Implementation of Preconditioned CG solver for FEM analysis on neutron transport

Marcel M.F.W. Wijnen
January 2009

Supervisor D. Lathouwers

Delft University of Technology
Department of Radiation, Radionuclides and Reactors
Mekelweg 15 · 2629 JB · Delft · Netherlands

Abstract

The solution of the first order Boltzmann equation that describes neutron transport in a nuclear reactor can be found by means of the least squares method. This report gives an outline on how the least squares method can be used to set up an FEM analysis and describes how the linear system coming from this analysis can be solved with the preconditioned conjugate gradient method by making use of the SPARSKIT [3] tool package. Finally the solver is tested on a test case coming from 1D FEM analysis on neutron transport.

Contents

Chapter 1

Introduction

In reactor physics one of the main topics researchers work on is predicting neutron transport within reactor cores. Neutron transport is very important since it tells a lot about the burn up of fuel in time and the heat distribution within the vessel. Neutron transport is described by the Boltzmann equation which can be solved using different approaches. One of the most elegant methods and computationally most efficient methods is variational calculus which in this case is the foundation for the finite element method (FEM) used for the analysis. FEM is used in a variety of industries ranging from automotive, civil engineering to space engineering. In most cases FEM analysis results in a large sparse linear system that has to be solved efficiently. Therefore the work described in this report has a broad range of applications.

First of all, the most important consequence of working with sparse matrices concerns their structure. The zero's in the matrix have no purpose and just consume memory and computation time. To handle these systems more efficiently different storage formats are used that are all based on exclusively storing the values of the nonzero matrix elements together with their indices. Second, there are different methods to find the solution of sparse systems which all have their advantages and disadvantages. Manipulation of sparse matrices has to be done with great efficiency since every step taken will be repeated for all rows or even to a power of that number. The best way to implement a solver therefore is by making use of one of the tool packages available, since those packages contain highly optimized routines.

This report gives an outline on how the FEM method is applied to the equations of neutron transport and on the different solver methods. It describes how these solvers can be implemented and configured and finally the implemented solver is tested on a series of systems coming from 1D FEM analysis on neutron transport.

Chapter 2

Discretization of neutron transport

Although the complete derivation is beyond the scope of this report an outline of the basic steps of the more theoretical aspects of the least squares is given. This illustrates the basic operations carried out to transform the transport equation to a linear system of equations. A more detailed derivation can be found in [1].

2.1 First order Boltzmann equation

Stationary neutron transport in a nuclear reactor is described by the first order Boltzmann equation. This equation for the neutron flux density $\psi(\mathbf{r}, \mathbf{\Omega})$ describes the production and loss of neutrons of energy $E + \delta E$ and direction of motion $\mathbf{\Omega} + \delta\mathbf{\Omega}$ and is written in differential form which implies it holds for all volume elements δV throughout the reactor core.

$$\mathbf{\Omega} \cdot \nabla\psi(\mathbf{r}, \mathbf{\Omega}) + \sigma\psi(\mathbf{r}, \mathbf{\Omega}) = \int \sigma_s(\mathbf{\Omega} \cdot \mathbf{\Omega}')\psi(\mathbf{r}, \mathbf{\Omega}')d\mathbf{\Omega}' + S(\mathbf{r}, \mathbf{\Omega}). \tag{2.1}$$

Where σ is the total cross section and σ_s the scattering cross section. The first term comes from the divergence of flow and accounts for the rate of change of number of neutrons due to leakage. The second term describes the loss of neutrons due to collisions. The first term on the right hand side is called the inscatter term and calculates the rate with which neutrons of different energy E' and direction of movement $\mathbf{\Omega}'$ are scattered to neutrons with energy E and direction of movement $\mathbf{\Omega}$. In a first approach only one energy group is considered and therefore integration over all possible energies is left out. The second term on the right hand side S is a general source term. The Boltzmann equation can be simplified by introduction of a linear operator $\mathcal{L}$.

$$\mathcal{L}\psi(\mathbf{r},\mathbf{\Omega}) = (\mathbf{\Omega}\cdot\nabla+\sigma)\,\psi(\mathbf{r},\mathbf{\Omega}) - \int \sigma_s(\mathbf{\Omega}\cdot\mathbf{\Omega}')\psi(\mathbf{r},\mathbf{\Omega}')d\mathbf{\Omega}' \tag{2.2}$$

Substitution of $\mathcal{L}$ into equation 2.1 yields

$$\mathcal{L}\psi(\mathbf{r},\mathbf{\Omega}) = S(\mathbf{r},\mathbf{\Omega}). \tag{2.3}$$

2.2 Least squares functional

The exact solution will be approximated by the function $\psi(\mathbf{r},\mathbf{\Omega})$. For an approximate solution there will be a difference in the left and right hand sides of equation 2.3. This error is a measure of how well the approximation represents the exact solution. The total error is obtained by adding the errors in all sub volumes δV, the square error is used in order to have only positive contributions. This method is known as the least square error method and for $\psi(\mathbf{r},\mathbf{\Omega})$ the squared error is given by the following functional $\mathcal{E}[\psi]$.

$$\mathcal{E}[\psi] = \int_{4\pi}\int_{-\infty}^{\infty} |\mathcal{L}\psi(\mathbf{r},\mathbf{\Omega}) - S(\mathbf{r},\mathbf{\Omega})|^2 d\mathbf{r}d\mathbf{\Omega} \tag{2.4}$$

In this case it is convenient to use the general notation for the inner product.

$$\int_{4\pi}\int_{-\infty}^{\infty} x(\mathbf{r},\mathbf{\Omega})y(\mathbf{r},\mathbf{\Omega})d\mathbf{r}d\mathbf{\Omega} = \langle x(\mathbf{r},\mathbf{\Omega}), y(\mathbf{r},\mathbf{\Omega})\rangle \tag{2.5}$$

With which the functional simplifies to

$$\mathcal{E}[\psi] = \langle \mathcal{L}\psi(\mathbf{r},\mathbf{\Omega}) - S(\mathbf{r},\mathbf{\Omega}), \mathcal{L}\psi(\mathbf{r},\mathbf{\Omega}) - S(\mathbf{r},\mathbf{\Omega})\rangle\,. \tag{2.6}$$

At the boundary the neutron flux density vanishes for all $\mathbf{\Omega}$ pointing inwards with respect to the boundary of the rector vessel, since neutron flow from outside the reactor can be neglected. This boundary condition is given by

$$\psi(\mathbf{r},\mathbf{\Omega}) = 0 \;:\; \forall\; \mathbf{r} \in \Gamma \cap \mathbf{\Omega}\cdot\hat{n} < 0. \tag{2.7}$$

After manipulations that are beyond the scope of this report this boundary condition can be added to the error functional.

$$\mathcal{F}[\psi] = \mathcal{E}[\psi] + 2\int_{\Gamma}\int_{\mathbf{\Omega}\cdot\hat{n}<0} d\Gamma d\mathbf{\Omega}|\mathbf{\Omega}\cdot\hat{n}|\psi^2(\mathbf{r},\mathbf{\Omega}) \tag{2.8}$$

Finding the best solution for the system comes down to minimizing the least square error functional $\mathcal{F}[\psi]$.

2.3 Approximation Polynomials

In order to perform the minimization both the scattering cross section and the neutron flux density will be represented by polynomials. This allows for the integral of the inscatter term to be evaluated and for the minimization to be carried out. The scattering cross section depends on the inner product $\mu_0 = \mathbf{\Omega} \cdot \mathbf{\Omega}'$ and therefore it is convenient to use the Legendre polynomials for its representation.

$$\sigma_s(\mu_0) = \sum_{l=0}^{\infty} \frac{2l+1}{4\pi} \sigma_{sl} P_l(\mu_0) \tag{2.9}$$

The elements σ_{sl} can be obtained from integrating the inner product of the polinomial terms with the true scattering cross section over all μ_0. The neutron flux density depends on both the position and the direction of flow. It turns out to be convenient to separate these dependencies and use spherical harmonics to represent the directional dependence.

$$\psi(\mathbf{r}, \mathbf{\Omega}) = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} Y_l^m(\mathbf{\Omega}) \psi_{lm}(\mathbf{r}) \tag{2.10}$$

The spatial part of the neutron flux density will be build up out of FEM basis functions $\phi_n(\mathbf{r})$ at N positions.

$$\psi_{lm}(\mathbf{r}) = \sum_{n=1}^{N} \phi_n(\mathbf{r}) \psi_{nlm} \tag{2.11}$$

FEM basis functions can be best thought of as interpolation functions which have value one at their own node and zero at all other nodes. What should always hold

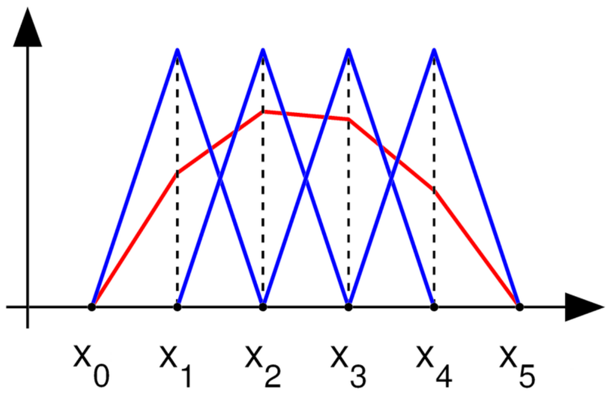

Figure 2.1: 1D finite elements basis functions (blue) used to interpolate a function (red)

is that the summation of all basis function of the different nodes together equals one at every position between the nodes. In the 1D case the linear basis functions are triangles as shown in figure 2.1. For example between x_2 and x_3 the only nonzero basis functions are the ones of node two and node three and it is easily verified that both conditions are met. The interpolation function reads

$$g(x) = \sum_{k=0}^{5} g(x_k)\phi_k(x). \tag{2.12}$$

In higher dimensional problems the shape of the basis functions will be different but the two conditions will always hold. It is also possible to use higher order basis functions which will result in a higher order interpolation.

