PRECONDITIONERS FOR ITERATIVE SOLUTIONS OF LARGE-SCALE LINEAR SYSTEMS ARISING FROM BIOT'S CONSOLIDATION EQUATIONS

Chen Xi

NATIONAL UNIVERSITY OF SINGAPORE
2005

PRECONDITIONERS FOR ITERATIVE SOLUTIONS OF LARGE-SCALE LINEAR SYSTEMS ARISING FROM BIOT'S CONSOLIDATION EQUATIONS

Chen Xi

(B.Eng.,M.Eng.,TJU)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF CIVIL ENGINEERING
NATIONAL UNIVERSITY OF SINGAPORE
2005

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Associate Professor Phoon Kok Kwang, for his continuous encourage, guidance and patience. Without his support and help during this period, the accomplishment of the thesis could not be possible. I would like to thank my co-supervisor, Associate Professor Toh Kim Chuan from the Department of Mathematics (NUS), for he has shared with me his vast knowledge and endless source of ideas on preconditioned iterative methods. The members of my thesis committee, Associate Professor Lee Fook Hou, Professor Quek Ser Tong and Associate Professor Tan Siew Ann, deserve my appreciation for their advices on my thesis.

Thanks especially should be given to my family and my friends for their understanding. Though someone could be left unmentioned, I would like to mention a few: Duan Wenhui and Li Yali deserve my acknowledgement for their help and encourage during the struggling but wonderful days. Zhang Xiying also gave me great influence and deserve my appreciation. I wish to thank Zhou Xiaoxian, Cheng Yonggang, and Li Liangbo for the valuable discussions with them.

TABLE OF CONTENTS

SUMMARY

The solution of the linear systems of equation is the most time-consuming part in large-scale finite element analysis. The development of efficient solution methods are therefore of utmost importance to the field of scientific computing. Recent advances on solution methods of linear systems show that Krylov subspace iterative methods have greater potentials than direct solution methods for large-scale linear systems. However, to be successful, a Krylov subspace iterative method should be used with an efficient preconditioning method.

The objective of this thesis was to investigate the efficient preconditioned iterative strategies as well as to develop robust preconditioning methods in conjunction with suitable iterative methods to solve very large symmetric or weakly nonsymmetric (which is assumed to be symmetric) indefinite linear systems arising from the coupled Biot's consolidation equations. The efficient preconditioned iterative schemes for large nonlinear consolidation problems also deserve to be studied. It was well known that the linear systems discretized from Biot's consolidation equations are usually symmetric indefinite, but in some cases, they could be weakly nonsymmetric. However, irrespective of which case, Symmetric Quasi-Minimal Residual (SQMR) iterative method can be adopted. To accelerate the convergence of SQMR, a block constrained preconditioner P_c which was proposed by Toh *et al.* (2004) recently was used and compared to Generalized Jacobi (GJ) preconditioner (e.g. Phoon *et al.*, 2002, 2003). P_c preconditioner has the same block structure as that of the original stiffness matrix, but with the (1, 1) block replaced by a diagonal approximation. As a further development of P_c, a Modified Symmetric Successive Over-Relaxation (MSSOR) preconditioner which modifies the diagonal parts of standard SSOR from a theoretical perspective was developed. The widely investigated numerical experiments show that MSSOR is extremely suitable for large-scale consolidation problems with highly varied soil properties. To solve the large nonlinear consolidation problems, Newton-Krylov (more accurately, Newton-SQMR) was proposed in conjunction with GJ and MSSOR preconditioners. Numerical experiments were carried out based on a series of large problems with different mesh sizes and also on more demanding heterogeneous soil conditions. For large nonlinear consolidation problems based on modified Cam clay model and ideal von Mises model, the performance of the Newton-SQMR method with GJ and MSSOR preconditioners was compared to Newton-Direct solution method and the so-called composite Newton-SQMR method with P_B (e.g. Borja, 1991). Numerical results indicated that Newton-Krylov was very suitable for large nonlinear problems and both GJ and MSSOR preconditioners resulted in faster convergence of SQMR solver than available efficient P_B preconditioner. In particular, MSSOR was extremely robust for large computations of coupled problems. It could been expected that the new developed MSSOR preconditioners can be readily extended to solve large-scale coupled problems in other fields.

Keywords: Biot's consolidation equation, nonlinear consolidation, three-dimensional finite element analysis, symmetric indefinite linear system, iterative solution, Newton-Krylov, quasi-minimal residual (QMR) method, block preconditioner, modified SSOR preconditioner

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

$\perp$	Orthogonality
$\{\cdot\}$	A vector
$(\cdot, \cdot)$	Inner product of two vectors
$\lvert\cdot\rvert$	Absolute value or modulus of a number
$\lVert\cdot\rVert$	Some kind of norm
$\lVert\cdot\rVert_2$	2-norm
$\mathbf{0}$	Zero vector
$\mathbf{1}$	Second-order Kronecker delta in vectorial form
2-D	2-dimensional
3-D	3-dimensional
$\mathbf{a}$	Gradient to the yield surface
A	Coefficient matrix or coupled stiffness matrix
AMD	Approximate minimal degree
b	Right hand side vector
$\mathbf{b}$	Vector $\mathbf{b} = \{0, 0, 1\}^T$ or the gradient to plastic potential
$\mathbf{b}_w$	Body force vector of pore fluid
$\mathbf{b}_s$	Body force vector of soil skeleton
$\mathbf{B}_p$	Derivative matrix of $\mathbf{N}_p$ about coordinates
$\mathbf{B}_u$	Strain-displacement matrix
BE	Boundary element
BFGS	Broyden-Fletcher-Goldfarb-Shanno
Bi-CG	Biconjugate gradient
Bi-CGSTAB	Biconjugate gradient stablized
$\mathbf{c}$	Gradient of plastic potential to internal variable
CG	Conjugate Gradient
CGNE	CG for normalized equation with A^T multiplied from right

CGNR	CG for normalized equation with A^T multiplied from left
CGS	Conjugate gradient squared
CPU	Central processing unit
CSC	Compress Sparse Column
CSR	Compress Sparse Row
D	Strictly diagonal part of a matrix
$\mathcal{D}$	Block diagonal part of a matrix
$\mathbf{D}^e$	Elastic stress-strain matrix
$\mathbf{D}^{ep}$	Elasto-plastic stress-strain matrix
$\mathbf{D}^{epc}$	Consistent tangent matrix
$\mathbf{D}^p$	Plastic stress-strain matrix
D-ILU	Diagonal ILU
$diag(\cdot)$	Diagonal matrix of a matrix
$div(\cdot)$	Divergence
e_i	$e_i = \{0, \ldots, \underbrace{1}_{i-th}, \ldots, 0\}^T$
e_k	Error vector defined by $e_k = x - x_k$
E'	Effective Young's modulus
EBE	Element-by-element
f	Load vector or yield surface function
$\mathbf{f}$	Load vector
$flops$	The number of floating point operations
g	Fluid flux or plastic potential function
$\mathbf{G}$	Global fluid stiffness matrix
$\mathbf{G}_e$	Element fluid stiffness matrix
G'	Elastic shear modulus
GJ	Generalized Jacobi
GMRES	Generalized minimal residual
H	Hessenberg matrix or preconditioning matrix
$\mathbf{I}$	Identity matrix
$\mathbb{I}_d$	The deviatoric component in $\mathbf{I}$
$\mathbb{I}_v$	The volumetric component in $\mathbf{I}$
IC	Incomplete Cholesky
ICCG	Incomplete Cholesky preconditioned conjugate gradient

ILU	Incomplete LU
ILUM	Multi-elimination ILU
ILUT	Incomplete LU with dual threshold
J	J matrix
$\mathbf{K}$	Global soil stiffness matrix
$\mathbf{K}_e$	Element soil stiffness matrix
K'	Elastic bulk modulus
$\mathcal{K}_k$	k-th Krylov subspace
$[\mathbf{k}]$	Permeability matrix
L	Strictly lower triangular part of a matrix
$\mathbf{L}$	Global connection or coupled matrix
$\mathbf{L}_e$	Element connection or coupled matrix
$\mathcal{L}$	Block lower triangular part of a matrix
M	Preconditioning matrix or slope of the critical state line
M_{Jac}	Jacobi iteration matrix
M_{GS}	Gauss-Seidel iteration matrix
M_{SOR}	SOR iteration matrix
M_{SSOR}	SSOR iteration matrix
max_it	Permitted maximum iteration number
MD	Minimal degree
MINRES	Minimal residual
MJ	Modified Jacobi
MMD	Multiple minimal degree
MSSOR	Modified symmetric successive over-relaxation
n	Dimension of a matrix
$\mathbf{n}$	Unit outward normal
$\mathbf{N}_p$	Pore pressure shape function vector for fluid element
$\mathbf{N}_u$	Displacement shape function matrix for solid element
ND	Nested Dissertation
$nnz(\cdot)$	The number of nonzero entries of a matrix
p	Total pore water pressure with $p = p^{st} + p^{ex}$
$\mathcal{P}$	Permutation or reordering matrix
p'	Mean effective stress

p'_c	Preconsolidation pressure
$\mathbf{p}_e$	Vector of element nodal pore water pressure
p^{st}	Static pore water pressure
p^{ex}	Excess pore water pressure
P_c	Block constrained preconditioner
P_d	Block diagonal preconditioner
P_{GJ}	Generalized Jacobi preconditioner
P_{MJ}	Modified Jacobi preconditioner
P_{MBSSOR}	Modified block SSOR preconditioner
P_{MSSOR}	Modified SSOR preconditioner
P_{SSOR}	SSOR preconditioner
P_t	Block triangular preconditioner
$P_k(\cdot)$	A k-degree polynomial satisfying $P_k(0) = 1$
PCG	Preconditioned conjugate gradient
PCR	Preconditioned conjugate residual
q	Deviatoric stress
$\mathbf{q}$	Vector of internal variable
$Q_j(\cdot)$	A j-degree polynomial satisfying $Q_j(0) = 1$
QMR	Quasi-minimal residual
QMR-CGSTAB	Quasi-minimal residual variant of the Bi-CGSTAB algorithm
r	Residual vector
$\mathbb{R}$	Set of real numbers
$\mathbb{R}^n$	vector space of real n-vectors
$\mathbb{R}^{n \times n}$	vector space of real n-by-n matrices
RAM	Random access memory
RCM	Reverse Cuthill-McKee
s	Shadow residual vector or stress deviator
S	Schur complement matrix
$\mathcal{S}$	Sparsity set of a matrix
S_e	Element surface boundary domain
SJ	Standard Jacobi
SOR	Successive over-relaxation
SPAI	Sparse approximate inverse

$span\{\cdots\}$	Space spanned by the given vectors
SPD	Symmetric positive definite
SQMR	Symmetric quasi-minimal residual
SSOR	Symmetric successive over-relaxation
stop_tol	Preset convergence tolerance
SYMMLQ	Symmetric LQ
$\mathbf{t}$	Vector of surface traction force
T	Tridiagonal matrix
$tr(\cdot)$	Trace of a vector
$\mathbf{u}$	Global nodal displacement vector
$\mathbf{u}_e$	Nodal displacement vector for solid element
U	Strictly upper triangular part of a matrix
$\mathcal{U}$	Block lower triangular part of a matrix
v	A vector
$\mathbf{v}$	Vector of superficial fluid velocity, $\mathbf{v} = \{v_x, v_y, v_z\}^T$
V_e	Element volume domain
w_p	Weighting scalar for pore pressure
$\mathbf{w}_u$	Weighting function vector for displacement
x	Solution vector or local spatial coordinate
α	Real number or scaler
β	Real number or scalar
γ	Real number or scalar
γ_w	Unit weight of pore water
γ_s	Unit weight of soil
δ	Small change in one variable
Δ	Increment of one variable
$\Delta\lambda$	Plastic multiplier
$d\lambda$	Differential of $\Delta\lambda$
ε_v	Volumetric strain
$\boldsymbol{\varepsilon}$	Strain vector
$\boldsymbol{\varepsilon}^e$	Elastic component of $\boldsymbol{\varepsilon}$
$\boldsymbol{\varepsilon}^p$	Plastic component of $\boldsymbol{\varepsilon}$
ζ	Real scalar

η	Real relaxation parameter
θ	Time integration parameter
ϑ	A scalar
κ	The swelling or recompression index
λ	The eigenvalue of a matrix
$\nu^{'}$	Effective Poisson's ratio
$\prod$	Continuous product
$\rho(\cdot)$	The spectral radius of a matrix
ϱ	a scalar with the value, $\varrho = \\|r_0\\|$
$\boldsymbol{\sigma}'$	Effective stress vector
$\sigma_i^{'}$	Normal effective stress in i-direction $(i = x, y, z)$
τ_i	Shear stress in i-direction $(i = xy, yz, zx)$
ξ	Real relaxation parameter
χ	The virgin compression index
Φ	Schwaz-Christoffel map
ω	Relaxation parameter or the weighted average parameter
ω_{opt}	Optimal relaxation parameter
$\sum$	Summation
$\widetilde{\nabla}$	A differential operator
$\int$	Integration in some kind of domain
$\otimes$	Tensor product
$\bar{\cdot}$	Permuted matrix
$\widehat{\cdot}$	Approximation of a matrix or a matrix with threshold
$\widetilde{\cdot}$	Preconditioned matrix

To my family

CHAPTER 1

INTRODUCTION

With the rapid developments of computer and computational technology, ordinary desktop computers have been widely used to solve engineering problems by scientists and engineers. It is well known that in finite element (FE) software packages, the solution of linear systems is one of the three classes of computationally intensive processes [1] (e.g. Smith, 2000). The solution of linear equations has received significant attentions because fast and accurate solution of linear equations is essential in engineering problems and scientific computing. Traditionally, direct solution methods are preferred to linear system of equations.

