Preconditioner methods applied to simulations of two-phase flow in porous media

Morten Grova
Department of Physics, NTNU, NO-7491 Trondheim, Norway
E-mail: Morten.Grova@ntnu.no

Abstract.
Simulations of two-phase flow in porous media require solving a set of linear equations which can be both large and sparse, and doing so repeatedly, to obtain local pressures under conditions of viscous flow. A variation of the incomplete cholesky preconditioned conjugate gradient solver has recently been developed for this purpose, and is presented here. It is found to be both stable and efficient. The variation consists of determining the sparsity structure of the preconditioner from a neighborhood search. Also, if the viscosities of the two fluids are different, the preconditioner should be updated during the course of the simulation. How often this should be done depends on the magnitude of the viscosity ratio, on the cost of updating the preconditioner and on how quickly the simulation configuration changes. To address this, a dynamic preconditioner update criterion is formulated.

1. Introduction
The transport of two immiscible fluids through a porous medium is greatly affected by the surface tension between the fluids [1, 2]. Menisci separating the two fluids within the pore space and the capillary pressure drops across them modify and are modified by flowing conditions. Understanding the resulting dynamics is of importance to oil recovery, for example.

Numerical simulations have a long history of complementing laboratory experiments in the study of this problem. Invasion percolation [3] and diffusion-limited aggregation [4] are two popular models which have been popular in the field of computational physics and have shed light on immiscible displacement processes.

When flow is intermediate between very slow (quasi-static) and very fast (dispersive) it is of key importance to solve the internal pressure- and flow-field given the external conditions. For example, it may be desired to solve for a given configuration of meniscus positions and a given total flow-rate. This requires solving a set of linear equations, assuming that the relations between flow and pressure drops are locally linear. Each equation expresses local volume conservation for the flow into and out of each pore. There are as many equations as there are pores, and each equation depends on the pressures within the adjacent pores as well as the pore itself. This results in a sparse, symmetric and positive-definite matrix $A$ which gives the general equation

$$A \cdot x = b,$$

to be solved for $x$, which is a vector of node pressures. The right-hand side $b$ is a vector of flow-rates caused by the menisci.
Eq. 1 may be solved by Cholesky factorization, $A = LL^T$, followed by forward and backward substitution. For a sparse matrix $A$ this direct method has worse scaling of performance than the iterative method of Conjugate Gradients (CG). This means that for some sufficiently large system, CG solvers will outperform the direct method.

The performance and scaling of CG solvers can be greatly improved by preconditioning the matrix $A$, meaning that the original problem is substituted with a new,

$$MA \cdot x = M \cdot b,$$

where the preconditioner $M$ approximates $A^{-1}$, producing a better conditioned problem with fewer iterations required by the CG solver. For ill-conditioned problems such as the one considered here, preconditioning is an essential technique.

The Incomplete Cholesky preconditioned Conjugate Gradient solver (ICCG(n), [5]) enforces the sparsity pattern of $A$ upon its Cholesky factors to produce a preconditioner $M$ which satisfies $M^{-1} = \tilde{L}\tilde{L}^T$, where $L$ is the incomplete Cholesky factor of $A$, which is now as sparse as $A$. The matrix $M$ is not explicitly calculated; multiplications with $M$ are done by forward and backward substitution with the incomplete Cholesky factors. The sparsity pattern of these factors may be augmented by including $n$ additional diagonals along the main diagonal. ICCG(n) is a popular method when applicable, combining the advantages of the Cholesky factorization with those of the iterative CG solver.

This work presents a variation of the ICCG(n) linear solver. After a brief description of the simulations which motivated its creation and implementation, the algorithm is presented. It consists of a neighborhood search which gives the sparsity pattern of the preconditioner. Some examples of solver performance are given, before a preconditioner update criterion is presented. The update criterion is used to determine when the preconditioner should be updated to provide optimal performance through the course of the simulation.

2. The network simulator

In order to study transients and steady-state properties of two-phase flow in porous media a network simulator has been developed. A brief description is given here, for more details see [6, 7].

