/2,j+1} + u^n_{i+1/2,j}\right)$$

The velocities needed for upwinding:

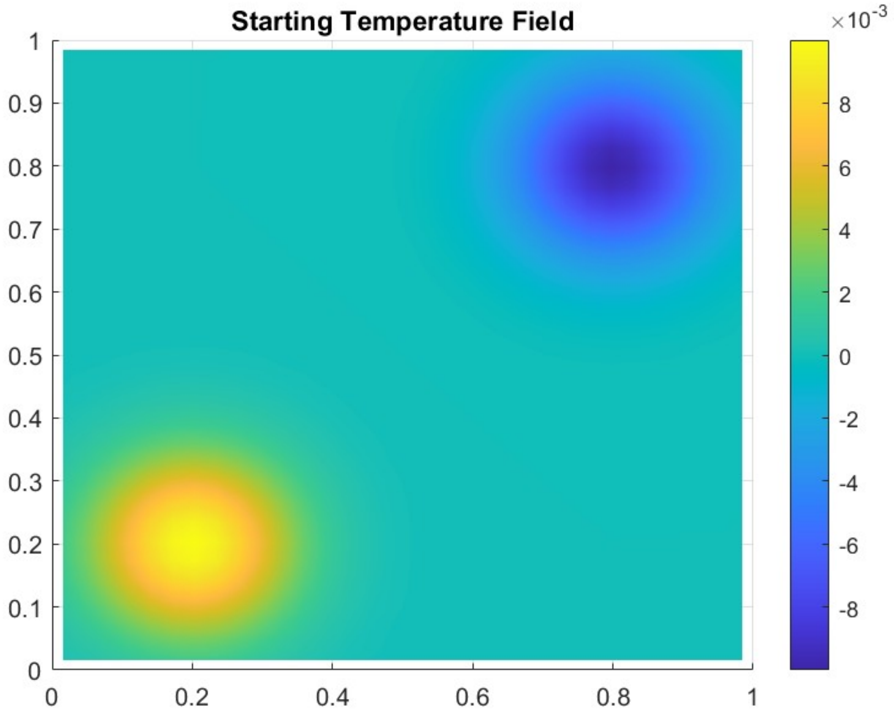
$$v^\pm_{i,j} = \frac{1}{2}\left(v^n_{i,j} \pm |v^n_{i,j}|\right), \quad u^\pm_{i+1/2,j+1/2} = \frac{1}{2}\left(u^n_{i+1/2,j+1/2} \pm |u^n_{i+1/2,j+1/2}|\right)$$

2 Code

The code is composed of a central part where all the relevant variables are defined and the SIMPLE algorithm is developed.
In this part, the computational grid is created and initialised with 32 points along x and y.
The temperature field T is initialized with two Gaussian bumps, one positive around $(0.2, 0.2)$ and one negative around $(0.8, 0.8)$.

$$T(x, y) = T_0 + 0.01 \left(\exp\left\{ -\frac{1}{2} \frac{(x-0.2)^2 + (y-0.2)^2]}{0.1^2} \right\} - \exp\left\{ -\frac{1}{2} \frac{(x-0.8)^2 + (y-0.8)^2]}{0.1^2} \right\} \right) \tag{9}$$

This setup introduces localised thermal perturbations, which are useful for simulating thermal convection and studying heat transfer.



The boundaries are set at $(0, 1)$ for x and y.
In the second part of the code, the temperature field is solved time-dependently.
The time step dt is defined as the minimum between 0.1 and the following

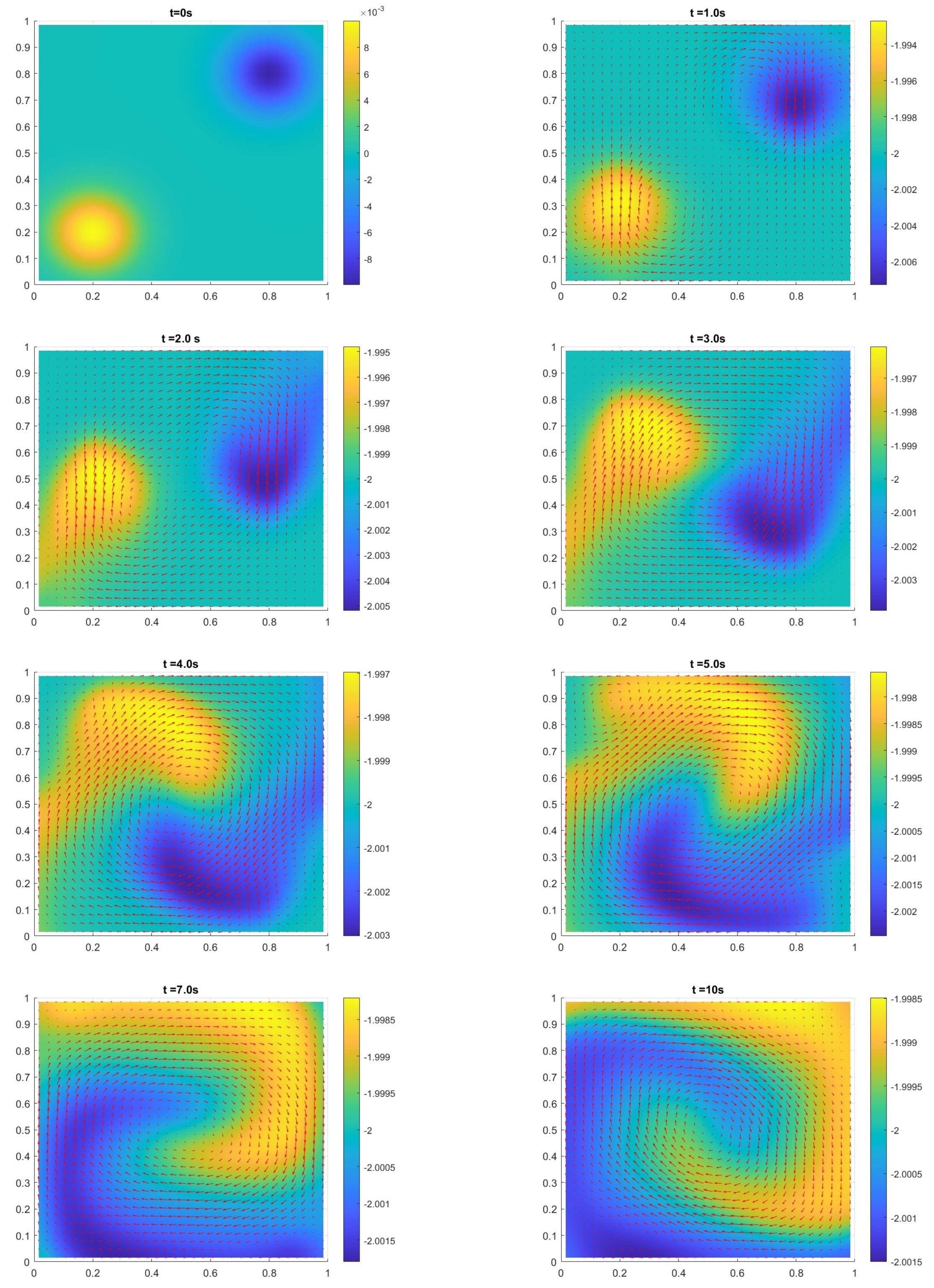
$$dt = \frac{CFL}{\frac{u_{max}}{dx} + \frac{v_{max}}{dy} + 2\nu(\frac{1}{dx^2} + \frac{1}{dy^2}) + 2\lambda(\frac{1}{dx^2} + \frac{1}{dy^2})} \tag{10}$$