A general form of linear system can be given as

$$Ax = b, \quad A \in \mathbb{R}^{n \times n} \, \text{and} \, x, b \in \mathbb{R}^n \tag{1.1}$$

It is uncertain whether direct solutions or iterative solutions are better because the boundaries between these two main classes of methods have become increasingly blurred (e.g. Benzi, 2002). However, as a result of recent advances of iterative methods and preconditioning techniques in scientific computing and engineering, more applications are turning to iterative solutions for large-scale linear systems arising from geotechnical engineering.

The traditional way to solve a non-singular square linear system is to employ direct solution methods or its variants which are based on the classical Gaussian elimination scheme. These direct methods can lead to the exact solution in the absence of roundoff

[1] The three classes of computationally expensive process are solution of linear equations, solution of eigenvalue equations and integration of ordinary differential equations in time domain

errors. However, especially for large sparse linear systems arising from 3-D problems, direct solution methods may incur a large number of fill-ins, and the large order n of the matrix makes it expensive to spend about n^3 floating point operations (additions, subtractions and multiplications) to solve such a large linear system. Therefore, for direct solution methods, the computing cost and memory requirement increase significantly with the problem size. On the contrary, iterative solution methods are becoming attractive for such large-scale linear equations because only matrix-vector products and inner-products are required in the iteration process.

1.1 Preconditioned Iterative Solutions in Geotechnical Problems

In recent years, preconditioned iterative methods have been used for some geotechnical problems. Wang (1996) successfully used element-by-element (EBE) based Preconditioned Conjugate Gradient (PCG) method to solve very large 3-D finite element pile groups and pile-rafts problems. The author also studied the characteristics of PCG in elasto-plastic analysis and concluded that although the PCG method was about 30% slower than the direct method for plastic analysis at that time, but he was very confident that with some improvements, PCG can be made faster in plastic analysis in the future. Payer and Mang (1997) proposed two iterative schemes to perform boundary-element (BE) and hybrid BE-FE simulation of excavations for tunneling. The investigated iterative methods were Generalized Minimum Residual (GMRES), Conjugate Gradient Square (CGS) and Stabilized Bi-CG (Bi-CGSTAB) preconditioned by diagonal scaling, Symmetric Successive Over-Relaxation (SSOR) as well as incomplete factorization, respectively. Hladík *et al.* (1997) suggested replacing direct solutions by preconditioned iterative methods in finite element packages. They tested Conjugate Gradient (CG) solver preconditioned by two different forms of preconditioners: one is global Incomplete Cholesky (IC) type preconditioners and the other is symmetrically scaled EBE based preconditioners. However, it was proved that the proposed EBE based preconditioner is less efficient than IC on a serial computer, but EBE storage is easily adapted to parallel computing. A range of numerical examples from geotechnical engineering problems showed that the Robust Incomplete Cholesky (RIC) preconditioner with a near-optimal choice of the relaxation parameter can be very efficient and reliable for practical 2-D and 3-D geotechnical problems in elasticity. Smith and Wang (1998) analyzed the piled rafts in their full three-dimensional complexity by PCG solver, but the numerical experiments were carried out on a parallel computer with the "element-by-element" or "mesh-free" strategies. By means of these strategies, the need to assemble the global coefficient matrix is removed. Kayupov *et al.* (1998) used CGS and GMRES iterative methods with simple Jacobi preconditioning to enhance Indirect Boundary Element Method (IBEM) for underground construction problems. They concluded that CGS and GMRES with simple Jacobi preconditioning appeared to be efficient and robust. By using sparse iterative methods, Mroueh and Shahrour (1999) studied the resolution of three-dimensional soil-structure interaction problems: a shallow foundation under a vertical flexible loading, a single pile subjected to a lateral loading and construction of a lined tunnel in soft ground. Because the elastic-perfectly plastic Mohr-Coulomb model was assumed for the soil materials, the resulting linear systems being nonsymmetric or symmetric depends on whether the plastic flow is non-associated or associated. For the sparse linear systems arising from these interaction problems, Bi-CG, Bi-CGSTAB, QMR-CGSTAB have been used, and the performances of SSOR and Jacobi preconditioners are investigated and compared. Numerical results show that left preconditioned SSOR preconditioner gives better performance compared to Jacobi preconditioner for the resolution of soil-structure interaction problems with high varied material heterogeneity and plasticity. Furthermore, for a full 3-D finite element analysis of the interaction between tunnels and adjacent structures, the authors proposed to use sparse Bi-CGSTAB solver coupled with the SSOR preconditioning.

1.2 Preconditioned Iterative Solutions in Biot's Consolidation Problems

In geotechnical engineering, the solution of Biot's consolidation problems has played an essential role since the pioneer work of Terzaghi (1925) and Biot (1941). To calculate soil settlement accompanied with dissipating pore water pressure, Terzaghi (1925) developed one-dimensional consolidation theory by introducing the interaction between soil skeleton and pore water pressure through the principle of effective stress. On the basis of Terzaghi's work and the continuity equation, Biot proposed three dimensional consolidation theory which have received wide applications in many engineering problems (e.g. Sandhu and Wilson, 1969; Abbo, 1997; Lewis and Schrefler, 1998). In Biot's theory, the interaction between the soil skeleton and pore water pressure is assumed to be dominated by the principle of effective stress and the continuity relationship. Therefore, soil consolidation process is time-dependent. Fast solutions of large coupled linear equations arising from 3-D Biot's consolidation problems are clearly of major pragmatic interest to engineers.

When Biot's consolidation equations are discretized by finite element method (FEM) in space domain and finite difference method in time domain, the coupled linear equations are coupled with displacement and pore water pressure unknown (e.g. Borja, 1991; Lewis and Schrefler, 1998; Zienkiewicz and Taylor, 2000). These resultant linear systems of equations are symmetric indefinite (e.g. Abbo, 1997; Smith and Griffiths, 1998), or sometimes, nonsymmetric indefinite (e.g., Gambolati *et al.*, 2001, 2002, 2003).

For nonlinear consolidation problems, Borja (1991) compared three different solution schemes: Newton with direct solution for linearized linear systems, composite Newton-PCG method and quasi-Newton method with Broyden-Fletcher-Goldfarb-Shanno (BFGS) inverse updating. Based on numerical experiments, the author concluded that the composite Newton-PCG technique, in which the tangent stiffness matrix at the first iteration of each time step was used as a preconditioner throughout the time step, possessed considerable potentials for large-scale computations. Smith and Griffiths (1998) employed PCG method preconditioned by Jacobi preconditioner based on EBE strategy to solve symmetric indefinite linear equations arising from 2-D consolidation problems. Furthermore, EBE based Jacobi preconditioned CG method was extended to solve large 3-D consolidation problems in parallel environment (e.g. Smith, 2000; Smith and Griffiths, 2004). Gambolati *et al.* (2001, 2002, 2003) studied the solution of nonsymmetric systems arising from Biot's coupled consolidation problems in a series of papers. The studies ranged widely including the investigation of the correlation between the ill-conditioning of FE poroelasticity equations and the time integration step, the nodal ordering effects on performance of Bi-CGSTAB preconditioned by Incomplete LU factorization with Threshold (ILUT) preconditioner (e.g. Saad, 1996), comparison study of direct, partitioned and projected solution to finite element consolidation models, and the diagonal scaling effect when incomplete factorization is used as a preconditioning technique. Because maintaining symmetry of linear systems can preserve computing and storage efficiency, symmetric indefinite formulation of 3-D Biot's consolidation problems was studied by Chan *et al.* (2001) and Phoon *et al.* (2002, 2003). For symmetric indefinite Biot's linear equations, a cheaper symmetric iterative method, Symmetric Quasi-Minimal Residual (SQMR) by Freund and Nachtigal (1994), was adopted. To combine with the SQMR solver, the authors developed two EBE-based efficient diagonal preconditioners, namely Modified Jacobi (MJ) and Generalized Jacobi (GJ) preconditioner. Numerical analyses and experiments showed that the two preconditioners performed far better than Standard Jacobi (SJ) preconditioner especially for large and ill-conditioned Biot's linear systems. A recent study by Toh *et al.* (2004) systematically investigated three forms of block preconditioners, namely, block diagonal preconditioner, block triangular preconditioner and block constrained preconditioner, for symmetric indefinite Biot's linear systems, and proposed correspondent efficient implementations.

The above review presents the recent advances on preconditioned iterative methods for Biot's consolidation problems. However, there are some important issues that need to be addressed. For example, for symmetric indefinite linear systems derived from 3-D Biot's consolidation problems, the specific performances of preconditioned iterative methods based on partitioned and coupled Biot's formulations have not been investigated. Secondly, although MJ and GJ preconditioned SQMR methods are significant improvements over SJ preconditioned counterpart, the convergence rates of the diagonal preconditioned methods may be still slow compared to sophisticated non-diagonal preconditioning methods, especially for more practical soil-structure interaction problems with highly varied material properties. In addition, the two proposed preconditioners were proposed based on EBE techniques, their applications with global sparse techniques have not been studied. Thirdly, the popular SSOR preconditioned methods recommended (e.g. Mroueh and Shahrour, 1999) for solving soil-structure interaction problems may not be effective for Biot's linear systems, and sometimes, breakdown can be observed. Fourthly, ILU-type preconditioning techniques have been shown to be effective in accelerating the convergence rate for an iterative method. However, solving linear systems with ILU-type preconditioners may incur some difficulties such as the need to choose proper pivoting strategies, the storage requirement may increase significantly if a small dropping tolerance is chosen, and the resultant iteration reduction may easily be counteracted by the increased computing cost in each iteration. Thus, it is difficult to address the inefficiency of ILU-type preconditioning methods on a single processor (e.g. Eijkhout, 2003). Last but not least, preconditioning techniques for large nonlinear consolidation problems are not well studied.

1.3 Objectives and Significance

It should be emphasized that although the studied problem in this thesis is large 3-D Biot's consolidation, the developed methodology in this thesis has a wide variety of applications in engineering problems and scientific computations. Because Biot's consolidation problem can be categorized into saddle point problems, the preconditioners developed in this thesis can be readily applied to this kind of problems. Benzi *et al.* (2005) gave a complete list of those applications which may lead to saddle point problems. This list of applications includes computational fluid dynamics (e.g. Elman *et al.*, 1997), constrained optimizations (e.g. Toh and Kojima, 2002; Toh, 2003), economics, finance, image processing, mixed finite element approximations of PDEs (e.g. Brezzi and Fortin, 1991; Perugia and Simoncini, 2000; Warsa *et al.*, 2002; Wang, 2004), parameter identification problems and so on.

The objectives of this thesis can be summarized as follows:

(a) To give a detailed comparison between block partitioned and global Krylov subspace iterative methods for discretized Biot's symmetric indefinite linear systems, and to suggest the efficient implementation for such symmetric indefinite linear systems.

(b) To develop more efficient preconditioners than the preconditioners proposed in the recent literatures in conjunction with the chosen iterative solver. In the past decade, much attention has been devoted to develop general preconditioners (e.g. Saad, 1996; Saad and van der Vorst, 2000; Benzi, 2002). A good preconditioner, however, should also exploit properties in the physical problem. Therefore, this thesis is to develop such good problem-dependent preconditioners.

(c) To carry out some application studies on large-scale 3-D linear elastic as well as nonlinear Biot's consolidation problems. The performance of the developed preconditioners will be investigated and compared to some available preconditioning methods in the future.

The numerical experiments carried out in this thesis were based on ordinary desktop platform. Thus, the proposed preconditioning methods, related conclusions and numerical results should be useful for the purpose of solving large-scale linear systems stemming from geotechnical engineering problems on a serial computer. These developed preconditioners may also be useful to enhance the preconditioned iterative solvers adopted in current engineering software packages. Furthermore, the ideas behind the proposed preconditioning techniques could be helpful in laying the groundwork for developing advanced preconditioning methods.

