A study of Jacobian-free Newton Krylov methods and Schur complement parameters for solving coupled systems.

by

Floris Brulleman

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Thursday August 22, 2019 at 3:00 PM.

Student number:	4553667	
Project duration:	September 1, 2018 – August 22, 2018	
Supervisors:	Dr. ir. D. Lathouwers,	TU Delft
	Prof. dr. ir. C. Vuik,	TU Delft
	M. Tiberga, MSc,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft Delft University of Technology

Abstract

Numerical models are paramount in describing the complex physical world around us. They are often based on non-linear functions, which have to be solved in order to run a simulation. The first goal of this thesis is to compare the Jacobian-free Newton-Krylov method against the regular Newton-Raphson method for solving non-linear equations. When doing this, preconditioning was only used for the Newton-Raphson method. As a second goal, the optimal parameters for solving a coupled system with the Schur complement method are determined. For computational purposes, these analyses are performed using PETSc.

The comparison between JFNK and Newton-Raphson were performed by solving a heat equation. The physical system which was analyzed is a one-dimensional radiating rod, with a heterogeneous thermal conductivity and Dirichlet boundary conditions. Firstly this rod was modelled as one single system. For this case, it was found that Newton-Raphson outperformed the JFNK method by a factor of 5-1700, depending on the type of preconditioning used. Secondly, the rod was modelled as a coupled system by solving two parts of the rod separately. In this case, it was found that the JFNK method outperformed the preconditioned Newton-Raphson method by a factor of 9.1 ± 0.3. It was concluded that the JFNK is only favorable over regular Newton-Raphson for coupled systems.

To achieve the second goal, an incompressible Navier-Stokes coupled system was solved using the Schur complement method. The coupled system originated from incompressible flow in a back-step pipe, using a finite element method. For the Schur complement parameters, it was shown that using an approximation of the Schur complement offered a significant increase in efficiency. The momentum and pressure subsystems were analyzed separately. The momentum subsystem performed best with a relative tolerance of $\tau_r = 10^{-4.5}$, while the pressure subsystem performed best with a tolerance of $\tau_r = 10^{-2}$. With these parameters, the Schur complement method had the same computational time as the pressure-correction method, which was used as a benchmark.

For future research, there are three main recommendations. Firstly, to use preconditioning for the JFNK method. This was not done in this thesis because of technical constraints, but it is theoretically possible. Secondly, if the JFNK method is preconditioned, it could be used for the incompressible Navier-Stokes simulation. Finally, there are combinations of parameters which were not tested for the Schur complement method. Trying out more combinations might result in finding an even more optimized method.

Contents

List of Figures

List of Tables

1 Introduction

Ever since the invention of the computer, numerical simulations have become increasingly important. They allow the simulation of complex situations, without the cost and problems that come with doing the actual experiment. These simulations are advantageous in many fields. They allow car companies to model their cars and see how efficient they will be without ever constructing the car [1]. They are even used by banks to model the financial market and try to predict what is going to happen [2]. There are many more branches where numerical models have become crucial in the functioning of the entire industry, which makes the accuracy of these models essential. In general, a higher accuracy requires more calculations, but more calculations take more time, which is often the limiting factor. It is possible to reduce the computational time by using supercomputers, but this is very expensive and sometimes insufficient. In practice, it is common to try and make numerical algorithms as efficient as possible. Optimizing an algorithm makes it possible for the same model to be simulated in less time, avoiding the need to invest in more computing power.

Numerical models are mostly used to simulate complex phenomena, so they are often governed by a non-linear function. An algorithm widely used to find the solution for any non-linear differentiable function is the Newton-Raphson algorithm. This method uses the Jacobian of a function to iterate towards its root. If the Jacobian is known, Newton-Raphson is generally easy to implement and is usually very fast in finding the right solution. However, the Jacobian is usually not available, which means the regular Newton-Raphson method can not be used. In these cases, the Jacobian-free Newton-Krylov (JFNK) method is a possible alternative since it approximates the Jacobian. The goal in the first part of this thesis is to compare the 'regular' Newton-Raphson method against the JFNK method. Many problems can be solved using the Newton-Raphson method, but there are two main types. The first type is the single-physics problem, which means that the model only focusses on one system. An example of this would be the velocity of an object as it falls, or more complexly: the temperature of a continuously cooled, running engine. For the second type, multi-physics problems (coupled systems), the model focusses on multiple systems simultaneously. The multi-physics problems are generally more complex to solve and are encountered, for example, in weather models. Weather models need to keep track of how clouds form, the velocity of the wind, interaction with land, et cetera. All of these systems interact with another, which makes these types of problems harder to solve.

First, both methods are tested on a single-physics model for the heat-flow in a 1D rod. The simplicity of this problem made it a useful test problem, which could be used to compare and optimize the regular Newton-Raphson and JFNK algorithms. Once these algorithms were both optimized, the 1D model rod was split up into two parts. The parts were then solved separately, making the system a coupled system. For the split rod, the JFNK and regular Newton-Raphson methods were then compared again to see how well they performed on a coupled system.

The second part of this thesis focussed on finding an optimal method for solving the incompressible Navier-stokes equations. The flow of an incompressible liquid in a backstep pipe was chosen because it is a widely used benchmark [3]. The liquid flow was solved as a coupled system because the incompressible Navier-Stokes equations involve both pressure and momentum systems. These incompressible Navier-Stokes problems play a large part in the simulation of new types of reactors at the Reactor Institute Delft. By optimizing the numerical methods for the incompressible flow in a pipe problem, potentially efficient methods for the more significant simulations were investigated.

The layout of this thesis is as follows: First, the basics of what Newton-Raphson is and how it works are discussed in Chapter 2, alongside the different methods that can be implemented inside Newton-Raphson. Chapters 3 & 4 focus on how well the JFNK and regular Newton-Raphson methods perform for the single-physics and multi-physics 1D rod problem, respectively. In Chapter 5, different parameters for the Schur complement method are tested for the linearized incompressible Navier-Stokes equation inside a pipe. Finally, all conclusions and recommendations are presented in Chapter 6.

This thesis is part of the double bachelor in physics and mathematics at the University of Technology Delft.

2

Newton methods for non-linear systems

In every field of physics, there are functions which can not be solved analytically. This means that finding x for a function f, such that $f(x) = b$ requires numerical methods. One of the fastest numerical methods available is the Newton-Raphson algorithm. The Newton-Raphson algorithm involves many processes, which will be explained and described in this chapter.

2.1. Newton-Raphson

The Newton-Raphson method is an iterative numerical method, which finds the root of a differentiable vector function $\mathbf{F}(\mathbf{x})$. Since Newton-Raphson is an iterative method, it does not find the exact value for the root $\mathbf{x}$, but instead, it returns an approximation of its value. Consequently, the solution will always be less precise than an analytical solution, but finding a root analytically may not always be feasible. For example, the root of a non-trivial polynomial of order 5 or higher is impossible to find analytically [4]. For these high order polynomials, Newton-Raphson can be used to find the roots.

The first step in applying Newton-Raphson is providing a starting point for the iterative sequence. Usually, the starting point is guessed such that it is close to a root. The next point is then calculated using

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \mathbf{J}_{\mathbf{F}}^{-1}(\mathbf{x}_i) \cdot \mathbf{F}(\mathbf{x_i}), \tag{2.1}$$

where $\mathbf{F}(\mathbf{x_i})$ is the value of the function at the i'th iteration, and $J_F(\mathbf{x_i})$ is the Jacobian at the i'th iteration.

For example, when using Newton-Raphson to find the root of $f(x) = x^2$, with initial guess $x_0 = 2$, the process of finding an approximation for the root looks like Figure 2.1. Once the approximated root x_n is close enough to an actual root, the sequence will converge to the root at a quadratic rate [5]. This implies that the error between the approximation and the exact value of the root, converges at least as fast as:

$$||\mathbf{x} - \mathbf{x_{i+1}}|| \propto ||\mathbf{x} - \mathbf{x_i}||^2, \tag{2.2}$$

where $\mathbf{x}$ is the exact value of the root. To save time, the Newton-Raphson method is stopped when the error in the approximation of the root satifies certain conditions. These conditions are otherwise known as stop criteria, and two criteria are used in this thesis. The first criterion is the relative step stop criterion given by

$$\frac{||\mathbf{x_i} - \mathbf{x_{i-1}}||}{||\mathbf{x_0}||} < \tau_S, \tag{2.3}$$

where τ_S is the relative step tolerance.
This criterion is very useful because it scales with the size of the system, so the error of the approximated root is always proportional to the actual root. The downside of the relative step criterion is that it does not work when the initial guess $\mathbf{x_0}$ is zero. The second criterion used in this thesis is the relative function stop criterion:

$$\frac{||\mathbf{F}(\mathbf{x_i}) - \mathbf{F}(\mathbf{x_{i-1}})||}{||\mathbf{F}(\mathbf{x_0})||} < \tau_F, \tag{2.4}$$

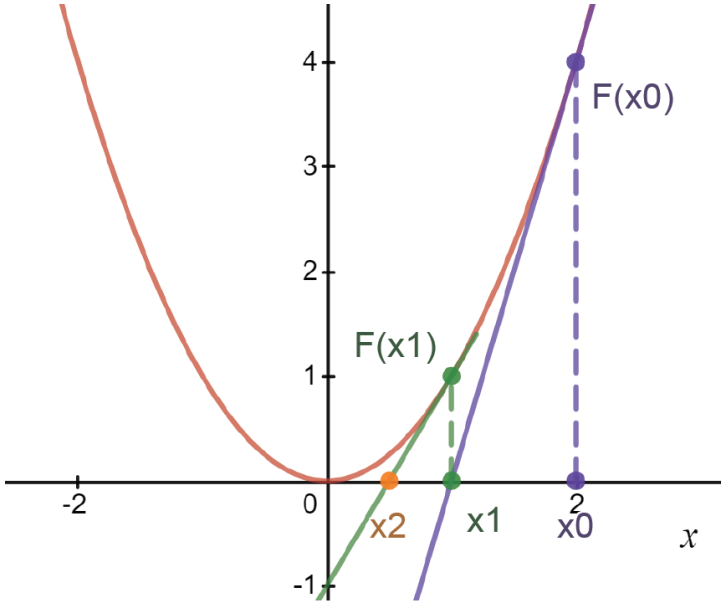


Figure 2.1: Illustation of the Newton-Raphson method approximating the root of x^2. Shows the first two iterations.

where τ_F is the relative function tolerance. Similar to the relative step stop criterion, the relative function stop criterion has as main advantage that the error in the system is always proportional to the size of the system.

In order to calculate the next Newton-Raphson step using Equation 2.1, it is required to know the inverted Jacobian $J_{\mathbf{F}}^{-1}(\mathbf{x}_i)$. In the one-dimensional case, for a differentiable function $F(x)$, the inverted Jacobian is given by

$$(J_{\mathbf{F}}(x))^{-1} = \frac{1}{\frac{\partial F}{\partial x}(x)}. \tag{2.5}$$

Because the right hand side of Equation 2.5 is a scalar value, which makes the inversion a simple matter. However, if $\mathbf{F}(\mathbf{x})$ does not map a scalar domain onto a scalar domain, the Jacobian becomes a matrix. Given a function $\mathbf{F}(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^m$, the Jacobian is given by

$$\mathbf{J_F} = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \cdots & \frac{\partial F_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x_1} & \frac{\partial F_m}{\partial x_2} & \cdots & \frac{\partial F_m}{\partial x_n} \end{bmatrix}. \tag{2.6}$$

This Jacobian could be inverted explicitly, and plugged into Equation 2.1 to find the next Newton-Raphson step. Although this way of finding the next step is theoretically possible, it requires storing the inverted Jacobian explicitly. This is very memory expensive, which makes that the algorithm scale poorly. It is important to note here that the Jacobian $\mathbf{J_F}$ is sparse in general, but the inverse $\mathbf{J_F}^{-1}$ is usually dense. Meaning that in some cases, storing the Jacobian scales well, while storing the inverse Jacobian scales poorly. In order to avoid storing the explicit Jacobian, Equation 2.1 is rewritten to

$$\mathbf{J_F}(\mathbf{x_i}) \cdot \delta\mathbf{x_{i+1}} = -\mathbf{F}(\mathbf{x_i}), \tag{2.7}$$

where $\delta\mathbf{x_{i+1}}$ is defined as

$$\delta\mathbf{x_{i+1}} = \mathbf{x_{i+1}} - \mathbf{x_i}. \tag{2.8}$$

Once $\delta\mathbf{x}_{i+1}$ is known, the next Newton-Raphson step is calculated with

$$\mathbf{x_{i+1}} = \mathbf{x_i} + \delta\mathbf{x_{i+1}}. \tag{2.9}$$

To find $\delta\mathbf{x_{i+1}}$ in Equation 2.7, a linear solver is required, of which there are two main types. The first type is the direct solver which returns the exact value of $\delta\mathbf{x_{i+1}}$. The second type is the iterative linear solver, which

returns an approximation of $\delta\mathbf{x_i}$. One of the most popular direct solvers is the LU decomposition method. All direct solver methods have some serious drawbacks, which cause issues with the scalability of the system. The first disadvantage is that it needs to store an $N \times N$ matrix to solve a $N \times N$ system, which can take up much memory. The second drawback is that the LU decomposition method (or any other direct solver) has a complexity of at least $\mathcal{O}(N^3)$ [6]. For large systems, direct solvers require a huge amount of computational power to solve a problem in any reasonable time frame. Because the systems in this thesis require good scalability, the direct solver methods were ruled out.

Since the direct solvers could not be used, iterative ones were chosen as the main type of linear solvers in this thesis. All the iterative solvers used for this work are discussed in depth in the next Subsection.

2.2. Krylov methods

The idea behind an iterative solver is quite similar to the Newton-Raphson method itself. They start with an initial guess and then use the properties of the system to get a better approximation in the next step. There are many different iterative linear solvers available, but only Krylov methods were used in this thesis. The Krylov methods were chosen because they are among the fastest iterative solvers available [7]. Because of their iterative nature, they do not have the $\mathcal{O}(N^3)$ time complexity of a direct solver. This makes Krylov methods very advantageous for ensuring the scalability of a problem.

Every Krylov method is by definition based on the Krylov subspace

$$\mathbf{K} = \{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \mathbf{A}^2\mathbf{r}_0, \ldots, \mathbf{A}^{N-1}\mathbf{r}_0\}, \tag{2.10}$$

where, $\mathbf{A}$ is the $N \times N$ matrix defining the linear system, $\mathbf{w_0}$ is the initial linear solution guess, $\mathbf{b}$ is the RHS of the linear system, and $\mathbf{r}_0$ is defined as $\mathbf{b} - \mathbf{A}\mathbf{w}_0$. The goal of all Krylov methods is to minimize the residual $\mathbf{r}$, which corresponds to solving the linear system.

There are many Krylov methods, but only two were used and discussed in this thesis; the conjugate gradient (CG) and generalized minimal residual (GMRES) methods.

2.2.1. Conjugate Gradient (CG)

Conjugate Gradient is an extremely effective numerical iterative solver, which always converges for symmetric and positive definite systems [8]. The iterative CG method is based on minimizing

$$||\mathbf{b} - \mathbf{A}\mathbf{w}||_2, \tag{2.11}$$

where $\mathbf{A}$ is a known matrix, $\mathbf{b}$ a known vector, and $\mathbf{w}$ the unkown vector. The complete CG algorithm is given in Algorithm 1.

It is not possible to perfectly predict how fast the CG method converges for a general system, but there are some bounds on the convergence. When solving a $N \times N$ system, the CG method has a complexity of $\mathcal{O}(N)$ [8]. Moreover, it always returns the exact solution within N iterations. Note that this does not mean that it finds the exact solution with a complexity of $\mathcal{O}(N)$, since iterations take more time when N increases. Together, these bounds promise much better scalability than the $\mathcal{O}(N^3)$ complexity of the LU decomposition method. With the downside being that the solution is not exact. In this thesis, the CG algorithm will only be used and not investigated. For further reading on how the basics of CG work, the reader is referred to [8].

2.2.2. Generalized Minimal Residual (GMRES)

The Generalized Minimal Residual method is an extension on the methods used in the CG algorithm [8]. GMRES is used to solve unsymmetrical systems since CG does not work for these types of problems. GMRES first orthogonalizes the Krylov subspace using Gramm-Schmidt. Because of this, GMRES will be less efficient compared to CG in most cases. So whenever the system is symmetric positive definite the CG method is chosen and if not, then GMRES is used.

The GMRES algorithm is given in Algorithm 2.

Data: System $\mathbf{A}$, initial guess $\mathbf{w}^{(0)}$, rhs of the system $\mathbf{b}$, and preconditioning matrix $\mathbf{M}$.
Result: Approximative solution $\mathbf{w}^{(i)}$

Compute $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{w}^{(0)}$
for $i = 1, 2, \ldots$ **do**
  solve $\mathbf{M}\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$
  $\rho_{i-1} = \langle \mathbf{r}^{(i-1)}, \mathbf{z}^{(i-1)} \rangle$
  if $i = 1$ **then**
    $\mathbf{p}^{(1)} = \mathbf{z}^{(0)}$
  else
    $\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$
    $\mathbf{p}^{(i)} = \mathbf{z}^{(i-1)} + \beta_{i-1}\mathbf{p}^{(i-1)}$
  end
  $\mathbf{q}^{(i)} = A\mathbf{p}^{(i)}$
  $\alpha_i = \rho_{i-1} / \langle \mathbf{p}^{(i)}, \mathbf{q}^{(i)} \rangle$
  $\mathbf{w}^{(i)} = \mathbf{w}^{(i-1)} + \alpha_i \mathbf{p}^{(i)}$
  $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha_i \mathbf{q}^{(i)}$

  if *converged* **then**
    return $\mathbf{w}^{(i)}$
  end
end

Algorithm 1: CG algorithm from [8], with preconditioning matrix $\mathbf{M}$ which is discussed later.