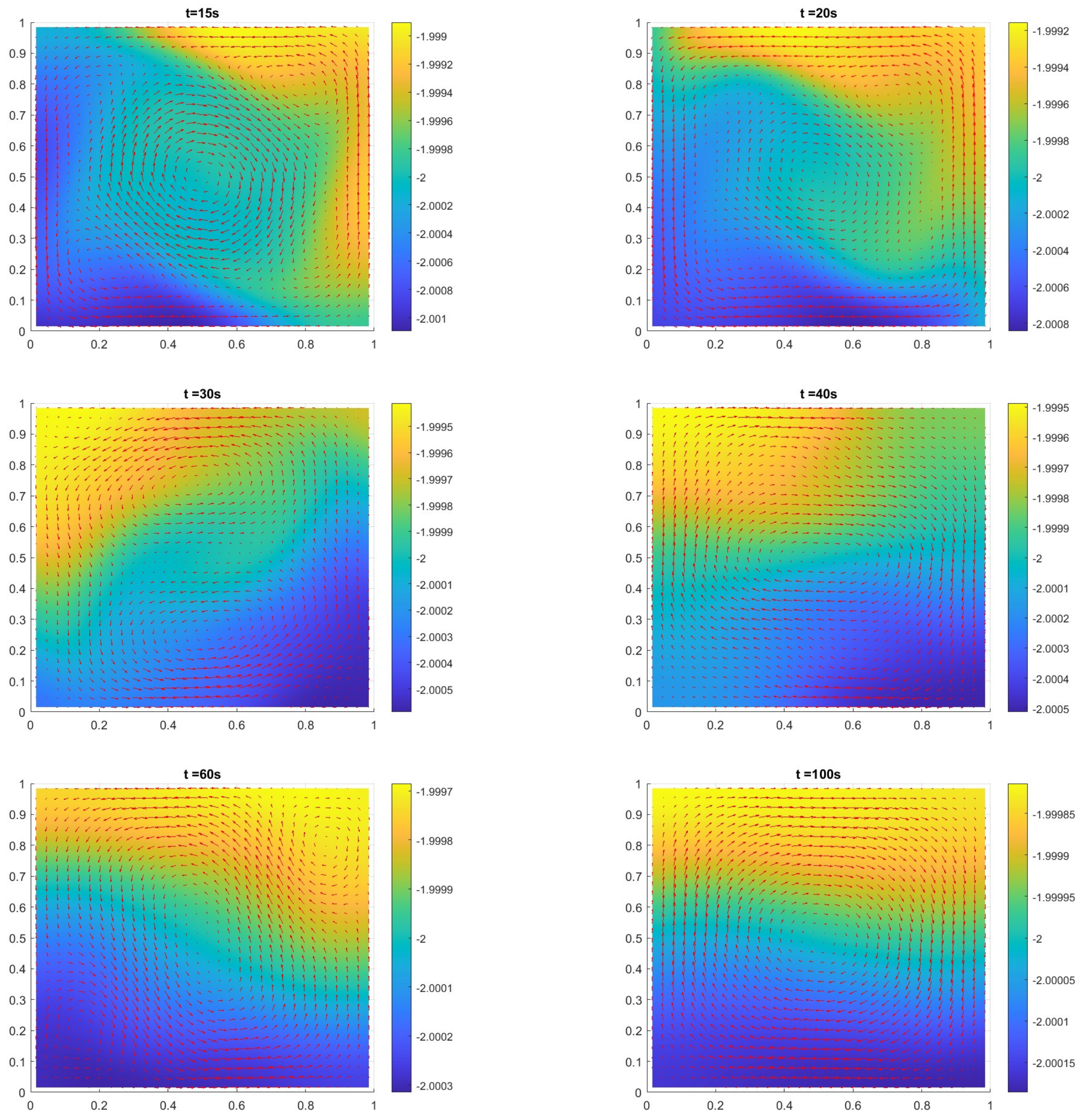
to ensure stability. Where CFL (Courant-Friedrichs-Lewy) number is set at 0.95, u_{max} and v_{max} are the maximum velocities along x and y, while dx and dy are the space steps defined as the difference

between the x and y boundaries divided by the number of points in the grid. dt is composed of 3 parts, the first part is related to the advection of the fluid along x and y, the second part represents the diffusion term related to the kinematic viscosity ν: it considers how velocity diffuses over time due to viscosity, while the third part represents the diffusion term related to thermal diffusivity λ: it considers how temperature diffuses over time due to thermal conductivity.
Then the SIMPLE algorithm is developed by exploiting some external functions.
First of all an initial guess of the pressure is chosen which is initialized with 0. After that the step 2 is performed calculating $\boldsymbol{v}^*$ using the external function "$MomConvectionDiffusion$" and "$AddGradP$" which solve the eq.1. T is computed by solving the Fourier law of the diffusion flux along x and y, which is:

$$\boldsymbol{q} = -\lambda \nabla T \tag{11}$$

where q is the heat flux.
Using the Matlab function "$CGsolver$" the eq.3 is solved. Finally, the velocity is updated using eq.4, so after this step the velocity field is divergence-free.
Then, the time is updated by adding dt and the algorithm starts again.
The temperature field is plotted for each time step as we can see in the following plots.