2.4 Minimization of the least square error functional

At this point the estimated solution of the neutron flux density is defined at all positions and for all flow directions by the following function

$$\psi(\mathbf{r}, \mathbf{\Omega}) = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} Y_l^m(\mathbf{\Omega}) \sum_{n=1}^{N} \phi_n(\mathbf{r})\psi_{nlm} \tag{2.13}$$

The basis functions and the spherical harmonics are fully defined and this leaves the set of constants $\{\psi_{nlm}\}$ to control the solution. The minimization of the functional is carried out with respect to this set. In order to find the best solution which in this case is defined to be the solution with the least squared error the derivatives of the functional with respect to the constants ψ_{nlm} should equal zero for the whole set $\{n, l, m\}$. The minimization condition becomes

$$\frac{\partial}{\partial \psi_{nlm}} \mathcal{F}[\psi] = 0 \ : \ \{n, l, m\}. \tag{2.14}$$

Treatment of the boundary term will not be considered in this outline. Rearranging the functional and taking the derivative with respect to $\psi_{n'l'm'}$ provides

$$\left\langle \mathcal{L}Y_{l'}^{m'}(\mathbf{\Omega})\phi_{n'}(\mathbf{r}), \mathcal{L}\psi(\mathbf{r}, \mathbf{\Omega}) \right\rangle = \left\langle S, \mathcal{L}Y_{l'}^{m'}(\mathbf{\Omega})\phi_{n'}(\mathbf{r}) \right\rangle. \tag{2.15}$$

This equation represents the $\{n', l', m'\}^{th}$ row of the linear system. In this case the primes are added to point out that the second element in the inner product on the left hand side of the equation above still contains the whole set $\{n, l, m\}$.

Only nearest neighbors produce results that are nonzero therefore the system will be sparse. The large sparse linear system can be written in the form $\mathbf{Ax} = \mathbf{b}$ for which $\mathbf{A}$ and $\mathbf{b}$ can be evaluated form equation 2.15. The elements of $\mathbf{A}$ at positions $n'l'm', nlm$ are given by

$$a_{n'l'm',nlm} = \left\langle \mathcal{L}Y_{l'}^{m'}(\mathbf{\Omega})\phi_{n'}(\mathbf{r}), \mathcal{L}Y_{l}^{m}(\mathbf{\Omega})\phi_{n}(\mathbf{r}) \right\rangle . \tag{2.16}$$

From this it can be seen that the obtained matrix is symmetric in case the operator $\mathcal{L}$ is real, which is the case. This feature turns out to be quite important since it allows for the system to be solved by the conjugate gradient method and memory can be saved since only half the matrix has to be stored. Solving this system provides the solution vector $\mathbf{x}$ representing the values of the set $\{\psi_{nlm}\}$ which defines the solution of the first order Boltzmann equation. How the linear system can be solved and how the solver can be implemented will be the subject of the next chapters.

Chapter 3

Solution methods to linear algebraic systems

As stated in the previous chapter the solution of the first order Boltzmann equation can be found from solving the sparse linear system originating from equation 2.15. Modeling physical problems and numerical analysis often results in a system of algebraic linear equations. Therefore, solving linear systems is an important topic in computational science. In the present case the number of equations to solve grows with system size, especially in a multi-dimensional setup. For large systems memory and computational costs become limiting factors. Different methods to solve systems have been developed in the past which can be divided into two main groups; direct methods and iterative methods.

3.1 Direct solvers

Direct methods are similar to the method one uses to solve small matrices by hand; Gaussian elimination. By performing elementary row operations the matrix is reduced to triangular form from which the solution can be obtained by back substitution. This method is very stable and well suited for smaller systems, but for larger systems this method is to expensive in both memory and computation time.

3.2 Iterative solvers

Iterative solvers differ from direct solvers in a sense that the iterative solvers start with an initial guess $\mathbf{x_0}$ that is altered during the solving procedure until the vector $\mathbf{x_n}$ after n iterations correspond to the solution. In that case the solution is called to be converged. There are several criteria to check for convergence; one commonly used is to check whether or not the square of the residual becomes smaller than a certain convergence error.

$$\mathbf{r^2} = (\mathbf{Ax} - \mathbf{b})^{\mathbf{T}}(\mathbf{Ax} - \mathbf{b}) < \epsilon \tag{3.1}$$

In most cases the convergence error is normalized with respect to the norm of the initial error or with respect to the norm of the right hand side vector **b**. The disadvantage of this method is that, in advance, it is not known how many iterations are needed for the solution to converge and whether or not the solution will converge at all. Especially for large sparse matrices this method can save a great amount of both storage and computation time.

3.3 Steepest Descent method

The steepest descent method is designed to reduce the residual of a guessed or a randomly created initial solution by adapting the solution vector in the direction of the gradient of the residual. The negative gradient of the residual defines the direction in **x**-space in which the residual decrease the most [1]. The new approximation for the solution is the previous solution to which the last step is added. Changing the solution vector reduces the residual and this approach will be repeated until the solution is converged.

3.3.1 Condition number

This method works very well in case the eigenvalues of the matrix are of the same order of magnitude. In that case the system is called to be well conditioned and has a low condition number. The condition number κ is defined to be

$$\kappa(\mathbf{A}) = \parallel \mathbf{A^{-1}} \parallel \cdot \parallel \mathbf{A} \parallel . \tag{3.2}$$

Which still depends on the choice of the norm that is used. The condition number can be evaluated with the Frobenius norm which is defined to be

$$\parallel \mathbf{A} \parallel_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{m} a_{ij}^2}. \tag{3.3}$$

When a system has a large condition number it will take a lot of iterations to find the solution. This can be explained in the two-dimensional case with the aid of

[1]Movement in this direction will result in a minimal residual in case the step size is such that the derivative of the residual with respect to change of position along this direction equals zero; this allows for the optimal step size to be calculate.

figure 3.1[2] in which the steps towards convergence are drawn for both the steepest descent method and the conjugate gradient method. The contours represent equal squared residuals so in 3D this figure would look like half a section of a rugby ball standing up and finding the solution efficiently comes down to finding the lowest point in as few steps as possible.

In this figure it is easy to identify the two directions of the eigenvectors of the two-dimensional matrix used for this example; they point toward the longest and the shortest side of the oval shaped contours. If the initial solution vector would accidentally be chosen in such a way that it would lie on one of the eigenvectors of the system, the solution would be found after one iteration[3]. However, when the initial solution vector is chosen somewhat more unfortunate as is shown in the figure it might take quite some steps to end up with a converged solution. This is due to the fact that in an elongated solution valley the gradient does not always point exactly in the direction where the solution can be found. The more the solution valley is elongated the more the direction of the gradient and the direction towards the solution might differ. The difference in eigenvalues and therefore the condition number is a measure for how much the solution valley is elongated and hence a measure for how fast the solution will converge.

3.4 Conjugate Gradient (CG) method

From the previous section it becomes clear that in case the eigenvalues of the system are not of the same order of magnitude the solution region is elongated and it will take more iterations to find the solution. As can be seen from figure 3.1 the steepest descent method takes multiple steps in the same direction. It would be better if in every direction only one step had to be taken. The conjugate gradient (CG) method is designed to do just that. It moves in n orthogonal search directions $\{\mathbf{d_1}, \mathbf{d_2}, ..., \mathbf{d_n}\}$ only once which has tremendous impact on the overall performance. From the figure it shows that the steepest descent method takes 2.5 more steps than the CG method and in practice the system might be conditioned even worse.

The orthogonal search directions have to be orthogonal with respect to the matrix $\mathbf{A}$.

$$\mathbf{d_i^T A d_j} = 0 \quad ; \quad i \neq j \tag{3.4}$$

Without providing a general proof for the statement that searching in $\mathbf{A}$-orthogonal

[2]Wikipedia, the free encyclopedia

[3]On an eigenvector, the gradient points in the direction of that eigenvector and moving in opposite direction; towards the point where the change in residual with respect to change in position along this eigenvector equals zero; this would result in finding the solution at once in the first iteration.