2.3. Preconditioning

An iterative linear solver can solve some systems more efficiently than other systems. This is due to the spectral properties of the matrix $\mathbf{A}$, which defines the system. A widely used measure for how well iterative solvers work for a certain system is the condition number:

$$\kappa(\mathbf{A}) = \frac{|\lambda_{max}(\mathbf{A})|}{|\lambda_{min}(\mathbf{A})|}, \tag{2.12}$$

where $\lambda(\mathbf{A})$ is an eigenvalue of $\mathbf{A}$ [5]. In general, a lower condition number results in the iterative linear solver being more efficient for this system. Therefore, if $\kappa(\mathbf{MA}) < \kappa(\mathbf{A})$ for some preconditioning matrix $\mathbf{M}$, the convergence can be sped up significantly by multiplying both sides of the system by $\mathbf{M}$, resulting in the preconditioned system:

$$\mathbf{MAx} = \mathbf{Mb}. \tag{2.13}$$

This preconditioned system has the same solution as the original system $\mathbf{Ax} = \mathbf{b}$, but has a lower condition number. Preconditioning plays a crucial role in ensuring that the iterative linear solvers find a solution with high efficiency.

In practice, algorithms are used to find a good preconditioning matrix for a certain system. Some of these preconditioning algorithms, such as symmetric successive over-relaxation (SSOR), take up almost no computing power to create and then apply the preconditioning matrix [8]. For other algorithms, such as the Incomplete Factorization preconditioners, this can take a significant portion of the total computational power used [8]. With the latter type of preconditioner algorithm, the problem becomes finding an optimum between the cost of making the preconditioner and the gain of using it.

Multiple preconditioners are used in this thesis, mainly: Incomplete Cholesky (icc), LU, Jacobi, and the FIELDSPLIT preconditioner. The preconditioner algorithms used in this thesis are all provided by PETSc. For further reading on the preconditioners, the reader is referred to the PETSc user manual [9].

Data: System **A**, initial guess $\mathbf{w}^{(0)}$, rhs of the system **b**, and preconditioning matrix **M**.
Result: Approximative solution $\mathbf{w}^{(i)}$

for *j = 1, ..., m* **do**
  solve $\mathbf{Mr} = \mathbf{b} - \mathbf{Aw}^{(0)}$
  $\mathbf{v}^{(1)} = \mathbf{r}/||\mathbf{r}||_2$
  $s = ||\mathbf{r}||_2 e_1$
  for *i = 1, ..., m* **do**
    solve $M\mathbf{q} = A\mathbf{v}^{(i)}$
    for *k = 1, ..., i* **do**
      $h_{k,i} = \langle \mathbf{q}, \mathbf{v}^{(k)} \rangle$
      $\mathbf{q} = \mathbf{q} - h_{k,i}\mathbf{v}^{(k)}$
    end
    $h_{i+1,i} = ||\mathbf{q}||_2$
    $\mathbf{v}^{(i+1)} = \mathbf{q}/h_{i+1,i}$
    apply $J_1, ..., J_{i-1}$ on $(h_{i,i}, ..., h_{i+1,i})$
    construct J_i, acting on ith and $(i+1)$st component
    of $h_{.,j}$, such that $(i+1)$st component of $J_i h_{.,i}$ is 0.
    $s = J_i s$
    if *$s(i+1)$ is small enough* **then**
      UPDATE($\hat{\mathbf{w}}$,i)
      break
    end
  end
  UPDATE($\hat{\mathbf{w}}$,i)
end

In this scheme UPDATE($\hat{\mathbf{w}}$,i)
replaces the following computations:

Compute **y** as the solution of $H\mathbf{y} = \hat{\mathbf{s}}$, in which
the upper $i \times i$ triangle part of H has $h_{i,j}$ as
its elements (in least square sense if H is singular),
$\hat{\mathbf{s}}$ represents the first i components of **s**.
$\hat{\mathbf{w}} = \mathbf{w}^{(0)} + y_1\mathbf{v}^{(1)} + y_2\mathbf{v}^{(2)} + ... + y_i\mathbf{v}^{(i)}$
$\mathbf{s}^{(i+1)} = ||\mathbf{b} - A\hat{\mathbf{w}}||_2$
if $\hat{\mathbf{w}}$ is an accurate enough approximation then **quit**
else $\mathbf{w}^{(0)} = \hat{\mathbf{w}}$

Algorithm 2: Restarted GMRES(m) algorithm from [8], where M is the preconditioning matrix which will be discussed later.

2.4. Jacobian-free Newton Krylov

So far in this thesis, it has been assumed that because $\mathbf{F}(\mathbf{x})$ is a differentiable function, that it is always possible to calculate the Jacobian $\mathbf{J_F}$. In reality, it is often not possible to find $\mathbf{J_F}$, and an alternative to the exact Jacobian must be found in order to use Newton-Raphson.

This is where Krylov methods become extremely effective. They only need the Krylov sub-space (Equation 2.10), which is solely based on the Jacobian-vector multiplication $\mathbf{J_F} \cdot \mathbf{r_0}$. Therefore, if $\mathbf{J_F} \cdot \mathbf{v}$ can be computed for a general vector $\mathbf{v}$, the whole Krylov sub-space can be computed by using

$$\mathbf{J_F}^{n+1} \cdot \mathbf{r_0} = \mathbf{J_F} \cdot (\mathbf{J_F}^{n} \cdot \mathbf{r_0}). \tag{2.14}$$

Here, the $\mathbf{J_F}^{n} \cdot \mathbf{r_0}$ term is always a vector, so $\mathbf{J_F}^{n+1} \cdot \mathbf{r_0}$ will always be a vector as well. Since the matrix in Equation 2.14 is a Jacobian, the matrix-vector product can be approximated by [10]

$$\mathbf{J_F}(\mathbf{x}) \cdot \mathbf{v} \approx \frac{\mathbf{F}(\mathbf{x} + \epsilon \mathbf{v}) - \mathbf{F}(\mathbf{x})}{\epsilon}, \tag{2.15}$$

where $\mathbf{F}(\mathbf{x})$ is the function for which the root is computed, $\mathbf{v}$ is some vector, and ϵ is a sufficiently small number (e.g., $1.5 * 10^{-8}$) [11]. Combining Equations 2.14 and 2.15 gives

$$(\mathbf{J_F}(\mathbf{x}))^{n+1} \cdot \mathbf{r_0} \approx \frac{\mathbf{F}(\mathbf{x} + \epsilon \mathbf{J}^{n} \cdot \mathbf{r_0}) - \mathbf{F}(\mathbf{x})}{\epsilon}, \tag{2.16}$$

which can be used to compute the entire Krylov sub-space, without explicitly knowing the Jacobian. Using this approximation in Newton-Raphson is known as the Jacobian-free Newton-Krylov method (JFNK).

2.5. Steady and transient solutions

Many fields of physics, are governed by a time-dependent differential-equation. These types of equations describe the way a certain process evolves, and due to conservation laws they are generally of form

$$\frac{\partial \mathbf{q}}{\partial t} = \nabla \cdot \mathbf{H}(\mathbf{q}, t) + \mathbf{S}(t), \tag{2.17}$$

where $\mathbf{q}$ is some property of the system (e.g., temperature, energy, or velocity), $\mathbf{H}$ is some suitable function, $\mathbf{S}$ is the source term, and t represents the time. The function $\mathbf{H}$ may be dependent on even more variables, such as position, pressure, volume, or electrical current. For now, only $\mathbf{q}$ and t are taken into account. To simplify Equation 2.17, it is rewritten to

$$\frac{\partial \mathbf{q}}{\partial t} = \mathbf{F}(\mathbf{q}, t), \tag{2.18}$$

where $\mathbf{F}$ equals $\nabla \cdot \mathbf{H}(\mathbf{q}, t) + \mathbf{S}(t)$.

There are two categories for Equation 2.18, the steady state (SS) problem, and the transient problem.

2.5.1. Steady state

The steady state problem occurs if all processes in a system are not changing over time [12]. Therefore, the LHS of Equation 2.18 is zero, and thus $\mathbf{F}$ has to be zero as well. Steady state only occurs when all processes in the system cancel out (e.g., flow in = flow out, energy gained = energy lost, etc). However, this does not mean everything has to be static. Laminar flow in a pipe can be a steady state process, even though the fluid moves

To solve a steady state problem, a solution needs to be found such that $\mathbf{F}(\mathbf{q}, t) = 0$. In other words; the solution is a root of $\mathbf{F}$. Newton-Raphson can be used to find this root, alongside with the other methods discussed in Sections 2.1-2.4.

2.5.2. Transient

The transient case occurs whenever the LHS of Equation 2.18 is not equal to 0. The variable $\mathbf{q}$ no longer has to be a constant over time, and the problem becomes a truly time-dependent differential equation. To solve the transient differential equation, Equation 2.18 is rewritten into

$$\mathbf{q}(t) = \int_{t_0}^{t} \mathbf{F}(\mathbf{q}(t'), t')\, dt' + \mathbf{q}(t_0), \tag{2.19}$$

where t_0 is the initial time, t is the time, and t' a dummy integration variable. To solve Equation 2.19, time integration methods are used. In this thesis only backward Euler is discussed and used. Backward Euler is a very popular, implicit time integration method. It uses the solution at the next time-step to compute the next time step. Since it is an implicit algorithm, it is unconditionally stable which means that any error in the initial conditions does not cause the solution to diverge. Furthermore, Backwards Euler has an accuracy of $\mathcal{O}(h)$, where h is time-step used to discretize the time. Backward Euler is based on approximating Equation 2.18 as

$$\mathbf{q}^{n+1} \approx \mathbf{q}^n + h \cdot \mathbf{F}(\mathbf{q}^{n+1}, t_{n+1}) \tag{2.20}$$

where t_n represents the time at $t_0 + h * n$, and $\mathbf{q}^n$ denotes $\mathbf{q}(t_n)$ [13]. Using Equation 2.20, the value of $\mathbf{q}^{n+1}$ can not be calculated directly, so numerical solvers such as Newton-Raphson are required to find $\mathbf{q}^{n+1}$. In order to use Newton-Raphson, the Backward Euler equation is rewritten to a form where $\mathbf{q}^{n+1}$ is the root of some vector function $\mathbf{G}$:

$$\mathbf{G}(\mathbf{q}^{n+1}) = \mathbf{q}^{n+1} - \mathbf{q}^n - h\mathbf{F}(\mathbf{q}^{n+1}, t_{n+1}) = 0. \tag{2.21}$$

For this system $\mathbf{G}(\mathbf{q}^{\mathbf{n+1}})$, the Newton-Raphson algorithm is given by

$$\mathbf{J_G} \cdot \delta\mathbf{q}_{i+1}^{n+1} = -\mathbf{q}_i^{n+1} + \mathbf{q}^n + h\mathbf{F}(\mathbf{q}_i^{n+1}, t_{n+1}), \tag{2.22}$$

where $\delta\mathbf{q}_{i+1}^{n+1}$ is defined as:

$$\delta\mathbf{q}_{i+1}^{n+1} = \mathbf{q}_{i+1}^{n+1} - \mathbf{q}_i^{n+1}. \tag{2.23}$$

It is imporant to note that $\mathbf{q}^n$ and t_{n+1} are all known constants in Equation 2.22, so the Jacobian of G is purely based on $\mathbf{q}_i^{n+1}$. The Jacobian $J_\mathbf{G}$ can then be written as

$$\mathbf{J_G} = \mathbf{I} - h\mathbf{J_F}, \tag{2.24}$$

where $\mathbf{I}$ is the identitity matrix.

Substituting Equation 2.24 into Equation 2.22 results in the final Newton-Raphson algorithm for the Backward Euler method:

$$(\mathbf{I} - h\mathbf{J_F})\delta\mathbf{q}_{i+1}^{n+1} = -\mathbf{q}_i^{n+1} + \mathbf{q}^n + h\mathbf{F}(\mathbf{q}_i^{n+1}, t_{n+1}). \tag{2.25}$$

In Equation 2.25, the system being solved is no longer a pure Jacobian of some function, but instead a mix of the identity matrix and a Jacobian. When the time-step h is small, the identity matrix begins to dominate the system, which generally makes the system easier to solve. A system is called diagonally dominant iff its elements e satisfy [14]

$$|e_{ii}| > \sum_{i \neq j} |e_{ij}| \quad \forall i, j. \tag{2.26}$$

If the time h step becomes smaller, the system is expected to become more diagonally dominant, which makes it easier to solve. This means that theoretically it is expected that a system takes less then twice as long to solve if the time-step h is halved.

3

Model problem: heat in a heterogeneous radiating 1D rod

From the theory, the regular Newton-Raphson method should always be faster than the JFNK method. However, if the analytic expression for the Jacobian is difficult to find, it is important to know how much worse the JFNK method performs, compared to the regular Newton-Raphson method. To find out how both methods compare, their performance on solving the temperature profile of a 1D model rod is tested.

3.1. Numerical model of a 1D rod

This subsection focuses on the model of the heat equation in a 1D rod. The rod, initially, has a uniform temperature of 600 K, is 2 m long, and is made of a material that has an inhomogeneous thermal conductivity $\lambda(x)$. The rod is modelled as a black body, and Dirichlet boundary conditions are imposed. The rod has a fixed temperature of 500 K at the left side ($x = 0$ m), and a fixed temperature of 700K at the right side ($x = 2$ m). The heat equation of the rod is given by [15]

$$\rho c_p \frac{\partial T}{\partial t} = \frac{\partial}{\partial x}\lambda(x)\frac{\partial T}{\partial x} - \sigma T^4, \tag{3.1}$$

where ρ is the density of copper ($8.69 * 10^3 \frac{kg}{m}$), c_p is the heat capacity of copper ($385 \frac{J}{kgK}$), σ is the Boltzmann constant ($5.67 * 10^{-8} \frac{J}{m^2 K^4 s}$), and T is the temperature in kelvin.

The thermal conductivity $\lambda(x)$ is defined as

$$\lambda(x) = 400 + 390 * \sin(\frac{3\pi}{2}x), \tag{3.2}$$

and is shown in Figure 3.1. The rod conducts heat relatively well at $x = \frac{1}{3}$ m and $x = \frac{5}{3}$ m, and insulates heat relatively well at $x = 1$ m. Because of these properties, the steady state solution of the heat equation should have a steep temperature gradient at the insulating part around $x = 1$ m [15].

In order to use numerical methods for the model problem, Equation 3.1 is discretized. The discretization in the x direction is done with a finite difference approximation, where the rod is modelled as N slices with a length $\Delta x = \frac{2}{N}$ m. The diffusion term of Equation 3.1 is approximated by:

$$\frac{\partial}{\partial x}\lambda(x)\frac{\partial T}{\partial x} \approx \frac{\lambda_{i+\frac{1}{2}}(T_{i+1} - T_i) - \lambda_{i-\frac{1}{2}}(T_i - T_{i-1})}{\Delta x^2} \qquad \{1 \leq i \leq N\}, \tag{3.3}$$

where $\lambda_{i\pm\frac{1}{2}} = \frac{1}{2}(\lambda_i + \lambda_{i\pm1})$. To implement the boundary conditions, T_0 is imposed to be 500 K, and T_{N+1} is imposed to be 700 K. Meaning that the system which is being solved does not include the boundary itself. A schematic drawing of the grid and boundary conditions are given in Figure 3.2.

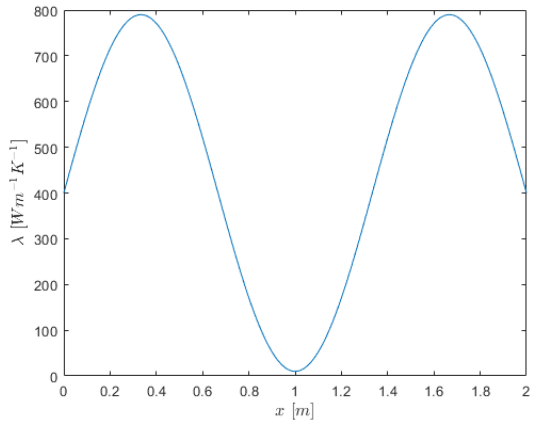


Figure 3.1: Thermal conductivity λ of the heterogeneous rod as function of x.

Figure 3.2: Schematic of the boundary conditions on the finite difference grid. The boundary nodes are given in black.

The time is discretized using a time-step Δt. The heat equation is then solved using the Backward Euler method. The fully discretized version of Equation 3.1 is given by

$$\rho c_p \frac{T_i^{n+1} - T_i^n}{\Delta t} = \frac{\lambda_{i+\frac{1}{2}}(T_{i+1}^{n+1} - T_i^{n+1}) - \lambda_{i-\frac{1}{2}}(T_i^{n+1} - T_{i-1}^{n+1})}{\Delta x^2} - \sigma(T_i^{n+1})^4 \qquad \{1 \le i \le N\}, \tag{3.4}$$

where the superscript is used to denote the time index, and the subscript denotes the spatial index.

The solution of the transient problem is plotted in Figure 3.3. It shows that the temperature profile becomes constant as time passes. This indicates that the transient solution converges to the steady state solution, given in Figure 3.4. When comparing the transient solution $\mathbf{x_{tr}}$ at $t = 1000$ s and the steady state solution $\mathbf{x_{SS}}$, the maximum difference between the solutions is 0.447 K. The norm of the difference between the temperature profiles $||\mathbf{x_{SS}} - \mathbf{x_{tr}}||_2$ is 11.7 K, which translates to an average error of about 0.004 K per grid point. These small differences confirm that the transient solution indeed converges to the steady state solution.

3.2. The algorithm in pseudo code

To understand the code that was used to solve the 1D model rod, the pseudo code of the transient simulation is given in Algorithm 3. It is made to be as generic as possible. For instance, when the matrix defining the linearized system is not symmetric, one would need to choose the GMRES method instead of the CG method as linear solver.

The simulation was written in Fortran using the PETSc library. PETSc can be difficult to learn, so a summary of how PETSc was implemented to solve the 1D model rod is given in Appendix B.

3.3. Newton methods for the steady state case

This subsection focuses on solving the steady state problem for the 1D model rod from Chapter 3.1. Both the regular Newton-Raphson and the JFNK method are tested for the steady state 1D problem to see which method works best in each case.

The 1D rod is simulated by taking $N = 10{,}000$ grid points, $\tau_F = 10^{-9}$ as relative Newton-Raphson function tolerance, $\tau_S = 10^{-4}$ as relative Newton-Raphson step tolerance, and $\tau_{CG} = 10^{-4}$ as relative CG tolerance.