The porous medium is represented as a transport network where pore throats are links and pores are nodes. Knowing the flow-rate through each throat requires knowing the pressures in each pore. In addition to the pressure difference between pores, flow is driven by the pressure drops across menisci. The relation between total pressure drop and flow-rate is assumed to be linear, with the permeability dependent on the cross-section of the throat. Disorder is assigned to throat radii.

Enforcing volume conservation at each pore leads to a set of linear equations which can be solved with respect to an external control parameter like total flow-rate. This problem is analogous to solving the Kirchhoff equations for a random resistor network.

Mappings of real porous media to networks are available, however for simplicity the networks considered in this work are square lattices. They are inclined at 45° relative to the main direction of flow. Boundary conditions are bi-periodic, meaning that the network may be mapped onto a torus. The reason for choosing this topology is to be able to study steady-state properties. However, this also causes the traditional method of adding extra diagonals to the incomplete preconditioner to become less efficient, as will be discussed.

The capillary pressure drops across menisci are given by the radius of the throat at their position. Starting with a set of meniscus positions the flow-rates contributed by them are calculated and placed in the right-hand side vector $b$ of Eq. 1. The system is iterated forward in time by solving for the pore pressures contained in $x$ to obtain the local flow-rates. Once these are known, the meniscus positions can be updated and the process is repeated.
Figure 1. Schematic illustration of the neighborhood search algorithm. Taking a 4x4 square lattice (top left) as an example, the pattern of the Laplacian matrix is shown (top right) with the red dots signifying the non-zero pattern of the adjacency matrix. Removing the upper part of the matrix (bottom right) and interpreting the resulting pattern in terms of a directed graph (bottom left), the idea behind the neighborhood search can be easily explained. The filled black circle has two nearest-neighbors (red) and three next-to-nearest neighbors (blue), when the direction of the arrows is enforced. With n equal to 1, column 5 now has non-zero elements in rows 7, 10 and 13, in addition to rows 6 and 9. For n equal to 2 the next-to-next-to-nearest neighbors should also be included, and so on.

The matrix $A$ contains local permeabilities and effective viscosities within each throat. If the two phases have unequal viscosities the local effective viscosity is volume-weighted in each throat. As the menisci move the local effective viscosity changes, changing $A$. If the two phases have identical viscosity, $A$ does not change for the duration of the simulation.

The viscosity ratio $M$ between the two phases is defined by

$$M = \frac{\mu_{nw}}{\mu_w},$$

where $\mu$ is viscosity and the subscripts refer to the non-wetting and wetting phase.

3. The linear solver

The single issue dominating computational performance is the rapid repeated solving of Eq. 1. For the simulations just described, $A$ has a static sparsity pattern. The off-diagonal elements may be interpreted as an adjacency matrix, with non-zero elements where two pores share a pore throat. Given that the porous medium is static (consolidated), the sparsity pattern is therefore also static.

If the phases have equal viscosity, the entries of $A$ do not change in the course of the simulation. It might then be expected that Cholesky factorization and repeated forward and backward substitution would be the most efficient solver. This is true only for small systems. Because the size of the Cholesky factors scale as $\sim N^{1.5}$ where $N$ is the number of pores, and CG methods scale better than this, large systems are not well treated by the direct method.
Also, the size of the factors may require more memory than today's workstations have available. This means that even if the phases have equal viscosity, CG methods are preferable to Cholesky factorization.

At this point a minor point needs to be addressed. The matrix $A$ has a unique solution $x$ to every $b$, up to an overall absolute pressure. In other words, only pressure differences enter into the equations; the absolute pressure is an irrelevant additive constant. As it stands, $A$ is therefore only semi-definite. In order to make it definite and hence invertible (stable with regards to Cholesky factorization) without destroying its symmetry a simple trick is used. One diagonal element of the matrix is multiplied by an arbitrary factor. As all but one of the equations remain unchanged, and the original matrix had exactly one redundant equation, all solutions to the new set of equations are also solutions to the old set. The only effect the arbitrary factor has is the only effect it can have, which is to fix the overall absolute pressure. As this is not important, the arbitrary factor is indeed arbitrary.