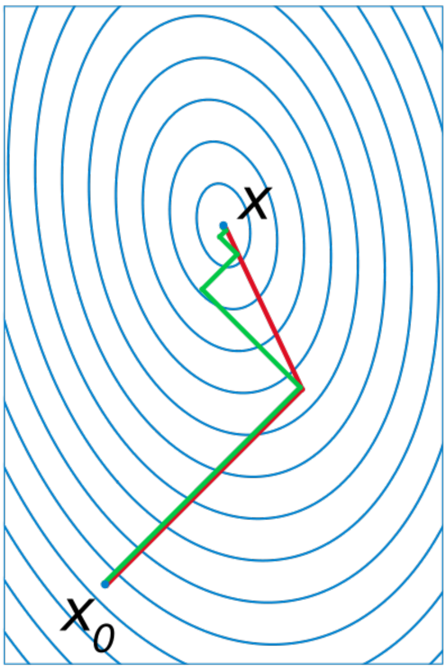

Figure 3.1: Steepest descent method (green) together with the CG method (red)

directions results in finding the solution within n steps, the argument is visualized by choosing the eigenvectors to be the search directions. The eigenvectors $\{\mathbf{e_1}, \mathbf{e_2}, ..., \mathbf{e_n}\}$ are clearly orthogonal in $\mathbf{A}$ since the eigenvectors are orthogonal and only change by a factor when multiplied by $\mathbf{A}$.

$$\mathbf{e_i^T A e_j} = \mathbf{e_i^T} \lambda_j \mathbf{e_j} = \lambda_j \cdot 0 \quad ; \quad i \neq j. \tag{3.5}$$

Any initial guess for the solution vector is a superposition of the eigenvectors. Therefore, minimizing the residual in the direction of the eigenvectors one by one would result in convergence within n iterations. Using the eigenvectors is a nice way to imagine how this works, but unfortunately using the eigenvectors is not an option since it requires to solve the system first in order to obtain them.

Again considering the two-dimensional example of figure 3.1; if it would be possible to define a set of $\mathbf{A}$-orthogonal search directions, the solution should be found within two iterations. As can be seen from the figure, this is exactly what happens. It turns out that it is possible to define a set of directions that is orthogonal in $\mathbf{A}$ and it can be proven that with an arbitrary set of $\mathbf{A}$-orthogonal directions it is always possible to find the solution within n iterations. Therefore, the conjugate gradient method solves systems more efficiently then the steepest descent method in case the system to solve is badly conditioned.

The conjugate gradient method only works for symmetric systems. This requirement is met as can be seen from equation 2.16.

Chapter 4

Preconditioning

The condition number has great influence on how efficient a linear system can be solved as discussed in the previous chapter. Using the conjugate gradient method prevents the solver from taking several steps in the same direction, but there is an additional method to further reduce the number of steps needed to find the solution.

4.1 Simple example

Again a two-dimensional example is used to help imagine how this method works. When the eigenvalues of a two by two matrix are the same, the contours of the square of the residual will be circular. The solution of the system lies in the center of these circles. Independent of the position in this 2D-space the negative gradient[1] will point towards the centre of the circle and therefore in the direction of the solution. Moving in the direction of the negative gradient starting from an arbitrary point in space will result in finding the solution at once, in the first iteration. The conjugate gradient method finds the solution in two or less iterations, but when the eigenvalues of the system are equal it requires only one iteration. For a three-dimensional problem the solution would be found by CG in three or less iterations, but with a set of identical eigenvalues the solution will be found in only one iteration. Finally the same holds for a n dimensional problem for which the difference between finding the solution in less than n iterations with respect to finding the solution in the first iteration is greatest.

[1]Gradient in this case refers to the gradient of the squared residual

4.2 Preconditioning

Changing the system in such a way that the eigenvalues become more similar or the condition number becomes smaller, has great influence on the efficiency of the solver as can be seen from the simple example stated above. Reducing the condition number can be realized by multiplying the system with a matrix $\mathbf{P^{-1}}$. This is called preconditioning.

$$\mathbf{P^{-1}Ax} = \mathbf{P^{-1}b} \tag{4.1}$$

When

$$\kappa(\mathbf{P^{-1}A}) < \kappa(\mathbf{A}) \tag{4.2}$$

the resulting system will be solved in less iterations than the original system. The best preconditioner would be $\mathbf{P^{-1}} = \mathbf{A^{-1}}$ which would result in a preconditioned system for which the solution can be found in the first iteration.

$$\mathbf{Ix} = \mathbf{P^{-1}b} \tag{4.3}$$

The problem catch in this case is that computing $\mathbf{A^{-1}}$ is more expensive than solving the original system itself. In case the identity matrix is used as a preconditioner or the system is not preconditioned, all computation time goes into solving the system with CG. When $\mathbf{A^{-1}}$ is used as preconditioner, all computation time goes into calculating the preconditioning matrix. Finding an optimum in this process is a trade off between gaining time in the solver process and losing time building the preconditioner.

4.3 Incomplete LU, zeroth order

Incomplete LU decompositions are very powerful preconditioners as explained in [2]. The method is derived from the general LU decomposition which is a direct method that calculates the lower **L** and upper **U** triangular matrices from **A**. The **L** and **U** matrices are used to solve the system by substitution of **LU** for **A**.

$$\mathbf{Ax} = \mathbf{b} \quad \rightarrow \quad \mathbf{LUx} = \mathbf{b}, \tag{4.4}$$

followed by forward substitution and backward substitution of the obtained triangular systems.

$$\mathbf{Ly} = \mathbf{b} \rightarrow \mathbf{Ux} = \mathbf{y} \tag{4.5}$$

As stated before, carrying out the LU decomposition is too expensive for large systems due to the required memory and computation time, but when constraints are added the method works very well to calculate preconditioning matrices. In the simplest incomplete LU decomposition the following constraint is added to the LU decomposition. The zero pattern of the matrix **A** is used as a mask for the matrices **L** and **U**. This means that only the values of the triangular matrices at positions that are nonzero in the original matrix are evaluated. Obviously, the product of the triangular matrices is not equal to the original matrix in that case, but it can be considered an estimate. The fact that only the terms from the mask are evaluated in the ILU decomposition saves a lot of storage and computation time. This ILU decomposition is called the ILU(0) since it uses the 0^{th} order mask of **A**. The algorithm used to perform the ILU(0) decomposition [2] is given in algorithm 1 and provided in the tool package that will be discussed later on.

Algorithm 1 ILU(0)

```
for $i = 2,...,n$ do
  for $k = 1,...,i-1 \cap (i,k) \in NZ(\mathbf{A})$ do
    $a_{ik} := a_{ik}/a_{kk}$
    for $j = k+1,...,n \cap (i,j) \in NZ(\mathbf{A})$ do
      $a_{ij} = a_{ij} - a_{ik}a_{kj}$
    end for
  end for
end for
```

4.4 Incomplete LU, p^{th} order

In many cases ILU(0) preconditioning may be insufficient to obtain significant improvement in the solving process. In that case it is possible to relax the constraint and higher order masks can be used. According to [2], more accurate ILU decompositions often result in more efficient and reliable solving processes. The p^{th} order mask results from the nonzero pattern from multiplication of $\mathbf{L}_{p-1}$ and $\mathbf{U}_{p-1}$ which were obtained by the ILU($p-1$) decomposition. For example the procedure to calculate ILU(1) is the same as that for ILU(0) except for using a different mask. The mask in this case is the nonzero pattern resulting from the product of the triangular matrices obtained from the ILU(0) decomposition. According to [2] the formalism might work for multi-diagonal matrices, but for general sparse matrices the ILU(p) decomposition has a few serious drawbacks. The amount of storage and computation time are not predictable when the order of the mask is higher than zero and more important it is not in general true that using a higher order mask results in a better performing preconditionering matrix. Therefore it is not guaranteed that doing more work in calculating a higher order preconditioner results in reduction of the number of iteration the solver needs to find the solution.

4.5 Incomplete LU, with thresholds

According to [2], the magnitude of the elements is more important then their position in the matrix. The ILUT factorization works with thresholds and the dropping strategy is based on the magnitude of the elements. There are two thresholds included in the ILUT approach. First, the magnitude of the elements is compared to a certain threshold[2] τ. Second, besides the diagonal element only the p largest values per row in $\mathbf{L}$ and $\mathbf{U}$ are kept. The second thresholds is called the level of fill and is applied to manage storage. By varying these two thresholds a spectrum of preconditioners can be created up to the full LU decomposition in case no elements are dropped. The ILUT decomposition is described by algorithm 2.

Algorithm 2 ILUT

```
for $i = 1,...,n$ do
  $w := a_{i*}$
  for $k = 1, ..., i-1 \cap w_k \neq 0$ do
    $w_k := w_k/a_{kk}$
    Apply a dropping($\tau$) rule to $w_k$
    if $w_k \neq 0$ then
      $w := w - w_k * u_{k*}$
    end if
  end for
  Apply a dropping($p$) rule to row $w$
  $l_{i,j} := w_j$ for $j = 1, ..., i-1$
  $u_{i,j} := w_j$ for $j = i, ..., n$
  $w := 0$
end for
```

4.6 Incomplete LU, modified

Discarding elements from the matrix will cause errors in the LU decomposition, but of course those errors are the price one pays for saving significant amounts of both memory and computation time. There are several techniques that aim to reduce the effect of dropping elements by compensating for the discarded elements. One way to do this is by adding the discarded elements and subtract them from

[2]The thresholds are applied to a whole row; in case an element is smaller than τ times the 2-norm of that row the element is dropped.

the diagonal elements. This method is called the diagonal compensation strategy and gives rise to the modified incomplete LU decomposition [2].

