(Proceedings ModelCARE2011 held at Leipzig, Germany, in September 2011) (IAHS Publ. 355, 2012).

Improvement of performance and applicability of MODFLOW-2005: new NWT solver and χMD matrix solver package

MOTOMU IBARAKI[1], SORAB PANDAY[2], RICHARD G. NISWONGER[3] & CHRISTIAN D. LANGEVIN[4]

1 Ohio State University, 125 South Oval Mall, Columbus, Ohio 43210, USA
ibaraki.1@osu.edu
2 AMEC Environment & Infrastructure, 12801 Worldgate Drive, Suite 500, Herndon, Virginia 20170, USA
3 US Geological Survey, 2730 N. Deer Run Road, Carson City, Nevada 89701, USA
4 US Geological Survey, 411 National Center, Reston, Virginia 20192, USA

Abstract MODFLOW has been widely used for many years to investigate groundwater flow systems. In most numerical models, including MODFLOW, >80% of memory and execution time is used for the matrix solver; thus, improving matrix solver performance is a key to improve simulation performance. A χMD solver package was developed for higher robustness, faster execution speed, and better memory efficiency. The preconditioning module of χMD consists of level-based incomplete LU (ILU) factorization with a drop tolerance scheme that can reduce memory usage and lead to faster execution speed. The acceleration part consists of the conjugate gradient method, Bi-CGSTAB and ORTHOMIN. The χMD solver package is adapted for MODFLOW-2005 and preliminary results show that level-based ILU factorization with a drop tolerance scheme greatly reduce memory usage compared to ILU-only factorization by a factor of two or more. In addition, execution times decrease by 40% or more.

Key words matrix solver; numerical modelling; groundwater

INTRODUCTION

MODFLOW (Harbaugh, 2005) has been used for many years to solve groundwater flow problems and is a widely accepted standard for groundwater modelling. More than 80% of memory requirements and execution time of most numerical simulation programs, including MODFLOW, are due to the matrix solver. Hence, improvement of the matrix solver is a key for overall improvement of simulation performance. A good matrix solver satisfies three essential conditions: (1) robustness – capability to solve non diagonally-dominant "hard-to-solve" matrices, (2) shorter execution times, and (3) memory efficiency.

Nonstationary iterative methods, i.e. conjugate gradient-type methods, have shown their effectiveness compared to direct methods and stationary iterative methods, such as Successive Over Relaxation and Gauss-Seidel. In nonstationary iterative methods, a preconditioning component is paired with an acceleration component. Even if the acceleration methods guarantee that the solution will be obtained within a finite number of iterations, higher quality of preconditioning is necessary to reduce the number of iterations and the computational cost. The convergence rate of a nonstationary method greatly depends on the spectrum of the coefficient matrix.

In order to increase its convergence rate, preconditioning of the coefficient matrix, which transforms the coefficient matrix to a matrix that has a favourable spectrum, is performed. Although this preconditioning process adds an extra cost, high-quality preconditioning can reduce the overall execution time, and thus overcome the extra cost caused by preconditioning. Hence, "preconditioned" conjugate gradient type methods are widely used and lead to the improvement of both preconditioning and iterative (i.e. acceleration) parts and are necessary to achieve the conditions for a good matrix solver described above.

Most preconditioned conjugate gradient-type methods use incomplete LU factorization, which is similar to Gaussian elimination, but decomposition is terminated at a certain level. With a higher level of factorization, fewer iterations are required because the quality of preconditioning is higher; however, cost-per-iteration increases, as illustrated in Table 1.

Table 1 The effects of incomplete LU factorization on the factors which affect overall solver performance.

Level of fills in preconditioning	Lower	⇔	Higher
Quality of preconditioning	Lower	⇔	Higher
Robustness	Lower	⇔	Higher
Number of iterations in acceleration	Higher	⇔	Lower
Cost/iteration	Lower	⇔	Higher

This is because memory costs of higher level factorization are greater than those of lower level factorization because a greater number of new fill-in entries appear in the process of decomposition. A higher quality of preconditioning (i.e. a higher level of factorization) is advantageous because it shows higher robustness but cost-per-iteration is higher, as described above.