1.4 Computer Software and Hardware

The FORTRAN source codes for three-dimensional FEM Biot's consolidation problems are based on the 2-D version given by Smith and Griffiths (1998). The FORTRAN codes on 2-D and 3-D Biot's consolidation problems are programmed by using Compaq Visual FORTRAN Professional Edition 6.6A (2000, Compaq Computer Corporation) and listed in Appendix C. The other software packages used in this thesis are listed as follows:

HSL Packages HSL (Harwell Subroutine Library) is a collection of ISO Fortran codes for large scale scientific computation. A free version is provided at http://www.cse.clrc.ac.uk/nag/hsl/

ORDERPACK ORDERPACK contains sorting and ranking routines in Fortran 90 and the package can be downloaded at http://www.fortran-2000.com/rank/index.html

SPARSKIT A basic tool-kit for sparse matrix computations. The software package can be obtained from Yousef Saad's homepage http://www-users.cs.umn.edu/∼saad/software

SparseM The software package is a basic linear algebra package for sparse matrices and can be obtained from http://cran.r-project.org/ or http://www.econ.uiuc.edu/∼roger/research/sparse/sparse.html

Template The software is a package for some popular iterative methods in Fortran, Matlab and C, and can be used to demonstrate the algorithms of the Template book (See http://www.netlib.org/templates/).

Except for Chapter 4, for which numerical experiments are performed on a Pentium IV, 2.0 GHz desktop computer, all numerical experiments are carried out at a Pentium IV, 2.4 GHz desktop computer. For these numerical studies, 1 GB physical memory without virtual memory is used.

1.5 Organization

This thesis is organized as follows. Chapter 2 gives a background of preconditioned iterative methods, and some popular iterative methods and some preconditioning methods are investigated for symmetric and nonsymmetric linear systems in many applications. Chapter 3 compares some applicable iterative methods for symmetric indefinite linear systems with coupled block coefficient matrix. Chapter 4 compares the performance between block constrained preconditioner, P_c and GJ preconditioner in details. Because the practical geotechnical problem is related to multi-layer heterogeneous soil conditions, Chapter 5 proposes a modified SSOR preconditioner which will be proved to be very robust in large-scale consolidation problems. To further study the numerical performance of GJ and MSSOR preconditioners in large nonlinear consolidation, Chapter 6 presents to use Newton-Krylov method in conjunction with these effective preconditioners and compares to available solution strategies. Finally, Chapter 7 gives a closure with some valuable conclusions and suggestions on future research work.

CHAPTER 2

OVERVIEW OF PRECONDITIONED ITERATIVE METHODS FOR LINEAR SYSTEMS

To solve a linear system, it is necessary to have some basic knowledge of the structure and the properties of the coefficient matrix so that the solution time and storage usage can be minimized as much as possible. Some special matrix structures or matrix properties can be summarized as

(a) **Symmetric positive definite matrix**

A matrix, which satisfies $A = A^T$ and $v^T Av > 0$ for an arbitrary vector $v \neq 0$, is *Symmetric Positive Definite* (SPD). By making use of symmetry, the solution time and storage usage can be halved compared to direct Gaussian elimination. For a SPD matrix, Cholesky factorization can achieve about 50% reduction.

(b) **Symmetric indefinite matrix**

If a symmetric matrix is neither positive definite nor negative definite, the matrix is called as *symmetric indefinite*. Similar to a SPD linear system, the solution time and storage usage can be reduced by making use of the symmetry, but subtle and complicated implementations such as pivoting strategies have to be applied (e.g. Duff and Reid, 1983; Demmel, 1997; Duff and Pralet, 2004).

(c) **Band matrix**

Band matrix is defined as the matrix with a bandwidth or a half bandwidth (symmetric) that is smaller than the dimension n. By making use of "banded" property, the runtime and storage can be reduced significantly. So far, this property has been extensively exploited for direct solutions.

(d) **Block matrix**

Organizing unknowns in terms of some properties such as unknowns type can lead to partitioned or block matrix, some popular methods may have their corresponding block variants, such as block Gaussian elimination, block SSOR and block ILU method and so on. Recent research showed that block implementations can be more suitable for parallel computing than their pointwise counterparts. However, block matrix implementations can also lead to some improvement on serial computers.

(e) **Sparse matrix**

A matrix with a large number of zero entries is called a *sparse matrix.* Incorporating sparse techniques to some methods leads to their sparse variants, for instance, direct Cholesky factorization can be extended to sparse Choleksy factorization by exploiting sparse storage and implementation. For iterative methods, sparse matrix-vector multiplications can lead to large reductions in solution time.

(f) **Dense and smooth matrix**

In some cases, dense matrices are unfavorable to storage and computing. However, dense and smooth (neighboring matrix entries are close in magnitude) can be viewed as a favorable property because fast wavelet transformation can be used in the construction of preconditioner or inexact matrix-vector multiplications of iterative methods (e.g. Chan *et al.*, 1998; Chen, 1999; van den Eshof *et al.*, 2003). In the application of inexact matrix-vector products, the true linear system to be solved is perturbed unless no entries are dropped after wavelet transforming.

It is obvious that a coefficient matrix may possess one or more properties described above. For a large linear system with sparse coefficient matrix, preconditioned iterative methods show significant potentials in computing and memory efficiencies.

2.1 Overview of Iterative Methods

The importance of iterative methods can be summarized very well by the quotation,

> "*. . . Iterative methods are not only great fun to play with and interesting objects for analysis, but they are really useful in many situations. For truly large problems they may sometimes offer the only way towards a solution, . . .*"
>
> H.A. van der vorst, 2003

It has been gradually recognized that for very large, especially sparse, linear systems, iterative methods come into the stage as an leading actor. The earliest iterative methods can be traced back to the great mathematician and physicist, Gauss, who demonstrated the basic idea behind iterative methods. Gauss' idea can be explained by making use of the matrix expression which has not been developed at his time. Firstly, we may look for an approximate nearby and more easily solved system, $Mx_0 = b$, instead of the original linear system $Ax = b$. Secondly, we correct the approximate solution x_0 with $x = x_0 + \delta x_0$, which results in a new linear system $A\delta x_0 = b - Ax_0 = r_0$ to be solved. Naturally, we can still use the approximate linear system $M\delta x_1 = r_0$ and lead to the iteration $x_1 = x_0 + \delta x_1$. By repeating this process, we can hopefully arrive at the desired solution. Different choice of M leads to different method, by choosing $M = diag(A)$, we get the Gauss-Jacobi method, while by choosing M to be the lower or upper triangular part of A, we find the Gauss-Seidel method. Although these methods have a long history, they are still widely employed in various practical applications.

Iterative methods comprise a wide variety of techniques ranging from classical iterative methods such as Jacobi (or Gauss-Jacobi), Gauss-Seidel, Successive Overrelaxation (SOR) and Symmetric SOR (SSOR) iterations, to comparatively more recent development such as Krylov subspace methods, multigrid and domain decomposition methods (e.g. Barrett *et al.*, 1994; Saad, 2003). Generally, iterative methods can be classified into two basic categories: classical stationary iterative methods and non-stationary iterative methods.

2.1.1 Stationary Iterative Methods

Stationary iterative methods are traditional methods for the solution of a linear system, $Ax = b$ with square and nonsingular coefficient matrix A. These iterative methods are called "stationary" because they follow the same recipe during the iteration process.

In this category, a well-known method is the Richardson iteration (1910): Given a strictly positive number $\alpha \in \mathbb{R}$, then the iteration is defined by

$$x_{k+1} = x_k + \alpha(b - Ax_k) = x_k + \alpha r_k \tag{2.1}$$

The method converges for α in the range $0 < \alpha < 2/\rho(A)$ where $\rho(\cdot)$ denotes the spectral radius of a matrix. The optimal value for the Richardson method is $\alpha_{opt} = 2/(\lambda_1 + \lambda_n)$, where λ_1 and λ_n are the largest and smallest eigenvalues of A, respectively (e.g. Saad, 2003).

The other basic stencils of stationary iterations (or simple iterations) can be derived from the following perspectives (e.g. Barrett *et al.*, 1994; Greenbaum, 1997; Eijkhout, 2003):

- Based on the matrix spliting, $A = M - N$ with M nonsingular, the linear system (1.1) becomes

  $$Mx = Nx + b$$

  This leads to the iteration

  $$x_{k+1} = M^{-1}Nx_k + M^{-1}b$$

  However, when considering Krylov subspace acceleration, it should be more useful to rewrite the above equation as

  $$x_{k+1} = x_k + M^{-1}(b - Ax_k) = x_k + M^{-1}r_k \tag{2.2}$$

  Clearly, the method is said to be convergent if the sequence $\{x_k\}$ converges to the exact solution, x, for any given initial vector.

- Let $r_k = b - Ax_k$ be the residual vector. By substituting $b = r_k + Ax_k$ into equation $x = A^{-1}b$, we get

  $$x = A^{-1}(r_k + Ax_k) = x_k + A^{-1}r_k$$

which leads to the following iteration

$$x_{k+1} = x_k + M^{-1} r_k$$

when M is use to approximate A.

- Let e_k to be the error vector defined by $e_k = x - x_k = A^{-1}(b - Ax_k)$. By using the approximation $M^{-1}(b - Ax_k)$, we gets the same iteration

$$x_{k+1} = x_k + M^{-1}(b - Ax_k) = x_k + M^{-1} r_k$$

Several important expressions can be derived from the above stationary iteration:

$$r_k = b - Ax_k = Ax - Ax_k = A(x - x_k) = Ae_k \tag{2.3a}$$

$$r_{k+1} = b - A(x_k + M^{-1} r_k) = (I - AM^{-1}) r_k = (I - AM^{-1})^{k+1} r_0 \tag{2.3b}$$

$$e_{k+1} = x - x_{k+1} = (x - x_k) - M^{-1} r_k = (I - M^{-1}A) e_k = (I - M^{-1}A)^{k+1} e_0 \tag{2.3c}$$

Here, we can write $(I - M^{-1}A)^{k+1} = P_{k+1}(M^{-1}A)$, where $P_{k+1}(\cdot)$ is a $(k+1)$-degree polynomial satisfying $P_{k+1}(0) = 1$. Clearly, the polynomial is $P_{k+1}(x) = (1 - x)^{k+1}$ for stationary iterative methods and the matrix $I - M^{-1}A$ or $I - AM^{-1}$ is called the iteration matrix. In most cases, Eq. (2.3b), instead of Eq. (2.3c), is evaluated iteratively because e_k can not be computed in practice, but it can be reflected through the Eq. (2.3a).

From Eq. (2.3b), we can conclude that the stationary iteration can converge provided

$$\|r_{k+1}\| \leq \|(I - AM^{-1})\| \|r_k\| \tag{2.4}$$

with

$$\|I - AM^{-1}\| < 1 \quad \text{or} \quad \rho(I - AM^{-1}) < 1 \tag{2.5}$$

Eqs (2.4) and (2.5) demonstrate that smaller spectral radius of the iteration matrix leads to the faster convergence rate. In other words, the closer to unit the eigenvalues of the preconditioned matrix AM^{-1} are, the faster an iterative method converges. Therefore, the eigenvalue distribution of preconditioned matrix becomes one of the guidelines to evaluate convergence rate. However, the above convergence conditions given in Eqs (2.4) and (2.5) may not be necessary for more advanced iterative methods.

By taking a weighted average (i.e., applying a relaxation) of the most recent two approximate solutions from Gauss-Seidel iteration, Young (1950) noticed that the convergence of Gauss-Seidel iterative method can be accelerated. As a result, he proposed the famous SOR iterative method. Usually, the relaxed version of the iteration in Eq. (2.2) can be derived from the linear system

$$\omega A x = \omega b \tag{2.6}$$

where ω is the relaxation parameter. Naturally, the stationary iteration with relaxation is given as

$$x_{k+1} = x_k + \omega M^{-1} r_k \tag{2.7}$$

for which the iteration matrix is $I - \omega M^{-1} A$ or $I - \omega A M^{-1}$, and different choice of M leads to different method such as weighted Jacobi and weighted Gauss-Seidel (SOR) methods.

Several popular stationary iterative methods are described in more details. Consider the decomposition $A = D+L+U$, where the matrices D, L and U represents the diagonal, the strictly lower triangular and the strictly upper triangular part of A, respectively. For different choices of M that derived from the above three factors, different stationary iterative methods can be obtained,

(a) *Jacobi method*:

$$M_{Jac} = D = diag(A) \tag{2.8}$$

(b) *Gauss-Seidel method*:

$$\begin{aligned} M_{GS} &= D + L, \quad \text{for forward Gauss-Seidel method} && (2.9a) \\ M_{GS} &= D + U, \quad \text{for backward Gauss-Seidel method} && (2.9b) \end{aligned}$$

(c) *SOR method*:

$$\begin{aligned} M_{SOR} &= D + \omega L, \quad \text{for forward SOR method} && (2.10a) \\ M_{SOR} &= D + \omega U, \quad \text{for backward SOR method} && (2.10b) \end{aligned}$$

For SPD matrix A, relaxation parameter, $\omega \in (0, 2)$, is chosen to guarantee convergence, but the determination of an optimal parameter ω_{opt} could be expensive.

(d) *SSOR method*:

$$M_{SSOR} = \frac{1}{2-\omega}\left(L + \frac{D}{\omega}\right)\left(\frac{D}{\omega}\right)^{-1}\left(U + \frac{D}{\omega}\right) \tag{2.11}$$

which can be regarded as the symmetric version of SOR method, that is, each iteration of the SSOR method is composed of one forward SOR sweep and one backward SOR sweep, but SSOR iteration is less sensitive to ω than SOR iteration (e.g. Langtangen, 1999).