Chapter 5

Implementation of preconditioned CG

As stated in the previous chapters a preconditioned conjugate gradient solver will be used. Multiple tool packages are available that can be used to build such a solver. Although it is possible to write the codes by hand it is advisable to use a tool package. In the first place to avoid doing work that already has been done, but more important the routines from a good tool package are optimized in a sense that they are efficient in usage of both memory and computation time. The larger the system to solve the larger the number of repetitions of the different routines. Bad implementation of the routines will have its impact on the overall performance of the solver.

5.1 Tool package criteria

The selection of a suitable tool package is done by comparing different packages available by the following criteria. For the case described in this report, a Fortran based tool package was preferred. Besides, it was also preferred that the solver process would not depend on parallel computing. The package should support the conjugate gradient method with preconditioning and should be well documented. SPARSKIT [3] was chosen for the implementation because it meets all the criteria and the author Yousef Saad is considered leading in the field of sparse matrix computations.

5.2 SPARSKIT

SPARSKIT consists of optimized Fortran subroutines to manipulate and work with sparse matrices. The difference in working with sparse matrices instead of dense matrices lies in the way sparse matrices are stored. For efficient usage of memory only the nonzero elements of a sparse matrix are stored. Sparse matrices are stored in three arrays; one for the values of the nonzero elements and two for the i and j indices. Examples of storage formats are; the coordinate (COO) format, the compressed sparse row (CSR) format and the modified compressed sparse row (MSR) format. CSR is the most efficient format from a storage point of view and the default format within SPARSKIT. In some routines different storage formats are used in case it provides computational advantages.

5.3 Implementation of various preconditioners

In a first approach to build a solver the CG method was implemented without a preconditioner. The CG code calls two functions. The first is **matvec** which does vector multiplication with $\mathbf{A}$ and the second is **precon** which does vector multiplication with the preconditioning matrix $\mathbf{P^{-1}}$. The linear system used to test the solver comes from the Poisson equation on a two-dimensional square lattice with a variable number of grid points n and a constant right hand side. The variable number of grid points allows to check the process easily when using a small n and investigate solver speed and its scalability by using greater n.

$$T_{\mathrm{CPU}} \propto cn^q \tag{5.1}$$

$$\ln T_{\mathrm{CPU}} \propto \ln c + q \ln n \tag{5.2}$$

The computation time T_{CPU} is proportional to a constant c times the dimension of the linear system n to the power q. The power can be determined by log-plotting the computation time as function of system size.

5.3.1 Implementation of ILU(0)

Implementation of the ILU(0) factorization comes down to using the right subroutine from SPARSKIT to obtain the **LU** matrix. Both the upper and the lower part are stored together in one matrix. SPARSKIT has a built in routine to efficiently perform the forward and back substitution which is used to create the **precon** routine.

5.3.2 Implementation of ILUT

For the implementation of the incomplete LU decomposition with thresholds small adjustments have to be made. In the previous case the fact that a mask was used implied that the resulting preconditioning matrix has the same number of nonzero elements as the original matrix $\mathbf{A}$. When working with a threshold τ in stead of a zeroth order mask, it is not known in advance which elements and how many elements will be nonzero. In order to manage storage a second threshold is added; the level of fill p as stated in the previous chapter. The level of fill determines the maximum number of nonzero element per row. After the first threshold has been applied to the entire row, only the p greatest values in the lower part of the row and the p greatest values of the upper part of the row will be kept. Including the diagonal elements, the maximum number of elements NZ_{max} in the final matrix will be

$$NZ_{max} = (2p + 1)n. \tag{5.3}$$

This number is used to allocate the variables needed in the ILUT routine. The preconditioning step is carried out by calling a subroutine from SPARSKIT which returns an error message reporting possible errors. When the preconditioning step was successful, the created preconditioning matrix will be used in the **precon** routine which is executed in a similar way as in the previous case. With **matvec** and **precon** working CG is used to find the solution. The flow chart of the program developed during the project is provided in figure 5.1.

5.3.3 Implementation of MILU

The implementation of the modified incomplete LU decomposition is analogous to the ILU(0) case, simply replace the call to the preconditioning subroutine.

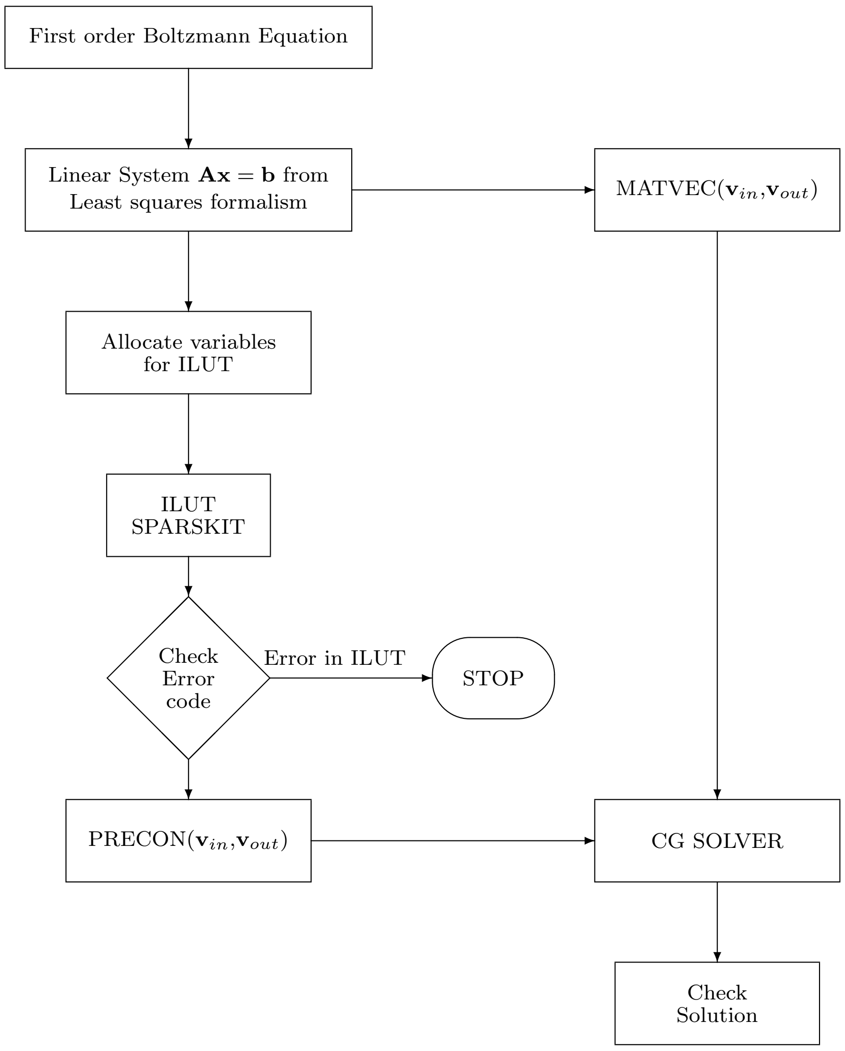
First order Boltzmann Equation
Linear System $\mathbf{A}\mathbf{x} = \mathbf{b}$ from Least squares formalism
MATVEC($\mathbf{v}_{in}$,$\mathbf{v}_{out}$)
Allocate variables for ILUT
ILUT SPARSKIT
Check Error code
Error in ILUT
STOP
PRECON($\mathbf{v}_{in}$,$\mathbf{v}_{out}$)
CG SOLVER
Check Solution

Figure 5.1: Flow chart of implementation of ILUT with SPARSKIT

Chapter 6

Unpreconditioned CG compared to ILU(0) preconditioned

In order to investigate how much the performance of a solver increases by using ILU(0) as preconditioner, the total solver time for both unpreconditioned CG and the preconditioned case is compared. Again the Poisson problem as discussed earlier is used as the test case. The results are provided in figures 6.1 and 6.2. In both cases the power with which the computation time as function of the system size scales is calculated according to the method explained in section 5.3. The system when solved with CG scales with $q = 1.57$ and when preconditioned with ILU(0) it scales with $q = 1.56$ with uncertainty 0.05 in both cases. Also the power with which the number of iterations scale with system size is calculated; the iterations scale with 0.44 and 0.44 for respectively CG and ILU(0), with uncertainty 0.03 in both cases. From this experiment it can be seen that indeed preconditioning results in a reduction of the computation time but at the same time it shows that although the calculation speeds up with a factor there is no significant difference in scalability. The scalability is of special interest because it determines how fast computation times will increase with system size. The difference in computation times for both cases which is about a factor two will not have great impact on computation time. On the other hand, a significant difference in scalability will have a tremendous impact. For example increasing system size with a factor of 10 with p equal to $\{1,0;1,25;1,5;2,0;3,0\}$ will result in an increase of computation times respectively by $\{10, 18, 32, 1.10^2, 1.10^3\}$. From the fact that for the two cases p is approximately indifferent, it can be concluded that either the ILU(0) preconditioner is not a very powerful preconditioner or that the Poisson system is not conditioned in such a way that preconditioning has great impact. According to [2] ILU(0) is not a very powerful preconditioning method, also the Poisson equation is considered to be a system that is not particularly hard to solve. Therefore this single experiment does not give an overall insight in the performance of ILU(0) and the conclusions are limited to those just described. Better view on the difference between plain CG and ILU(0) can be obtained by comparing their performances on different systems with various condition numbers in order to see which of the previous stated arguments outweighs the other.