Therefore, even if a selected ILU factorization level has higher robustness and the number of iterations is decreased, the total computational cost, which is the product of the number of iterations and cost-per-iteration, is not necessarily reduced. An optimal combination of cost-per-iteration and number of iterations can be obtained at lower level (Fig. 1).

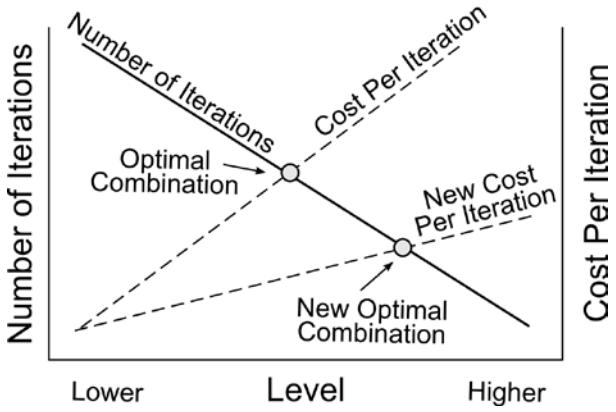

Fig. 1 Schematic diagram showing the relationship between level of fill, number of iterations, and cost per iteration.

If we can reduce cost per iteration (i.e. decrease the slope angle of the line representing cost-per-iteration) and obtain a new optimal combination point, as shown in Fig. 1, the total cost can be greatly reduced.

In order to achieve this goal, a drop tolerance scheme has been developed to obtain a higher level factorization (i.e. higher quality of preconditioning and less expensive computational work per iteration) was implemented (Munksgaard, 1980; Zlatev, 1982). In this scheme, the small new fill-in entries that tend to appear in higher factorization and have little or no effect on the quality of preconditioning are discarded. This leads to a decrease in execution time. Because the new optimal combination point will be obtained at higher level, the matrix solver can enhance its robustness, which is very important when we have to solve non diagonally-dominant "hard-to-solve" matrices.

A new χMD solver package (Niswonger *et al*., 2011) was developed for higher robustness, reduced execution times, and better memory efficiency. The preconditioning part of the solver includes ILU factorization with a drop tolerance scheme, which can reduce memory usage. The acceleration part consists of the conjugate gradient method (Hestenes & Stiefel, 1952) for a symmetric matrix, and Bi-CGSTAB (van der Vorst, 1992) or ORTHOMIN (Vinsome, 1976) for a non-symmetric matrix. The χMD solver package is adapted for both an Un-Structured Grid Version of MODFLOW (Panday *et al*., 2011) and MODFLOW-2005. Its performance was

compared with other matrix solvers in MODFLOW-2005 and examined through a regional groundwater flow simulation problem.

Solver performance comparisons

As the first step, the performance of the newly implemented χMD solver package was compared with the GCG (Hestenes & Stiefel, 1952) and GMRES (Saad & Schultz, 1986) matrix solvers available for MODFLOW-2005. Two test problems are used for the comparisons (Fig. 2). The 1-layer and complex test problems constitute 16 046 and 96 800 nodes, respectively. Note that GCG failed to converge for the 1-layer problem and GMRES L1 and L2 failed in the preconditioning section for the complex problem.

It can be seen from Fig. 2 that the χMD solver package shows better performance than the other solvers. In the 1-layer test problem, the execution time for the χMD solver package is about a factor of four less than the other solvers and by a factor of 14 in the complex test problem. It also shows robustness of the χMD solver package relative to other solver packages because other solver packages failed in the preconditioning or acceleration parts.