Steady-state simulations require the use of bi-periodic boundary conditions. This means that the bottom boundary of the network connects to the top boundary, and the left to the right. While ICCG(0) represents a considerable improvement over non-preconditioned CG and inverse diagonal preconditioned CG for these simulations, the traditional ICCG(n) method of adding additional diagonals does not result in any substantial additional increase of performance. This is in contrast to what is reported for several other problems, which typically deal with solving differential equations using some finite-difference method. The suspected reason for this lack of improvement is the use of bi-periodic boundary conditions. These lead to numerically large elements in the full Cholesky factors which are far from the main diagonal. Thus, the bi-periodic boundary conditions are a strong motivation for using a neighborhood search to determine the sparsity pattern.

The neighborhood search algorithm is illustrated in Fig. 1. The search proceeds in the direction of increasing node index on a directed graph obtained from the lower half of $A$. The maximum number of nodes allowed between the origin and the most distant node still considered a neighbor is $n$. Because the search proceeds along a directed graph the ordering of node indices is significant; this mirrors the significance of node ordering in the Cholesky factorization algorithm. An ordering which keeps near neighbors close in index value will result in a sparsity pattern which is most dense in the proximity of the main diagonal.

Proceeding only in the direction of increasing node index when mapping out the neighborhood structure is not a strictly necessary condition. If the neighborhood search is performed on the undirected graph a stable preconditioner with similar performance is obtained. There will however be more entries in the sparsity pattern which are zero-valued in the Cholesky factorization. These elements can be identified and eliminated as part of the factorization. The main reason for only searching on the directed graph is the symmetry with the factorization algorithm, which also proceeds on a directed graph.

Fig. 2 illustrates the result of the procedure. The original matrix is shown together with a full Cholesky factor (which is semi-sparse) and an $n = 4$ incomplete factor. In this example, the neighborhood search was performed on an undirected graph.

Even though the example shows a very small system, it should be clear from this that the size of the incomplete factor can be controlled using $n$, and that the neighborhood search produces a sparsity pattern which captures features of the full factor. For example, the entries in the lower-left corner and the outlying horizontal lines reflect the bi-periodic boundary conditions. These entries would not be present if the method of adding additional diagonals was used. It can also be noticed that the near-main diagonals are not completely regular. The slightly jagged pattern seems innocent, but is important with respect to parallelizability.

For large problems, parallelizability is an important consideration. Consider a factorization $A = LL^T$. Solving this requires forward backward substitution, which is not perfectly
Figure 2. Visualization of matrix structure. System size is 16x32 pores. The top row is the lower half of $A$, the middle row is the complete Cholesky factor $L$ and the bottom row is the incomplete Cholesky factor $\tilde{L}$, where the neighborhood search algorithm has been applied with $n = 4$. The left column shows the full matrices with entries color coded by their values. The entries are indexed by $i$ and $j$. The right column shows a zoom in. Non-zero entries are black, highlighting the sparsity pattern.
parallelizable. This is the most significant challenge in parallelizing the ICCG(n) method. Consider a factor \( L \) which consists of only the main diagonal and the elements directly below it. This corresponds to a directed graph which is simply a one-dimensional chain. That directed graph gives the sequence in which the substitution must be carried out. It should be clear that this problem is not parallelizable at all. In general, complete diagonals prevent parallelization. Thus, the method of adding extra diagonals results in a method with poor parallelizability.

In contrast, using a directed neighborhood search results in a directed graph which can be partitioned as easily as a domain decomposition of the original problem. Because the neighborhood search preserves the original spatial layout it provides a method with excellent parallelizability.

The stability of a given sparsity pattern is not guaranteed. For example, it might be tempting to threshold the full Cholesky factor, keeping only the largest non-zero values and setting the rest to zero. This procedure results in a sparsity pattern which can then be used for subsequent calculations of incomplete Cholesky factors. However, numerical experiments have shown that this procedure results in an unstable preconditioner.

The sparsity pattern determined by a neighborhood search has resulted in a stable preconditioner for all realizations of the simulations so far, and is now a trusted method. Stability is not trivial, which makes this an important point.