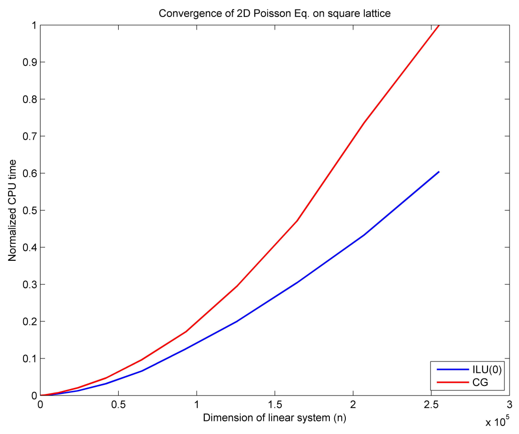

Figure 6.1: Normalized time needed to solve the linearization of the Poisson equation on a two-dimensional square lattice with number of unknowns n

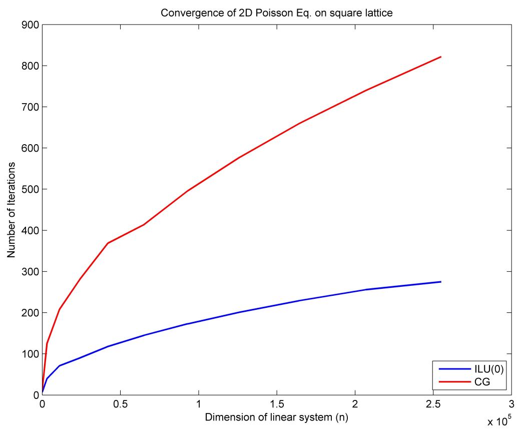

Figure 6.2: Number of iterations needed to solve the linearization of the Poisson equation on a two-dimensional square lattice with number of unknowns n

Chapter 7

Optimization of thresholds

In order to test the performance of the solver a linear system is created that represents the Poisson equation on a two-dimensional square lattice with a variable number of grid points n as explained in section 5.3. At this point the systems can be solved by CG with either no preconditioning, with ILU(0) preconditioning, with ILUT(τ, p) preconditioning and with MILU(0) preconditioning. The performance of the ILUT preconditioner strongly depends on the thresholds τ and p which determine the dropping strategy as stated before.

7.1 Mapping preconditioner performance

In order to investigate how well the solver performs for different thresholds the process of preconditioning and solving with CG is repeated with different values for τ and p. Some threshold combinations will result in poor convergence and they may delay the investigation tremendously. To still get a sense of how well the solver performs for all possible combinations only a limited number of iterations is carried out for each of them. The order of magnitude of the remaining relative error ϵ is a measure of the performance of the preconditioner. The order of magnitude is determined from the logarithm.

$$\log \epsilon = \log c \times 10^{d} = \log c + d. \tag{7.1}$$

Where $\log c$ is smaller than one and d represents the power of the relative error. Since this approach is rather qualitative than quantitative the outcomes are rendered on a range between not converging and fast converging. In case d equals zero the solver does not converge at all and in case d equals minus four it implies fast convergence. The experiment is carried out for $n = 1, 0.10^{4}$, $n = 6, 25.10^{4}$ and $n = 2, 5.10^{5}$. The initial guesses were generated by a pseudo random number generator and for some combinations the calculations were repeated to check whether or not the initial position influences the calculations. It turned out not to be the case. The results of this experiment are shown in figure 7.1 to 7.3. From these

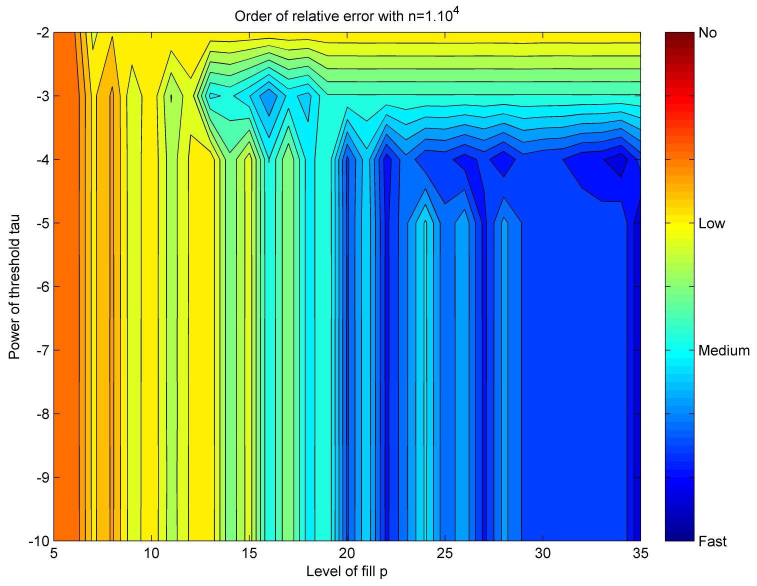

Figure 7.1: Preconditioner performance map for different combinations of threshold settings for system of size $n = 1, 0.10^4$

experiments it can be concluded that for certain values of τ and p the solver is not able to find a solution. It is believed that this has to do with the dropping strategy. By dropping elements from the system errors are introduced since some dependences are fully neglected and others are therefore exaggerated. How severe the introduced errors are is quite random since it is not known how many elements are dropped and what their magnitudes are. If a number of elements with similar magnitudes is dropped from a row and a smaller number of elements with magnitudes only slightly greater is kept; it is obvious that the introduced error will be great. The introduced error thus depends on the spread of the elements and on how many elements are cut off. Also with system size the chance of an inconvenient dropping event increases and this might have a great negative effect on the efficiency of the solver. What can be seen from the experiments is that with decreasing τ and increasing p, which will be referred to as relaxing the thresholds, the performance of the solver increases. To this general trend a random oscillating behavior is added; coming from the random influence of the dropping events on the accuracy of the preconditioning matrix. The horizontal lines in the figures show the points where increasing the level of fill has no influence on the solver performance. From this, it can be concluded that at this point the threshold τ limits

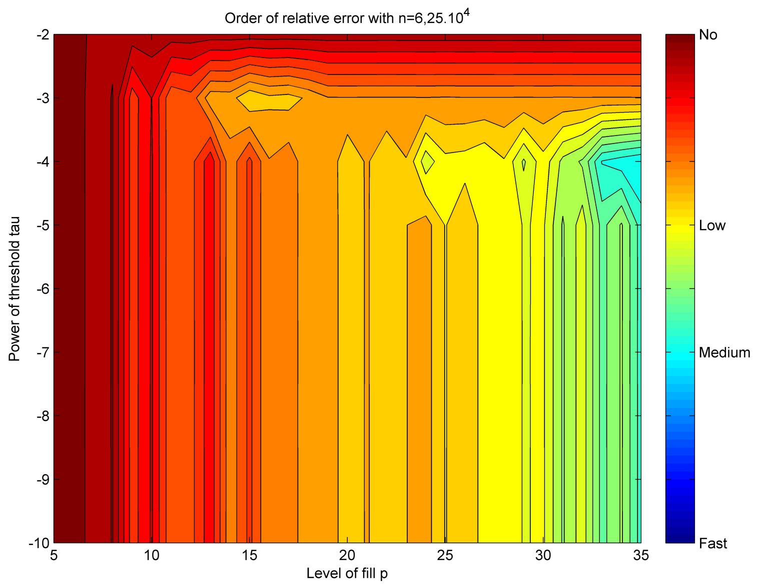

Figure 7.2: Preconditioner performance map for different combinations of threshold settings for system of size $n = 6, 25.10^4$