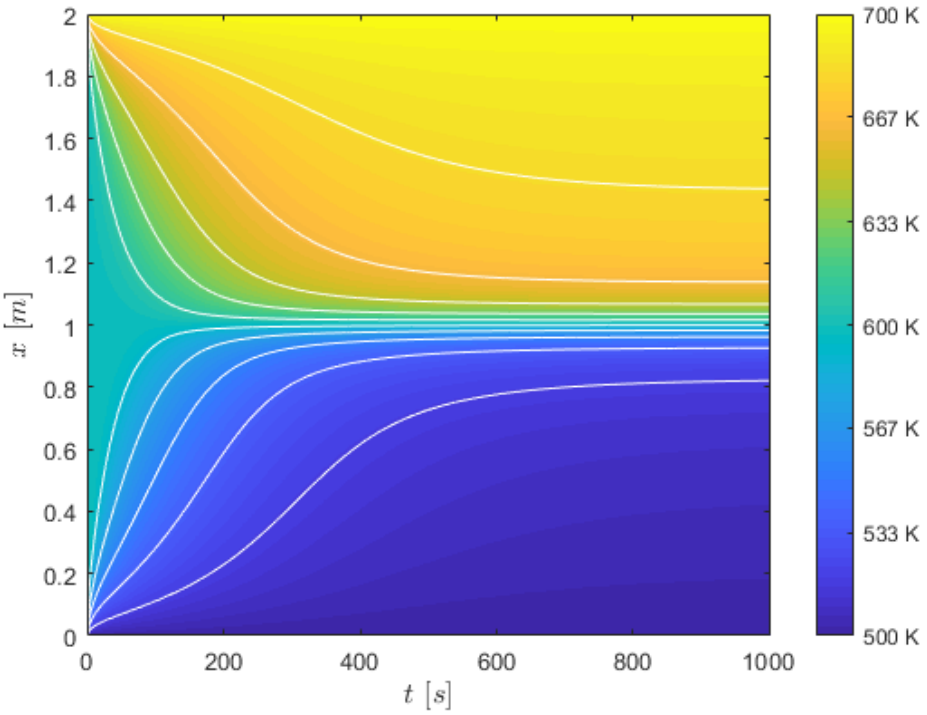


Figure 3.3: Solution for the transient heat equation in a 1D rod. The temperature is represented by the color of the plot in Kelvin. One thousand Backward Euler iterations were computed, each with a time-step $\Delta t = 1$ s. The white lines are contour curves that are drawn at 519 to 681 kelvin, with steps of 18 kelvin from bottom to top.

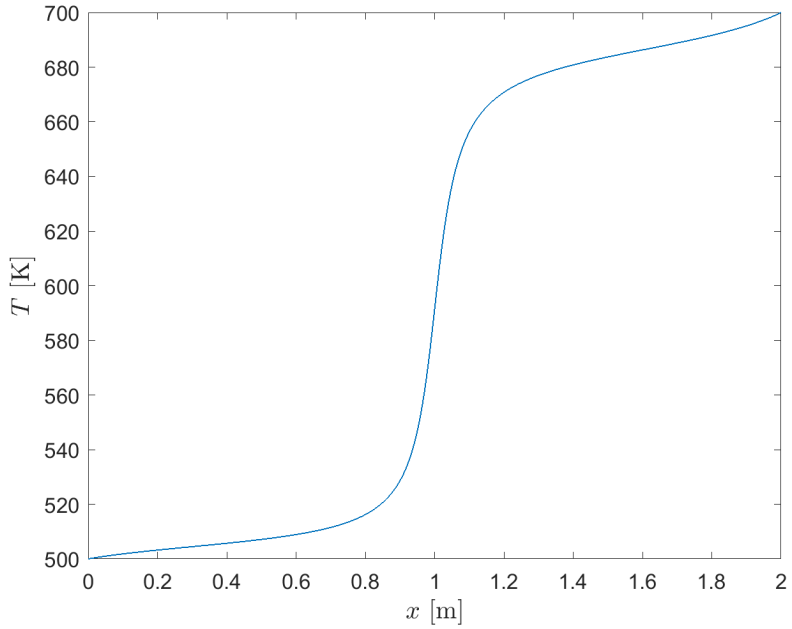


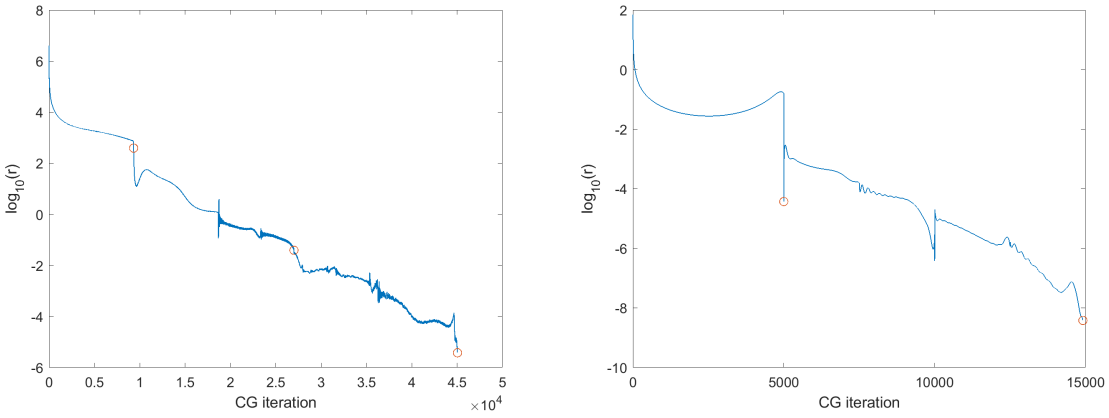
Figure 3.4: Steady state heat profile for the 1D model of a radiating rod with heterogeneous thermal conductivity.

Data: Non-linear function $\frac{\partial \mathbf{T}}{\partial t} = \mathbf{F}(\mathbf{T})$, Jacobian $\mathbf{J_F}$, initial temperature profile $\mathbf{T}^0$, initial time t_0, time-step Δt.
Result: The temperature profile $\mathbf{T}$ at time t.

Choose linear solver type (LS);
Choose preconditioning algorithm;
$\mathbf{T} = \mathbf{T}^0$;
$t = t_0$;
while $t < t_{max}$ **do**
  while *NR tolerance not met* **do**
    while *LS tolerance not met* **do**
      if *Using Jacobian* **then**
        Calculate the preconditioning matrix $\mathbf{M}$;
        Solve next iteration of: $\mathbf{M}(\mathbf{I} - \Delta t \mathbf{J_F})\delta \mathbf{T}_{i+1}^{n+1} = \mathbf{M}\left[\mathbf{T}^n + h\mathbf{F}(\mathbf{T}_i^{n+1}) - \mathbf{T}_i^{n+1}\right]$;
      else
        Approximate the Jacobian with $\hat{\mathbf{J}}_\mathbf{F}$;
        Solve next iteration of: $(\mathbf{I} - \Delta t \hat{\mathbf{J}}_\mathbf{F})\delta \mathbf{T}_{i+1}^{n+1} = \mathbf{T}^n + h\mathbf{F}(\mathbf{T}_i^{n+1}) - \mathbf{T}_i^{n+1}$;
      end
    end
    $\mathbf{T}_{i+1}^{n+1} = \mathbf{T}_i^{n+1} + \delta \mathbf{T}_{i+1}^{n+1}$;
  end
  $\mathbf{T}^{n+1} = \mathbf{T}_{i+1}^{n+1}$;
  $t = t + \Delta t$;
end

Algorithm 3: Pseudo code of an algorithm which solves a transient problem using Newton-Raphson and Backward Euler.

These values were chosen such that the error in the computational time was acceptable, while still having a feasible computational time. The relative computation time was calculated with respect to the computation time of the regular Newton-Raphson method with unpreconditioned CG as linear solver (3.3 ± 0.1 s).



(a) CG residual r with no preconditioning.

(b) CG residual r with Jacobi preconditioning.

Figure 3.5: Overview of the CG residual r of the unpreconditioned and preconditioned systems, from the Newton-Raphson method solving the SS heat equation in a 1D heterogeneous rod. The plots depict multiple Newton-Raphson iterations, the last CG iteration of each Newton-Raphson cycle is marked with an 'o'.

3.3.1. Regular Newton-Raphson

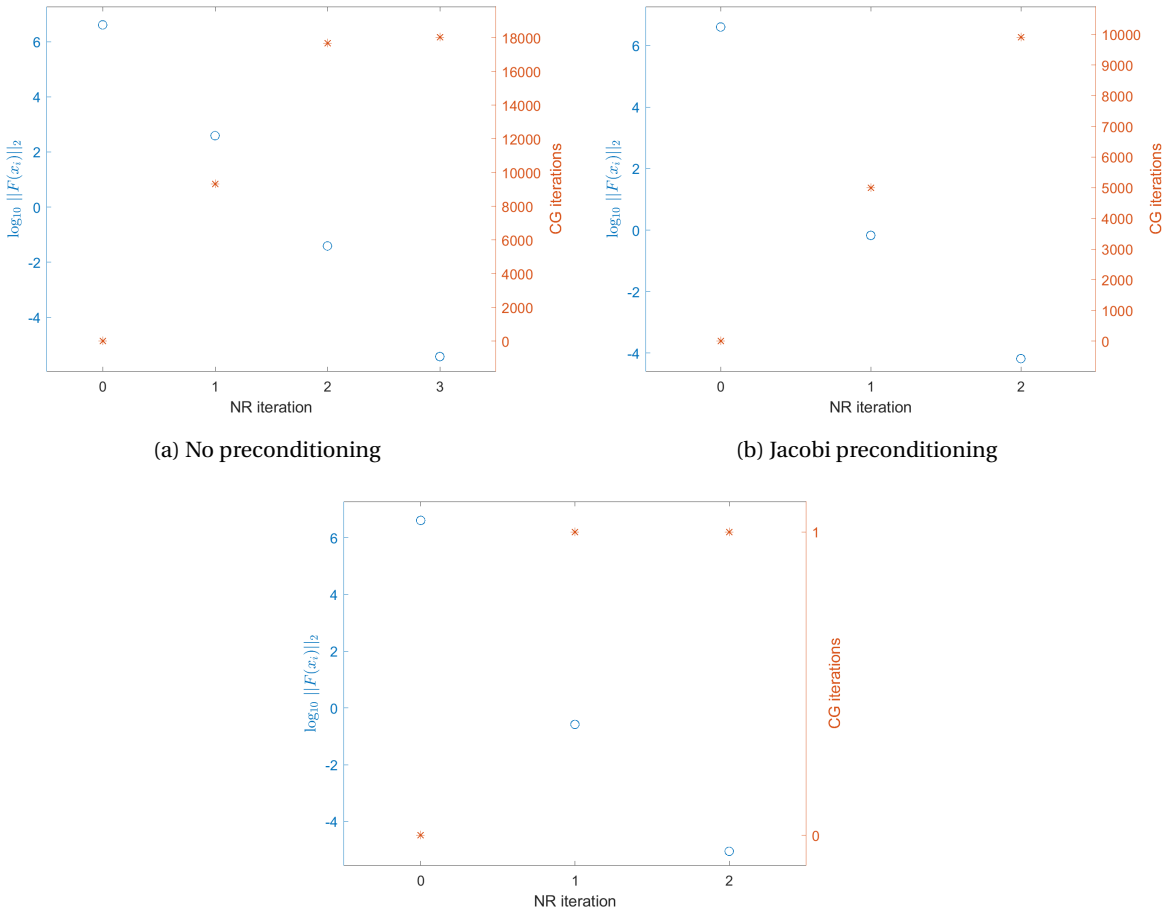
The first method being tested on the model rod is the regular Newton-Raphson method, which means the exact Jacobian is used. A big advantage of this method is that preconditioners can be used since the Jacobian is known. To get the best performance, multiple preconditioners are tested. The results for the steady state 1D model problem are given in table 3.1. It clearly shows that preconditioning plays a huge role in speeding up the iterative linear solver. In this case, the complete and incomplete Cholesky preconditioning methods are the same; in fact, the tridiagonality of the system makes the Cholesky preconditioners act as a direct solver. This resulted in a very short computation time, but should not be seen as realistic for systems which are not tridiagonal. Jacobi preconditioning also gave a significant decrease in computation time. To investigate the effect of preconditioners further, the CG convergence with and without Jacobi preconditioning is given in Figure 3.5. The CG convergence when using the Cholesky preconditioners are left out because these methods converged in a single step.

All the preconditioned systems take one less Newton-Raphson iteration to be solved, compared to the case where no preconditioning is used. For the complete and incomplete Cholesky preconditioning methods, this is due to the exact solution having an error which is smaller than the bounds on the CG algorithm. This results in the Newton-Raphson steps being computed with a greater accuracy, which in turn results in less Newton-Raphson iterations being needed to reach the tolerances.
Even though the Jacobi preconditioner did not act as a direct solver, it can be seen in Figure 3.5b that the Jacobi preconditioned system makes an improvement of 4 orders at iteration 5,000 of the CG algorithm. This resulted in the error of the Jacobi preconditioned system being over 100 times smaller than the tolerances set on the linear solver. The increased accuracy for this first Newton-Raphson iteration was enough the reduce the total amount of Newton-Raphson steps needed by one.

When using the Jacobi preconditioner, the CG method makes big improvements at iteration 5,000 and 10,000. This indicates that Jacobi preconditioning makes the CG method a quasi-direct solver after N/2 steps.

Preconditioner	Relative computation time	Newton iterations	CG iterations
none	1	3	45,041
Incomplete Cholesky	$(3.2 \pm 0.2) * 10^{-3}$	2	2
Complete Cholesky	$(3.3 \pm 0.2) * 10^{-3}$	2	2
Jacobi	$(3.6 \pm 0.2) * 10^{-1}$	2	14,912

Table 3.1: The results of the Newton-Raphson method for solving the SS heat equations in a 1D rod, with heterogeneous thermal conductivity. The Newton and CG iterations refer to the total amount of iterations used to solve the problem. All times are relative to the 3.3 ± 0.1 s computation time of the regular Newton-Raphson method with unpreconditioned CG as linear solver.



(a) No preconditioning

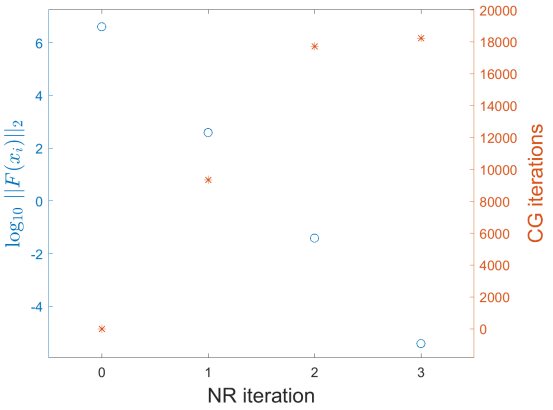
(b) Jacobi preconditioning

(c) Complete / Incomplete Cholesky preconditioning

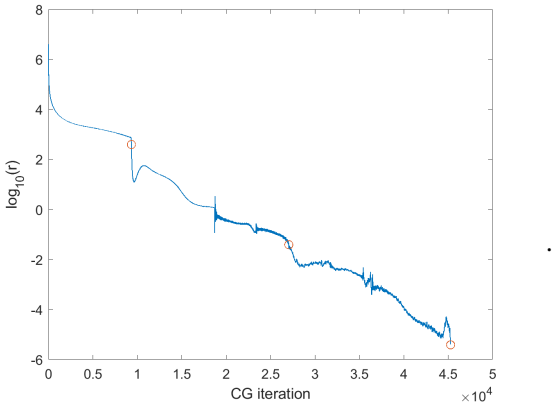
Figure 3.6: Newton-Raphson convergence and CG iterations per Newton-Raphson iteration for the steady state 1D rod model problem.

The goal is now to find out more about how the Newton-Raphson method converges and how much computing power each step takes. To investigate this, the Newton-Raphson residuals and CG iterations per Newton-Raphson iteration are plotted for each of the used preconditioning methods in Figure 3.6. For both no preconditioner and the Jacobi preconditioner, it takes more CG iterations to solve the later Newton-Raphson steps compared to the first Newton-Raphson steps. This indicates that the spectral properties of the Jacobian become less favorable for the CG method after the first Newton-Raphson iteration.

The rate of convergence p of the Newton-Raphson method without preconditioning has an approximate value of $p = 2.1$ (see appendix A for the definition of p). This is close to the theoretical value of $p = 2$, associated with the quadratic convergence of Newton-Raphson. The difference can be explained by the error caused by the relative tolerance.



(a) Newton-Raphson residual of the JFNK method, alongside with the CG iterations per Newton-Raphson iteration.



(b) CG residual r of the JFNK method. Multiple Newton-Raphson cycles are depicted, the last CG iteration of each Newton-Raphson cycle is marked with an 'o'.

Figure 3.7: The convergence of the JFNK method and the CG convergence for solving the steady state 1D rod model problem.