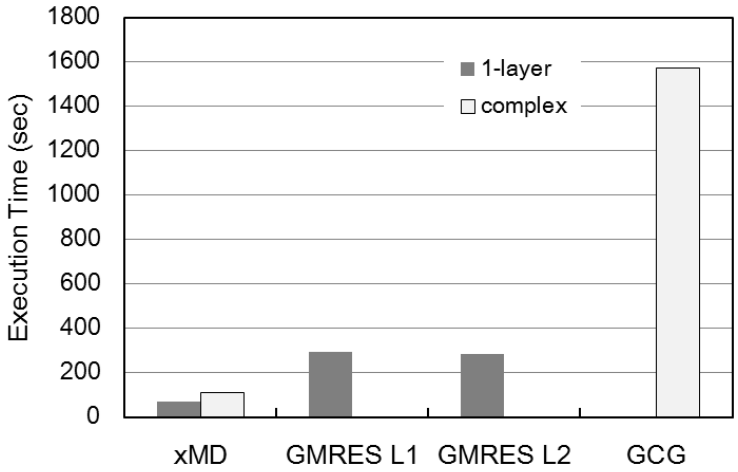

Fig. 2 Comparisons of execution time among matrix solvers. Note that GCG failed to converge for the 1-layer problem and GMRES L1 and L2 failed in the preconditioning section for the complex problem.

χMD SOLVER PERFORMANCE

Level-only preconditioning

In order to evaluate the effects of preconditioning in the χMD solver package, sensitivity analyses were performed. The simulation problem for these analyses consists of two layers of geological units with an area of 56.3 km^2. The domain was discretized with 730 000 model cells and contains 40 000 wells and 23 000 streams, which are represented as head-dependent boundary cells. 1170 cells are connected to the streams and evapotranspiration and recharge were simulated. All model layers are convertible between confined and unconfined conditions, and all cells remain active during wetting and drying of cells. The hydraulic conductivity of porous materials of the model domain ranges between 0.003 and 3 m/day.

Figure 3 illustrates the effects of the level of ILU on the number of elements in the preconditioning matrix (i.e. memory usage) and execution time. It can be seen from the diagram that as the level of ILU increases, the quality of preconditioning increases and leads to less iterations in the acceleration part, and accordingly, execution time decreases. We can see this trend up to the ILU level of four. However, an increase in elements in the preconditioning matrix results in an increase in cost-per-iteration. Although the number of iteration decreases as the quality of preconditioning matrix increases (i.e. the ILU level increases), the overall execution time does not decrease.

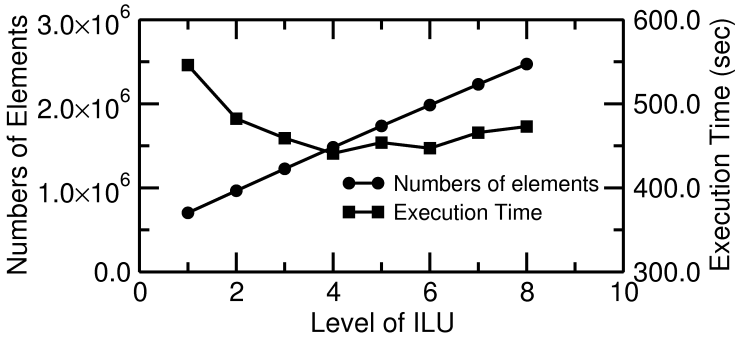

Fig. 3 Number of elements in preconditioning matrix (i.e. memory usage) and execution time for level-only preconditioning.