Though stationary iterative methods are easier to understand and simpler to implement, they may be slow or even diverge (e.g. Barrett *et al.*, 1994; Greenbaum, 1997).

2.1.2 Non-stationary Iterative Methods

The non-stationary iterative methods differ from stationary iterative methods in that there is a varied optimal parameter for convergence acceleration in each iteration. That is, iterative stencil for a preconditioned non-stationary iterative methods follows

$$x_{k+1} = x_k + \alpha_k M^{-1} r_k \tag{2.12}$$

Here α_k is a scalar defined by some optimal rule. Thus the new iteration is determined by three ingredients: the previous iteration, the search direction vector (in this case, it is preconditioned residual) and the optimal scalar. In this class of iterative methods, Krylov subspace method is the most effective, and thus, it has been ranked as one of the top ten algorithms of 20-th century by Sullivan (2000). Krylov subspace iterative methods are so called because a solution can be approximated by a linear combination of the basis vectors of a Krylov subspace, that is,

$$x_k \in x_0 + \mathcal{K}_k(A, r_0), \quad k = 1, 2, \dots \tag{2.13}$$

where

$$\mathcal{K}_k(A, r_0) = span\{r_0, Ar_0, \dots, A^{k-1}r_0\}, \quad k = 1, 2, \dots \tag{2.14}$$

is called a k-th Krylov subspace of $\mathbb{R}^n$ generated by A with respect to r_0 (for the convenience of expressions in the following sections, both A and r_0 in Eqs. (2.13) and (2.14) represent the preconditioned versions). Thus the dimension of the subspace increases by one for each new iteration. It is also clear that the subspace depends on the initial vector, and it may be more natural to use r_0 though other choices are possible. To extend the subspace, only the matrix-vector multiplications are required for Krylov subspace methods, and to obtain the approximate solution, only vector related operations are involved.

There are two essential ingredients for Krylov subspace methods (e.g. Coughran and Freund, 1997): one is to construct a sequence of basis vectors with orthonormal property and the other is to generate approximate solutions $\{x_j\}_{j\leq k}$.

The obvious choice for constructing orthonormal basis vectors are based on Krylov subspace, $\mathcal{K}_k(A, r_0)$. However, this choice may not be attractive since $A^j r_0$ tend to point in the direction of dominant eigenvector corresponding to the largest eigenvalue for increasing subspace dimension, and thus, leads to ill-conditioned set of basis vectors (e.g. Demmel, 1997; van der Vorst, 2003). In addition, constructing an ill-conditioned vectors $\{A^j r_0\}_{j\leq k}$ followed by their orthogonalization still does not help from numerical viewpoint, which means that a new sequence of basis vectors $span\{v_1, v_2, \ldots, v_k\} = \mathcal{K}_k(A, r_0)$ with good properties should be constructed.

The Arnoldi and Lanczos algorithms can be used for constructing orthonormal basis vectors. Let subspace $V_k \in \mathbb{R}^{n\times k}$ be constructed with columns, $v_j = A^{j-1}r_0, (j = 1, 2, \ldots, k)$, then

$$AV_k = [v_2, v_3, \ldots, v_k, v_{k+1}] = V_k[e_2, e_3, \ldots, e_k, 0] + v_{k+1}e_k^T = V_k E_k + v_{k+1}e_k^T \tag{2.15}$$

Here, $E_k \in \mathbb{R}^{k\times k}$ with zero entries except for $E_k(j+1, j) = 1$ $(j < k-1)$ can be viewed as a special upper *Hessenberg matrix*. e_i is a n-dimensional zero vector except for the unit entry at ith row. Then, QR decomposition[1] of V_k, that is, $V_k = Q_k R_k$ with $Q_k \in \mathbb{R}^{n\times k}$ and upper triangular matrix $R_k \in \mathbb{R}^{k\times k}$, results in

$$AQ_k R_k = Q_k R_k E_k + v_{k+1}e_k^T \tag{2.16}$$

that is,

$$\begin{aligned} AQ_k &= Q_k R_k E_k R_k^{-1} + v_{k+1}e_k^T R_k^{-1} = Q_k \hat{H}_k + \frac{v_{k+1}e_k^T}{R(k,k)} \\ &= Q_k \hat{H}_k + h_{k+1,k} q_{k+1} e_k^T (= Q_{k+1} H_{k+1,k}) \end{aligned} \tag{2.17}$$

[1] Basically, there are two ways to compute QR decomposition of a matrix $A_{m\times n}, m \geq n$: one is to apply modified Gram-Schmidt algorithm to orthogonalize the columns of A; The other way is to apply a sequence of unitary matrices to transform it into the upper triangular $R_{n\times n}$, and the product of the sequence of unitary matrices is the inverse of $Q_{m\times n}$. For the second way, the often used unitary matrices are Householder reflection and Givens rotation (e.g. Greenbaum, 1997).

where $H_{k+1,k}$ is a $k+1$ by k matrix with $\hat{H}_k$ at the top k by k block and the zero $k+1$ row except for the $(k+1,k)$ entry which has the value, $h_{k+1,k}$. Thus, we get

$$Q_k^T A Q_k = \hat{H}_k + Q_k^T h_{k+1,k} q_{k+1} e_k^T = H_k \tag{2.18}$$

where, $\hat{H}_k, H_k \in \mathbb{R}^{k\times k}$ are both upper Hessenberg matrices because both R_k and R_k^{-1} are upper triangular can not change the upper Hessenberg property of E_k; $R(k,k)$ is the (k,k) entry of R_k. Equating the columns on two sides of equation $AQ_k = Q_k H_k$ with $Q_k = [q_1, q_2, \dots, q_k]$ results in

$$Aq_j = Q_k h_j = \sum_{i=1}^{j+1} h_{i,j} q_i = \sum_{i=1}^{j} h_{i,j} q_i + h_{j+1,j} q_{j+1}, \quad (j \le k-1) \tag{2.19}$$

where h_j represents the jth column of the Hessenberg matrix H_k and $h_{i,j}$ represents the entry at ith row and jth column. These entries can be computed by multiplying q_l, $l \le j$ to two sides of Eq. (2.19), that is,

$$q_l^T A q_j = q_l \Big(\sum_{i=1}^{j} h_{ij} q_i + h_{j+1,j} q_{j+1} \Big), \quad (l \le j) \tag{2.20}$$

and thus,

$$h_{lj} = q_l^T A q_j, \quad (l \le j) \tag{2.21}$$

then from Equation (2.19), we get

$$h_{j+1,j} q_{j+1} = Aq_j - \sum_{i=1}^{j} h_{i,j} q_i \tag{2.22}$$

or

$$q_{j+1} = \frac{1}{h_{j+1,j}} \Big(Aq_j - \sum_{i=1}^{j} h_{i,j} q_i \Big)$$

which indicates that the computation of the new basis vector q_{j+1} requires all previous basis vectors. Following the above procedures to compute the coefficient entries $h_{i,j}$ and the basis vectors $\{q_j\}_{j\le k}$, we get the *Arnoldi algorithm*. If A is symmetric, then the reduced Hessenberg matrix, H_k, become symmetric and tridiagonal, that is, $H_k = T_k$. The algorithm which reduces symmetric A to tridiagonal form is the famous *Lanczos algorithm*, in which,

$$Aq_j = \beta_{j-1} q_{j-1} + \alpha_j q_j + \beta_j q_{j+1}, \quad (j \le k-1) \tag{2.23}$$

where

$$T_k = \begin{bmatrix} \alpha_1 & \beta_1 & & \\ \beta_1 & \ddots & \ddots & \\ & \ddots & \ddots & \beta_{k-1} \\ & & \beta_{k-1} & \alpha_k \end{bmatrix} \tag{2.24}$$

Usually, the *modified Gram-Schmidt* method is adopted to construct an orthogonal space, but there are some other alternate choices. For Lanczos algorithm, the new basis vector is only required to be orthogonal to previous two vectors, and thus, Lanczos algorithm has a cheap three-term recurrence. In addition, two-sided (or nonsymmetric) Lanczos algorithm can be used to solve nonsymmetric linear system. In two-sided Lanczos algorithm, T_k is a nonsymmetric tridiagonal matrix with the form

$$T_k = \begin{bmatrix} \alpha_1 & \beta_1 & & \\ \gamma_1 & \ddots & \ddots & \\ & \ddots & \ddots & \beta_{k-1} \\ & & \gamma_{k-1} & \alpha_k \end{bmatrix} \tag{2.25}$$

Based on either Lanczos or Arnoldi algorithm for orthonormal basis vectors, iterative methods differ from each other in that they generate approximate solutions $\{x_j\}_{j\leq k}$ in terms of different optimal rule. In general, iterative methods are developed based on four different approaches to generate $\{x_j\}_{j\leq k}$: the *Ritz-Galerkin* approach, the *minimal residual norm* approach, the *Petrov-Galerkin* approach and the *minimal error norm* approach (e.g. van der Vorst, 2003).

The history of Krylov subspace methods can be traced back to the middle of 20-th century. One of the most famous iterative methods is Conjugate Gradient (CG) method developed by Hestenes and Stiefel (1952) for solving symmetric positive definite linear systems. CG method belongs to the *Ritz-Galerkin* approach which requires $r_k \perp \mathcal{K}_k(A, r_0)$. That is, the approximate solution x_k is computed satisfying

$$\tilde{R}_k^T(b - Ax_k) = 0 \tag{2.26}$$

where $\tilde{R}_k = [r_0, r_1, \ldots, r_{k-1}]$. However, the CG method did not show it is a promising competitor to direct methods until it was combined with preconditioning method which led to the later popular and robust PCG solver. The Minimal Residual (MINRES) and Symmetric LQ (SYMMLQ) methods were discovered by Paige and Saunders (1975), the kth step of MINRES compute x_k from

$$x_k = x_0 + Q_k y_k \tag{2.27}$$

where y_k can be obtained by minimizing

$$r_k = r_0 - AQ_k y_k \tag{2.28}$$

in the least squares sense with Eq. (2.17). That is,

$$\begin{aligned} \|r_k\|_2 &= \min_y \|r_0 - AQ_k y\|_2 = \min_y \|r_0 - Q_{k+1}T_{k+1,k}y\|_2 \\ &= \min_y \|Q_{k+1}(\varrho e_1 - T_{k+1,k}y)\|_2 = \min_y \|\varrho e_1 - T_{k+1,k}y\|_2 \end{aligned} \tag{2.29}$$

where $\varrho = \|r_0\|$. From various literatures, it can be seen that MINRES has a wide applications for symmetric and possibly indefinite linear systems. Some authors (e.g. Warsa *et al.*, 2002) may prefer to use the MINRES method because it is thought more reliable based on extensive testing. The bi-conjugate gradient (Bi-CG) method developed by Fletcher (1996) was a “natural” generalization of CG method for nonsymmetric linear systems.

Significant development in Krylov subspace iterative methods occurred during the period of 1985∼1995, the Generalized Minimum Residual (GMRES) algorithm, which was developed by Saad and Schultz (1986) for nonsymmetric systems, is one of the most widely used iterative method for a general linear system. In GMRES method, the approximate solution x_k is also taken from Eq. (2.27) with y_k minimizing the relation given in Eq. (2.28) in the least squares sense. That is,

$$\begin{aligned} \|r_k\|_2 &= \min_y \|r_0 - AQ_k y\|_2 = \min_y \|r_0 - Q_{k+1}H_{k+1,k}y\|_2 \\ &= \min_y \|Q_{k+1}(\varrho e_1 - H_{k+1,k}y)\|_2 = \min_y \|\varrho e_1 - H_{k+1,k}y\|_2 \end{aligned} \tag{2.30}$$

where, $H_{k+1,k}$ is the rectangular upper Hessenberg matrix given in Eq. (2.17). However, as GMRES iterates, the work and storage requirement in each iteration grow linearly because the method is based on *Arnoldi* algorithm (e.g. Saad, 1996; Toh, 1997; Saad, 2003; van der Vorst, 2003). Therefore, the full version of GMRES may be prohibitively expensive and is not widely used in practical applications. Requiring long recurrences for nonsymmetric CG-like iterative solvers such as GMRES may incur large memory requirement and computational cost. To overcome the problems related to long recurrences caused by nonsymmetric property, two different artificial strategies can be used: one is to truncate the last m basis vectors for the next update, which is referred to as *truncation* (e.g. de Sturler, 1999); the other is to restart the iteration after m iterations and input the current approximate solution as the initial guess of next iteration cycle, for instance, GMRES(m) method restarts in a m-iteration cycle (e.g. Meerbergen and Roose, 1997; Simoncini, 2000). However, both the truncation and the restarting strategies inevitably result in partial subspace methods, thus they may lead to different convergence behavior (such as a slower convergence rate) from the idealized counterparts (e.g. Bruaset, 1997). Different from the truncation and restarting strategies, Bi-CG takes another approach to obtain short recurrences by replacing the orthogonal sequence of residuals by two mutually orthogonal sequences: apart from the space given in Eq. (2.14), an auxiliary space $\mathcal{K}_k(A^T, s_0)$ is constructed, where s_0 is an arbitrary vector, but usually $s_0 = r_0$ is chosen. In Bi-CG, both r_k and s_k sequences are generated on coupled two-term recurrences at the price of no longer providing a minimization[2] so that the convergence of Bi-CG may exhibit an irregular or so-called zigzagging behavior (e.g. Pommerell and Fichtner, 1991). Since the auxiliary space involves with A^T, Bi-CG incurs an additional multiplication with A^T without further reduction of the residual. Sonneveld (1989) noticed that both r_k and s_k sequences converge to zero, but only the convergence of first sequence is exploited, and thus he developed Conjugate Gradients-Squared (CGS) method by a minor modification to Bi-CG (e.g. Sonneveld, 1989; van der Vorst, 1992; Golub and van der Vorst, 1997; Eijkhout, 2003). Notice that Bi-CG's residual vector r_k and the so-called "shadow" residual vector s_k can be written as

$$r_k = P_k(A)r_0, \quad \text{and} \quad s_k = P_k(A^T)s_0 \tag{2.31}$$

Therefore, the bi-orthogonal relation, i.e., $r_k \perp \{s_j\}_{j<k}$ and $s_k \perp \{r_j\}_{j<k}$, in Bi-CG leads to

$$(r_k, s_j) = (P_k(A)r_0, P_j(A^T)s_0) = (P_j(A)P_k(A)r_0, s_0) = 0, \quad (j < k) \tag{2.32}$$