3.3.2. Jacobian-free Newton-Raphson

Because the Jacobian is not known when using the JFNK method, preconditioning could not be used with this method. This is an inherent disadvantage of the JFNK method, but even when compared to the regular Newton-Raphson method without preconditioning, the computational time was (5.0 ± 0.2) times as slow. The JFNK method took a total of 4 Newton-Raphson iterations and 45,273 CG iterations to converge. More details on the convergence of the JFNK method are plotted in Figure 3.7. It is interesting to note that the JFNK method used a total of 45,273 CG iterations to solve the steady state 1D rod, while the unpreconditioned regular Newton-Raphson method only took a total of 45,041 CG iterations to solve the same system. The relative difference in the amount of CG iterations is less than 1%, which clearly indicates that the JFNK method results in almost exactly the same linearized system as when using the regular Newton-Raphson method. To measure how well the JFNK approximation works, the relative difference β between the first Newton-Raphson step for both methods were calculated using:

$$\beta = \frac{||\delta\hat{\mathbf{x}}_1 - \delta\mathbf{x}_1||_2}{||\delta\mathbf{x}_1||_2}, \tag{3.5}$$

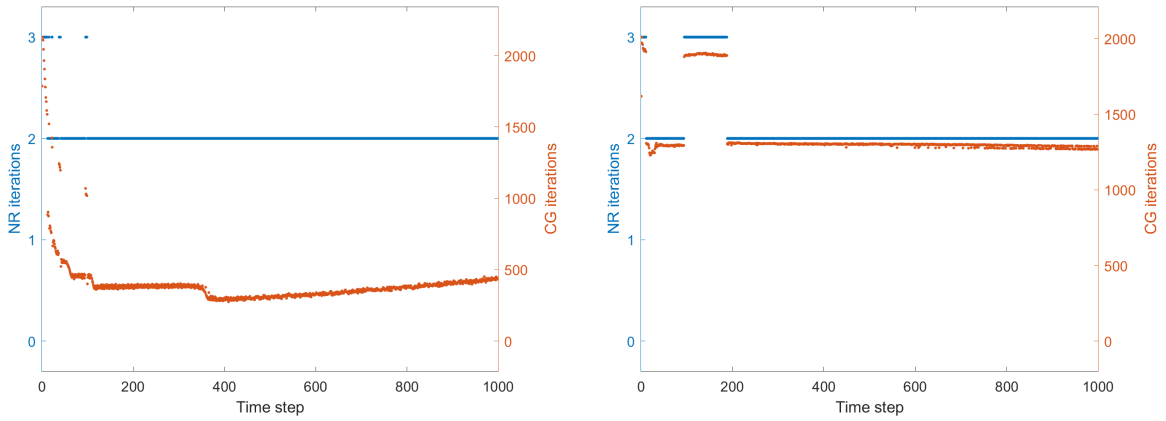
where $\delta\hat{\mathbf{x}}_i$ is the first step of the JFNK method, and $\delta\mathbf{x}_i$ is the first step of the regular Newton-Raphson method. This resulted in a relative difference of β = 6.0E-6, which shows that the approximated Jacobian is accurate. The relative error for the later steps was not calculated, because the small error causes $\hat{\mathbf{x}_2}$ and $\mathbf{x_2}$ to be slightly different, which makes β diverge.

The JFNK method has the same convergence rate $p = 2.1$ as the regular Newton-Raphson method.

An important note is that the default value $\epsilon = 1.5 * 10^{-8}$ as stepsize in the forward difference approximation of the JFNK method did not work well. No theoretical optimal value could be established for ϵ, since this would require a bound on the second derivative of the LHS in Equation 3.4 [5]. Instead an empirical approach was taken. From testing, $\epsilon = 4.7 * 10^{-4}$ was found to be the best value (using β as a measure), so this value was used for all JFNK approximations regarding the 1D model problem.

3.4. Newton methods for the transient case

In contrast to the steady state model problem, the linearized system that is being solved in the transient case is more diagonally dominant, due to the Backward Euler method. Diagonally dominant systems are easier to solve for iterative solvers compared to non-diagonally dominant systems [14]. This means that the regular Newton-Raphson and JFNK method perform differently for transient problems. To see how both methods are affected, this subsection focuses on comparing the regular Newton-Raphson and the JFNK methods for solving the transient 1D rod model problem described in Chapter 3.1.



(a) No preconditioning

(b) Jacobi preconditioning

Figure 3.8: The number of Newton-Raphson and CG iterations needed for each time-step while solving the transient solution of the 1D model rod. The simulation used a time-step of $\Delta t = 1$ s, for a total of 1,000 steps.

The transient 1D rod problem is simulated by taking $N = 3,000$ grid points and 1,000 time-steps of $\Delta t = 1$s. The tolerances are the same as in the steady state case. The relative computation time is respective to the computation time of the regular Newton-Raphson with no preconditioner method (9.6 ± 0.4 seconds).

3.4.1. Regular Newton-Raphson

The regular Newton-Raphson method was tested for the transient single-physics problem with a number of different preconditioners. The computation time of the exact Newton-Raphson method with each preconditioner is given in Table 3.2. Just like the steady state problem, the incomplete and complete Cholesky preconditioners were the same and gave the best result. This is again because the Cholesky preconditioner acted as a direct solver for tridiagonal systems. However, it should be noted that the effectiveness of the Cholesky preconditioner has decreased significantly compared to the steady state case. Furthermore, Jacobi preconditioning did not improve the computation time at all; it even made it worse with a factor of 3. To find out why preconditioning is less effective for transient problems, the Newton-Raphson and CG iterations per time-step are plotted in Figure 3.8. It shows that the number of CG iterations needed to solve the Jacobian without preconditioning decreases significantly after the first few time-steps. This is because the initial guess gets closer to the exact solution as the system goes to steady state. This makes the CG iterations drop from 2130 at the first time-step to only 441 at the final (1,000th) time-step, with an average of 403 CG iterations. In Figure 3.8b, it can be seen that the CG iterations per time-step stay almost constant at 1280 iterations when using the Jacobi preconditioner. The discontinuity at $t = 200$ is due to the Newton-Raphson method needing one additional iteration for these points. It is assumed that this was caused by the Jacobi preconditioned system having a slightly worse solution than the unpreconditioned system.

Preconditioner	Relative computation time	Newton iterations	CG iterations
none	1	2,023	4.0E5
Cholesky	$(1.8 \pm .1) * 10^{-1}$	2,000	2,000
Jacobi	3.0 ± 0.2	2,105	1.4E6

Table 3.2: The results of the regular Newton-Raphson method for solving the transient heat equation in the 1D rod. Cholesky refers to both the incomplete and complete preconditioning method since they gave the same result. The relative computation time is relative to the 9.6 s computational time of the unpreconditioned regular Newton-Raphson method, solving the transient heat equation in the 1D rod. The Newton and CG iterations refer to the total amount of iterations used to solve the problem, so the sum over all time-steps.

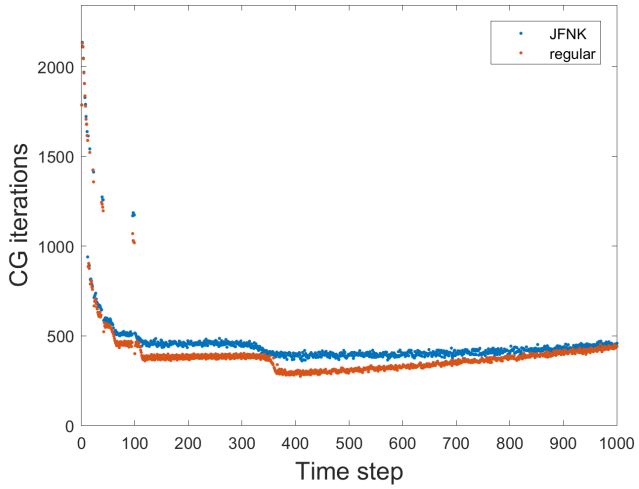


Figure 3.9: Plot of the Newton-Raphson and CG iterations per time-step, for the transient simulation of the 1D model rod with a thousand time-steps of $\Delta t = 1$ s. The amount of Newton-Raphson iterations per time-step are not shown because they were the same for both methods.

After the first few time-steps, the Jacobi preconditioned system needs more CG iterations per time-step than the unpreconditioned system. Furthermore, the Jacobi preconditioner also takes time to be applied to the system. This makes it clear why the Jacobi preconditioned system had a worse computational time compared to the unpreconditioned system. There are a couple of ways in which the bad computational time of the Jacobi preconditioned system can be explained. Firstly, because of the diagonal dominance, the system is already well suited to be solved by iterative solvers. So even if the preconditioner could offer improvement in solving the system efficiently, there would not be a lot to improve. Secondly, preconditioners are usually not made for already well-conditioned systems, so preconditioning these systems could make them harder to solve [8].

To find out if the effectiveness of preconditioning is dependent on the Δt, the problem was solved again with 5000 time-steps of $\Delta t = 0.2$ s. With this smaller Δt, the unpreconditioned and Jacobi preconditioned systems still had the same relative computational time compared to each other. The computational time of both the simulations was $2.3 \pm .1$ times larger compared to their counterpart simulation with $\Delta t = 1$ s. However, both of the methods calculated 5 times more time-steps in this time-frame, so each time-step was calculated 2.2 ± 0.1 times faster compared to the $\Delta t = 1$ s simulation. By using this smaller Δt, the average number of CG iterations per time-step was reduced to 39% and 46% of the original amount, for the unpreconditioned and Jacobi preconditioned systems respectively. From these results, it is concluded that the time-step does not significantly influence the effectiveness of preconditioning.

Due to the method converging almost always within 2 Newton-Raphson iterations, no reliable rate of convergence could be determined.

3.4.2. Jacobian-free Newton-Krylov

Just like the steady state case, the JFNK method was slower than the exact Newton-Raphson method for the transient single-physics case, by a factor of (5.9 ± 0.2). The JFNK method has the same number of Newton-Raphson iterations at each time-step, but has a significantly different amount of CG iterations per time-step. The JFNK method used a total of 461,000 CG iterations, which corresponds to an average of 61 (14%) more CG iterations per time-step compared to the regular Newton-Raphson method. The CG iterations per time-step for both methods are shown in Figure 3.9.

The JFNK method gave almost exactly the same solution as the regular Newton-Raphson method for the transient case. The average absolute difference was $2.3 * 10^{-9}$ K over all grid points and all time-steps, with a maximum difference of $2.8 * 10^{-9}$ K between the two solutions. This meant that both methods resulted in the same solution. The different number of CG iterations used per time-step can be explained by the fact that the JFNK method will always have a numerical error in the approximated Jacobian. This error did not play a significant role in the steady state case since the relative difference in the number of CG iterations was less than 1% for this problem.

4

Newton methods for multi-physics problems

At the core of every simulation is a model which is governed by its relevant physical laws. These models can usually be solved using numerical algorithms such as Newton-Raphson, but what happens if multiple models describe a system? If the models do not interact with each other, both models can be solved separately without even considering each-other. However, if the systems do interact, the models can not be solved separately anymore, and it becomes a multi-physics (coupled) system. This type of problem often occurs in reactor physics, which is why they are investigated in this thesis. This chapter discusses how coupled systems work and how well Newton methods perform for this type of problems.

4.1. Newton-Raphson for coupled systems

Suppose n subsystems make up a coupled system. If a function $\mathbf{F_i}(\mathbf{x})$ describes each subsystem, then the coupled system is given by

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} \mathbf{F_1}(\mathbf{x}) \\ \mathbf{F_2}(\mathbf{x}) \\ \vdots \\ \mathbf{F_n}(\mathbf{x}) \end{bmatrix}. \tag{4.1}$$

Just as with any regular system, Newton-Raphson is a useful method to solve coupled systems. The Jacobian of a coupled system consists of all the Jacobians of the individual subsystems. This gives the Jacobian a block-like structure, which is why it is referred to as a block Jacobian. For the coupled system given by $\mathbf{F}$, the block Jacobian is given by:

$$J_{\mathbf{F}} = \begin{bmatrix} \mathbf{J_{F_1}} & \mathbf{C}_{1,2} & \cdots & \mathbf{C}_{1,n} \\ \mathbf{C}_{2,1} & \mathbf{J_{F_2}} & \cdots & \mathbf{C}_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{C}_{n,1} & \mathbf{C}_{n,2} & \cdots & \mathbf{J_{F_n}} \end{bmatrix}, \tag{4.2}$$

where $\mathbf{J_{F_i}}$ is the Jacobian of the subsystem $\mathbf{F_i}$ and $\mathbf{C}_{i,j}$ is the coupling term between subsystem i and j.

A common problem in multi-physics is finding the coupling term $\mathbf{C}$. It is often not known or very hard to calculate, which is a significant obstacle to the regular Newton-Raphson method. However, the JFNK method does not need to know this coupling term, since it does not need to compute the Jacobian explicitly. This makes the JFNK method a promising candidate for these type of problems, where the Jacobian is (partially) unknown.

4.2. Multi-physics example problem

In Chapter 5 of this thesis a saddle point coupled system is encountered, which has the following form:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B}^T \\ \mathbf{B} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix}, \tag{4.3}$$

with **A** and **B** some matrices. Ideally, the 1D rod model problem of Chapter 3.1 would be rewritten to a saddle point problem. This would make it possible to investigate methods which might work well for the coupled system encountered in Chapter 5. However, the Jacobian of the 1D rod model problem is tridiagonal, which means that setting the lower right block to **0** makes the matrix singular. Thus the 1D model rod can not be used to test the performance of Newton methods for solving saddle point coupled systems.

Instead, the 1D rod model problem is used to test how well Newton methods perform on a coupled system with unknown coupling terms. This is achieved by imaginarily splitting the rod at $x = 0.5$ m into two smaller rods, as shown in Figure 4.1. The rod is split off-center (at 0.5 m while the rod has a length of 2 m), to avoid any unwanted symmetries.

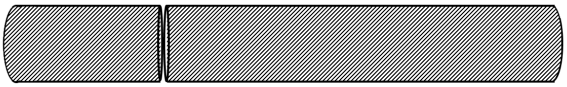


Figure 4.1: A rod being imaginarily cut at a quarter of its length.

Suppose left rod is described by $\mathbf{F_1}$, and the right rod by $\mathbf{F_2}$. These two functions together still describe the original problem before the cut, so the original solution to the heat equation of the un-splitted rod is still correct. The goal of this simulation is to mimic a system where the coupling term can not be calculated. To mimic such a system, the coupling terms of the split rod are imposed to be zero, even though in reality, they are not. The Jacobian of the coupled system, with the missing coupling terms, is then given by

$$\mathbf{J_F} = \begin{bmatrix} \mathbf{J_{F_1}} & \mathbf{0} \\ \mathbf{0} & \mathbf{J_{F_2}} \end{bmatrix}, \tag{4.4}$$

where $\mathbf{J_{F_1}}$ and $\mathbf{J_{F_2}}$ are the Jacobians of $\mathbf{F_1}$ and $\mathbf{F_2}$ respectively.

Using this incomplete Jacobian to solve the coupled system is called incomplete Newton-Raphson throughout this thesis.

4.3. Block algorithms

The inverses of the individual subsystems largely determine the inverse of a block Jacobian. This makes it possible for specialized block algorithms to solve the block Jacobian system, using the solutions of these individual subsystems. This subsection discusses the theory behind block algorithms that were used to solve the 1D model multi-physics problem.

4.3.1. Theory of block algorithms

The goal of a block algorithm is to solve a system $\mathbf{Ax} = \mathbf{b}$, where **A** consists of block matrices. Because only coupled systems with two subsystems are encountered in this thesis, **A** always has the form

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}, \tag{4.5}$$

where $\mathbf{A}_{11}$ and $\mathbf{A}_{22}$ represent the individual system discretization matrices, and $\mathbf{A}_{12}$ and $\mathbf{A}_{21}$ represent the coupling terms. The block algorithm used in this report then iterates towards the solution using

$$\mathbf{x}^{i+1} = (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})\mathbf{x}^i + \mathbf{B}^{-1}\mathbf{b}, \tag{4.6}$$

where **B** is either the block Jacobi or block Gauss-Seidel matrix [16].

4.3.2. Block Jacobi

The block Jacobi algorithm is based on just the diagonal block matrices in the nested matrix. First, the individual solutions for each diagonal element are computed, and then the block Jacobi method starts iterating until it converges.

The block matrix of the block Jacobi method is given by [16]

$$\mathbf{B} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{22} \end{bmatrix}. \tag{4.7}$$

From the definition of **B**, it is evident that the block Jacobi algorithm does not depend on the coupling term in the Jacobian. This does not mean that the problem is uncoupled, because the functions $\mathbf{F}_1$ and $\mathbf{F}_2$ may still depend on each other.

4.3.3. Block Gauss-Seidel

The Gauss-Seidel algorithm is based on solving one part of the coupled system first and then using that solution to solve the other part. The Gauss-Seidel algorithm is guaranteed to converge if the matrix is either symmetric and positive definite or diagonally dominant [8].

The block matrix for the Gauss-Seidel method is given by [16]

$$\mathbf{B} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{0} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}. \tag{4.8}$$

From the definition of B, it is clear that the solution of subsystem 1 is independent of the solution of subsystem 2. This allows the algorithm first to solve subsystem 1 independently, and then use this solution to approximate the solution for subsystem 2.

4.4. Newton methods for the steady state multi-physics 1D rod

So far, it has been shown that the JFNK method is slower than the regular Newton-Raphson method for the single-physics case, but it is still unknown how they compare for the multi-physics case. Similar to the single-physics case of Chapter 3.1, the multi-physics model problem can be split up into two different cases: steady state and transient. Both of these cases are solved in a very different manner, so they are discussed separately. This chapter focuses on testing the performance of the JFNK and incomplete Newton-Raphson method for the 1D split rod multi-physics model problem.

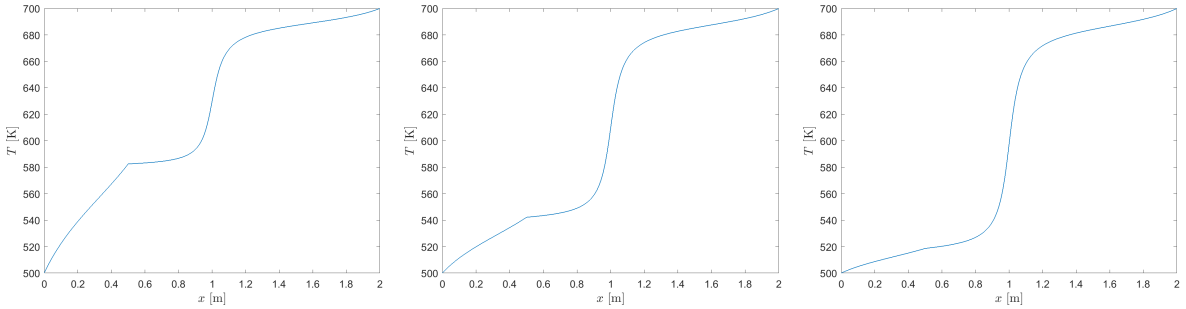
The multi-physics problem is modeled by taking $N = 3{,}000$ grid points, where the rod is split up at $N = 750$, corresponding to $x = 0.5$m. The heat equation is the same as for the single-physics problem, but the Jacobian is replaced with the incomplete Jacobian from Equation 4.4. This results in missing Jacobian elements for $x = 0.5$m. The relative Newton-Raphson and CG tolerances are taken the same as for the single-physics problem ($\tau_F = 10^{-9}$ and $\tau_{CG} = 10^{-4}$ respectively). The relative step tolerance on the Newton-Raphson algorithm is disabled because the newton steps become extremely small before the solution is close to convergence.