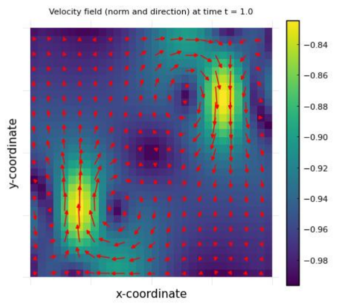

t=0s
t =1.0s
t =2.0 s
t =3.0s
t =4.0s
t =5.0s
t =7.0s
t =10s

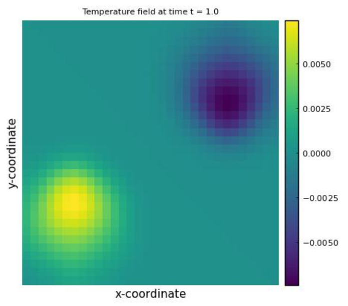


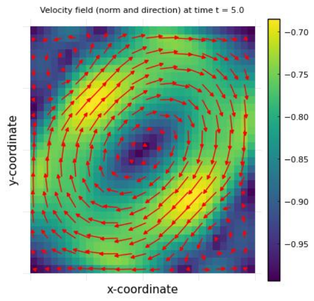
We can see that initially the two temperature peaks are localised in the upper right (coldest) and lower left (hottest) corners. Then, over time, the two peaks diffuse in a spiral fashion, and after $t = 100s$ the upper part of the plot is hotter and the lower part is colder, the temperature gradient is approximately constant with a direction from the bottom to the top of the plot, and the velocity field is convective.

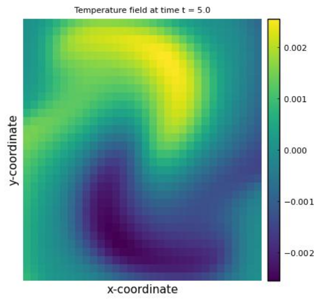
3 Julia

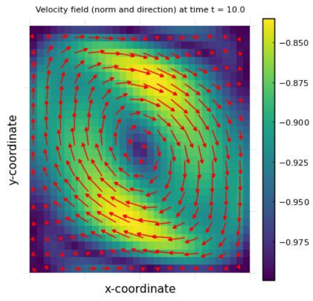
The main objective of this work is to adapt the Matlab code of 2D SIMPLE with the FTCS discretization of viscous and thermal fluxes developed during the numerical methods lecture into a Julia programming code and to compare the time performance between the two language types.
Julia is designed to be very fast, similar to *C* and *Fortran* performances. This is due to its just-in-time(JIT) compiler which optimizes code during execution. Julia should be faster than Matlab, particularly for non-vectorised code and complex numerical operations. Julia has also the advantage of being open-source and free, while Matlab requires a costly license, which can limit access. Furthermore, Julia offers more flexible support for parallelism and multithreading compared to Matlab, which requires additional toolboxes for advanced parallel computing capabilities.
The code has been converted to Julia programming language adding specific macros in order to speed up the running operation in the Julia improved version.
"@*inbounds*" macro is used to disable array bounds checking within a loop indeed Julia checks by default that the indices used to access array elements are within valid ranges to prevent runtime errors.
"*threads*" macro is used to enable multithreading in Julia, allowing code to run on multiple CPU cores simultaneously, useful for parallelizing loops and taking advantage of multiple-core processors to improve performance.
"@*views*" macro is used to create "views" of arrays rather than making full copies of subarrays, this can save memory and improve performance.
"@*elapsed*" macro is used to measure the time it takes to execute a block of code or a function call; it is used to measure the time it takes for each mesh grid size.
Lastly, the "@." macro is used to automatically broadcast every function and operator in an expression, making the code more concise and avoiding the need to explicitly use the dot operator multiple times.
In Julia, it is possible to choose packages from other programming languages. "*Pyplot*" is chosen to make the plots and visualisations, it provides a Julia interface to the "*matplotlib*" library in Python.
To further speed up the Julia improved version, some "for" cycles are introduced to fill the vectors in the various functions mentioned in the "Code" part, as it turns out that in the Julia programming language is faster.
In the following graphs we can see some snapshots of the temperature and velocity fields using Julia instead of Matlab.

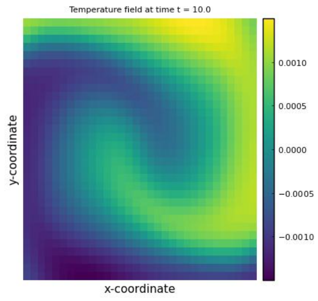

Velocity field (norm and direction) at time t = 1.0
y-coordinate
x-coordinate
−0.84
−0.86
−0.88
−0.90
−0.92
−0.94
−0.96
−0.98

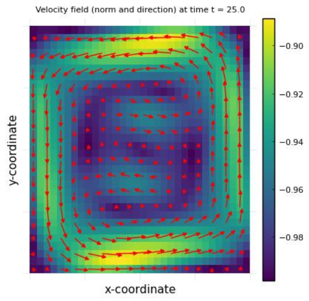

Temperature field at time t = 1.0
y-coordinate
x-coordinate
0.0050
0.0025
0.0000
−0.0025
−0.0050