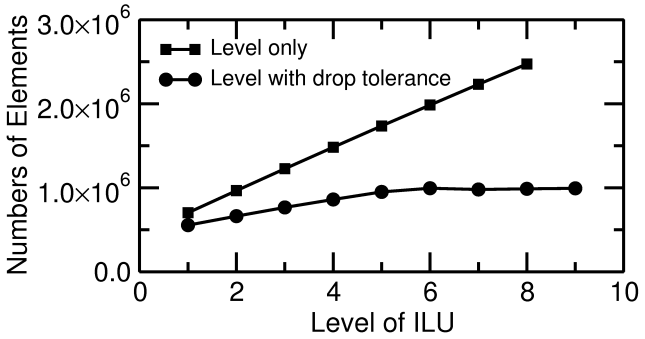

Fig. 4 Number of elements in preconditioning matrix (i.e., memory usage) for level only and level with drop tolerance preconditionings.

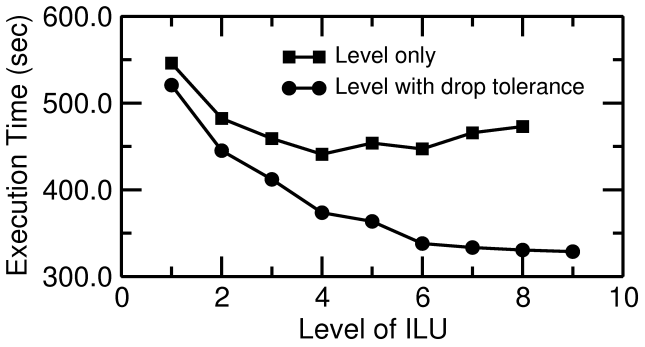

Fig. 5 Execution time for level only and level with drop tolerance preconditioning.

Level with drop tolerance pre-conditioning

As the level of ILU increases, new small fill-ins appear in the preconditioning matrix. In a drop tolerance scheme, small entries that have little or no effect on the quality of the preconditioning matrix are discarded to maintain the higher quality of the matrix while decreasing cost-per-iteration. Figure 4 illustrates the effects of the level of ILU on the number of elements in the preconditioning matrix.

It can be seen from the diagram that the number of elements in the preconditioning matrix increases as the level of ILU increases in a similar way to that observed in level-only preconditioning. However, the number of elements becomes constant when the value of the ILU level is greater than or equal to six because new small entries are dropped.

The effects of the level of ILU on execution time are shown in Fig. 5. The execution time, on the other hand, decreases as the level of ILU increases. Although we observed that the execution

time increases after the value of level of four in level-only preconditioning, this preconditioning shows that the execution time continues to decrease. This is because the quality of preconditioning becomes higher, which reduces the number of iterations in the acceleration part, while maintaining lower cost per iteration by discarding smaller new entries.

It can be seen from Fig. 4 that level-based ILU factorization with a drop tolerance scheme greatly reduces memory usage compared to ILU-only factorization by a factor of two or more. Furthermore, the execution time decreases by 40% or more (Fig. 5).

Sensitivity analysis of the drop tolerance value

We demonstrated that level-based ILU factorization with a drop tolerance scheme shows better performance compared to level-based ILU-only factorization. In order to examine the effect of drop tolerance value, a sensitivity analysis is performed. Figure 6 shows that the effect of the drop tolerance value on the number of elements in the preconditioning matrix (i.e. memory usage) and execution time. In this analysis, the ILU level is assigned a value of 10. It can be seen from the diagram that as the value of drop tolerance increases, the number of elements in the preconditioning matrix decrease. Accordingly, the execution time decreases because of a decrease in cost-per-iteration in the acceleration part.

However, an increase in the execution time is observed when the value of drop tolerance is greater than or equal to 4×10^{-4}. This increase in the execution time is caused by degradation in the quality of the preconditioning matrix. In the case of factorization with a larger drop tolerance value, the factorization discards not only smaller inconsequential entries, but also larger entries, which are essential to maintain the higher quality of the matrix. A degraded preconditioning matrix results in higher numbers of iterations in the acceleration part and increased overall execution time although cost-per-iteration is small. This indicates that choosing an appropriate value of tolerance is essential for this factorization scheme. Through a series of sensitivity analyses for other simulation cases, we found that an appropriate value of tolerance ranges from 1×10^{-4} to 1×10^{-3}.