The neighborhood search to determine the preconditioner sparsity pattern is expensive when implemented for general networks. However, because the sparsity pattern does not change in the course of the simulation the search only has to be done once. Because the solver is called a large number of times (\( > 10^6 \)) the cost of the neighborhood search is negligible. In other words, the high initial cost is offset by repeated application.

Fig. 3 shows some performance results for typical simulation parameters. The number of iterations maps proportionally to overall performance for a given \( n \), however larger \( n \) gives larger incomplete Cholesky factors and thus a larger cost per iteration. Scaling results can be found for constant \( n \), however it is also found that the optimal \( n \) increases for larger systems. The actual scaling of the ICCG(n) linear solver with \( N \) therefore requires some scheme for varying \( n \) with system size. This scheme can not be considered independent of the architecture on which

Figure 3. Scaling of solver performance measured in the number of iterations required, with system size \( N \). Various values of \( n \) are shown. Counting from above, \( n \) is 0, 2, 4, 6 and 8. Scaling exponents are 0.41, 0.39, 0.36, 0.34 and 0.33, respectively.
The solver is to be run.

If the scaling exponent of CG iterations with $N$ is referred to as $\alpha$ and the scaling exponent of CG solver time with $N$ as $\beta$, an idealized implementation of the CG method will scale as $\beta = \alpha + 1$. This is because every CG iteration consists of scalar-vector and sparse matrix-vector operations, all of which scale linearly with $N$. In any actual implementation of a CG method the situation is more complicated, as hardware issues such as memory locality become relevant and give implementation dependent scaling exponents.

If 0.33 is taken as the value for $\beta$ and an idealized implementation is assumed, a value of 1.33 is obtained as an upper bound on the scaling of the neighborhood search ICCG($n$) method. This is for some constant value of $n$. Better scaling can be achieved, because $n$ should be increased when $N$ increases. The exponent should be compared with 1.45, which is how these simulations scale when the inverse diagonal is used as preconditioner, and 1.5, which is for non-preconditioned CG as well as Cholesky factorization.

Perhaps more important for practical purposes is the reduction of the prefactor. For system sizes and parameters of typical interest a speed-up of a factor 4 has been achieved, when comparing additional diagonals ICCG($n$) with neighborhood search ICCG($n$).

Eventually, memory constraints will place a limit on $n$, as the size of the incomplete Cholesky factor grows rapidly with increasing $n$. Partitioning the factor across nodes on a distributed memory architecture will ease this constraint, but then there are additional considerations of message passing cost to take into account.

It should be clear by now that the parameter $n$ in the ICCG($n$) linear solver is a crucial parameter. Finding its optimal value is difficult to do in general, as it depends on the hardware. Of course, it can always be determined by trial and error. Tuning $n$ allows the solver to balance various constraints of the hardware, such as available memory and message passing bandwidth.

4. The preconditioner update criterion

When $A$ changes, the preconditioner based on an old $A$ becomes progressively worse. The cost of re-calculating the preconditioner can be high compared to the cost of a single call to the linear solver, so this can not be done for every timestep. However, the benefit of keeping the number

![Figure 4](image)

*Figure 4.* Visual description of preconditioner update criterion. $n_0$ is the number of iterations required by the CG method immediately after a preconditioner update. The area marked by a red wiggle represents the benefit of keeping the number of iterations on the black line compared to the blue line. For the interval between preconditioner updates this area should be kept equal to the cost of updating the preconditioner.
Figure 5. Visual argument for criterion optimality. The areas marked by red wiggles are meant to be identical in size to those in Fig. 4. Left: the areas marked by green wiggles represents additional cost caused by updating the preconditioner too infrequently. Right: the area between the green and blue dashed lines represents additional cost caused by updating the preconditioner too frequently.

of iterations of the conjugate gradient method low can also be large. Therefore, a criterion is needed to judge whether to upgrade the preconditioner or not.