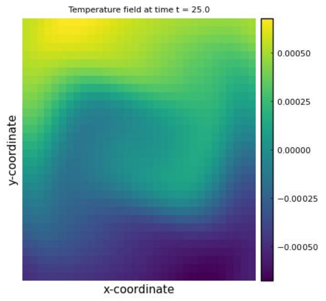

Velocity field (norm and direction) at time t = 5.0
y-coordinate
x-coordinate
−0.70
−0.75
−0.80
−0.85
−0.90
−0.95

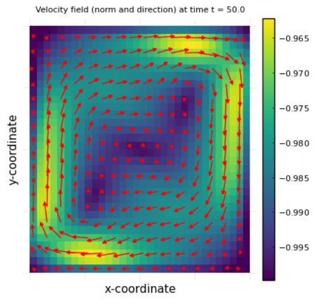

Temperature field at time t = 5.0
y-coordinate
x-coordinate
0.002
0.001
0.000
−0.001
−0.002

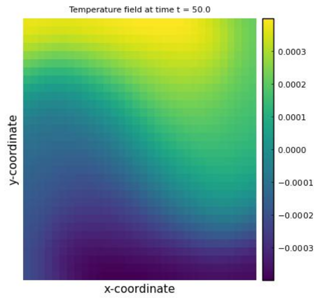

Velocity field (norm and direction) at time t = 10.0
y-coordinate
x-coordinate
−0.850
−0.875
−0.900
−0.925
−0.950
−0.975

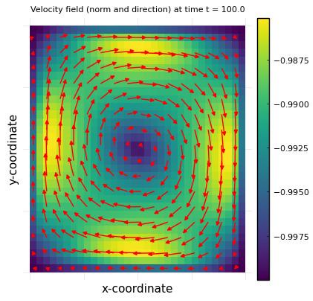

Temperature field at time t = 10.0
y-coordinate
x-coordinate
0.0010
0.0005
0.0000
−0.0005
−0.0010

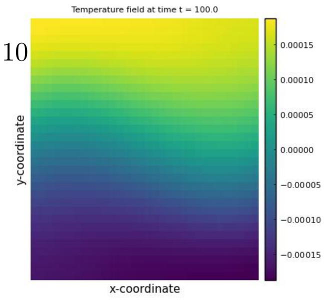

Velocity field (norm and direction) at time t = 25.0
y-coordinate
x-coordinate
−0.90
−0.92
−0.94
−0.96
−0.98


Temperature field at time t = 25.0
y-coordinate
x-coordinate
0.00050
0.00025
0.00000
−0.00025
−0.00050


Velocity field (norm and direction) at time t = 50.0
y-coordinate
x-coordinate
−0.965
−0.970
−0.975
−0.980
−0.985
−0.990
−0.995


Temperature field at time t = 50.0
y-coordinate
x-coordinate
0.0003
0.0002
0.0001
0.0000
−0.0001
−0.0002
−0.0003


Velocity field (norm and direction) at time t = 100.0
y-coordinate
x-coordinate
−0.9875
−0.9900
−0.9925
−0.9950
−0.9975


Temperature field at time t = 100.0
y-coordinate
x-coordinate
0.00015
0.00010
0.00005
0.00000
−0.00005
−0.00010
−0.00015

4 Results

The performance between the two programming languages is compared using different mesh grid sizes namely 50^2, 100^2, 150^2, 200^2, 250^2, 300^2, 350^2, 400^2, 450^2, 500^2 with an end time of $t_{end} = 0.5s$. The simulation is run 5 times for all code versions to get better statistics and the average of the time measurements is taken.

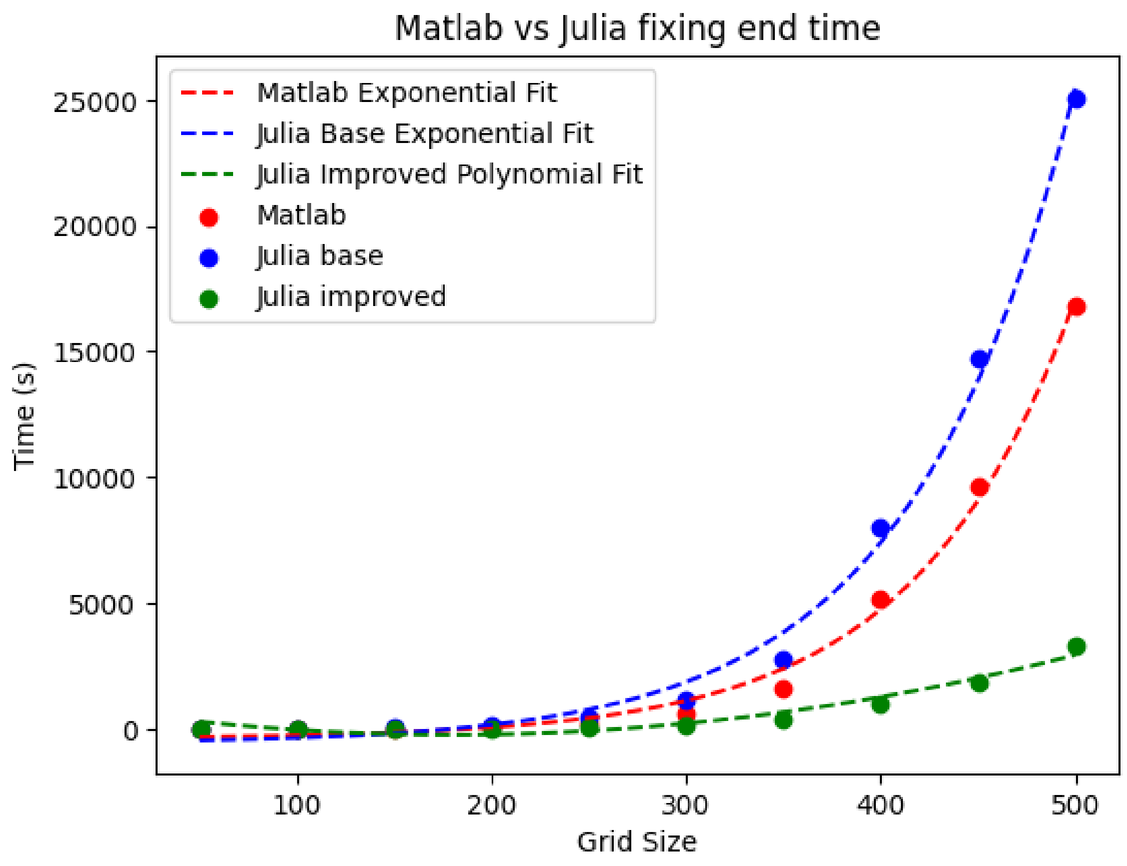


Figure 4: Computational time as a function of the grid size fixing t_{end}

We can see qualitatively that the Matlab and "Julia base" computation times increase exponentially in grid points. In contrast, the "Julia improved" time increases approximately quadratically in the range of grid points considered. From the graph, we can note that the difference between "Julia improved" and Matlab starts to become significant from approximately 300^2 grid points. Julia Base and Matlab both have exponential behaviour, but Julia's rises faster than Matlab's.