Then, we can define that

$$r_j = P_j^2(A)r_0 \tag{2.33}$$

This inner-product transformation makes the distributed convergence efforts concentrate to the residual r_j, and thus results in a stronger contraction on the initial residual r_0 than that in Bi-CG as shown in Eq. (2.31) and avoids the auxiliary space. Usually, the CGS method converges almost twice as fast as Bi-CG, and it is recognized as the start of

[2] This indicates that convergence associated with residual norm reduction may not occur in every iteration.

hybrid iterative methods. However, at the same time of accelerating convergence, CGS also inherits the irregularity of Bi-CG and even amplifies irregular behavior due to the squaring effect in Eq. (2.33). The seriousness of irregular convergence behavior has been discussed in detail by van der Vorst (1992, 2003) and Golub and van der Vorst (1997).

From the 1990s onward, some robust and so far widely used Krylov subspace iterative methods arose in succession. Freund and Nachtigal (1991) proposed three-term recurrences Quasi-Minimal Residual (QMR) algorithm, and then more robust coupled two-term recurrences version, to overcome the breakdown and numerical instability problems associated with the original Bi-CG algorithm using look-ahead strategy. Using Lanczos algorithm constructed basis, the actual QMR iterates are defined by a relaxed minimization (quasi-minimal) property which ensure QMR converge smoothly. This relaxation comes from the following relations,

$$r_k = r_0 - AV_k y_k = V_{k+1}(\varrho e_1 - T_{k+1,k} y_k) \tag{2.34}$$

Thus, we have

$$\|r_k\|_2 \leq \|V_{k+1}\|_2 \cdot \|\varrho e_1 - T_{k+1,k} y_k\|_2 \tag{2.35}$$

Instead of minimizing $\|r_k\|_2$ (because the columns of V_{k+1} are not orthogonal and it is difficult to choose y_k), QMR algorithm compute y_k from the following least squares problem,

$$\min_y \|\varrho e_1 - T_{k+1,k} y\|_2 \tag{2.36}$$

Thus, QMR is so called because it does not truly minimize the residual. In some cases, the operations with A^T is impossible, the transpose-free version of QMR method (TFQMR), which was developed by Freund (1993), can be used. Furthermore, a simplified or symmetric version of QMR (SQMR) was also proposed by Freund and Nachtigal (1994) for symmetric and possibly indefinite systems . As we know, two-sided Lanczos process requires matrix-vector products Av for right Lanczos vectors and products $A^T w$ for left Lanczos vectors. QMR is simplified as long as one can find a nonsingular matrix J satisfying

$$A^T J = JA \tag{2.37}$$

(in this case, A is *J-symmetric*) and if the initial vector w_1 is chosen as $w_1 = \zeta_1 J v_1$ with scalar $\zeta_1 \neq 0$. As a result, the left Lanzcos vectors can be obtained as follows

$$w_j = \zeta_j J v_j, \quad \zeta_j \in \mathbb{R} \text{ and } \zeta_j \neq 0 \tag{2.38}$$

and matrix-vector products with A^T can be replaced by matrix-vector products with J (e.g. Freund and Nachtigal, 1994). The scalars $\{\zeta_j\}$ depend on the particular normalization used in Lanczos process. When matrix-vector products with J are cheap, the storage and computing cost are both reduced.

Bi-CGSTAB, a more smoothly converging variant of Bi-CG or CGS proposed by van der Vorst (1992), generates the sequences r_k and s_k in terms of different polynomials as long as the requirement given in Eq. (2.32) is satisfied. That is,

$$(r_k, s_j) = (P_k(A)r_0, Q_j(A^T)s_0) = (Q_j(A)P_k(A)r_0, s_0) = 0 \tag{2.39}$$

where, the new polynomial is chosen as

$$Q_j(x) = \prod_{i=1}^{j}(1 - \omega_i x) \tag{2.40}$$

with suitable ω_i to minimize the current residual $r_j = Q_j(A)P_j(A)r_0$ in 2-norm. In the Bi-CGSTAB method, residuals r_j are generated in an apparently rather stable and efficient way. So far, Bi-CGSTAB has been one of most widely used solvers for nonsymmetric linear systems due to its several obvious advantages such as the fast and smooth convergence and short recurrences.

It may be difficult to make a general statement on how fast these Krylov methods converge, however, one certain thing is that Krylov methods converge much faster than the classical iterative methods and often the convergence of a Krylov iterative method takes place in a wide range of matrix classes. Because different Krylov iterative methods are applicable for different linear systems, Figure 2.1 gives the flowchart with suggestion for the selection of iterative methods.

For classical stationary iterative methods and non-stationary Krylov subspace methods, there is a common phenomenon: the more ill-conditioned a linear system is, the more iterations an iterative method would take. However, multigrid and domain decomposition methods may provide the exceptions. Multigrid methods may be very suitable for large problems since they require less memory and are much faster than a direct solver (e.g. Ekevid *et al.*, 2004). Thus, it is interesting to introduce the idea behind the multigrid iterative method here: the residual based on the fine mesh is a vector composed of several components of frequency. when a classical iterative method is applied, the high-frequency component of the residual vector is reduced rapidly, but the low-frequency component of the residual vector is difficult to reduce. However, it can be proved that when the studied problem is transferred to a coarse mesh, the low-frequency become higher component which can be reduced rapidly. Thus, multigrid iterative methods with different residual reducing cycles can be developed (e.g. Barrett *et al.*, 1994; Greenbaum, 1997; Briggs *et al.*, 2000).

2.2 Overview of Preconditioning Techniques

Many iterative methods can converge to the true solutions in *exact arithmetic* with bounded iteration counts, for examples, the upper limit iteration counts should be n (n is the dimension of the linear system) for CG, MINRES and GMRES methods with the minimization property. However, the number is still very large for large-scale linear systems. For example, the 3-D aquifer's consolidation problem with $n = 127100$ was studied by Gambolati *et al.* (2002). The possibility to overcome this difficulty is preconditioning (e.g. Keller *et al.*, 2000). As we mentioned previously, preconditioning contributes to the popularity of iterative methods and it has become an indispensable part in the notion of preconditioned iterative methods. The essential role of preconditioning can be accurately expressed by the following quotation (e.g. Benzi, 2002):

> "*...In ending this book with the subject of preconditioners, we find ourselves at the philosophical center of the scientific computing of the future .... Nothing will be more central to computational science in the next century than the art of transforming a problem that appears intractable into another whose solution can be approximated rapidly. For Krylov subspace matrix iterations, this is preconditioning.*"
>
> Trefethen and Bau, 1997

For an ill-conditioned linear system, it may be difficult for an iterative method to converge. Preconditioning has the role to improve the situation, more accurately, to improve the *spectrum*[3] of the original coefficient matrix of the linear system so that the

[3] The set of all the eigenvalues of a matrix is called its spectrum.

preconditioned linear system can converge at a faster rate. To apply a preconditioner, three formats can be chosen. For example, given a preconditioner

$$M = M_L M_R \in \mathbb{R}^{n\times n} \tag{2.41}$$

where M_L, M_R represent left and right preconditioner, respectively. Then, the preconditioned system of Eq. (1.1) can be written as

$$M_L^{-1} A M_R^{-1} M_R x = M_L^{-1} b \quad \text{or} \quad \widetilde{A}\tilde{x} = \tilde{b} \tag{2.42}$$

Therefore,

When, $M_L \neq I$, $M_R \neq I$, it is left-right preconditioning approach;

When $M_R = I$, it is left preconditioning approach;

When $M_L = I$, it is right preconditioning approach.

In practice, the choice of preconditioning side often depends on the selected iterative method to choose and on properties of the coefficient matrix A. The obvious advantage for right preconditioning approach is that in exact arithmetic, the residuals for right-preconditioned system are identical to the true residuals, and thus, the convergence behavior can be monitored accurately. For the left-right preconditioning approach, it is required that the preconditioner can be expressed explicitly as in Eq. (2.41). It is worth mentioning that the manner in which a preconditioner is applied has obvious influence on nonsymmetric problems. Another noteworthy point is that for the simplified QMR method shown in Eq. (2.38), one can choose $J = M_L^T M_R^{-1}$ because it can be verified that

$$\widetilde{A}^T J = M_R^{-T} A M_R^{-1} = M_L^T M_R^{-1} M_L^{-1} A M_R^{-1} = J\widetilde{A} \quad (M = M_L M_R = M_R^T M_L^T = M^T) \tag{2.43}$$

Thus, $\widetilde{A}$ is J-symmetric with the choice $J = M_L^T M_R^{-1}$ (e.g. Freund and Nachtigal, 1995).

Generally, an efficient preconditioner is largely problem-dependent although there are some robust preconditioners developed for general purpose. Regardless, it is helpful to keep in mind the following three basic criteria:

(a) The preconditioned system should converge rapidly, i.e., the preconditioned matrix should have good eigenvalue clustering which indicates $M \approx A$.

(b) The preconditioner should be cheap to construct and easy to "invert" within each iteration.

(c) Last but not least, the preconditioner should not consume a large amount of memory. It is also preferable to avoid massive indirect memory addressing operations to exploit the cache architecture in CPUs.

These criteria seem to be self-conflicting, or in other words, there is a conflict between good approximation and efficiency. Therefore, it seems that designing a good preconditioner requires a delicate balance. A variety of preconditioners are available and a brief sketch of some popular preconditioning techniques is presented below.

2.2.1 Diagonal Preconditioning

The simplest and probably the most widely used preconditioning technique is to scale the coefficient matrix with a diagonal matrix D. Diagonal preconditioning is among the cheapest and most memory effective ones (e.g. Pini and Gambolati, 1990).

2.2.1.1 Standard Jacobi (SJ)

For problems with a diagonally dominant stiffness matrix A, the choice $D = diag(A)$ results in the preconditioner $M = diag(A)$ which is known as *Jacobi preconditioning* or *diagonal scaling* and is efficient enough for many applications. However, if $diag(A)$ contains comparatively smaller entries with several order's difference in magnitude than off-diagonal entries, this preconditioner is often not suitable and possibly counter-productive.

2.2.1.2 Modified Jacobi (MJ)

For the coefficient matrix of an ill-conditioned linear system discretized from Biot's consolidation equations, there may exist unbalanced columns or rows corresponding to the pore pressure unknowns. That is, the pore pressure part of $diag(A)$ contains significantly smaller entries in absolute magnitude than the off-diagonal ones such that the SJ scaling may "under-scale" the coefficient matrix conditioning. To avoid this problem, Chan *et al.* (2001, 2002) proposed a modified Jocobi preconditioning method, which scales the rows (left preconditioning) or columns (right preconditioning) corresponding to the pore pressure unknowns by using the largest absolute value in the corresponding columns.

2.2.1.3 Generalized Jocobi (GJ)

Compared with the heuristic MJ preconditioner, the motivation for the construction of GJ preconditioner comes from a theoretical eigenvalue clustering result given by Murphy *et al.* (2000) for a linear system with the similar block form as that of Biot's problems, but with zero $(2,2)$ block. Similar to MJ preconditioner, an element-by-element (EBE) form of GJ was proposed, and numerical examples show that GJ is an efficient choice for solving Biot's linear equations due to its cheap diagonal form and robustness to accelerate the convergence of SQMR (e.g. Phoon *et al.*, 2002, 2003; Toh, 2003; Toh *et al.*, 2004). However, GJ was only proposed to solve the symmetric case. In fact, it can also be used as a scaling technique for nonsymmetric Biot's linear systems (e.g. Gambolati *et al.*, 2003).