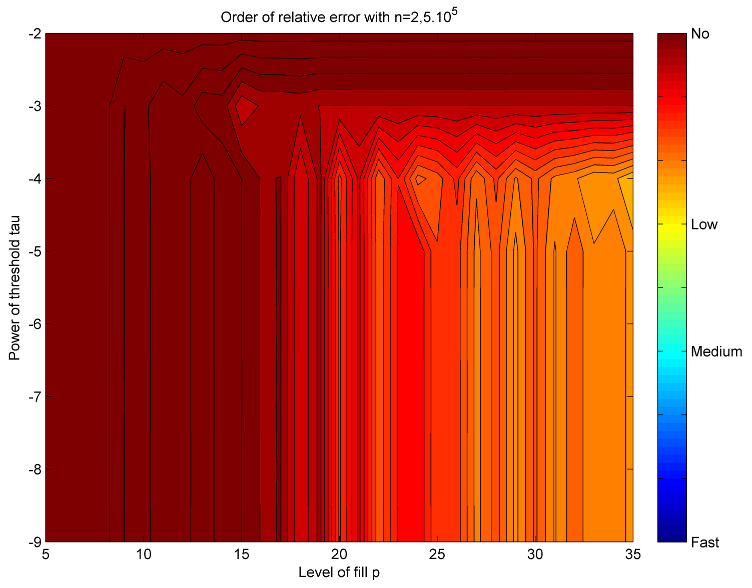

Figure 7.3: Preconditioner performance map for different combinations of threshold settings for system of size $n = 2, 5.10^5$

the number of element in the preconditioning matrix and allowing more elements per row therefore does not change the preconditioning matrix. The vertical lines in the figures come from the reverse argument, in case more elements are kept due to relaxing τ the level of fill becomes a limiting factor. Another feature that catches the eye is the peaks that appear in the figures. They suggest that some combinations of the thresholds are a more desirable than others and one might ask what the reason for that would be. This effect comes form the random influence the dropping events have on the accuracy of the preconditioning matrix. A first glance at figure 7.1 for example suggests that for $\{p = 17, \tau = 1.10^{-3}\}$ the solver works better than for any other point in that region. The fact of the matter is that with increasing p and decreasing τ a horizontal respectively vertical line is reached. At these points the solver performance gets fixed for the reasons stated above. Chances are that at these points the value representing the performance gets fixed somewhat halfway the domain of oscillating values and the likelihood of getting the performance fixed at a maximum value is very small. Therefore it is to be expected that performance peaks exist just before these fixation points, which thus falsely suggest that those threshold combinations are more desirable.

Also it can be seen that there are two regions in the figures; stable regions where performance is approximately constant and unstable regions in which the performance oscillates. What becomes clear from varying the system size is that with increasing system size the thresholds have to be relaxed in order to provide a stable solver. For the system with $n = 2,5.10^5$ no stable region exists for the used thresholds.

7.2 Optimal level of fill

The computational costs of preconditioning increases with decreasing τ and increasing p. From the previous experiment it can be seen that the performance of the solver increases strongly at approximately $\tau = 1.10^{-4}$. Therefore, the optimum is expected to be at about this value for τ. It is expected that smaller values of τ will result in more accurate preconditioning matrices and therefore faster convergence, taking τ too small on the other hand will result in a more expensive preconditioning process. This is in agreement with [2] in which it is stated that the best performance for the ILUT preconditioner will be found for $\tau \in [1.10^{-4}, 1.10^{-5}]$. It is believed that for larger systems this threshold should be taken smaller together with larger p. In order to investigate what would be the optimal level of fill, the system is solved until convergence for p varying between 5 and 35. This upper limit comes from the fact that for $p = 40$ a system of $n = 1,0.10^6$ results in a memory usage of $2[GB]$ which is the limit for this setup. Although it is possible to use greater values of p for smaller systems, such a setup would not be applicable in case of greater systems. This experiment is carried out for systems of $n = 1,0.10^4$ and $n = 6,25.10^4$, for which the results are provided in figure 7.4 and 7.5. Both results show normalized computation times and number of iterations. The figures show what was already suspected from the

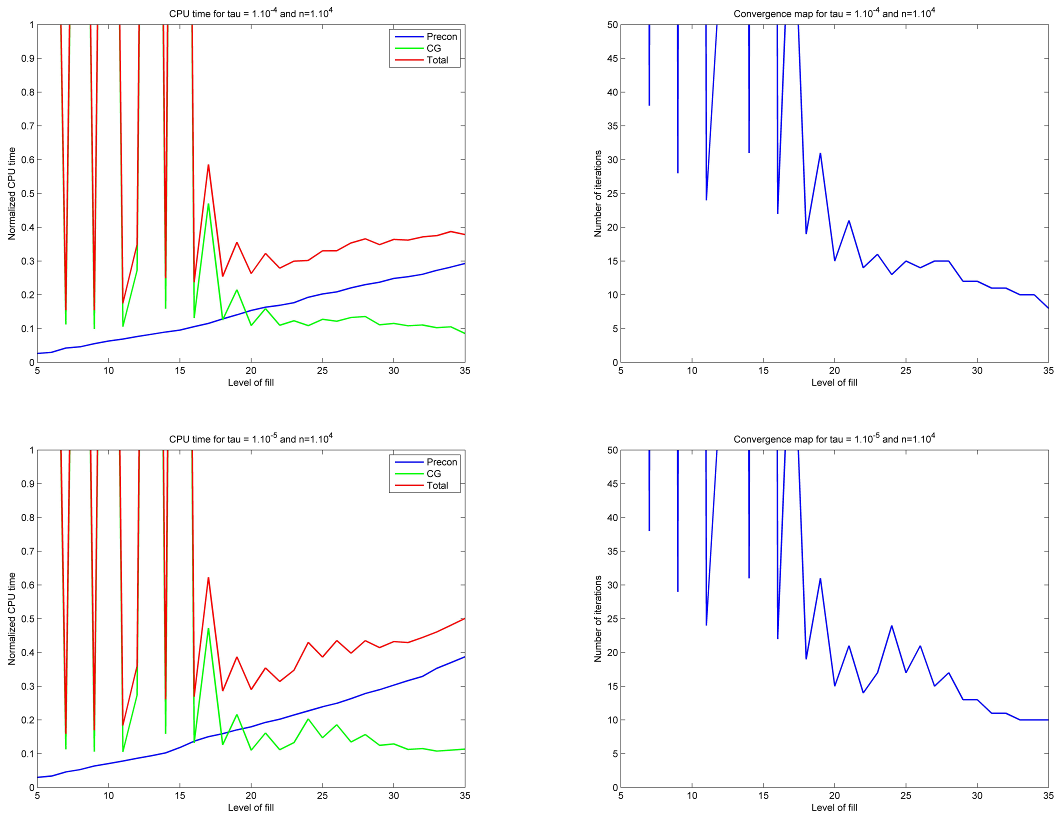

Figure 7.4: Normalized computation times and number of iterations as function of level of fill for system of size $n = 1,0.10^4$, for $\tau = 1,0.10^{-4}$ respectively $\tau = 1,0.10^{-5}$

previous experiments; for small p the solver behaves unstable and in this region solver time blows up for some values of p. The solver is not reliable to use in this unstable region . For $n = 1.10^4$ the transition from unstable to stable takes place at p is approximately 20 and for $n = 6,25.10^4$ this transition takes place at approximately p equal to 30. As expected, the preconditioning time increases with the relaxation of both thresholds. In all cases the minimum in computation time is found near the boundary between the unstable and the stable region. Although, using the boundary value saves some computation time, it is wise to take p to be somewhat greater in order to prevent an unstable solver process. In the case where $n = 1.10^4$ and $\tau = 1.10^{-5}$ the solver seems not to be improved with respect to the

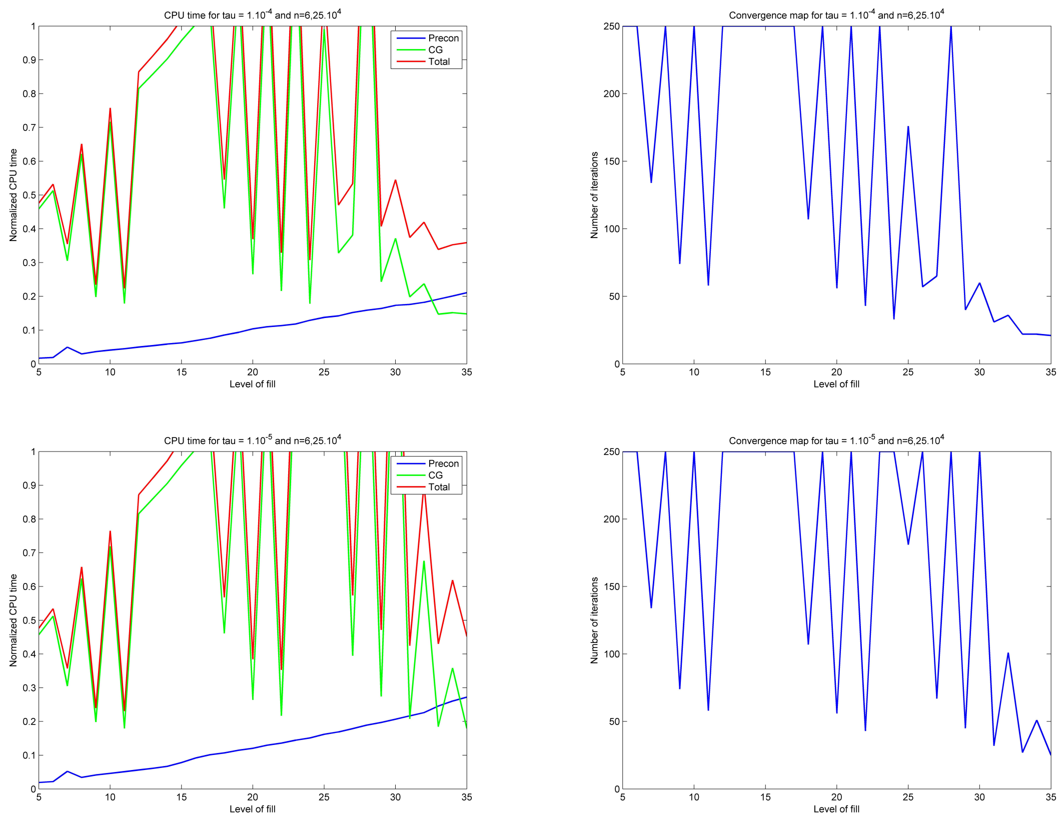

Figure 7.5: Normalized computation times and number of iterations as function of level of fill for system of size $n = 6, 25.10^4$, for $\tau = 1, 0.10^{-4}$ respectively $\tau = 1, 0.10^{-5}$