The difference between the three performances can be better appreciated by plotting the percentage between the Julia and Matlab computation times($\frac{t_{julia}}{t_{matlab}} \cdot 100$).

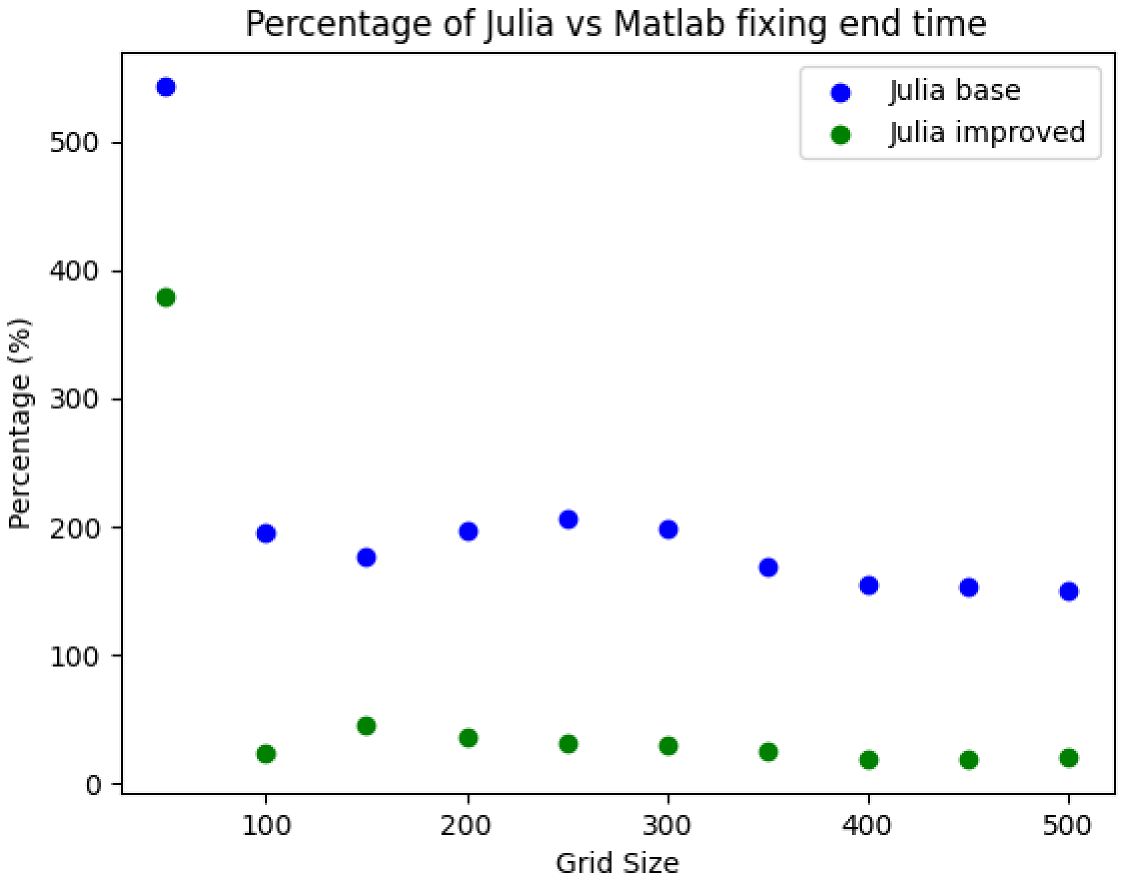


Figure 5: Computational time percentage of Julia vs Matlab as a function of the grid size

We observe that Matlab's performance is initially faster than Julia's improved version at 50^2 grid points. This is likely because Julia performs just-in-time (JIT) compilation the first time the code is run, which introduces some delay. Then, "Julia improved" performs better than Matlab, particularly we can see the percentage after 150^2 points decreases continuously, indeed at 500^2 grid points Julia takes $\approx 19.6\%$ of the Matlab computation time.
The "Julia base" version is always slower than Matlab, the first point at 50^2 grid size is particularly slower for the just-in-time compilation, and then at 500^2 grid points Julia takes $\approx 150\%$ of Matlab computation time.

In the following plot we can see the total computation time with the respective error:

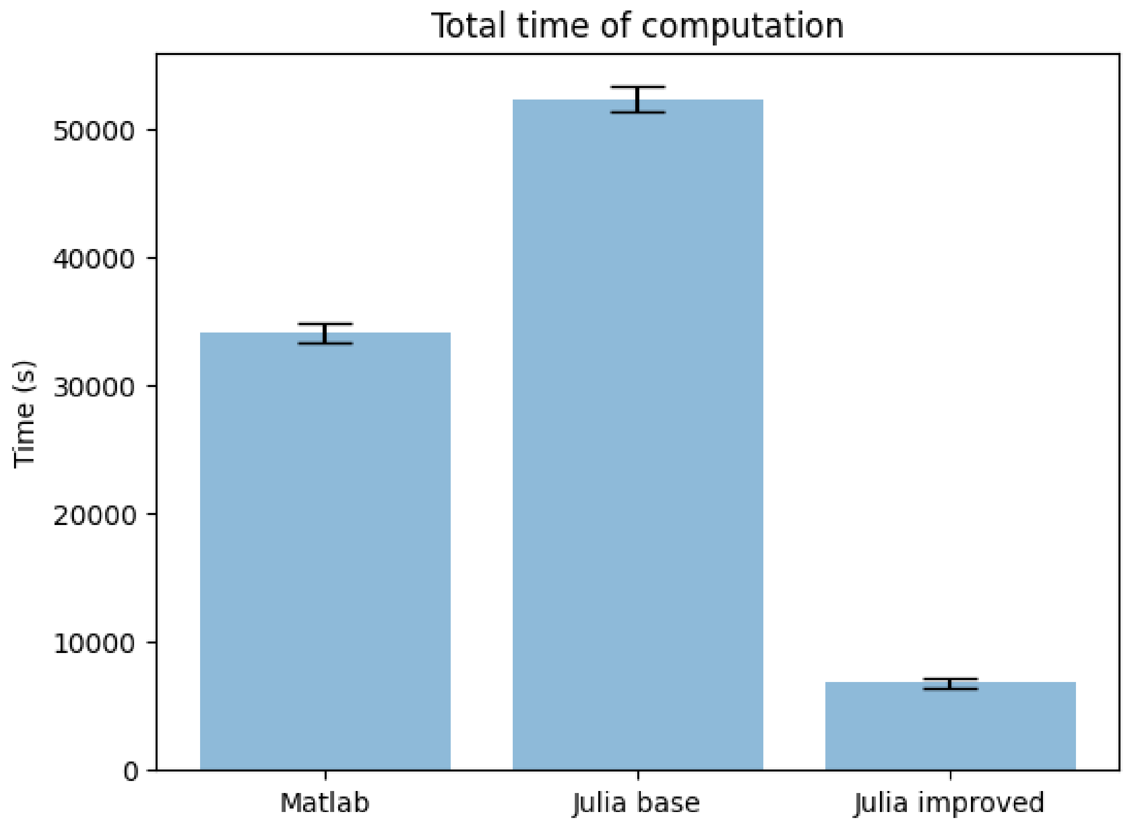


Figure 6: Total computation time comparison fixing t_{end}

Matlab takes $34153 \pm 728s$, "Julia base" takes $52327 \pm 1000s$: $153 \pm 23\%$ of Matlab computation time, "Julia Improved" takes $6801 \pm 391s$: $19.9 \pm 1.6\%$ of Matlab computation time.

A table summarises the data obtained:

Grid Size	**Matlab (s)**	**Julia Base (s)**	**Julia Improved (s)**	**Julia Base (%)**	**Julia Improved (%)**
50	0.60 ± 0.12	3.24 ± 0.08	2.27 ± 0.56	543 ± 108	380 ± 95
100	4.04 ± 0.88	7.90 ± 0.27	0.97 ± 0.22	196 ± 43	24.1 ± 5.8
150	21.7 ± 3.1	38.3 ± 1.2	9.96 ± 1.68	177 ± 27	45.9 ± 8.3
200	76.8 ± 8.2	152 ± 3	27.4 ± 2.8	197 ± 24	35.7 ± 4.6
250	225 ± 16	464 ± 12	70.2 ± 0.7	206 ± 17	31.2 ± 2.2
300	586 ± 55	1158 ± 17	170 ± 2	198 ± 19	29.1 ± 2.8
350	1625 ± 601	2736 ± 61	404 ± 18	168 ± 63	24.9 ± 9.2
400	5158 ± 96	7995 ± 183	970 ± 40	155 ± 4	18.8 ± 0.8
450	9638 ± 180	14693 ± 291	1816 ± 81	152 ± 3	18.9 ± 0.9
500	16817 ± 342	25066 ± 538	3313 ± 261	149 ± 2	19.7 ± 1.6

Table 1: Comparison of Computational Times for Julia Base, Julia Improved, and Matlab fixing t_{end}

The error in the time performance is calculated as the standard deviation, while the error in the percentage is calculated using the error propagation.

Another study is carried out to better understand the difference in time performance. The grid size is kept constant at 50^2 points while the end time is varied: $t_{end} = 1s$, $5s$, $10s$, $25s$, $50s$, $100s$, $250s$, $500s$, $1000s$, $2000s$, $4000s$.