4.4.1. Incomplete Newton-Raphson

The incomplete Newton-Raphson method was tested for the steady state multi-physics problem with two different block algorithms: block Jacobi and block Gauss-seidel. Both the Jacobi and Gauss-Seidel block algorithms had the same computation time and gave exactly the same results. This can be explained by the fact that the block matrix only has diagonal elements, which makes it trivial to solve. The results are given in Table 4.1.

Computation time	Newton iterations	CG iterations
35 ± 0.6	20,000	20,000

Table 4.1: The results of the incomplete Newton-Raphson method for solving the steady state and multi-physics version of Equation 3.1. The absolute computation time is used since no preconditioner methods could be used.



(a) 300 NR iterations. (b) 1400 NR iterations. (c) 3000 NR iterations.

Figure 4.2: Plots of the intermittent solutions of the incomplete Newton-Raphson method for the steady state multi-physics model problem.

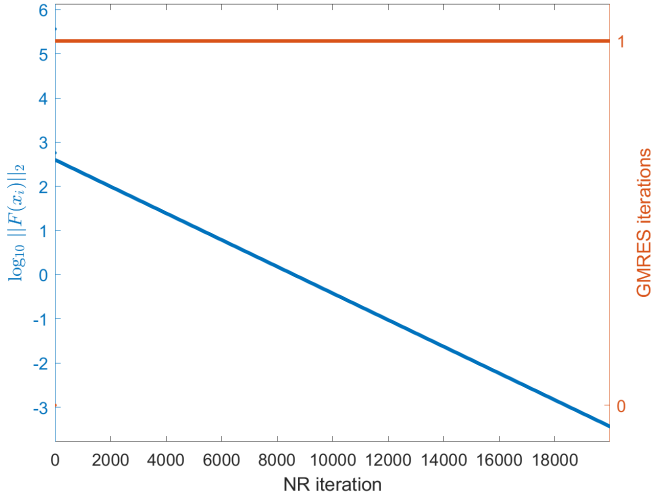


Figure 4.3: Plots of the convergence properties of the incomplete Newton-Raphson for the 1D split rod model problem. The GMRES iterations refer to the number of outer block iterations being performed per Newton-Raphson step.

To investigate why 20,000 Newton-Raphson iterations are needed to converge, the intermittent Newton-Raphson iterations are plotted in Figure 4.2. This shows that $N = 750$ ($x = 0.5$m) is indeed the point that prevents the incomplete Newton-Raphson method to converge properly. This is to be expected, since this is the point where the Jacobian is incorrect.

To investigate why the incomplete Newton-Raphson method works poorly for the coupled problem, the convergence properties are plotted in Figure 4.3. It shows that the Jacobi & Gauss-Seidel block algorithms only need one GMRES iteration per Newton-Raphson iteration to solve the outer block system. The convergence rate of the Newton-Raphson method has an approximated average value of $p = 1.00$, and takes a huge amount of iterations to converge properly because of the incomplete Jacobian.

Both subsystems (the part from 0-0.5 m and 0.5-2 m) were solved using GMRES with ILU(0) preconditioning. Just as with the incomplete Cholesky preconditioning, the ILU(0) preconditioner acted as a direct solver because the system is tridiagonal. This resulted in all the inner subsystems being solved with only one GMRES iteration. These settings were not tweaked further because this was already done in Chapter 3.1.

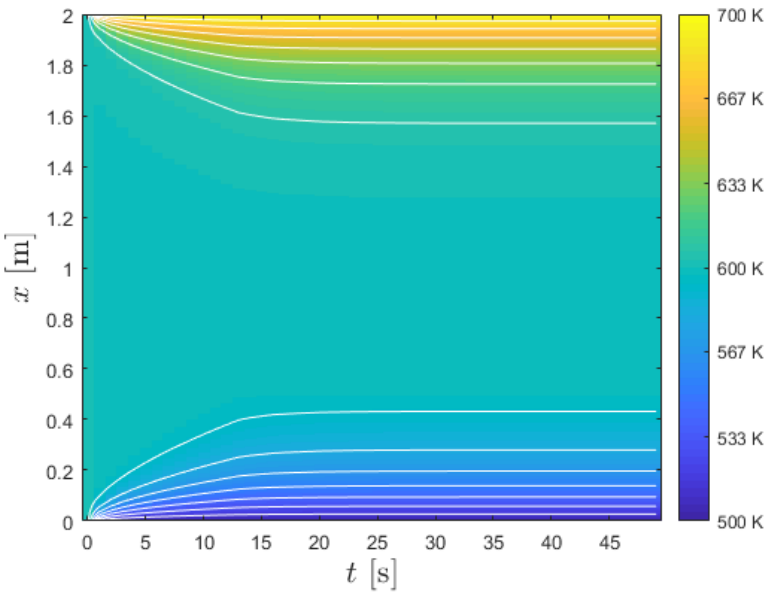


Figure 4.4: The wrong transient solution of the incomplete Newton-Raphson. The color depicts the temperature of the rod in Kelvin.

4.4.2. Jacobian-free Newton-Krylov

Because the Jacobian is not used in the JFNK method, the incomplete Jacobian did not affect the JFNK method at all. This resulted in the JFNK method having the exact same convergence properties as if the rod was not split at all. The problem was solved as a single-physics problem, just like in Chapter 3.1. The JFNK method was (9.1 ± 0.3) times faster compared to the incomplete Newton-Raphson method, and a total of 4 Newton-Raphson iterations and 13,430 CG iterations were computed. The difference in the number of linear solver iterations in the multi-physics case compared to the single-physics case can be explained by the number of grid points being only $N = 3,000$ instead of $N = 10,000$. Since the results of the JFNK method are the same as in Chapter 3.1, they are not discussed here.

The JFNK method outperformed the regular Newton-Raphson method in every aspect. From this, it was concluded that the JFNK method is the better method for the steady state coupled system with missing coupling terms.

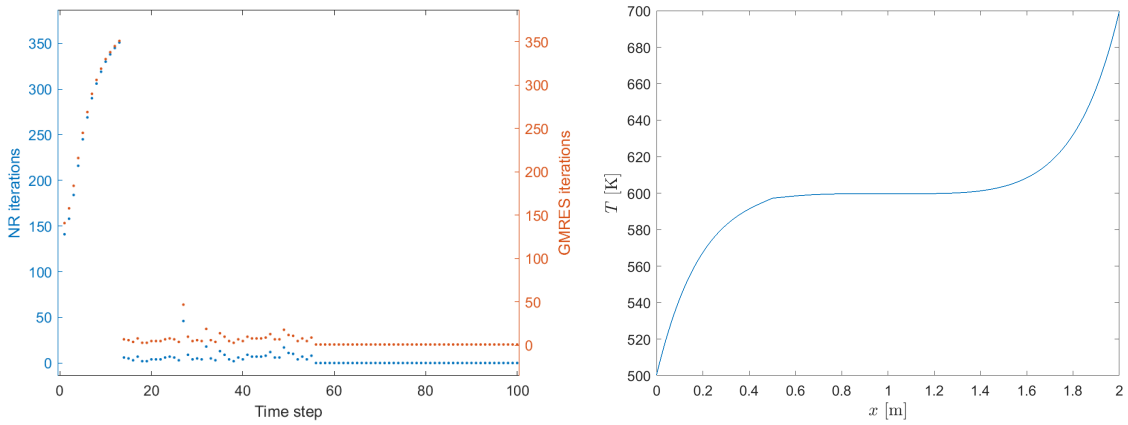
4.5. Newton methods for the transient multi-physics 1D model problem

In this subsection, the incomplete Newton-Raphson and JFNK methods are compared for solving the transient split rod multi-physics problem.

The single-physics problem is simulated by taking 1,000 time-steps of $\Delta t = 1$ s. All other parameters are the same as with the steady state split-rod simulation.

4.5.1. Incomplete Newton-Raphson

The incomplete Newton-Raphson method is extremely unfit for this type of problem and was not able to solve it with a time-step of $\Delta t = 1$ s. The unfinished transient solution of the incomplete Newton-Raphson method with $\Delta t = 1$ s is plotted in Figure 4.4. It shows that the first few Backward Euler iterations show the physically correct behavior of the boundaries heating up first. However, as soon as the heat starts to diffuse deeper into the rod and reaches the imaginary cut, the incomplete Newton-Raphson method breaks down and stops working. The cause of the breakdown, according to PETSc, is that the Newton-Raphson solution did not improve. A multitude of conditions can cause this error, but in this case, it is most likely that the incomplete Jacobian is the root of the problem. The convergence of the Newton-Raphson method and the final solution are plotted in Figure 4.5. It shows that before breaking down, the number of incomplete Newton-Raphson iterations per time-step increases to over 350. Then the iterations per time-step suddenly drop, and



(a) NR and CG iterations per time-step.

(b) Transient solution for $t > 15$s.

Figure 4.5: The failed solution of the incomplete Newton-Raphson method for the coupled 1D rod model system.

the method stops working. This behavior could not be explained. It should be noted that the amount of incomplete Newton-Raphson iterations appears to form a parabolic function, which is almost at its top when the breakdown occurs.

In Figure 4.5b, a slight discontinuous bend at the border of the two rods ($x = 0.5$ m) can be seen. This bend indicates that the Newton-Raphson method is not able to solve the problem correctly since the curve should be smooth.

No rate of convergence could be calculated since the problem could not be solved.

To test if the breakdown could be fixed by using a smaller time-step Δt, three additional runs were done using $\Delta t = 10$ s, $\Delta t = 0.5$ s, and $\Delta t = 0.1$ s. All of these simulations broke down as well. From this, it is concluded that the incomplete Newton-Raphson method should not be used for transient coupled systems with missing coupling terms.

4.5.2. Jacobian-free Newton-Krylov

The JFNK method was again unaffected by the incomplete Jacobian. The results of the transient multi-physics simulations were the same as the JFNK simulation for the transient single-physics case in Chapter 3.4.2.

The JFNK method was able to solve the transient multi-physics problem, which made it automatically the best method out of the two.

4.5.3. Newton methods for other types of incomplete coupled systems

It is possible to encounter a system where only one of the coupling terms is known, which is why the following two Jacobians were tested as well:

$$\mathbf{J_F} = \begin{bmatrix} \mathbf{J_{F_1}} & \mathbf{0} \\ \mathbf{C_{2,1}} & \mathbf{J_{F_2}} \end{bmatrix}; \qquad \mathbf{J_F} = \begin{bmatrix} \mathbf{J_{F_1}} & \mathbf{C_{1,2}} \\ \mathbf{0} & \mathbf{J_{F_2}} \end{bmatrix}. \tag{4.9}$$

Both of the incomplete Jacobians in Equation 4.9 gave exactly the same results.

When removing only one of the coupling terms from the Jacobian, the total amount of GMRES iterations was only 52% of the case where both coupling terms were removed. This resulted in these systems being faster than the system with two missing coupling terms, but asides from that no other differences were found. Because of the similar behavior of these coupled systems, the Jacobians in Equation 4.9 are not investigated further.

5

Methods for solving linearized incompressible Navier-Stokes systems

The Navier-Stokes (NS) equations are encountered in many engineering and scientific fields. The NS equations are partial differential equations which describe how the flow and pressure inside a system evolve over time. They are used for almost anything related to flow, e.g. the weather, flow in a pipe, or the aerodynamics of an airplane. This makes the Navier-Stokes equations very important for the simulations at the Reactor Institute Delft.

This thesis investigates which methods work best to solve the linearized incompressible Navier-Stokes system.

5.1. Basics of the Incompressible Navier-Stokes equations

The general idea of the Navier-Stokes equations is to start with the conservation of momentum for a system, and then use some properties of the system to find the correct differential equations. For the incompressible flow problem in this thesis, the liquid is assumed to be Newtonian, so the conservation of momentum can be rewritten into

$$\rho \frac{\partial \mathbf{u}}{\partial t} + \rho \nabla \cdot (\mathbf{u} \otimes \mathbf{u}) - \eta \nabla^2 \mathbf{u} + \nabla p = \mathbf{f}, \tag{5.1}$$

where $\mathbf{u}$ is the velocity, p is the pressure, $\mathbf{f}$ is a known source term, ρ is the fuild pressure, and η is the dynamic viscosity [17]. This is called the Navier-Stokes equation and can not be solved directly because the convective $\nabla \cdot (\mathbf{u} \otimes \mathbf{u})$ term is nonlinear. This means that numerical methods have to be used to solve this equation. However, there are two unknowns ($\mathbf{u}$ and p), while there is only one differential equation. This means that a second assumption is needed in order to use numerical methods for this system. The second assumption is based on fluids being incompressible, which means that the net flow (inflow - outflow) over every control volume must be zero. This incompressibility condition is given by

$$\nabla \cdot \mathbf{u} = \mathbf{0}. \tag{5.2}$$

Equation 5.2 is also called the continuity equation. When the continuity equation is combined with Equation 5.2, they are called the incompressible Navier-Stokes equations.

The incompressible Navier-Stokes equations are time-dependent, so time integration methods are needed to find the transient solution. Similar to the previous chapters, the Backward Euler algorithm is chosen. This time, however, the system is linearized first and then plugged into the Backward Euler algorithm, eliminating the need for Newton-Raphson. This was done because performing a full Newton-Raphson cycle in each time-step would take up too much computing power.

Because this incompressible Navier-Stokes equation deals with the interacting pressure and momentum systems, it makes sense to solve it as a coupled system. To transform the incompressible Navier-Stokes equation into a coupled system, it was spatially discretized using a discontinuous Galerkin method finite element. This

discretized system was then implemented in the 'DGFLOW' code at the RPNM department of the TU Delft. The system has a corresponding mass matrix **M**. For more information about the mass matrix and discontinuous Galerkin methods, the reader is referred to [17]. The linearized incompressible Navier-Stokes equations are then given by:

$$\mathbf{M}\frac{\partial \mathbf{u}}{\partial t} + \mathbf{N}(\mathbf{u})\mathbf{u} - \mathbf{S}\mathbf{u} + \mathbf{D}^T\mathbf{p} = \mathbf{f} \tag{5.3}$$

$$\mathbf{D}\mathbf{u} = \mathbf{g}, \tag{5.4}$$

where **g** is used to impose the inflow boundary conditions, **S** respresents the linearized viscous term, **N**(**u**) represents the linearized convective term, and **D** represents the discretized divergence which is given by Equation 2.6. Note that the transposed discretized divergence matrix is the discretization of the gradient matrix. Furthermore, the pressure **p** is now a vector, as it now represents the pressure at multiple elements. By writing this system down as a coupled system, the following relation is found:

$$\begin{bmatrix} \mathbf{M}\frac{\partial \mathbf{u}}{\partial t} \\ 0 \end{bmatrix} + \begin{bmatrix} \mathbf{A}(\mathbf{u}) & \mathbf{D}^T \\ \mathbf{D} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \mathbf{g} \end{bmatrix}, \tag{5.5}$$

where $\mathbf{A}(\mathbf{u}) = \mathbf{N}(\mathbf{u}) - \mathbf{S}$.

To solve **u** and **p** the Backward Euler method is chosen as time integration method in this thesis. The first step of rewriting the system into the Backward Euler form is to discretize the coupled incompressible Navier-Stokes system temporally:

$$\begin{bmatrix} \mathbf{M}\frac{\mathbf{u^{n+1}}-\mathbf{u^n}}{\Delta t} \\ \mathbf{0} \end{bmatrix} + \begin{bmatrix} \mathbf{A}(\mathbf{u}^n) & \mathbf{D}^T \\ \mathbf{D} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{u^{n+1}} \\ \mathbf{p^{n+1}} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \mathbf{g} \end{bmatrix}. \tag{5.6}$$

By taking adding $\frac{\mathbf{u}^n}{\Delta t}$ to both sides, the following relation is found:

$$\begin{bmatrix} \mathbf{M}\frac{\mathbf{u}^{n+1}}{\Delta t} \\ \mathbf{0} \end{bmatrix} + \begin{bmatrix} \mathbf{A}(\mathbf{u}^n) & \mathbf{D}^T \\ \mathbf{D} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{u^{n+1}} \\ \mathbf{p^{n+1}} \end{bmatrix} = \begin{bmatrix} \mathbf{f} + \mathbf{M}\frac{\mathbf{u}^n}{\Delta t} \\ \mathbf{g} \end{bmatrix}. \tag{5.7}$$

The final Backward Euler linearized system is then found by combining the terms in the LHS:

$$\begin{bmatrix} \mathbf{F}(\mathbf{u}^n) & \mathbf{D}^T \\ \mathbf{D} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{u^{n+1}} \\ \mathbf{p^{n+1}} \end{bmatrix} = \begin{bmatrix} \mathbf{f} + \mathbf{M}\frac{\mathbf{u}^n}{\Delta t} \\ \mathbf{g} \end{bmatrix}. \tag{5.8}$$

where the $\mathbf{F}(\mathbf{u}^n)$ is defined as $\mathbf{A}(\mathbf{u}^n) + \frac{\mathbf{M}}{\Delta t}$. Equation 5.8 is referred to as a saddle-point problem, because of the zero block in the lower right corner. Furthermore, it is a coupled system as it consists of a momentum and pressure subsystem, which both interact with each-other.

5.2. Fieldsplit

All the simulations in this thesis are written using the PETSc library. In PETSc, all coupled systems are solved using the fieldsplit preconditioning context. Even though fieldsplit is technically implemented as a preconditioner in PETSc, it is much more than simply a matrix which conditions the system [9]. Fieldsplit is a context which gives the user much flexibility in how the coupled system is solved. It allows the user to chose the linear solver, preconditioning type and tolerances for both the outer, as well as the inner systems. This offers many different variables to optimize. The goal of the remainder of this chapter is to find the optimal fieldsplit settings for the incompressible Navier-Stokes problem.