case where τ equals 1.10^4. Figure 7.1 shows that for p smaller than about 30 the performance values are vertical lines. This means that in these cases not p but τ is the limiting factor and that the performance at these points is the result of a different combination of the thresholds. The true combination of the thresholds can be found by following the straight lines. To see this, have a look at the peaks in the green lines between $p = 15$ and $p = 22$. They are the same for different thresholds of τ which goes together with vertical lines in figure 7.1 reaching up to $\tau = 1.10^{-4}$ passing $\tau = 1.10^{-5}$. Then between $p = 23$ and $p = 30$ the green lines differ corresponding to vertical lines that stop at approximately $\tau = 1.10^{-5}$. Besides how the solver behaves as a result of the thresholds something else is learned from this experiment. For these large plots only a small part of it represents actual threshold settings being the region below and to the left of horizontal lines and above and on the right of vertical lines. This is something that has to be kept in mind when changing the settings of ILUT preconditioner. One has to be aware of the fact that in a lot of cases the chances are that new setting will not result in a different preconditioner and this has to be checked.

7.3 Qualitative preconditioner performance for larger systems

Special interest is in solving systems as large as possible. In the first experiment the results on the largest system showed poor convergence. Since it is not known how the remaining relative error develops during the solution process, the experiment is repeated with 45 iterations instead of 15 iterations. The results of this experiment are shown in figure 7.6. In this case, fast convergence label is assigned to d equal to minus six because of the increase in number of iterations. This might seem arbitrary but it is sufficient for this qualitative analysis and it is believed to be a good choice since after 45 iterations a fast solver process should be fully or nearly converged and the power of minus six is believed to correspond to an acceptable convergence error. From this it can be seen that for p smaller that 35 this system has no stable region. Therefore the required resources for successful ILUT preconditioning are simply too great for systems of this size. Besides the random oscillations have increased in size which is to be expected, since after performing more iterations the difference in performance due to different preconditioning setting only increases. In some cases the instabilities are caused by pivot elements with very small values. One way to avoid this is to use pivoting during the ILU decomposition. This showed no pivot events were carried out which implies that the instabilities are not caused by small valued pivot elements.

7.4 Development of relative error

Finally for a system of $n = 2,5.10^5$ the development of the remaining error for a ILU(0) preconditioned system and for a ILUT($\tau = 1,0.10^{-5}, p = 40$) preconditioned system is plotted in figure 7.7, which will give more insight in the behavior of the two different preconditioned systems. This experiment shows an very important difference between the way ILU(0) and ILUT preconditioned systems move towards their solution during iterating. It can be seen that initially the ILUT system moves toward the solution faster than the ILU(0) system. So for great convergence errors ILUT will be faster in most cases. The greater the convergence error, the greater the difference. It also shows that for smaller and smaller convergence errors the convergence speed of the ILUT systems decreases drastically. This explains what happens in the unstable regions. In some cases the solution is found very fast, but in other cases when the preconditioning matrix is less accurate

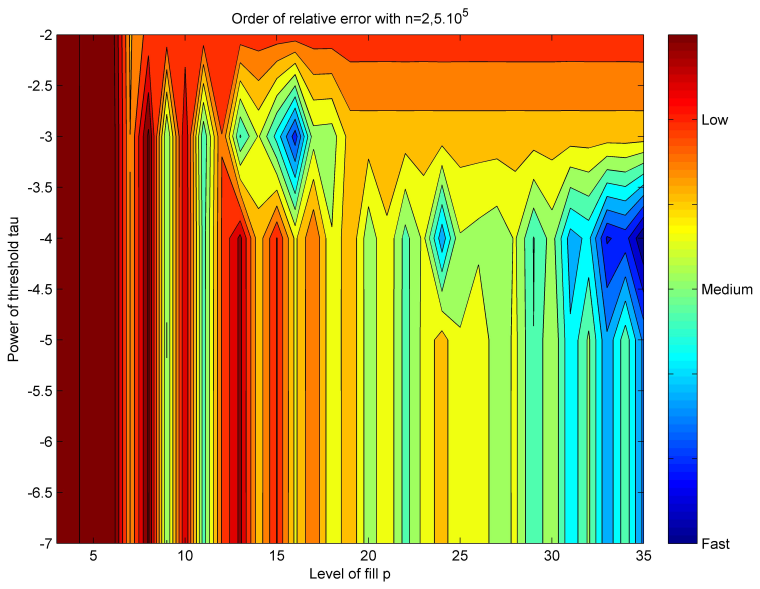

Figure 7.6: Preconditioner performance map for different combinations of threshold settings, for system of size $n = 2, 5.10^5$, after 45 iterations have been performed

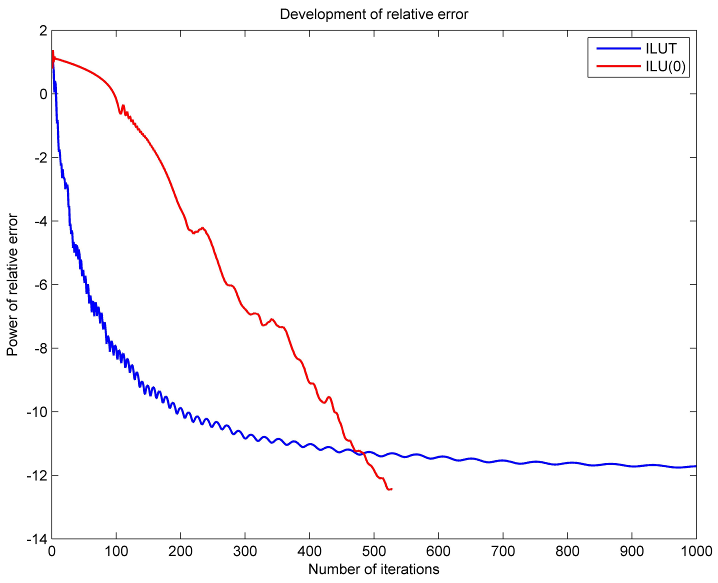

Figure 7.7: Development of relative error for ILU(0) preconditioned and ILUT preconditioned systems

the number of iterations needed to get to the convergence error builds up. Together with the fact that iterations after preconditioning with ILUT have greater computation times compared to the IILU(0) case causes the solver time to blows up. This results in alternating great performances and exceptionally poor performances. Up to this point, no satisfactory explanation is found for these different behaviors. For further investigation of this effect it might be helpful to have a look at the distributions of the eigenvalues of the preconditioned systems.

Chapter 8

Test Case: 1D FEM on neutron transport

After selecting a tool package, implementing the solver with different preconditioners and investigating the influence of the settings of the thresholds, everything is set to move on to the final stage of this research; solving linear systems representing problems from nuclear reactor physics. In the next experiment the solver with its various preconditioning options is used to solve the linear systems coming from neutron transport. As explained in Chapter 2 by means of the least square error method neutron transport in a reactor vessel can be translated into a linear system of equations. How efficient the various solver method will work depends for large part on how well conditioned the systems to solve will be. No specific research is done on the condition number of both the Poisson system and the one-dimensional FEM system. The number of equations is determined by the number of elements the vessel consists of and by how many polynomials are used to represent the scattering cross section and neutron density flux. For a particular problem coming from the neutron transport equation the setup was made with the number of segments varying to be $2000, 4000$ and 8000 and the number of terms in the polynomials to be $1, 3$ and 5. The resulting systems are solved using the three preconditioners as discussed in the previous chapters ILU(0), ILUT(τ, p) and MILU(0). It would be possible to optimize the threshold for every single case but that contradicts the need for a general method to solve matrices. Also it is not known in advance whether or not the chosen thresholds will result in a stable or an unstable process. This is one of the hardest parts of solving systems of various size with the ILUT(τ, p) preconditioning. Taking a large threshold τ and small level of fill might result in an highly effective solver process but also might just as well result in a system not converging at all. On the other hand, relaxing the thresholds surely will result in slowing down the process but in that case the solver is more likely to perform with higher consistency. In this case τ is chosen to be

$1,0.10^{-7}$ and p is chosen to be 70 which are believed to be conservative choices and will probably result in a stable solver process. As stated before, it's not convenient to investigate what thresholds suit best for every different system, still different configurations were tried out and all of them resulted in approximately the same computation times. In some cases the number of iterations increases by a small amount, which in those cases is compensated by the decrease in computation time per iteration and reduced preconditioning time. Less expensive threshold settings showed poor convergence for some cases, therefore these conservative settings were chosen in order to be able to compare performances for all cases. For the various systems the number of iterations needed to reach convergence as well as the preconditioning time and solver time are logged. The results of this experiment are shown in figures 8.1 and 8.2.