If \( n_{it}(i) \) is the number of iterations required in the CG method \( i \) timesteps after a preconditioner update, the quantity \( n_{\text{sum}} \) can be defined as

\[
n_{\text{sum}} = \sum_{i} (n_1 - n_{it}(i)),
\]

(4)

where the sum starts at \( i = 0 \) and \( n_1 \) is some value which \( n_{it} \) should not exceed. \( n_1 \) should satisfy

\[
n_{\text{sum}} = n_{\text{cost}},
\]

(5)

where \( n_{\text{cost}} \) is the cost of updating the preconditioner, measured in units of single iteration cost. This cost can be calculated the first time the preconditioner is updated. A visual description of Eqs. 4 and 5 is given in Fig. 4. \( n_{\text{sum}} \) can be estimated without knowing \( n_1 \),

\[
n_{\text{sum}} \approx \sum_{i} i \cdot (n_{it}(i) - n_{it}(i-1)),
\]

(6)

where the sum starts at \( i = 0 \), and only runs over those values of \( i \) for which the difference within the parentheses is positive. In some rare cases \( n_{it} \) may decrease for some random reason, which will cause Eq. 6 to only be an approximation.

The application of the preconditioner update criterion in Eq. 5 can now be described as follows. The sum of Eq. 6 is updated for each timestep following a preconditioner update. Considering this to be a good approximation to \( n_{\text{sum}} \) as defined in Eq. 4, the preconditioner is updated when \( n_{\text{sum}} \) exceeds \( n_{\text{cost}} \). Then, the sum is zeroed out and the process is repeated. Notice that \( n_1 \) never needs to be explicitly calculated.

The criterion and its application has now been described, but not yet fully motivated. The reason for considering Eq. 5 to be a criterion for the optimal preconditioner update frequency is easily understood from considering Fig. 5. If the preconditioner is updated less frequently, the CG method will require an unnecessarily large number of iterations in some interval preceding a preconditioner update. If the preconditioner is updated more frequently, the cost of the update is never fully regained by the benefit within a single interval between updates.

Fig. 6 compares two simulations with and without applying the preconditioner update criterion. The viscosity ratio used in this example is \( M = 10 \). For viscosity ratios larger than this, the importance of updating the preconditioner increases even more. In the example, the difference is about a factor 4. It can be seen how the preconditioner update criterion enforces a constant ceiling on the number of iterations.
Figure 6. Number of iterations in the conjugate gradient algorithm as a function of the simulation timestep. Comparison of updating (black) and not updating (red) the preconditioner. System size is 32x64, M = 10.

5. Conclusion
The need for repeatedly solving a set of linear equations with static neighbor structure motivated this work. A neighborhood search is used to determine the sparsity structure of the incomplete cholesky preconditioner. This preconditioner has proven to be stable, and more efficient for the use considered here than previously implemented preconditioners. It is also more efficient than the method of augmenting the incomplete cholesky factors with additional diagonals.

Due to the high initial cost of the neighborhood search the method is unlikely to be generally useful. However, problems where the solver needs to be called a large number of times could benefit. In particular, the method should be beneficial to problems with complex topologies or irregular neighbor structure and problems where parallel performance is required.

A preconditioner update criterion has also been formulated, which could be of general use for problems where the solver is called repeatedly and preconditioner performance decreases gradually. Its implementation is straight-forward. It dynamically balances the cost of updates with the benefit of a reduced number of iterations.

Discussions with Knut Gjerden and Arne Stormo on the topic of preconditioners have been useful, and are greatly appreciated.

References
[1] Dullien F A L 1992 Porous Media: Fluid Transport and Pore Structure, 2nd ed. (Academic Press, San Diego)
[2] Sahimi M 1995 Flow and transport in porous media and fractured rock (VCH Verlagsgesellschaft)
[3] Wilkinson D and Willemsen J F 1983 Journal of Physics A: Mathematical and General 16 3365
[4] Witten T A and Sander L M 1981 Physical Review Letters 47 1400–1403
[5] Meijerink J A and van der Vorst H A 1977 Mathematics of Computation 31 148–162
[6] Aker E, Måloy K J, Hansen A and Batrouni G G 1998 Transport in Porous Media 32 163–186
[7] Knudsen H A, Aker E and Hansen A 2002 Transport in Porous Media 47 99–121