5.3. Navier-Stokes Model problem

The coupled system which is being solved comes from a turbulent backward-facing step flow problem, where the fluid has a density of $\rho = 1\ \frac{kg}{m^3}$. As boundary condition, the velocity at the walls is zero, the inflow is uniform, and the outflow is given normal access. A schematic of the backward step flow problem is given in Figure 5.1. The incompressible Navier-Stokes equation is used to find the velocity in the x and y direction for each point inside the pipe. The mesh that was used is given in Figure 5.2. The initial conditions for the velocity in the x and y directions shown in Figure 5.3 and 5.4.

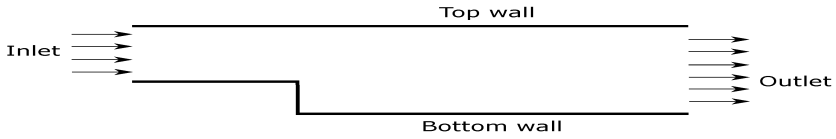


Figure 5.1: Schematic drawing of the backward step flow problem.

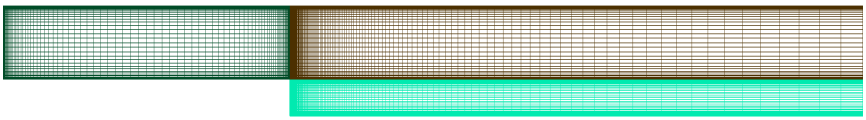


Figure 5.2: The edges of the 2D mesh used for the finite element method.

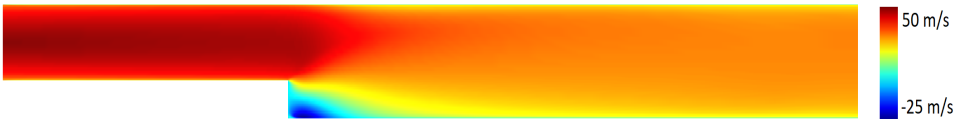


Figure 5.3: The velocity in the x direction of the turbulent flow in the backward step problem.

Figure 5.4: The velocity in the y direction of the turbulent flow in the backward step problem.

In each simulation for this problem, a total of 10 time-steps are computed, each time-step of $\Delta t = 1$ ms long.

The goal of solving this problem is not to investigate the solution itself, but instead the efficiency of the methods that are used to solve it. In the next subsections, two different methods are discussed to solve the problem for 10 Backward Euler time-steps.

5.4. Pressure-correction

All fieldsplit methods are compared with a very efficient algorithm called pressure-correction. By comparing them with the pressure-correction method, all simulations can be tested fairly. The pressure-correction algorithm is based on approximating the momentum $\hat{\mathbf{u}}^{m+1}$ for the next time-step by using the current pressure $\mathbf{p}^m$. The approximated momentum is then used to find the next pressure value $\mathbf{p}^{m+1}$, which is in turn used to calculate the next momentum step $\mathbf{u}^{m+1}$. It is not important to explicitly know how the pressure-correction method works for this thesis. If the reader is interested in this method, they are referred to [17].

The pressure-correction method only has one iteration per time-step and thus is expected to be very fast. The computation time of the pressure-correction method (45 ± 2 s) is used as a reference point for the relative computation time of the entire backward step simulation. Most of the computation time (over 90%) was used to solve the linear system, so any improvements in the efficiency of the linear solver should make a significant improvement in computational time.

5.5. Schur Factorization

The inverse of the saddle-point system from Equation 5.8 is given by [18][1]

$$\begin{bmatrix} \mathbf{F^{-1}} - \mathbf{F^{-1}D^T}(\mathbf{DF^{-1}D^T})^{-1}\mathbf{DF^{-1}} & \mathbf{F^{-1}D^T}(\mathbf{DF^{-1}D^T})^{-1} \\ (\mathbf{D^TF^{-1}D^T})^{-1}\mathbf{DF^{-1}} & -(\mathbf{DF^{-1}D^T})^{-1} \end{bmatrix}. \tag{5.9}$$

The inverted matrices can not be stored explicitly as they are dense, so solving the saddle-point system this way requires solving 13 subsystems. This is very computationally expensive, so a block algorithm is required to solve the saddle-point system efficiently.

For the 2x2 saddle-point coupled system that arises from the incompressible Navier-Stokes equation, the Schur complement method can be used as a very efficient block algorithm [19]. All of the Schur complement methods are based on the Schur decomposition, which is given by

$$\begin{bmatrix} \mathbf{F(u^n)} & \mathbf{D}^T \\ \mathbf{D} & \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{DF(u^n)^{-1}} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{F(u^n)} & \mathbf{0} \\ \mathbf{0} & \mathbf{S(u^n)} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{F(u^n)^{-1}D}^T \\ \mathbf{0} & \mathbf{I} \end{bmatrix}, \tag{5.10}$$

where $\mathbf{S}(\mathbf{u}^n)$ is the Schur complement given by

$$\mathbf{S(u^n)} = -\mathbf{DF(u^n)^{-1}D^T}. \tag{5.11}$$

The Schur decomposition consists of a lower triangular, diagonal, and upper triangular block matrix, all of which are computationally inexpensive to solve compared to a full 2x2 block system [18]. Inverting the Schur decomposed system in a block-wise fashions results in

$$\begin{bmatrix} \mathbf{F(u^n)} & \mathbf{D}^T \\ \mathbf{D} & \mathbf{0} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{I} & -\mathbf{F(u^n)^{-1}D^T} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{F(u^n)}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{S(u^n)}^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{DF(u^n)^{-1}} & \mathbf{I} \end{bmatrix}. \tag{5.12}$$

By solving the Schur decomposition of the saddle-point system, the focus changes of solving the coupled system as a whole to solving the **S** and **F** subsystems. Both of these subsystems can be solved independently, so this is done in parallel. The solution of the Schur decomposed system is then passed to an 'outer' linear solver to make sure that the solution of the coupled system as a whole satisfies the stop criteria. The goal for this part of the thesis is to find the optimal method to solve the Schur factorized system.

5.6. Schur approximation

In practice, the Schur complement is not computed using the inverse of momentum subsystem $\mathbf{F} = \mathbf{A} + \frac{\mathbf{M}}{\Delta t}$ when solving a transient problem, but instead by using an approximation of **F**. This approximation is based on **F** being almost completely defined by $\frac{\mathbf{M}}{\Delta t}$, if a small Δt is used. Meaning that the $\mathbf{F}^{-1}$ term in the Schur complement can be approximated with $\mathbf{M}^{-1}$, resulting in the approximated Schur complement

$$\hat{\mathbf{S}} = -\Delta t\, \mathbf{DM}^{-1}\mathbf{D}^T. \tag{5.13}$$

There are two main reasons why this approximation was used. Firstly, for the approximated Schur complement $\hat{\mathbf{S}}$, the inverted mass matrix $\mathbf{M}^{-1}$ only needs to be calculated once in the entire simulation, since **M** is constant. For the real Schur complement $\mathbf{S} = -\Delta t\, \mathbf{DF}^{-1}\mathbf{D}^T$, the momentum matrix **F** changes in each time-step, so **F** needs to be inverted at every time-step to compute the exact Schur complement. A lot of computing power is saved by not inverting the momentum subsystem **F** every time-step. Secondly, **M** is a symmetrical matrix which makes $\hat{\mathbf{S}}$ symmetrical as well. This makes the approximated Schur complement easier to invert, since the very efficient CG algorithm can be used for symmetrical systems.

Besides the Schur complement being approximated, it is also possible to build the preconditioner of the Schur complement using the Schur approximation. Similar to using the approximation for the Schur complement, using the approximation for the preconditioner means that **F** does not need to be inverted every time-step in order to build the preconditioner.

[1] A simpler inverse is available if **D** is a square matrix, but this is not the case in the incompressible flow simulation.

Approximation used for	rel time	avg. iters outer (**F**, **S**)
complement & preconditioning*	$1.4 \pm .1$	1.1 (32.9, 2562)
only complement	$2.5 \pm .1$	1.0 (23.9, 1264)
only precondition	442 ± 2	1.0 (1.6e5, diverged)
none	291 ± 2	1.0 (3.6e4, diverged)

Table 5.1: Results of the default case using the Schur approximation for the complement **S** and for the preconditioning of the Schur complement. All relative times are relative to the pressure-correction simulation. *default case.

5.7. The default case

Using a different linear solver, preconditioner, or tolerance on a system can change the efficiency with which the problem is solved drastically. So it is well worth the effort to try and find out what methods work best for each of the subsystems of Schur decomposed system.

In order to get an overview of which options work best for solving the back-step problem, a default set of options is created. The goal of these default settings is to see what effect each option has on this default scenario, which allows each option to be compared fairly. The default options from PETSc are chosen as the default case, which means:

- The linear solver for the outer system is FGMRES
- The outer system is solved with a relative tolerance $\tau_r = 10^{-4}$
- The Schur complement **S** and the Schur preconditioner are approximated with $\hat{\mathbf{S}}$
- The preconditioning is based on the full Schur decomposition
- Each subsystem is solved using GMRES with block Jacobi as preconditioner method
- Each subsystem is solved with a relative tolerance $\tau_r = 10^{-4}$

The default case had a relative computation time of 1.4 ± 0.1 compared to the pressure-correction method. In the 10 time-steps (of $\Delta t = 1$ ms) an average of 1.1 iterations were used to solve the outer (coupled) system, an average of 32.9 iterations for the momentum subsystem, and an average of 2,562 iterations for the pressure subsystem. An important note is that whenever a reference is made to the average amount of iterations per time-steps, this is defined as:

$$\frac{\#\text{total iterations used for system } x}{\#\text{time-steps}}. \tag{5.14}$$

So if an inner system is solved multiple times in one time-step, the average amount of iterations for that system will be higher.

In Subsections 5.8.1 - 5.8.7, each option of the default case except the first two are tested, to see which options work best for the backward step problem. The FGMRES method is always chosen as solver for the outer system because PETSc does not offer a feasible alternative. The outer tolerance is not changed because there is no optimum for this value, as explained in Chapter 5.8.6.

In order to fairly compare all the cases with each other, it had to be checked that all the results were the same (within a reasonable limit). As a measure for the correctness of each method, it was demanded that the maximum change in momentum at the 10'th time-steps had to be within 1% of the maximum momentum change in the 10'th time step of the default case. The pressure-correction method was within the 1% limit. For all of the simulations in the next subsections, this demand was met.

5.8. Testing the Schur approximations

To see how using the Schur approximations affects the computational time, four cases were tested. The cases consisted of; $\hat{\mathbf{S}}$ being used for both the preconditioning and for the Schur complement, $\hat{\mathbf{S}}$ is only used to replace the Schur complement, $\hat{\mathbf{S}}$ is used only for the preconditioning, and finally where $\hat{\mathbf{S}}$ is not used at all. The results of these simulations are given in Table 5.1. From which it is clear the Schur complement must

	rel time	avg iters outer ($\mathbf{F}$, $\hat{\mathbf{S}}$)
full*	1.4 ±.1	1.1 (32.9, 2,562)
diagonal	3.8 ±.2	4.9 (131 , 6,779)
upper	1.5 ±.1	1.1 (15.7, 2,353)
lower	2.3 ±.1	3.9 (93.7, 4,931)

Table 5.2: Performance of different Schur preconditioners for the back-step linearized system. The time is relative to the computation time of the pressure correction method (45 ± 2 s). The average amount of iterations refers to the average amount per per time-step to solve each respective system. *default case.

be approximated in order to get an acceptable computational time. Whenever the real Schur complement was used for the pressure subsystem, the pressure subsystem diverged. Note that divergence meant that the pressure subsystem did not reach the tolerance within 10,000 iterations. The momentum subsystem took a huge amount of iterations per time-step because it had to be inverted many times for each pressure iteration. The huge amount of momentum iterations completely bottle-necked the simulation, resulting in a very slow computational time.

When the approximated Schur complement was used, it is favourable to also use this approximation for preconditioning the Schur subsystem. This preconditioning made all systems converge with more iterations, but it reduced the computational time by 44%.

Even though the simulations with the real Schur complement had a diverged pressure solution, the maximum change in momentum at the 10'th iteration was still within 1% of the default case.

5.8.1. Schur factorization preconditioners

In this subsection, four different types of Schur factorized preconditioners are discussed for the Schur approximated system: full, diagonal, lower, upper. These preconditioners are all at least partially based on the full Schur preconditioner, which is the inverse of Equation 5.10 using the approximated Schur complement. The full Schur preconditioner matrix is given by

$$\begin{bmatrix} \mathbf{I} & -\mathbf{F}^{-1}\mathbf{D}^T \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{F}^{-1} & \mathbf{0} \\ \mathbf{0} & \hat{\mathbf{S}}^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{D}\mathbf{F}^{-1} & \mathbf{I} \end{bmatrix}. \tag{5.15}$$

This 'preconditioner' is the exact inverse of the linearized coupled system in Equation 5.8, meaning that it acts as a direct solver when applied to the system.

The diagonal Schur preconditioner consists of only the diagonal matrix of the full Schur preconditioner, resulting in the following preconditioner matrix:

$$\begin{bmatrix} \mathbf{F}^{-1} & \mathbf{0} \\ \mathbf{0} & \hat{\mathbf{S}}^{-1} \end{bmatrix}. \tag{5.16}$$

Unlike the full Schur preconditioner, this preconditioning matrix does not work as a direct solver, which means that the outer block still needs more than one iteration to be solved using this preconditioner.

The upper and lower Schur preconditioners are made by taking only the upper or lower triangular part of the full preconditioner. This results in the following matrices, for the upper and lower preconditioners, respectively:

$$\begin{bmatrix} \mathbf{F}^{-1} & -\mathbf{F}^{-1}\mathbf{D}^T\hat{\mathbf{S}}^{-1} \\ \mathbf{0} & \hat{\mathbf{S}}^{-1} \end{bmatrix}, \quad \begin{bmatrix} \mathbf{F}^{-1} & \mathbf{0} \\ -\hat{\mathbf{S}}^{-1}\mathbf{D}\mathbf{F}^{-1} & \hat{\mathbf{S}}^{-1} \end{bmatrix} \tag{5.17}$$

Each of these Schur preconditioner methods were tested. An overview of how well the different preconditioners performed is given in Table 5.2.

From a theoretical point of view, the full, triangular and diagonal preconditioners should ensure that the outer system is solved within 1, 2 and 4 iterations respectively [9]. However, Table 2.13 shows values which

	rel time	avg. iters outer (**F**)
Block Jacobi*	1.4 ± 0.1	1.1 (32.9)
ASM	1.4 ± 0.1	1.0 (23.9)

Table 5.3: Comparison of the ASM and Block Jacobi preconditioner for solving the momentum subsystem **F** in parallel. The time is relative to the (45 ± 2 s) of the pressure-correction method, and the average amount of iterations is the average over all time-steps. * default case.

	rel time	avg. iters outer (**F**)
Block Jacobi / ILU(0)	1.3 ± 0.1	1 (23.9)

Table 5.4: Performance of using the Block Jacobi and ILU(0) preconditioning methods for solving the momentum subsystem. The block Jacobi and ILU(0) methods were identical to each other. The time is relative to the computation time of the sequentially computed pressure-correction method (83 ± 3 s). The average amount of iterations refers to the total average over all time-steps.

exceed this theoretical limit. The reason why the actual average is higher is because the subsystems are not solved exactly, and thus have an error that is within the relative tolerance of $\tau_r = 10^{-4}$. This error makes it possible for the outer block to converge slower than is theoretically predicted.

Furthermore it should be noted that the full Schur preconditioner computes the inverse of the momentum subsystem **F** twice to produce the preconditioner. The momentum subsystem is solved twice due to the way PETSc computes the full Schur preconditioner:

$$\begin{bmatrix} inv(\mathbf{F}) & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & -\mathbf{D}^T \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & inv(\hat{\mathbf{S}}) \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{D} * inv(\mathbf{F}) & \mathbf{I} \end{bmatrix}, \tag{5.18}$$

where "$inv()$" represents computing the action of the inverse [9].

All other preconditioners computed the inverse of **F** and $\hat{\mathbf{S}}$ only once to construct the preconditioner.

5.8.2. Preconditioners for the momentum subsystem

In order to solve the coupled system, the momentum subsystem needs to be solved individually. Each iteration takes up a lot of computational resources, which makes it very important to solve this subsystem efficiently. Just like with the model problem of Section 3.1, preconditioning could play a huge role in ensuring that the system is solved efficiently.

However, changing the preconditioner for the subsystems is more difficult than changing the other options from the default case. This is due to only the block Jacobi and Additive Schwarz Method (ASM) preconditioning algorithms being available when the systems are solved in parallel, which is generally the fastest way to solve the coupled system. This leaves two options:

1. Solve the coupled subsystems in parallel with Block Jacobi/ASM preconditioning
2. Solve the coupled system sequentially and try out better types of preconditioners

Option one is explored first. The simulation is performed using the Block Jacobi and ASM preconditioning. The results are given in Table 5.3. The ASM preconditioning method did manage to solve the momentum system using fewer iterations than the block Jacobi preconditioning method. At the same time, the block Jacobi algorithms used less computational time to be made and applied, resulting in the computational time of the two preconditioners being the same. It is interesting to note that the ASM preconditioning method caused the outer system to be solved in a single iteration, while the block Jacobi preconditioning did not.

For the second option, two preconditioner types were tested for the sequential simulation: Block Jacobi and Incomplete LU-factorization ('ILU(0)'). The Incomplete Cholesky preconditioner was not tested because the momentum subsystem is asymmetrical. The Block Jacobi and ILU(0) preconditioning method gave the same results shown in Table 5.4. By comparing results of the Block Jacobi method for the parallel and sequential simulations, it becomes clear that this preconditioner acts different depending on if the code is run in parallel or sequentially. This is due to the Block Jacobi preconditioner being based on the complete system when it is used in parallel, and only on the individual subsystems when it is used for the sequential case. Furthermore,

	rel time	avg. iters outer ($\hat{\mathbf{S}}$)
Block Jacobi*	1.4 ± 0.1	1.1 (1562)
ASM	1.4 ± 0.1	1.1 (1410)

Table 5.5: Comparison of the ASM and Block Jacobi preconditioner for solving the pressure subsystem $\hat{\mathbf{S}}$ in parallel. The time is relative to the (45 ± 2 s) of the pressure-correction method, and the average amount of iterations is the average over all time-steps. * default case.

	rel time	avg. iters outer ($\hat{\mathbf{S}}$)
Block Jacobi / ILU(0)	1.3 ± 0.1	1.0 (1264)
ICC	1.5 ± 0.1	1.0 (1604)

Table 5.6: Performance of using the Block Jacobi, ILU(0) and ICC preconditioning methods for solving the pressure subsystem. The ILU(0) and Block Jacobi preconditioners acted identically. The time is relative to the computation time of the sequentially computed pressure correction method (83 ± 3 s). The iterations represent the average amount of iterations over all time-steps.

it should be noted that when the sequential results are compared to the computation time of the parallel pressure-correction method (using two processors), they have a relative computational time of $2.3 \pm .1$ s. This clearly indicates that it is not worth it to run the simulation sequentially in order to use better preconditioning methods for the momentum subsystem.