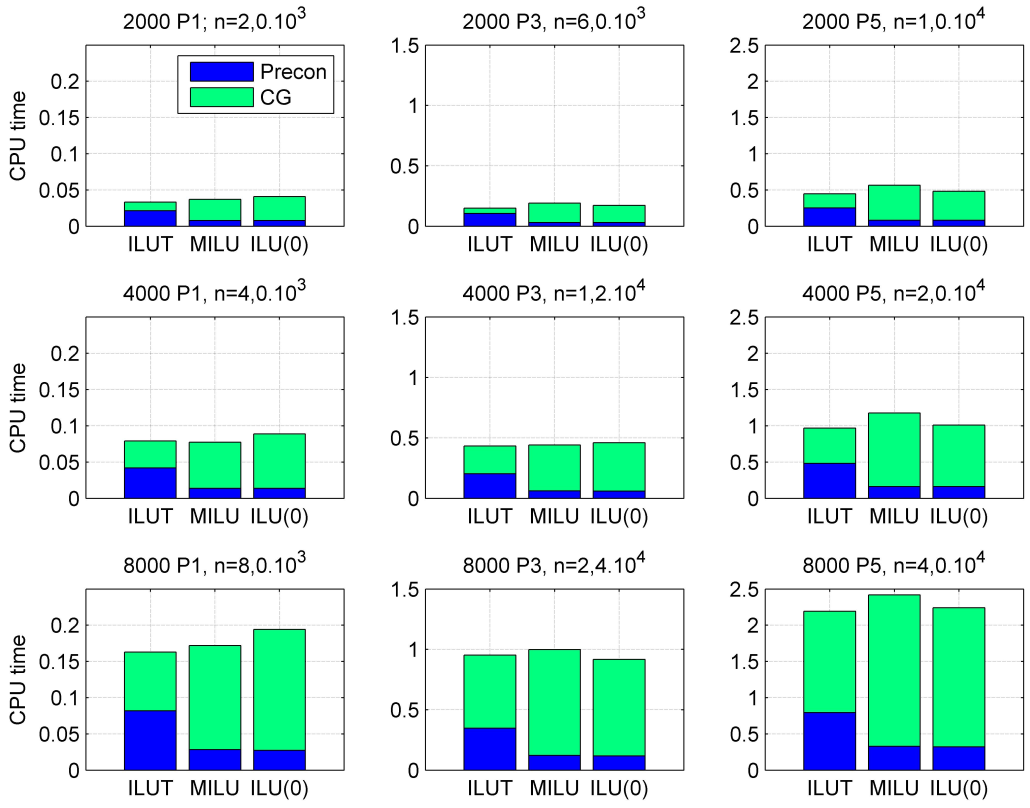

Figure 8.1: Preconditioning time and CG solver time needed to solve the various systems

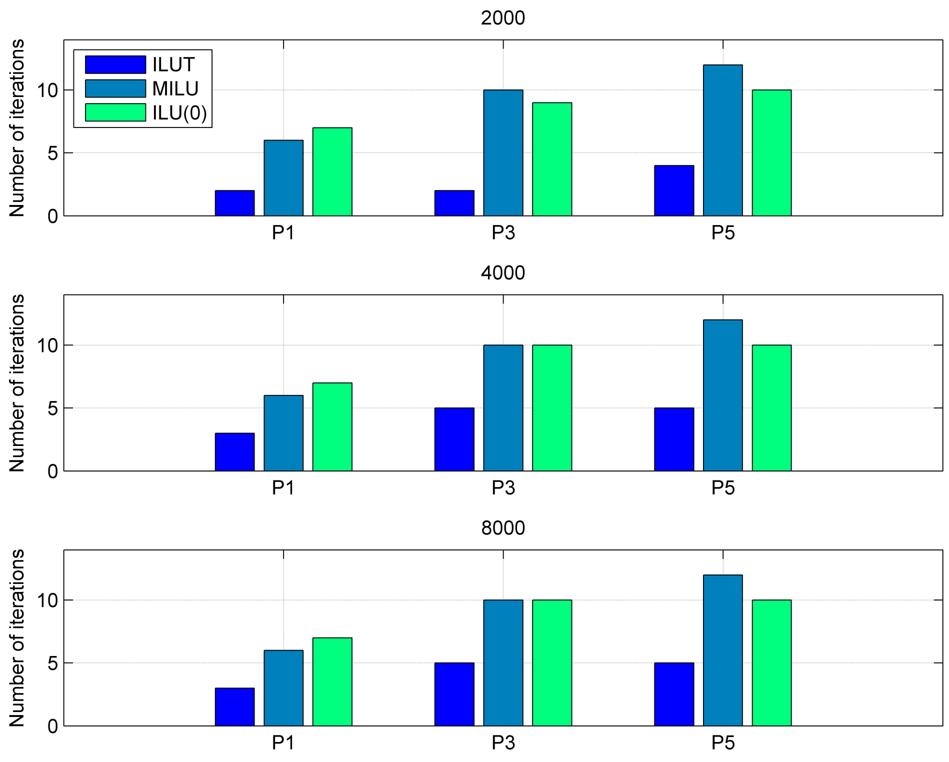

Figure 8.2: Number of iterations needed to solve the various systems

In this case the computation time rendered is the actual computation time in seconds. All calculations are performed in a sequence on the same machine and therefore the actual computation time can be used for comparing the different preconditioners. Besides, it is interesting to see how much time it actual takes to solve the systems. The axis scales are set differently for different number of polynomials used in the problem; this is done in order to make visual comparison of the results within these groups easier. From the results it can be seen that the different methods have different approaches and differ mainly in how computation time is divided over the preconditioning phase and the solver phase. From these plots it becomes clear that ILUT is able to reduce the number of iterations needed to find the solution, but since the iterations with a larger preconditioning matrix take longer no significant reduction in computation time is realized. As with the Poisson problem, it is not clear whether the differences in computational cost are small due to the fact that the systems are fairly well conditioned to begin with, or that the different method are approximately equally powerful. ILU(0) which makes use of a mask is expected to perform best in case of diagonal matrices and ILUT is expected to perform best for systems without any pattern. The pattern comes from the mesh and when this mesh is changed ILUT might show to be better equipped to handle the system, but at this point there are no experiments performed to confirm this. The scalability of the ILUT preconditioned system is not calculated. The reason for this is that computation times are well comparable with those of ILU(0) which scales approximately with a power of 1.5 and besides it will depend on the threshold settings. Therefore, further investigation would not provide additional information. For the special case considered it is believed that using either ILU(0) of MILU would be the best option since the memory requirements are significantly lower, no investigation for threshold settings is required and both solvers are more robust than the ILUT preconditioner.

Chapter 9

Conclusions and discussion

Initially, ILUT was expected to outperform the other preconditioners, but after performing the test case, it was found that the 0^{th}-order preconditioners were better suited than the ILUT preconditioner. Cases in which ILUT might provide an advantage are those where one system has to be solved for multiple right hand sides. In that case investing in a very accurate preconditioning matrix could save computation time. Also in case the system to solve is not diagonal or nearly diagonal, ILUT could provide an advantage because of its different dropping strategy which might proof better suited in those case. To draw any conclusions on that, further research on computational cost for systems with different patterns and different condition numbers would be required. As stated before performance on this specific test case is of interest and not a general comparison.

From the test case it can be concluded that solving systems of $n = 4,0.10^4$ takes about $2,5$[s]. Considering the scalability this implies approximately a 5[min] solution time for systems of size $n = 1,0.10^6$, which is the expected size for 3D cases. Whether or not this is acceptable depends on the final application.

One way to further improve the solver efficiency would be to find a more powerful preconditioning routine. Although it has to be noted that most available preconditioning methods are based on the LU decomposition with an implemented dropping strategy based either on thresholds, a nonzero element pattern or both. Therefore it is not expected that one of those methods will outperform the preconditioning routines described in this report. Another way to further optimize the performance could be to exploit the symmetry of the system, but the improvements will be bound to a factor of approximately two which has no great effect on the scalability, though the reduction in storage would be significant.

Finally, not considered in this report because the implementation is expected to be too time consuming to fit the window of this research, scalability properties could be improved by using a multigrid approach. This might result in almost linear scalability, but will be more complicated to implement. Especially for large systems, this will result in a tremendous reduction of computation time. Therefore the multigrid approach is believed to be a very interesting alternative for those cases.

Bibliography

[1] D. Lathouwers, Neutron Transport Discretization using a Least Squares Approach, Internal report; Delft University of Technology, Department of Radiation, Radionuclides and Reactors, November 2007.

[2] Y. Saad, Iterative Methods for Sparse Linear Systems, PWS, 1996.

[3] Y.Saad, manual of SPARSKIT version2; A basic tool-kit for sparse matrix computations, 2005.