2.2.2 SSOR Preconditioning

Symmetric Successive Over-Relaxation (SSOR) iteration belongs to the classical iterative methods as we mentioned in Section 2.1.1, and it is still used widely in some fields. SSOR preconditioning can be regarded as a single iteration of SSOR method. A natural idea is to apply a classical iterative method with only one or several iteration to the preconditioning step. For example, there are some other preconditioning methods obtained from one iteration of classical iterative methods such as Gauss-Seidel and SOR preconditioners. Eq. (2.11) in fact gives the SSOR iteration in a combination form, and more often, it is used with the split form

$$M_L = \frac{1}{2-\omega}\left(L + \frac{D}{\omega}\right), \quad M_R = \left(\frac{D}{\omega}\right)^{-1}\left(U + \frac{D}{\omega}\right) \tag{2.44}$$

The scale factor $1/(2-\omega)$ may be neglected when SSOR preconditioner is used with Krylov subspace iterative methods, but it may be important if the iterative method is not scale invariant (e.g. Chow and Heroux, 1998). In addition, the performance of SSOR preconditioner is not as sensitive to ω as the SSOR iterative method (e.g. Bruaset, 1997).

2.2.3 Block Preconditioning

Block preconditioning methods are based on the fact that variables can be grouped in terms of physical properties, and thus, the stiffness matrix can be partitioned into block form, and the corresponding preconditioners based on the block form are called block preconditioners. For Biot's consolidation problems, if we partition the displacement variables and the pore pressure variables separately, the stiffness matrix becomes a 2 × 2 block matrix. For the Biot's linear equation with 2 × 2 block coefficient matrix, Toh *et al.* (2004) studied systematically three classes of block preconditioners from which many variants can be derived. However, numerical results show that low iteration count does not imply that a preconditioner is good in practical sense. In practice, choosing a good preconditioner is not a simple task, but an art of science. For a detailed description on block preconditioners for saddle point problems, refer to Benzi *et al.* (2005).

2.2.3.1 Block Diagonal Preconditioner

Following the notations given by Toh *et al.* (2004), the proposed block diagonal preconditioner takes the form

$$P_d = \begin{bmatrix} \widehat{K} & 0 \\ 0 & \alpha\widehat{S} \end{bmatrix} \tag{2.45}$$

where α is a given nonzero real scalar and possibly negative, but whether α is negative or positive may be dependent on which iterative method is chosen. In Eq. (2.45), the matrices with hat symbol are approximations to their counterparts, respectively. For the ideal case, $\widehat{K} = K$ and $\widehat{S} = S$, the eigenvalues of the preconditioned matrix are clustered around at most 3 points, namely 1 and $(1 \pm \sqrt{1 + 4/\alpha})/2$. A negative α is shown to be important when $\widehat{K}$ is an inexact form, and $\alpha = -4$ may lead to at most two clusters of eigenvalues around at 1 and 1/2 (e.g. Phoon *et al.*, 2002). Obviously, when α is negative, then SQMR becomes a cheap choice, otherwise, MINRES or SYMMLQ is applicable. It should be noted that some researchers are becoming interested in using PCG in conjunction with indefinite preconditioners (e.g. Lukšan and Vlček, 1998; Smith and Griffiths, 1998; Smith, 2000; Rozložník and Simoncini, 2002).

2.2.3.2 Block Constrained Preconditioner

This class of preconditioners are so called since they have the same block structure as the original block coefficient matrix, but the difference is that one or more sub-blocks are approximated or 'constrained'. In terms of Toh *et al.* (2004), the (1, 2) and (2, 1) sub-blocks are preserved, and only the (1, 1) and (2, 2) blocks are approximated. However, it can be seen that a more general form is possible as long as the preconditioner is still nonsingular. Thus, we rewrite the block constrained preconditioner as

$$P_c = \begin{bmatrix} \widehat{K} & \widehat{B} \\ \widehat{B}^T & -\widehat{C} \end{bmatrix} \tag{2.46}$$

where, $\widehat{B}$ is also the approximation to B, but with a different approximation strategy compared with $\widehat{K}$ and $\widehat{C}$. Here, we propose that $\widehat{B}$ can be obtained by applying a small threshold tolerance[4] to the entries of B, and some numerical results show that computing time really can be reduced by this strategy at the price of a slightly increased iteration count. In addition, it is worth noting that B is only geometry-dependent and does not change with soil properties. When the preconditioner is applied, its inverse can be considered as suggested by Toh *et al.* (2004) because the multiplication of $P_c^{-1}v$ is required in the iterative steps.

2.2.3.3 Block Triangular Preconditioner

Compared with the previous two classes of block preconditioners, block triangular preconditioners do not possess the symmetric property, which is usually useful in iterative methods.

For left preconditioning method,

$$P_t = \begin{bmatrix} \widehat{K} & 0 \\ B^T & -\widehat{S} \end{bmatrix} \tag{2.47a}$$

or for right preconditioning method,

$$P_t = \begin{bmatrix} \widehat{K} & B \\ 0 & -\widehat{S} \end{bmatrix} \tag{2.47b}$$

[4] How to choose the threshold tolerance can be important and worthy further study.

Naturally, block matrix B can be replaced by $\widehat{B}$ applied with a threshold strategy. Due to the nonsymmetric property of P_t, the nonsymmetric iterative methods such as Bi-CGSTAB, TFQMR or GMRES are required.

2.2.4 Incomplete Factorization Preconditioning

A very popular and broad class of preconditioners is developed based on the sparse incomplete factorization of the coefficient matrix. More or less, the incomplete factorization preconditioners simulate the direct solution methods. The previously introduced SSOR can be categorized into this class. It is already well known that when a sparse matrix is factored, it usually introduce some *fill-ins*, that is, triangular factors L and U factors have nonzero entries in the positions which are zero in the original matrix. When the fill-ins are reduced or ignored based on some criteria, we obtain the incomplete factorization methods such as ILU or IC.

2.2.4.1 ILU-Type or IC-Type Preconditioners

The incomplete LU (ILU) factorization is obtained by factorizing the coefficient matrix into sparse triangular factors, while incomplete Cholesky factorization (IC) is designed for symmetric positive definite system. To derive the triangular factors for ILU or IC, some criteria should be applied such as ignoring all fill-ins or part of fill-ins to derive level-of-fill based or threshold based incomplete factorization preconditioners. There are many powerful ILU-type preconditioners available such as ILUT and ILUM (e.g. Saad, 1994, 1996, 2003). For SPD linear systems, the efficient ICCG (Incomplete Cholesky preconditioned CG) method was proposed by Meijerink and van der Vorst (1977, 1981). The no-fill ILU(0), IC(0) and their modified variants are inexpensive, simple to implement and usually effective enough, and thus, they have a wide application in science and engineering. Obviously, the incomplete factors (that is, a sparse lower triangular factor L and a sparse upper triangular factor U) stemming from incomplete factorization are not a real factorization of the coefficient matrix, they are derived as easily inverted sparse triangular factors. Generally, the incomplete factorization computes the factors as follows (e.g. Barrett *et al.*, 1994; Benzi, 2002),

$$For\,each\,k,\; and\; i,j>k: \quad a_{ij} \leftarrow \begin{cases} a_{ij}-a_{ik}a_{kk}^{-1}a_{kj}, & if\;(i,j)\in\mathcal{S} \\ a_{ij}, & otherwise \end{cases} \tag{2.48}$$

where $\mathcal{S}$ is the predefined restriction subset. When computing ILU(0) or IC(0) factor, this subset becomes the subset for nonzero sparsity structure of A. As a result, when implementing each preconditioning step with an incomplete factorization preconditioner, two triangular solves are involved.

Another incomplete factorization preconditioner called diagonal ILU (D-ILU) is worth mentioning due to its success in some packages and its potential to extend to many other applications. This so-called D-ILU preconditioner has been proposed by Pommerell and Fichtner (1991) with the stencil,

$$M_{D\text{-}ILU} = (\widehat{D}+L)\widehat{D}^{-1}(\widehat{D}+U) \tag{2.49}$$

where L and U are still the strictly lower and upper triangular of coefficient matrix A, respectively, but $\widehat{D}$ is computed using ILU factorization procedure.

There are limitations associated with incomplete factorizations. Incomplete factorization methods may fail in strongly nonsymmetric or indefinite cases though they are recognized to be very effective in many applications. For example, failure may occur due to zero pivots, very small pivots, or negative pivots in some case. In addition, failure can also happen due to numerical instability (e.g. Chow and Saad, 1997; Benzi and Tůma, 1999; Eijkhout, 1999).

2.2.4.2 Reordering or Matrix Permutation Algorithms

As we mentioned above, when a sparse matrix is factored, it usually introduces some fill-ins. To reduce the fill-ins and accelerate the factorization, reordering or permutation of the variables should be performed before the factorization. Once the permutation matrix $\mathcal{P}$ is obtained, the system required to be factorized becomes (here, it is a symmetric permutation)

$$(\mathcal{P}A\mathcal{P}^T)(\mathcal{P}x) = \mathcal{P}b \tag{2.50}$$

Several popular reordering algorithms are given as follows:

(a) *Reverse Cuthill-McKee (RCM) or Banded Ordering*

The original Cuthill-McKee ordering method is primarily designed to reduce the profile of a matrix. The idea of Cuthill-McKee ordering is that numbering begins from one point, and numbers the neighbors of the point, and then continues to number the neighbors of the already numbered points. George (1971) observed that reversing the Cuthill-McKee ordering turns out to be superior to the original ordering. Figure 2.2(a) shows the sparsity pattern of the naturally ordered stiffness matrix of Biot's linear systems, while Figure 2.2(b) gives the sparsity pattern of the matrix after RCM ordering for naturally ordered matrix showed in Figure 2.2(a).

(b) *Minimum Degree Ordering*

Minimum Degree (MD) ordering algorithm by George and Liu (1981) is known as a very fast fill-in-reduction reordering algorithm and it has received much attention over the last two decades. MD is based on greedy approach such that the ordering is chosen to minimize some quantities at each step of a simulated-step symmetric Gaussian elimination process. In the variants of MD, Multiple Minimal Degree (MMD) algorithm by George and Liu (1985, 1989) and Approximate Minimum Degree (AMD) algorithm by Davis *et al.* (1994) are more efficient and popular. Figure 2.2(c) is the sparsity pattern of MMD reordered matrix, which is very similar to the sparsity pattern of AMD reordered matrix in Figure 2.2(d).

(c) *Nested Dissection*

Another alternate approach, Nested Dissection (ND) ordering by George (1971), is fundamentally a divide and conquer approach. Though ND has some appealing theoretical properties, it is less competitive to minimum degree ordering. Nested dissection algorithms are generally much slower than MMD.

(d) *Multi-color ordering*

In this ordering process, the uncoupled variables are organized into different groups in terms of colors, but this ordering is best suited for parallel environments because the variables in each color can be dealt with independently.

There are also some new sparse matrix ordering algorithms developed recently, but most of them are variants of the ND algorithm or MD algorithm, or the hybrid of the two algorithms. The ordering algorithms described above are automatic and adaptive.

However, it may be necessary to reorder a matrix artificially in terms of the physical property of variables. Following the definitions of variable ordering given by Gambolati *et al.* (2001), we get the following three orderings for 3-D Biot's consolidation problems,

- *Natural ordering (iord1)*: $[x_1, y_1, z_1, p_1, x_2, y_2, z_2, p_2, \ldots]$;
- *Block ordering (iord2)*: $[x_1, y_1, z_1, x_2, y_2, z_2, \ldots, p_1, p_2, \ldots]$;
- *Block ordering (iord3)*: $[x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots, p_1, p_2, \ldots]$.

Here, x, y and z correspond to the displacement unknowns in three direction, respectively; p corresponds to the pore pressure unknown.

2.2.5 Approximate Inverse Preconditioning

Approximate inverse preconditioning received little attention though the ideas of approximate inverse had been proposed in 1970s. The possible reason is due to the difficulty in automatically determining an effective nonzero sparsity patten $\mathcal{S}$. The recent developments of sparse approximate inverse may attribute to the advances of parallel processing and the good convergence rates of approximate inverses (e.g., Benzi *et al.*, 1996, 1998, 1999, 2000; Grote and Huckle, 1997; Chow and Saad, 1997, 1998; Gould and Scott, 1998). Basically, there are three classes of approximate inverse preconditioning methods: one is based on least squares norm-minimization technique, and second one is based on the factorized sparse approximate inverses, and the last one is based on incomplete factorization followed by an approximate inverse of the incomplete factors.

The proposed approximate inverse preconditioners by Grote and Huckle (1997), Chow and Saad (1997), which are all based on Frobenius norm minimization, are thought the most successful. Chow and Saad (1997) noticed that the popular incomplete factorization preconditioners may fail in some cases, especially when the preconditioned error matrix is very large, the resulted identity matrix may have large perturbations. For example, given the factorization

$$A = LU + E \tag{2.51}$$

where E is the error matrix, the resultant preconditioned error matrix $L^{-1}EU^{-1}$ can be too large in some applications. Based on this observation, they proposed to use Minimal

Residual (MR) method to obtain sparse approximate inverse with the prescribed sparsity pattern. Grote and Huckle (1997) adopted adaptive sparsity pattern to construct the approximate inverse, leading to current popular the SParse Approximate Inverse (SPAI) preconditioners.