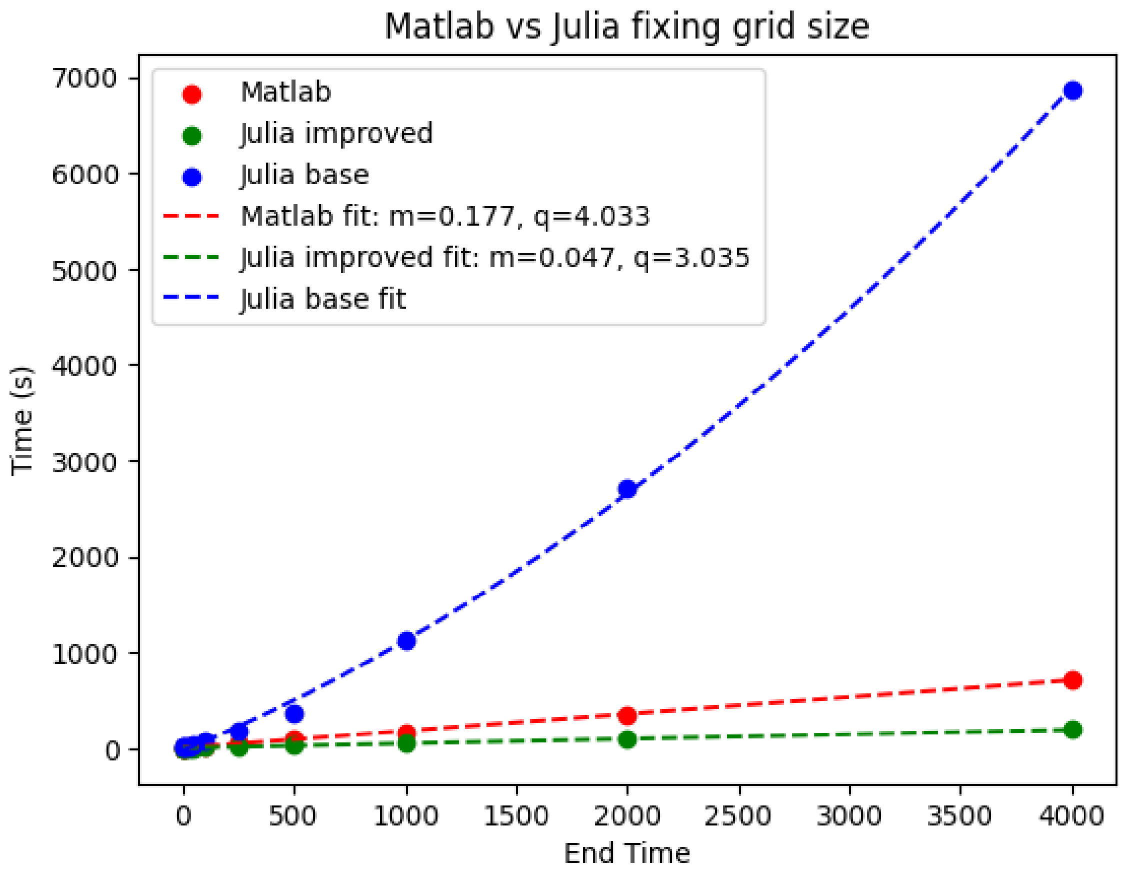


Figure 7: Computational time as a function of t_{end} fixing the grid size

As before, we can see from the plot that "Julia improved" performs better than Matlab, while "Julia base" performs worse. In the range $t_{end} = 1 - 4000s$, both "Julia improved" and Matlab have a qualitatively linear behaviour as a function of t_{end}, but the Julia linear fit has an angular coefficient of $m_{julia} = 0.047$ while $m_{matlab} = 0.177$, an order of magnitude difference, while the "Julia base" has a quadratic behaviour.
To better understand the difference in performance, as before, the percentage between Julia and Matlab computational time is plotted as a function of t_{end}.

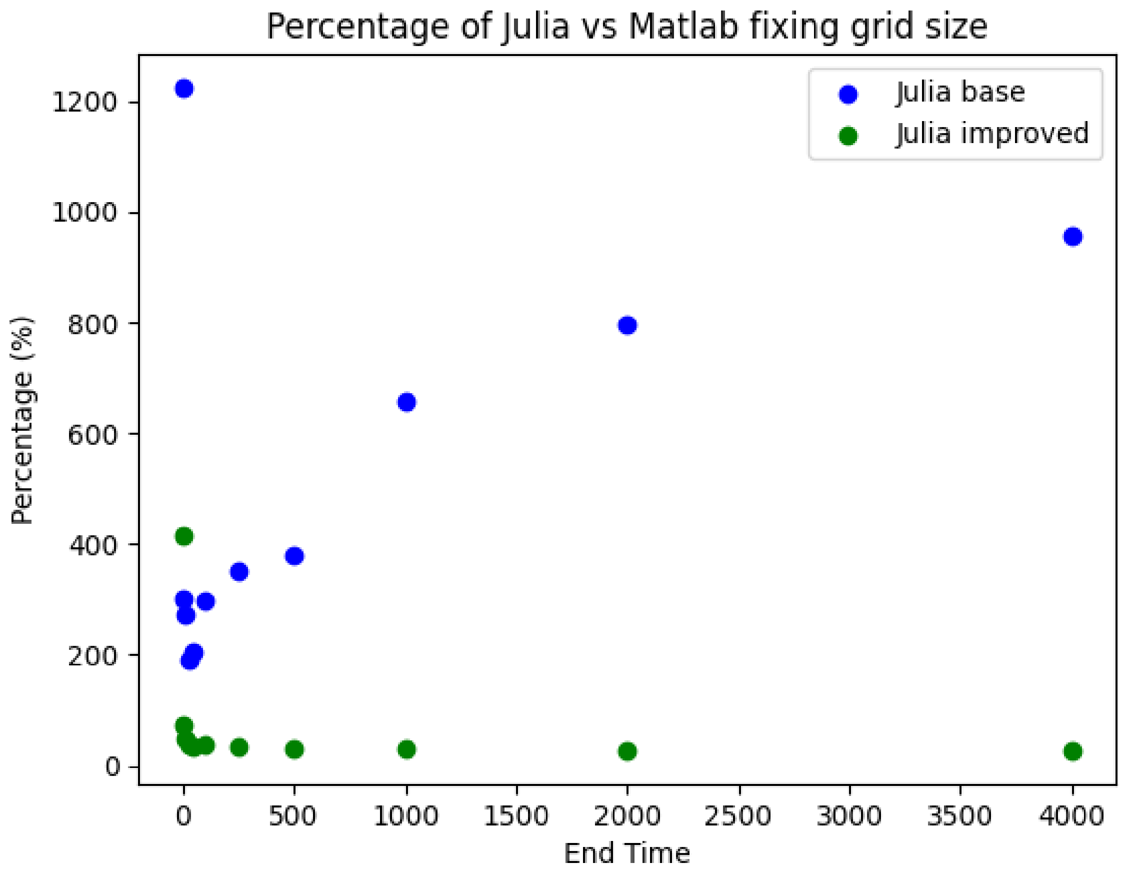


Figure 8: Computational time percentage of Julia versions vs Matlab as a function of t_{end}

We can see that Matlab is initially faster, as in the previous case, this is due to JIT compilation. Then, "Julia improved" becomes continuously faster than Matlab as t_{end} increases. At $t_{end} = 4000s$ the percentage is ≈ 27.10; while "Julia base" is always slower than Matlab, particularly at the first point which takes $\approx 1224\%$ of Matlab computation time, at the last point($t_{end} = 4000s$) it takes $\approx 955\%$ of Matlab computation time.