5.8.3. Preconditioners for the pressure subsystem

The choices for preconditioning methods for the pressure subsystem are bound to the same restrictions as the momentum subsystem. The two options (sequential and parallel), are again tested for the pressure subsystem.

The Block Jacobi and ASM preconditioning methods were again the only preconditioning methods for the parallel case, so only these two are tested. The results are given in Table 5.5. Using the Block Jacobi and ASM preconditioner for the pressure subsystem resulted in the same computational time. When used for the pressure subsystem, the ASM preconditioning method did not cause the outer system to always converge in one iteration, as it did with the momentum subsystem.

For the sequential simulation, three preconditioning methods were tested: block Jacobi, ILU(0) and ICC. The ILU(0) and block Jacobi preconditioners gave the same results again, similar to when they were used for the momentum subsystem. The results of the three preconditioning methods are given in Table 5.6. The ICC preconditioning method did not manage to outperform the ILU(0) or block Jacobi preconditioning methods. The relative computational time of the sequential simulation, compared to the parallel pressure correction method (using two processors), was again worse than the default case. For both subsystems, it is now shown that using any of the mentioned preconditioning methods does not save enough time to make running the code sequentially worth it.

5.8.4. Linear solvers for the momentum subsystem

The momentum subsystem is a non-symmetrical, non-positive definite system. This subsystem is very diagonally dominant due to the small time-step of $\Delta t = 1$ ms, which makes it easy for most iterative methods to solve. Because of the asymmetric momentum matrix, the very efficient CG algorithm could not be used. As an alternative, an extension to the CG method called the Biconjugate Gradient Stabilized method (BCGS) is tested in this subsection. In [20], it has been shown that the BCGS method is very efficient for the steady state incompressible Navier-Stokes system. In this subsection, it is tested if the BCGS solver outperforms GMRES for the transient case.

Because most other methods available in PETSc are either based on one of these methods or poorly suited for the problem, only the BCGS and GMRES algorithms are compared. The results of this comparison are shown in Table 5.7. The BCGS method had the same computational time as the GMRES method. In general, GMRES is one of the most efficient iterative linear solvers available. This implies that the BCGS method works well for the transient incompressible Navier-Stokes system, even though it did not outperform GMRES.

	rel time	avg iters outer (**F**)
gmres*	1.4 ± 0.1	1.1 (32.9)
bcgs	1.4 ± 0.1	1.0 (14.9)

Table 5.7: Comparison of the BCGS and GMRES algorithm for solving the momentum subsystem **F**.

	rel time	avg iters for $\hat{\mathbf{S}}$
gmres*	1.4 ± 0.1	2562
cg	1.1 ± 0.1	551

Table 5.8: Comparison between the CG and GMRES algorithm solving the approximated Schur complement system.

5.8.5. Linear solvers for the pressure subsystem

As can be seen from the results of the default case, solving the pressure subsystem takes the most iterations by far. This makes the linear solver used for the pressure subsystem very important because much computational power is used to solve this system.

Because the approximated Schur complement is symmetrical and positive definite, the CG algorithm can be used. The CG algorithm is compared with the GMRES algorithm in table 5.8. Just as the theory predicts, the CG algorithm is the best solver for the approximated Schur subsystem.

5.8.6. Tolerances for the momentum subsystem

The tolerance on the outer solver is chosen such that the error in the final solution is acceptable. The problem with choosing the 'best' outer tolerance is the following. If the outer tolerance is chosen to be 'small', the linear solver needs more iterations (and thus time) to solve the system. Conversely, if the outer tolerance is chosen to be 'big', the system is solved quicker, but with relatively bad accuracy. This means that there is not a single 'best' outer tolerance because the trade-off between time and accuracy is subjective. Therefore, the outer tolerances were not investigated in this thesis.

The tolerance on the subsystems **F** and $\hat{\mathbf{S}}$ however, only determine the accuracy of the subsystem and not the accuracy of the final solution. This leads to the situation where the choice of subsystem tolerances only affects how fast the outer solver can compute the final solution, and how fast the inner solver can compute the local solution. This means that there is an unambiguous optimum for the tolerances set on the inner subsystems, which results in minimal computational time. The goal of this subsection is to find this optimal relative tolerance τ_r for the momentum subsystem. To see which tolerances resulted in the best computational time, tolerances from 10^{-6} to 10^{-1} were tested in steps of $10^{-\frac{1}{2}}$. The results are given in Figure 5.5. It shows that the default tolerance value of $\tau_r = 10^{-4}$ is already very close to the optimal tolerance $\tau_r = 10^{-4.5}$. To see why changing the tolerance for the momentum subsystem did not lead to any significant improvements, the amount of iterations that were used to solve the momentum and outer system are shown in Figure 5.6. It shows that the amount of outer iterations is very sensitive for the tolerance on the momentum subsystem. This meant that increasing the tolerance on the momentum subsystem made the outer system harder to solve, which in turn resulted in a larger computational time. Conversely, making the tolerance extremely small ($\tau_r = 10^{-6}$), caused the momentum subsystem to bottle-neck the process, which also resulted in a bad computational time. This resulted in the optimal tolerance for the momentum subsystem, being in between those two extremes at $\tau_r = 10^{-4.5}$. The peak in computational time and momentum iterations around $\tau_r = 10^{-3}$ could not be explained.

5.8.7. Tolerances for the pressure subsystem

This subsection has the goal to find the optimal relative tolerance for the pressure subsystem. Unlike the previous subsection, tolerances are tested in the range of 10^{-6} to 1, where $\tau_r = 1$ corresponds to not solving the pressure subsystem at all. The relative computational time with a variable tolerance τ_r on the pressure subsystem is shown in Figure 5.7.

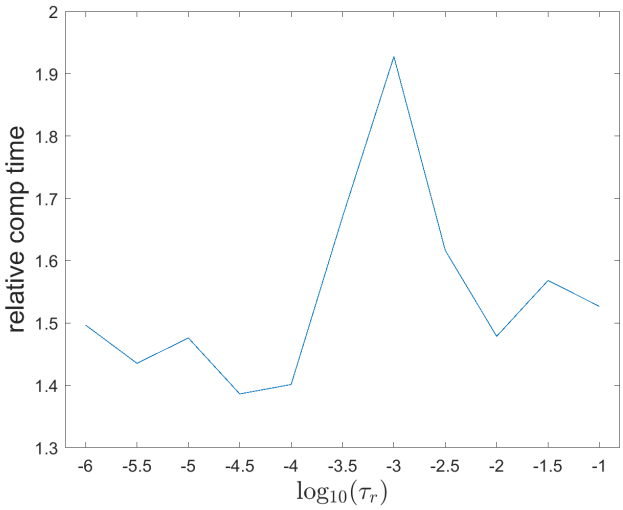


Figure 5.5: Relative computational time for the default case with a variable tolerance τ_r for the momentum subsystem.

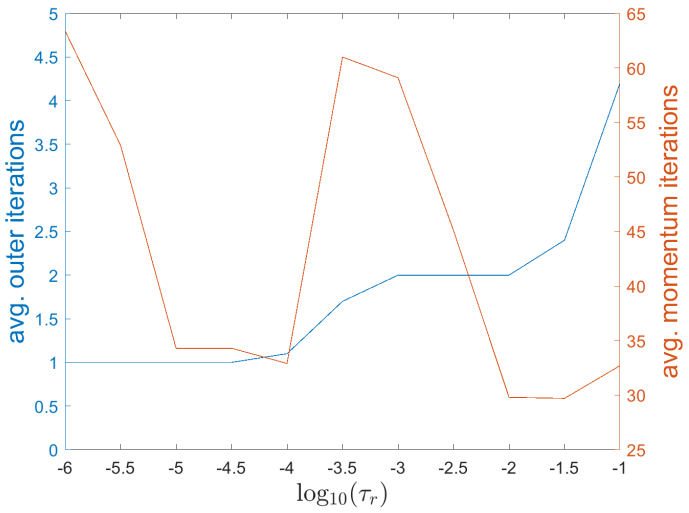


Figure 5.6: Average amount of iterations per time-step used to solve the momentum and outer system, with a variable tolerance τ_r on the momentum subsystem.

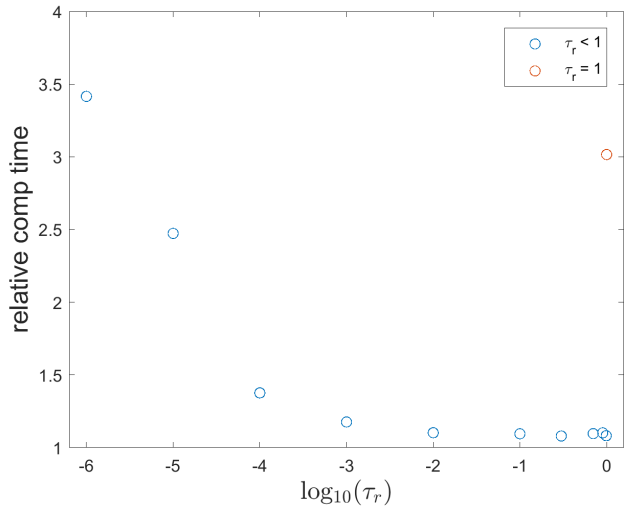


Figure 5.7: Computation time relative to the pressure correction algorithm of the default case with variable tolerance for the pressure subsystem. For $\tau_r < 1$, the average amount of outer iterations was 1.1 (the same as the default case), and for $\tau_r = 1$ the average amount of outer iterations was 6.5.

Surprisingly, the tolerance on the pressure system can be extremely big. When a tolerance bigger then 0.9 is used, the pressure subsystem almost always converges within one linear solver iteration, which makes the solution of the pressure subsystem very unreliable. Yet, The number of outer iterations did not increase in the simulation of 10 time-steps. Furthermore, the maximum change in momentum at the 10'th iteration was within the 1% limit from the default case. This suggest that the pressure subsystem does not need to be solved at all. However, when $\tau_r = 1$ is chosen as tolerance, the computational time increases significantly. To see why the system becomes harder to solve when the pressure becomes a constant ($\tau_r = 1$), the outer iterations per time-step are shown in Figure 5.8. For the first four iterations, the outer system only takes one iteration to be solved. This is because the actual pressure has not changed much, so the constant pressure is a good approximation. After more time has passed, the constant pressure and the actual pressure start to diverge, which results in the outer solver using more iterations to compensate for this discrepancy. This makes the computation time increase as more time-steps are computed, making this tolerance unfeasible. Furthermore, even though the momentum change at the 10'th iteration was within 1% of the default case, it seems very realistic that this method returns a wrong solution when more time-steps are simulated.

Figure 5.7 seems to suggest that $\tau_r = 0.99$ does not lead to the same problem, because the computation time very low, and the average amount of outer iterations is the same as in the default case. However, if the simulation is run for 1.000 time-steps, it becomes clear that $\tau_r = 0.99$ does, in fact, lead to an increase in outer iterations, as shown in Figure 5.9. The problem is now to find a relative tolerance for the pressure subsystem, that leads to a minimal computational time, whilst not resulting in an eventual increase in outer iterations. From Figure 5.7, it is seen that $\tau_r = 10^{-2}$ is the smallest tolerance which does not lead to a significant increase in computational time.

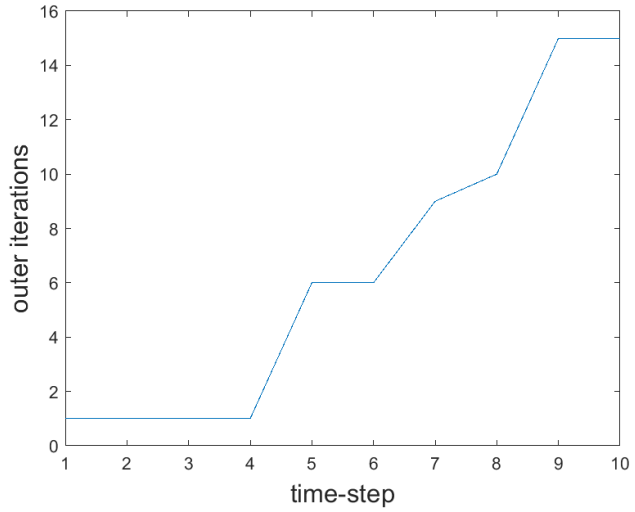


Figure 5.8: Iterations required to solve the outer incompressible navier-Stokes system, when the pressure subsystem is not solved ($\tau_r = 1$).

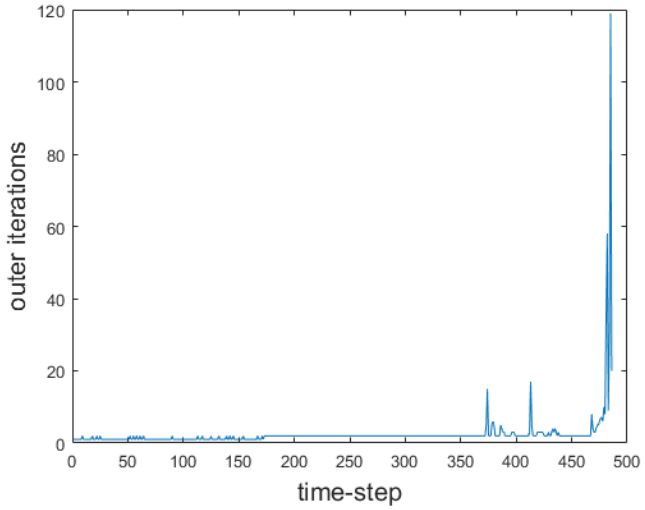


Figure 5.9: The amount of outer iterations needed per time-step for a coupled, incompressible Navier-Stokes problem. The pressure subsystem is solved with a relative tolerance of $\tau_r = 0.99$. Only the first 480 time-steps are shown since the later ones require too much computational time to be calculated.

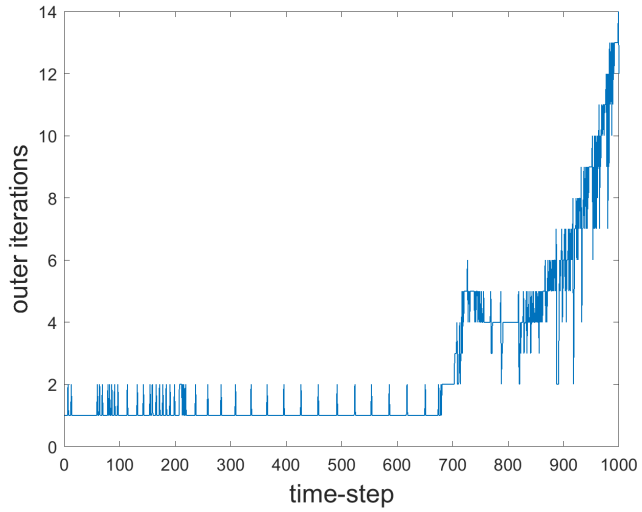


Figure 5.10: Number of iterations required to solve the outer system per time-step, with a tolerance of $\tau_r = 10^{-2}$ for the pressure subsystem.

A 1,000 step simulation is ran with $\tau_r = 10^{-2}$, to verify that this tolerance is small enough to prevent this increase in the outer iterations from happening. The results for the 1,000 time-steps simulation with $\tau_r = 10^{-2}$ are shown in Figure 5.10. Figure 5.10 shows that the outer system needs more iterations to converge in the later time-steps. This could be due to the tolerance being too big, but $\tau_r = 10^{-2}$ should be small enough to at least ensure that the pressure subsystem gets updated sufficiently in each time-step. To confirm that this increase is not due to the relative tolerance on the pressure subsystem, another 1.000 step simulation is run with $\tau_r = 10^{-4}$ as tolerance on the pressure subsystem. This simulation showed exactly the same behavior as in Figure 5.10, from which it is concluded that the increase in outer iterations after time-step 700 is not due to the tolerance on the pressure subsystem.

As a conclusion, $\tau_r = 10^{-2}$ is deemed to be the optimal relative tolerance for the pressure subsystem, even though the amount of outer iterations does increase after 700 time-steps.

5.8.8. Optimal Schur method

In conclusion, the optimal conditions that were found for the default case are:

- The Schur complement $\mathbf{S}$, and the Schur preconditioner is approximated with $\hat{\mathbf{S}}$
- Fully factorized Schur component
- The momentum subsystem is solved using GMRES with block Jacobi preconditioning
- The pressure subsystem is solved using CG with block Jacobi preconditioning
- $\tau_r = 10^{-4.5}$ as relative tolerance for the momentum subsystem
- $\tau_r = 10^{-2}$ as relative tolerance for the pressure subsystem

The simulation using the optimal Schur conditions resulted in a relative computational time of $1.0 \pm .1$, compared to the pressure-correction method. The average amount of iterations for the outer, momentum, and pressure systems were respectively 1.0, 34.4, and 123. Most of the gain in computational time was made by setting the tolerance on the pressure subsystem to $\tau_r = 10^{-2}$. The reason why the CG solver did not offer a significant computational time reduction is due to the large relative pressure tolerance. This large tolerance decreased the number of iterations spent on the pressure system significantly, which resulted in the CG method not having to solve many iterations.

It has now been shown that using the right settings the Schur complement method can perform equally well as the pressure-correction method.