To compute the sparse approximate inverse M of matrix A, the idea is to minimize $\|I - MA\|_F$ (or $\|I - AM\|_F$ for right preconditioning) subject to certain sparsity constraints. The Frobenius norm, $\|\cdot\|_F$, is usually chosen since it allows the decoupling of the constrained minimization problem into independent linear least-squares problems as

$$\min_{M \in \mathcal{S}} \|I - AM\|_F^2 = \sum_{j=1}^{n} \min_{m_j} \|e_j - Am_j\|_F^2 \tag{2.52}$$

Where, $M = [m_1, m_2, \ldots, m_n]$, $I = [e_1, e_2, \ldots, e_n]$ is the identity matrix.

For the other two classes of sparse approximate inverse preconditioners, refer to Benzi *et al.* (1996), Benzi and Tůma (1998), Benzi and Tůma (1999), Benzi *et al.* (2000), for details.

2.2.6 Other Preconditioning Methods

There are some other popular preconditioning methods which can be adopted in different applications. *Polynomial preconditioners* only require matrix-vector multiplications with A to approximate the preconditioning steps in iterative methods, they may have some potential for parallelization due to a series of matrix-vector products, but it has been shown that they can not compete with the above discussed incomplete factorization preconditioners in terms of iteration counts (e.g. Bruaset, 1997; Benzi and Tůma, 1999). For the other preconditioners such as domain decomposition and multigrid methods, see Saad (2003); Ekevid *et al.* (2004) for the details.

2.3 Data Storage Schemes

In finite element analysis, several different data structures can be used to store the coefficient matrices such as the element-by-element storage, the global compressed sparse storage and so-called edge-based storage. The comparison between these storages based on PCG iterative method was carried out recently, and the advantages and the disadvantages of each storage scheme was clarified by Ribeiro and Coutinho (2005). In this thesis, only the element-by-element storage and global compressed sparse storage are studied because they are still popular storage schemes in current finite element software packages.

The element-by-element implementation have essentially removed the requirement for assembling matrices. In addition, it seems that by making use of EBE-level stiffness matrices, the storage and even computational cost can be reduced further by grouping the identical element stiffness matrices into one (e.g. Smith and Wang, 1998). This potential can be exploited, but it may be limited to regular element property. In EBE-based FE applications, the applied iterative method and the preconditioner must be implemented at element level so that the matrix-vector products in each iteration have to be adapted to the unassembled matrix, which lead to the so-called “Gather” and “Scatter” procedures. In practice, EBE based preconditioned iterative methods are more suitable for parallel environment than serial computers in that the matrix-vector products with the element-level stiffness matrices are more readily parallelized, but they have to be implemented sequentially on serial computers (e.g. Smith, 2000; Smith and Griffiths, 2004).

On the contrary, it is well known that stiffness matrices derived from discretization of partial differential equations by finite element method are usually large and sparse due to the maximum connections of each element to others and there are few nonzero entries in each row. Global compressed sparse storage as its name implied requires the stiffness matrix to be assembled and only nonzero entries of stiffness matrix to be stored. The obvious advantages of sparse technique are that the final storage for the stiffness matrix may be significantly lower than that of EBE format and much less computer execution time is spent because only nonzero entries are involved in arithmetic operations. If the stiffness matrix is symmetric, sparse symmetric storage can be used. Figure 2.4 shows a simple flow chart of applying sparse preconditioned iterative method to FE applications.

Here, it is necessary to give a brief comparison between EBE and sparse technique in terms of storage and floating point operations (flops) based on some 3-D Biot’s consolidation models. Given a stiffness matrix $A \in \mathbb{R}^{n\times n}$, the EBE-based storage required for the matrix can be computed by

$$S_{ebe} = \frac{n_{el}^2 \times n_e \times 8}{1024^2}\ (MBytes) \tag{2.53}$$

where n_e is the number of total elements; n_{el} is the dimension of element matrix which depends on the adopted element type, for the 20-nodal solid quadrilateral element coupled with 8-nodal fluid element used in 3-D Biot's consolidation, $n_{el} = 20 \times 3 + 8 = 68$; The number '1024' means the conversion, $1\,MBytes = 1024\,KBytes = 1024^2\,Bytes$. "8" in the numerator of Eq. (2.53) means one double-precision number requires the storage of 8 bytes.

For a sparse matrix, three vectors are required for Compress Sparse Row storage (CSR) or Compress Sparse Column storage (CSC). However, to form such compressed sparse storage, three temporary vectors are needed to collect the element-level nonzero entries in upper (or lower) triangular part. The storage requirement for the three temporary vectors is almost as same as that for EBE based storage requirement. When summing up the sorted entries with the same global row and global column number, the already summed storage can be used to store the global compressed sparse matrix. Therefore, the truly required memory storage can be computed by

$$S_{sps} = \frac{\dfrac{n_{el}^2 + n_{el}}{2} \times n_e \times (4 + 4 + 8)}{1024^2}\ (MBytes) \tag{2.54}$$

The number '8' means one double-precision real number requires 8 bytes, and '4' means one integer requires 4 bytes. Clearly, the proposed symmetric sparse storage scheme (only three vectors) uses about $\dfrac{n_{el} \times n_e \times 8}{1024^2}(MBytes)$ more memory storage compared to EBE storage scheme. For a 3-D consolidation problem using totally 100000 consolidation elements (that is $n_e = 100000$) and given 20-node displacement elements coupled with 8-node fluid elements (that is, $n_{el} = 68$) as an example, the additional incurred storage by this symmetric sparse storage scheme is only 52 MBytes. A demonstration example on how to form such a symmetric sparse storage is provided in Appendix B.1.2.

The flops required by one matrix-vector multiplication Av for dense matrix, EBE matrix and sparse matrix are computed, respectively, as follows:

$$flops(Av)_{den} = 2n^2 \tag{2.55a}$$

$$flops(Av)_{ebe} \approx (2n_{el}^2 + n_{el}) \times n_e \tag{2.55b}$$

$$flops(Av)_{sps} \approx 2n^2 r_s \tag{2.55c}$$

It is also possible to exploit the sparsity of the vector v when computing $flops(Av)_{sps}$, but to be conservative, this sparsity is not considered. Table 2.1 presents the comparison between EBE technique and sparse technique in terms of storage requirement and flops on some Biot's consolidations problems. The comparison shows that for 3-D Biot's consolidation problems, both element-by-element and compressed sparse storages can save computing cost and storage significantly, but more achievements can be obtained by adopting compressed sparse storage. This increase with the number of elements is roughly linear for EBE and sparse storages.

In this thesis, the compressed sparse technique is adopted and this compressed sparse storage is constructed by assembling global stiffness matrix in the element loop as shown in Figure 2.4 by collecting nonzero entries in upper (or lower) triangular part of each newly generated element matrix into three vectors, that is, `iebea`, `jebea` and `ebea`, which correspond to global row number, global column number and element entry value, respectively. Taking the natural ordered matrix as an example, the three vectors, `iebea`, `jebea` and `ebea`, should be sorted (by quick sorting algorithm) in ascending order in terms of row number, `iebea` (vectors `jebea` and `ebea` should be changed correspondingly with `iebea`). Next, for the same `iebea`, quick sorting is applied to `jebea` unless the number of `jebea` with the same `iebea` is less than 16, in this case, insertion sorting is adopted for efficiency. Note that `ebea` changes correspondingly with `jebea` (see Numerical Recipes by Press *et al.* and ORDERPACK package for these sorting algorithms). Finally, the entries with the same `iebea` and `jebea` number are summed up, the new three global vectors, `icsra`, `jcsra` and `csra` (or compressed column storage, `icsca`, `jcsca` and `csca`) are formed for global sparse stiffness matrix A. However, as mentioned above, The three vectors, `iebea`, `jebea` and `ebea`, can be overwritten by `icsca`, `jcsca` and `csca`) for efficient storage purpose. That is to say, the vectors, `icsca`, `jcsca` and `csca`) may not be necessary any more. Clearly, for symmetric matrix, CSR storage of upper triangular part is also the CSC storage of lower triangular part. For the operations involving block matrices, each block should be stored, separately. That is,

- `icsrk`, `jcsrk` and `csrk` (or `icsck`, `jcsck` and `csck`) for upper triangular part of K;
- `icsrb`, `jcsrb` and `csrb` (or `icscb`, `jcscb` and `cscb`) for B;
- `icsrc`, `jcsrc` and `csrc` (or `icscc`, `jcscc` and `cscc`) for upper triangular part of C.

2.4 Summary

This chapter gives a broad, but not in-depth, introduction to some popular iterative methods, preconditioning methods, and some storage schemes. Krylov subspace methods are especially emphasized because they are widely applied in engineering and science. During the introduction of preconditioned iterative methods, some related topics are also described. Finally, we compared the characteristics of the EBE based preconditioned iterative methods and sparse preconditioned iterative methods, and proposed to use sparse preconditioned iterative methods in this thesis.

For more details of iterative methods and preconditioning methods, the following well-written literature are recommended: Axelsson (1994); Barrett *et al.* (1994); Kelley (1995); Bruaset, (1995, 1997); Demmel (1997); Greenbaum (1997); Trefethen and Bau (1997); Dongarra *et al.* (1998); Duff and van der Vorst (1998); Meurant (1999); Saad and van der Vorst (2000); Benzi (2002); Eijkhout (2003); Saad (2003); van der Vorst (2003); Benzi *et al.* (2005); Simoncini and Szyld (2005); Elman *et al.* (2005). For sparse matrix storage schemes and sparse matrix operations, refer to Barrett *et al.* (1994); Saad (2003).

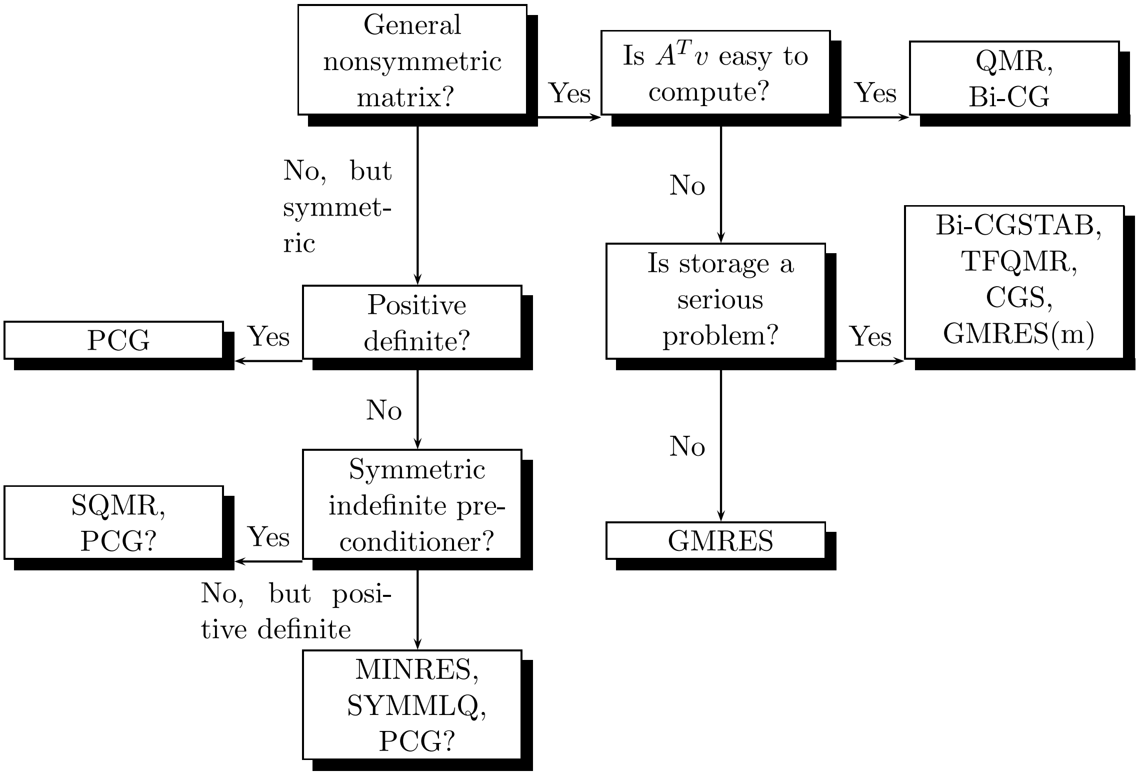
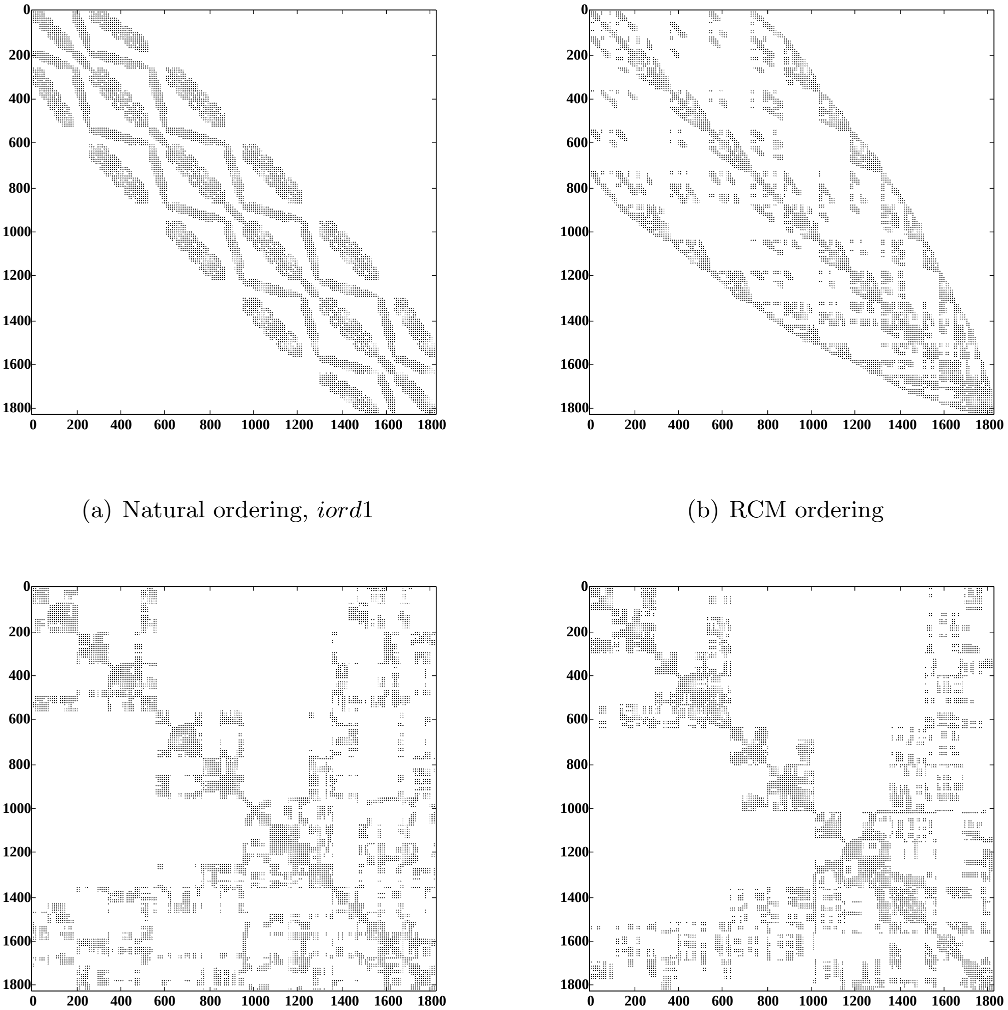