In the following plot we can see the total computation time with the respective error:

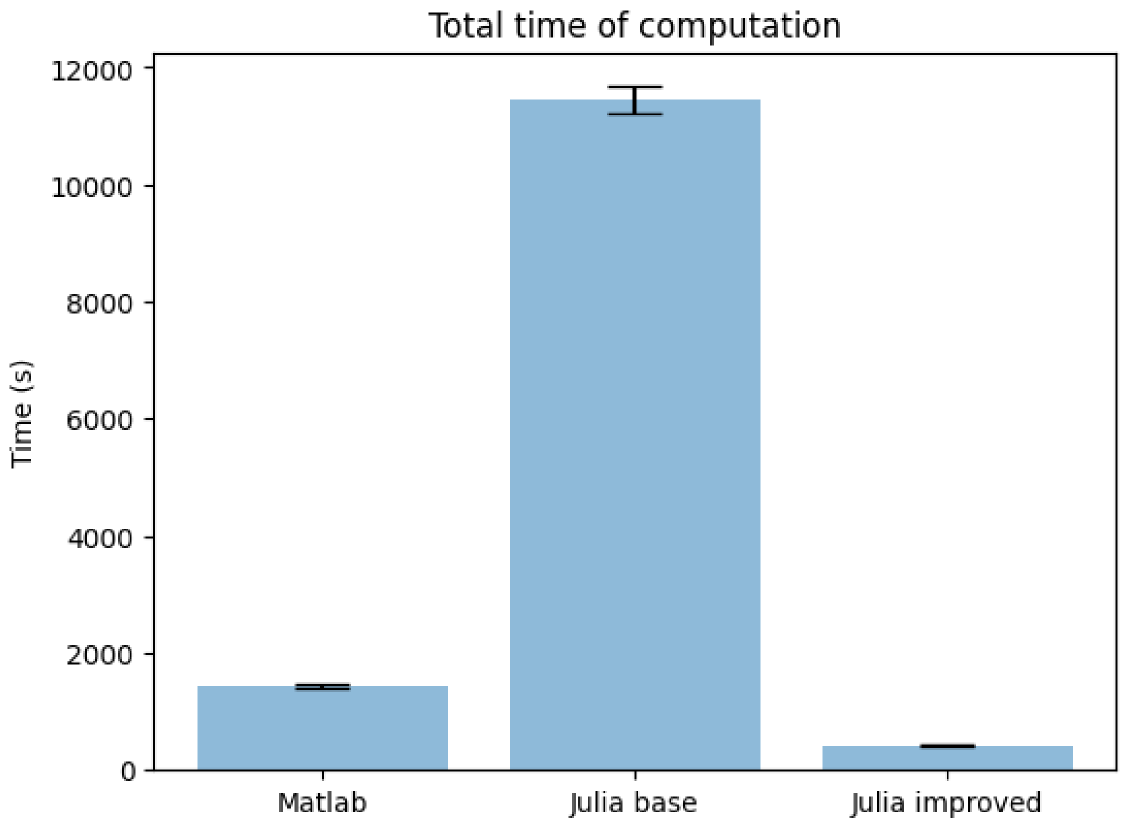


Figure 9: Total computation time comparison fixing the grid size

Matlab takes 1448 ± 31, "Julia base" takes $11443 \pm 230s$: $790 \pm 72\%$ of Matlab computation time, "Julia Improved" takes $410 \pm 18s$: $28.3 \pm 4.2\%$ of Matlab computation time.

A table summarises the data obtained.

t_end (s)	**Matlab (s)**	**Julia Improved (s)**	**Julia Base (s)**	**Julia Improved (%)**	**Julia Base (%)**
1	0.61 ± 0.04	2.55 ± 0.20	7.48 ± 0.50	416 ± 30	1240 ± 90
5	3.00 ± 0.20	2.16 ± 0.15	9.01 ± 0.60	72.1 ± 5.0	300 ± 25
10	6.27 ± 0.40	2.94 ± 0.25	17.0 ± 1.0	46.9 ± 4.0	272 ± 20
25	11.6 ± 0.8	4.39 ± 0.35	21.9 ± 1.5	37.9 ± 3.5	189 ± 15
50	18.2 ± 1.2	6.11 ± 0.50	37.1 ± 2.5	33.5 ± 3.0	204 ± 17
100	25.8 ± 1.7	9.47 ± 0.75	76.5 ± 5.0	36.7 ± 3.5	296 ± 25
250	50.1 ± 3.5	16.8 ± 1.3	177 ± 10	33.5 ± 3.5	352 ± 30
500	98.9 ± 7.0	28.4 ± 2.0	375 ± 25	28.7 ± 3.5	379 ± 40
1000	171 ± 12	49.2 ± 3.5	1130 ± 80	28.7 ± 4.0	658 ± 50
2000	342 ± 24	92.5 ± 7.0	2720 ± 180	27.1 ± 4.0	794 ± 70
4000	720 ± 50	195 ± 15	6880 ± 500	27.1 ± 5.0	955 ± 80

Table 2: Comparison of Computational Times for Julia Base, Julia Improved, and Matlab for Varying t_end

The error in the time performance is calculated as the standard deviation, while the error in the percentage is calculated using the error propagation.

5 Conclusions

The main objective of this work is to compare the computational performance between Matlab and Julia.
It is found that when increasing the grid size "Julia improved" performs far better than Matlab, in particular, "Julia improved" follows a quadratic behaviour while Matlab has an exponential behaviour increasing the grid size. At 500^2 grid size points Julia takes $\approx 19.6\%$ of Matlab computational time. "Julia base", instead, performs worse than Matlab: it has an exponential-like behaviour as Matlab but rises faster.
When the grid size is fixed and t_{end} is varied, "Julia improved" performs better compared to Matlab and "Julia base" worse, as before, but in this case, "Julia improved" and "Matlab" have a linear behaviour, while "Julia base" has a quadratic behaviour. Matlab linear fit has an angular coefficient one order of magnitude greater than "Julia improved": $m_{matlab} = 0.177$ while "Julia improved" has $m_{julia} = 0.047$. At $t_{end} = 4000s$ Julia takes $\approx 27.1\%$ of Matlab computational time.
We can conclude that increasing the complexity and the size of the simulation "Julia improved" performs better and better than Matlab, while "Julia base" is still slower than Matlab, probably because the original code was designed for the Matlab programming language instead of Julia.