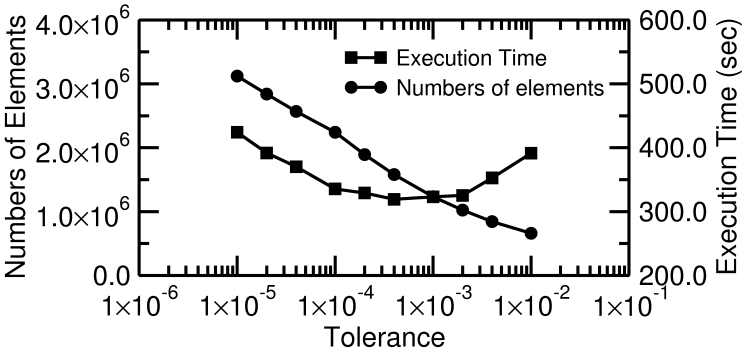

Fig. 6 Effects of drop tolerance value on memory usage and execution time.

CONCLUSIONS

A χMD solver package was developed for higher robustness, reduced execution times, and better memory efficiency. Its performance was compared with other matrix solvers available for MODFLOW-2005. In addition, its performance was examined through a series of sensitivity analyses. Through performance comparisons, the χMD solver package outperforms the other solvers evaluated by a factor of four or more in the execution time. In the sensitivity analyses, it is illustrated that a drop tolerance scheme significantly decreases memory usage and execution time. It is also shown that the value of drop tolerance and the level of ILU greatly affect overall execution time and memory usage. An appropriate choice for those values is essential to maximize performance.

A drawback of a drop tolerance scheme is that it is harder to implement in practice. This is because the amount of storage needed for the ILU factorization is not easy to predict since entries in the coefficient matrix and new fill-in entries appear in the process of decomposition are not known *a priori*. However, this drawback can be overcome by using programming languages which can handle dynamic memory allocation, such as C and newer FORTRAN90 programing languages. The amount of storage needed in the factorization process involving a drop tolerance scheme is efficiently handled in the χMD solver package, which is written in FORTRAN90.

Although we showed performance of the χMD solver package which was adapted for MODFLOW-2005, the χMD solver package can be adapted for any numerical simulator that solves matrices which are assembled through numerical discretization processes. As demonstrated in this paper, it has a potential to improve performance of models with respect to execution time, robustness, and memory usage.

REFERENCES

Harbaugh, A. W. (2005) MODFLOW-2005, the US Geological Survey modular ground-water model-the Ground-Water Flow Process: US Geological Survey Techniques and Methods 6-A16 (variously paginated).

Hestenes, M. R. & Stiefel, E. (1952) Methods of conjugate gradients for solving linear systems. *J. Res. National Bureau of Standards* 49, 409–436.

Munksgaard, N. (1980) Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients. *ACM Trans Mathematical Software* 6, 206–219.

Niswonger, R.G., Panday, S. & Ibaraki, M. (2011) MODFLOW-NWT, A Newton Formulation for MODFLOW-2005. US Geological Survey Techniques and Methods 6-A37, 44 p.

Panday, S., Niswonger, R. G., Langevin, C. D. & Ibaraki M. (2011) An Un-Structured Grid Version of MODFLOW, MODFLOW and More 2011.

Saad, Y. & Schultz, M. H. (1986) GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput* 7(3), 856–869.

van der Vorst, H. A. (1992) Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing* 13(2), 631–644.

Vinsome, P. K. W. (1976) ORTHOMIN—An iterative method for solving sparse banded set of simultaneous linear equations. In: *Fourth SPE Symposium on Numerical Simulation of Reservoir Performance* (Los Angeles, 19–20 February), 149–159, SPE. paper SPE 5729.

Zlatev, Z. (1982) Use of iterative refinement in the solution of sparse linear systems. *SIAM J. Numerical Analysis* 19, 381–399.