6

Conclusions and recommendations

Choosing the right numerical method to solve a non-linear function is critical for ensuring a good efficiency. In order to get an overview of which methods work best for certain problems, different numerical methods were investigated. In the first part of this thesis, the Jacobian-free Newton-Raphson method has been compared with the regular Newton-Raphson method. Both methods have been used to solve the heat equation in a 1D rod. The regular Newton-Raphson method outperformed the JFNK method in both the steady state and in the transient case. Without preconditioning the regular Newton-Raphson method was 5 to 6 times as fast as the JFNK method for both the steady state and the transient case. Preconditioning has been proven to be very useful for the steady state case. The ICC and Jacobi preconditioning methods sped up the regular Newton-Raphson method by a factor of $(3.1 \pm .2) * 10^2$ and $(2.8 \pm .2)$ respectively. For the transient case, preconditioning was less effective. The ICC only sped up the regular Newton-Raphson method by a factor of $(5.6 \pm .3)$, while Jacobi preconditioning slowed the regular Newton-Raphson down by a factor of $(3.0 \pm .2)$. It has been concluded that the regular Newton-Raphson method should be used over the JFNK method, whenever the Jacobian can be calculated.

Next, the regular Newton-Raphson and JFNK methods have been tested for the 1D rod problem again, but this time as a multi-physics problem with the coupling terms missing. When using the regular Newton-Raphson method, the subsystems were solved using the GMRES linear solver with ILU(0) preconditioning. For the steady state multi-physics problem, the performance of the regular Newton-Raphson method was extremely poor. The JFNK method, however, was completely unaffected by the missing coupling terms. This caused the JFNK method to be (9.1 ± 0.3) times as fast as the regular Newton-Raphson method without preconditioning for the steady state multi-physics case. For the transient multi-physics rod, the regular newton-Raphson method broke down completely and was not able to compute the transient solution. The JFNK method was again unaffected by the missing coupling terms and gave the same solution as in the transient single-physics case. It was concluded that the JFNK method outperforms the regular Newton-Raphson method for coupled systems with missing coupling terms in both the steady state and the transient case.

Finally, an incompressible Navier-Stokes problem has been solved as a coupled system. This has been done using different parameters for the Schur complement method and different solver settings for the subsystems. Considerable efficiency has been gained by approximating the Schur complement with $\Delta t\ \mathbf{D}\mathbf{M}^{-1}\mathbf{D}^T$, where Δt is the time-step, $\mathbf{M}$ is the mass matrix from the discretization mesh, and $\mathbf{D}$ is the discretized divergence term. This worked especially well when the preconditioning for the Schur subsystem was also based on this approximation. The CG solver worked best for the Schur subsystem, while the GMRES solver worked best for the momentum subsystem. Only the tolerances on the inner subsystems have been tweaked in this thesis. Changing the tolerance on the momentum subsystem had a clear effect on the number of outer iterations. Making the tolerance bigger increased the number of outer iterations, and making the tolerance smaller decreased the number of outer iterations. This made the computational time relatively sensitive to the tolerance on the momentum subsystem. The optimal relative tolerance for the momentum subsystem has been found to be $\tau_r = 10^{-4.5}$. The tolerance on the pressure subsystem did not work in the same fashion. This tolerance could be increased significantly, without the outer solver needing to compensate with more iterations. Meaning that it was advantageous to use a relatively big tolerance of $\tau_r = 10^{-2}$ for the pressure

subsystem. Furthermore, the full Schur factorization has been found to be the best factorization type for the Schur complement method.

When all optimal settings were combined, the coupled system could be solved with a relative computational time of (1.0 ± 0.1) compared to the pressure-correction method, which was used as a benchmark.

For future research, there are three areas for improvement.

Firstly, the JFNK method has been implemented without preconditioning. This has been done because PETSc does not allow a matrix-free context to work with any preconditioning method since it does not have any matrix it can use to build the preconditioner. Theoretically, it is possible to apply a user-defined preconditioning matrix to all the Krylov subspace vectors, and manually precondition the system in this way. If this was done, the JFNK method could potentially be competitive with the regular Newton-Raphson method for non-coupled systems.

Secondly, there are still more combinations of options that could be tested out for the Schur complement method. Due to time restraints, the effect of each parameter was tested separately in this thesis. By doing this, it is possible that efficient combinations might have been missed. For example; it has not been tested how changing the tolerance for both subsystems affects the computation time.

Finally, the JFNK method could be applied to the incompressible Navier-Stokes problem. From the 1D model rod, it has been shown that the JFNK was very efficient for coupled problems. The JFNK method was not used to solve the incompressible backward step problem in this thesis because no preconditioner could be used.

Appendices

A. Calculating the rate of convergence

To approximate the rate of convergence, the following relation is used[5]:

$$p_n = \frac{\log(||\mathbf{x_{n+1}} - \mathbf{x_n}||_2) - \log(||\mathbf{x_n} - \mathbf{x_{n-1}}||_2)}{\log(||\mathbf{x_n} - \mathbf{x_{n-1}}||_2) - \log(||\mathbf{x_{n-1}} - \mathbf{x_{n-2}}||_2)}. \tag{1}$$

The average convergence rate p of a set of indexed S is calculated with:

$$p = \frac{1}{\#S} \sum_{n \in S} p_n, \tag{2}$$

where $\#S$ is the amount of elements in S.

B. PETSc implementation of 1D model problem

This Appendix shows how the 1D model problem was implemented using the PETSc toolkit in Fortran 90. Given the length of the code, only a summary of the steady state simulation with explaination is given.

PETSc should not be seen as its own coding language. Instead, PETSc should be seen as a Fortan 90 object. To work with this PETSc object, appropriate PETSc functions have to be called.

To start of, every PETSc code needs to be intialized (make the 'PETSc object') and import certain PETSc packages. A Scalable Nonlinear Equations Solvers ('SNES') object is used to solve the non-linear system with Newton-Raphson, so this need to be defined as well. All the initialization for the 1D model rod is done with the following lines:

```
!where '#main_name' is the name of the main program file.
program #main_name

    !import #package
    #include "petsc/finclude/#package.h"
    use #package

    !a separate module called 'base_mod' for constants which are accessed often is created.
    !the constants in this module are loaded with:
    use base_mod

    !define the type of each Variable (both PETSc and non PETSc).
    !define the type for the temperature vector x, boundary vector b, and function vector f.
    Vec :: x
    Vec :: b
    Vec :: f

    !define the Jacobian matrix J
    Mat :: J

    !the amount of grid points is defined as a parameter, which makes it unchangeable.
    PetscInt, parameter :: n = 1000

    !the initial temperature (K) of the entire rod is also defined as a parameter.
    petscInt, parameter :: initTemp = 600

    !define some constants
    PetscScalar :: val, b_left, b_right, eps

    !all the used PETSc objects
    SNES :: snes
    PetscErrorCode ierr

    !initialize PETSc
    call PetscInitialize(PETSC_NULL_CHARACTER,ierr);
```

```
...
```

The package used to solve the 1D rod is called 'petscsnes', so replace '#package' with 'petscsnes'.

Important: every variable, in every PETSc function, needs to be a PETSc variable and not a fortran variable. For example; use PetscReal instead of real in all PETSc functions.

Now that PETSc is initialized, the next goal is to make the function **F** as described in Chapter 3.1, so that the SNES objects can solve it. The vectorialized version of the heat equation **f** was constructed in the following way for the 1D rod simulation:

$$\mathbf{f} = A\mathbf{x} - \sigma\mathbf{x}^4, \tag{3}$$

where A is the discretized diffusion term, $\mathbf{x}$ is the temperature vector, σ is the Boltzmann constant.

Before this function can be made in Fortran, matrix A and vectors $\mathbf{x}, \mathbf{f}$ need to be defined. Since A is accessed by both the main program and the function which creates **f**, so it is initialized in the 'base_mod' module instead of in the main program. This is done in the following fashion:

```
module base_mod
    !is called since all variables will be defined
    implicit none

    !define the diffusion matrix A
    Mat :: A

    !define the boltzmann constant 'boltz', since it will be accessed by multiple functions
    PetscScalar, parameter :: boltz = 5.67E-8
    ...
end module
```

In the main program file, the diffusion matrix A, and the vectors **f**, and **x** can now be created:

```
...
!create the function vector 'f', temperature vector 'x', and diffusion matrix 'A'

!first the dimensions of 'x' are defined
call VecCreateSeq(PETSC_COMM_WORLD,n,x,ierr)

!set all values of 'x' to 'initTemp'
call VecSet(x,initTemp,ierr)

!assemble the vector x, this needs to be done.
call VecAssemblyBegin(x,ierr)
call VecAssemblyEnd(x,ierr)

!make 'f' have the same dimensions as 'x'
call vecDuplicate(x, f, ierr)

!create the diffusion matrix A (was only a type before)
call MatCreate(PETSC_COMM_WORLD,A,ierr)

!set the dimensions to n by n
call MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,n,n,ierr)

!set the values of the matrix
!loop over all rows
do i=1,n
    !loop over all columns
    do j=1,n
        !determine the diffusion term g of the i,j'th element.
```

```
        !this can be done with the methods in chapter 3.
        val = g(i,j)

        !set the i,j'th element to this value (i and j have to be PetscInt type)
        call MatSetValue(A, i, j, val, INSERT_VALUES, ierr)
    end do
end do

!Assemble the Matrix
call MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY,ierr)
call MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY,ierr)
...
```

In Fortran it is common to use subroutines instead of functions. This is because in Fortan a function can only have one output variable, which can not also be an input variable. A subroutine can have more then one output variable, which can also be an input variable. By PETSc convention, the subroutine for **f** is typically called 'FormFunction'. The subroutine to make **f** is defined for the 1D rod model as:

```
subroutine FormFunction(snes, x, f, ierr)

    !basis module containing all constants.
    !the diffusion term A, boltzmann constant sigma, and boundary condition b are used.
    use base_mod

    !4 is needed later, so we create a PetscScalar which will get the value 4.
    PetscScalar :: four

    !define the petsc objects
    PetscErrorCode ierr
    SNES snes

    !define the PETSc vectors f (function value), and x (temperature)
    Vec f, x

    !create another dummy vector for the diffusion
    Vec diff

    !-1 is needed later, so we create a PetscScalar which will get the value -1.
    PetscScalar negOneScal

    !set the appropriate values
    negOneScal = -1.0
    four = 4.0

    !since x is already defined, the properties of 'x' are duplictated into 'diff'
    call VecDuplicate(x, diff, ierr)

    !the current value of f is not used, so the values of 'x' are copied into 'f'
    call VecCopy(x, f, ierr)

    !make f -> f^4. note that f was x before this operation, so f = x^4 afterwards.
    call VecPow(f, four, ierr)

    !multiply f by the boltzmann constant to get f = boltz*x^4.
    call VecScale(f, boltz, ierr)

    !set the diffusion term 'diff' to Ax. So diff = Ax after this operation.
    call MatMult(A, x, diff, ierr)
```

```
    !subtract the radiation term from the diffusion term, resulting in f = Ax - boltz*x^4.
    call VecAYPX(f, negOneScal, diff, ierr)

    !since this is a subroutine, the final value of f is automatically returned.
    !note that the value of x remained unaltered in this subroutine.
end subroutine
```

Besides the function vector **f**, SNES also needs to know the Jacobian if not using the JFNK method. By PETSc convention, the function which makes the Jacobian is called 'FormJacobian', and for the 1D model simulation it was defined as:

```
subroutine FormJacobian(snes, x, jac, ierr)
    !import the base constants
    use base_mod

    !define the input variables.
    SNES :: snes
    PetscErrorCode :: ierr
    Mat :: jac
    Vec :: x

    !define some dummy vector elements.
    Vec :: elements

    !the value of 3 and -1 will be needed later.
    PetscScalar :: three, negOneScal

    !set the values of the constant variables.
    three = 3.0

    !give 'elements' the same properties and values as 'x'
    call VecDuplicate(x, elements, ierr)
    call VecCopy(x, elements, ierr)

    !multiply elements to the power 3, so elements = x.^3
    call VecPow(elements, three, ierr)

    !elements = -4*boltz*x^.3 (the derivative of -boltz*x.^4)
    call VecScale(elements, -4 * boltz, ierr)

    !copy the values of the diffusion A into the Jacobian jac
    call MatCopy(A, jac, DIFFERENT_NONZERO_PATTERN, ierr)

    !add the radiation term to get the final jacobian
    call MatDiagonalSet(jac, elements, ADD_VALUES, ierr)

    !assemble the jacobian, this must be done
    call MatAssemblyBegin(jac,MAT_FINAL_ASSEMBLY,ierr)
    call MatAssemblyEnd(jac,MAT_FINAL_ASSEMBLY,ierr)
end subroutine
```

As a final step before the 1D rod model can be solved, the boundary conditions vector **b** has to be imposed. This is done with the following code in the main program:

```
!give the boundary condition vector the same type as the heat vector
call VecDuplicate(x, b, ierr)

!set the all elements in the boundary vector to 0 (used 'boltz*0' because just '0' would
    not be of type PetscInt).
```

```
call VecSet(b,boltz*0,ierr);

!Set the value for the left boundary condition (using the discretization in chapter 3)
call VecSetValue(b, 0, b_left, INSERT_VALUES, ierr)

!Set the value for the right boundary condition (using the discretization in chapter 3)
call VecSetValue(b, n-1, b_right, INSERT_VALUES, ierr)

!assemble the boundary vector, this needs to be done
call VecAssemblyBegin(b,ierr)
call VecAssemblyEnd(b,ierr)
```

If it is chosen to solve the heat equation using the regular Newton-Raphson method, this can be done with the following code in the main program:

```
...
!allows the user to set the preconditioner type, linear solver type, etc from the
!command line. e.g. '-ksp_type cg -pc_type icc'
call SNESSetFromOptions(snes, ierr)

!tell SNES to use FormFunction to compute f
call SNESSetFunction(snes,f,FormFunction,PETSC_NULL_CHARACTER,ierr)

!give the Jacobian J the same dimensions as the diffusion matrix A.
call MatDuplicate(A, MAT_SHARE_NONZERO_PATTERN, J, ierr)

!tell SNES to use FormJacobian to compute the Jacobian J
call SNESSetJacobian(snes,J,J,FormJacobian,PETSC_NULL_CHARACTER, ierr)

!solves the heat profile such that F(x) = b, using Newton-Raphson
call SNESSolve(snes, b, x, ierr)

!'x' should now be the steady state heat profile
...
```

If it is chosen to solve the heat equation using the JFNK method, this can be done with the following code in the main program:

```
...
!tell SNES to use FormFunction to compute f.
call SNESSetFunction(snes,f,FormFunction,PETSC_NULL_CHARACTER,ierr)

!create the matrix free SNES environment.
call MatCreateSNESMF(snes, J, ierr)

!set the epsilon value in the JFNK approximation.
eps = 4.7E-4

!pass this value into the JFNK SNES context.
call MatMFFDSetFunctionError(J, rerror, ierr);

!tell SNES that the Jacobian is approximated using JFNK.
call SNESSetJacobian(snes, J, J, MatMFFDComputeJacobian ,PETSC_NULL_CHARACTER, ierr)

!solves the heat profile such that F(x) = b, using JFNK
call SNESSolve(snes, b, x, ierr)

!'x' should now be the steady state heat profile
...
```

Finally, it is very useful to be able to control certain aspects of the code from the command line. For example, the following line can be used to set the PetscInt a to x by typing 'program_name -a x' in the command line:

```
call PetscOptionsGetInt(PETSC_NULL_OPTIONS,PETSC_NULL_CHARACTER, '-a', a, flg, ierr)
```

This concludes the summary of the PETSc implementation for the 1D rod model problem.

Bibliography

[1] Bartosz Bartczak, Jacek Mucha, and Tomasz Trzepieciński. Stress distribution in adhesively-bonded joints and the loading capacity of hybrid joints of car body steels for the automotive industry. *International Journal of Adhesion and Adhesives*, 45:42–52, 2013.

[2] HK Moffatt et al. *Numerical methods in finance*, volume 13. Cambridge University Press, 1997.

[3] Olivier Verdier. *Benchmark of femlab, fluent and ansys*. Univ., 2004.

[4] Yan-Bin Jia. Roots of polynomials. *Com S*, 477:577, 2017.

[5] Cornelis Vuik, P Van Beek, F Vermolen, and J Van Kan. *Numerical Methods for Ordinary differential equations*. VSSD, 2007.

[6] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[7] Henk A Van der Vorst. *Iterative Krylov methods for large linear systems*, volume 13. Cambridge University Press, 2003.

[8] Richard Barrett, Michael W Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*, volume 43. Siam, 1994.

[9] Satish Balay, Shrirang Abhyankar, Mark Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, W Gropp, et al. *PETSc users manual*. Argonne National Laboratory, 2019.

[10] Dana A Knoll and David E Keyes. Jacobian-free newton–krylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193(2):357–397, 2004.

[11] Dion Koeze. A study of possible applications for jacobian-free newton krylov methods in nuclear reactor physics. Technical report, TU Delft, 2009.

[12] Paul A Gagniuc. *Markov chains: from theory to implementation and experimentation*. John Wiley & Sons, 2017.

[13] John Charles Butcher and Nicolette Goodwin. *Numerical methods for ordinary differential equations*, volume 2. Wiley New York, 2008.

[14] Bruce M Maggs, Gary L Miller, Ojas Parekh, R Ravi, and Shan Leung Maverick Woo. Solving symmetric diagonally-dominant systems by preconditioning. 2002.

[15] Robert F. Mudde Harry van den Akker. *Transport phenomena*. Delft Academic Press, 2014.

[16] C. Vuik. *Iterative solution methods*. J.M. Burgercentrum, 2019.

[17] A. Segal. *Finite element methods and Navier-Stokes equations*, volume 22. Springer Science & Business Media, 1986.

[18] Tzon-Tzer Lu and Sheng-Hua Shiou. Inverses of 2× 2 block matrices. *Computers & Mathematics with Applications*, 43(1-2):119–129, 2002.

[19] Barry Smith, Petter Bjorstad, and William Gropp. *Domain decomposition: parallel multilevel methods for elliptic partial differential equations*. Cambridge university press, 2004.

[20] Mehfooz Rehman, Cornelis Vuik, and Guus Segal. A comparison of preconditioners for incompressible navier–stokes solvers. *International Journal for Numerical methods in fluids*, 57(12):1731–1751, 2008.