General nonsymmetric matrix?
Yes
Is $A^T v$ easy to compute?
Yes
QMR, Bi-CG
No
No, but symmetric
Is storage a serious problem?
Yes
Bi-CGSTAB, TFQMR, CGS, GMRES(m)
Positive definite?
Yes
PCG
No
No
GMRES
Symmetric indefinite preconditioner?
Yes
SQMR, PCG?
No, but positive definite
MINRES, SYMMLQ, PCG?

Figure 2.1: Flowchart on the selection of preconditioned iterative methods



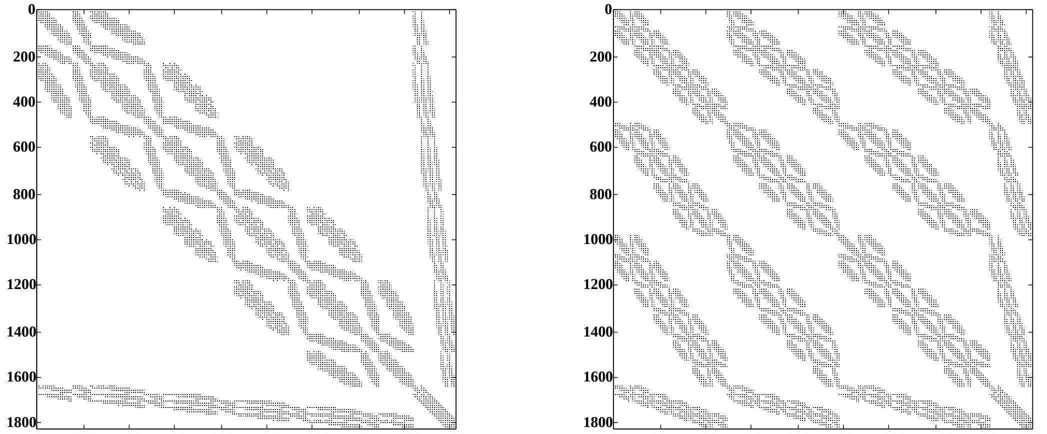
(a) Natural ordering, *iord*1

(b) RCM ordering

(c) MMD ordering

(d) AMD ordering

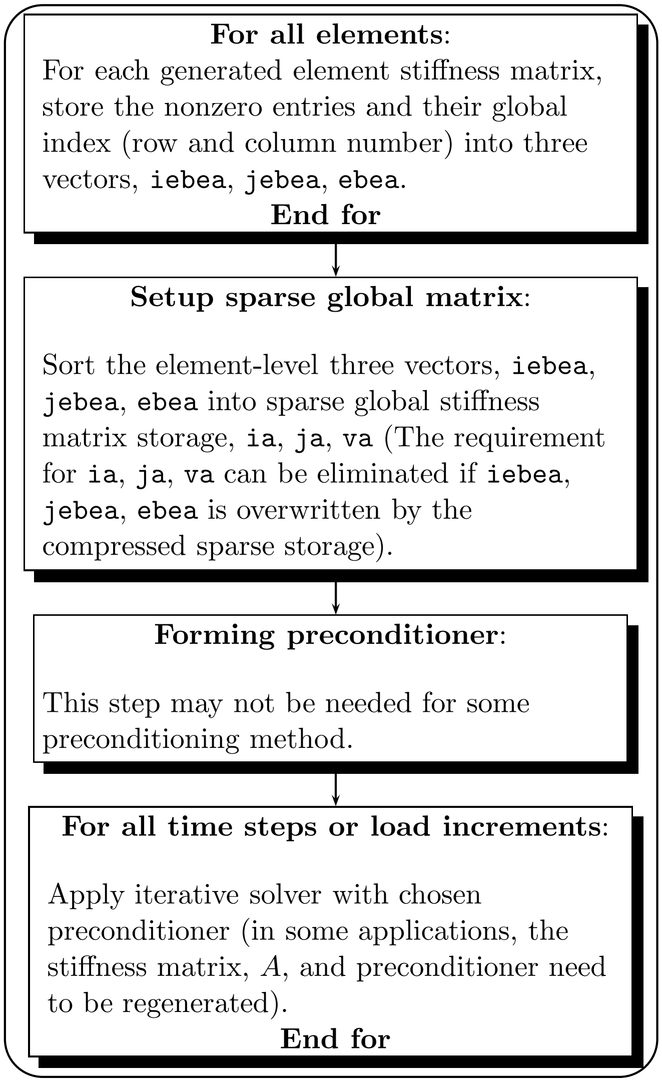
Figure 2.2: Sparsity pattern of matrices after reordering.



(a) Block ordering, *iord*2

(b) Block ordering, *iord*3

Figure 2.3: Sparsity pattern of matrices after block reordering.



For all elements:
For each generated element stiffness matrix, store the nonzero entries and their global index (row and column number) into three vectors, `iebea`, `jebea`, `ebea`.
End for

Setup sparse global matrix:

Sort the element-level three vectors, `iebea`, `jebea`, `ebea` into sparse global stiffness matrix storage, `ia`, `ja`, `va` (The requirement for `ia`, `ja`, `va` can be eliminated if `iebea`, `jebea`, `ebea` is overwritten by the compressed sparse storage).

Forming preconditioner:

This step may not be needed for some preconditioning method.

For all time steps or load increments:

Apply iterative solver with chosen preconditioner (in some applications, the stiffness matrix, A, and preconditioner need to be regenerated).
End for

Figure 2.4: Flow chart of applying sparse preconditioned iterative method in FEM analysis

Table 2.1: EBE technique versus sparse technique

Mesh size	n	n_e	$r_s(\%)$	S_{ebe} (MBytes)	S_{sps} (MBytes)	$flops(Av)_{den}$	$flops(Av)_{ebe}$	$flops(Av)_{sps}$
$8 \times 8 \times 8$	7160	512	2.15	18.1	18.3	1.025×10^8	4.770×10^6	2.204×10^6
$12 \times 12 \times 12$	23604	1728	0.72	61.0	61.9	1.114×10^9	1.610×10^7	8.023×10^6
$16 \times 16 \times 16$	55280	4096	0.32	144.5	146.6	6.112×10^9	3.816×10^7	1.956×10^7
$20 \times 20 \times 20$	107180	8000	0.17	282.2	286.4	2.298×10^{10}	7.453×10^7	3.906×10^7

CHAPTER 3

ITERATIVE SOLUTIONS FOR BIOT'S SYMMETRIC INDEFINITE LINEAR SYSTEMS

3.1 Introduction

For symmetric and positive definite linear systems, the PCG method is the obvious choice. However, there are several choices for symmetric indefinite problems with coupled 2×2 block coefficient matrices though MINRES and SYMMLQ, which exclude symmetric indefinite preconditioners, are regarded as standard iterative methods for such problems. Apart from MINRES and SYMMLQ methods, the other possible choices can be summarized as:

Partitioned Iterative Methods

For the symmetric indefinite linear systems with 2×2 block coefficient matrix, one possibility is to decouple the linear system, then apply possible different iterative schemes on the two decoupled systems. For example, the popular Uzawa-type iterative methods can be categorized into this class.

Coupled Iterative Methods

(a) **Normal Equation Methods**

A general nonsymmetric or indefinite linear system can be transformed into a symmetric positive definite system for which PCG iterative method can be applied. The “normalized” linear system has the form

$$A^T Ax = A^T b, \quad \text{or} \quad AA^T y = b \text{ with } x = A^T y \tag{3.1}$$

where the matrix product $A^T A$ or AA^T is not explicitly formed. CGNE and CGNR methods are variants of this class, the difference between them is that CGNR solves the normalized system with A^T multiplied from left, while CGNE solves the normalized system with A^T multiplied from right (e.g. Barrett *et al.*, 1994; Kelley, 1995; Dongarra *et al.*, 1998). However, their convergence is usually slow because the condition number is squared after the normalization. Therefore, applying PCG to normalized equation are not considered in our problems.

(b) **PCG Method**

As we mentioned above, PCG is the first-choice iterative solver for SPD linear systems. However, PCG has been used with some success for the symmetric indefinite linear systems, and breakdown was observed rarely in practical problems (e.g., Lukšan and Vlček, 1998, 1999, 2000; Smith, 2000; Lee *et al.*, 2002; Smith and Griffiths, 1998, 2004). Recent advance shows that symmetric indefinite CG method (e.g. Lukšan and Vlček, 1998) should be equivalent to the simplified Bi-CG method for a special choice of the auxiliary vector (e.g. Rozložník and Simoncini, 2002).

(c) **PCR Method**

Preconditioned Conjugate Residual (PCR) method has been mentioned in many literatures for symmetric indefinite problems (e.g. Ashby *et al.*, 1990; Hackbusch, 1994; Klawonn, 1995; Toh and Kojima, 2002; Toh, 2003). In fact, PCR is an alternate name of MINRES algorithm (e.g. Wathen and Silvester, 1993).

(d) **SQMR Method**

An investigation on iterative methods for symmetric indefinite problems shows that so far, SQMR can be preconditioned by an arbitrary symmetric preconditioner with theoretical guarantee. When a symmetric positive definite preconditioner or no preconditioner is used, SQMR is mathematically equivalent to MINRES (e.g. Freund and Nachtigal, 1994; Freund and Jarre, 1996).

This chapter is intended to compare some currently popular iterative methods for the solution of Biot's symmetric indefinite linear systems from perspectives of computer runtime and memory usage.

3.2 Linear Systems Discretized from Biot's Consolidation Equations

Since the pioneering work of Terzaghi (1925), soil consolidation has played an important role in soil mechanics. As a further development, three-dimensional soil consolidation theory was put forward by Biot (1941) and it was widely applied in the past decades. In terms of Biot's definition, soil consolidation is the process of a gradual adaptation of the soil to the load variation. The following derivation of the coupled finite element formulation of 3-D Biot's consolidation problem is based on the weighted residual Galerkin method which has been used by Abbo (1997). More detailed derivation of finite element Biot's consolidation formulation are given by Sandhu and Wilson (1969), Abbo (1997), Smith and Griffiths (1998), Zienkiewicz *et al.* (1998), Lewis and Schrefler (1998) and Zienkiewicz and Taylor (2000).

3.2.1 Biot's Consolidation Equations

In Biot's theory, soil is regarded as a porous skeleton filled with water, and the interaction between soil skeleton and pore water is determined by the principle of effective stress and the continuity relation. When taking an infinitesimal soil element, the equilibrium equations of this element can be expressed as

$$\frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + \frac{\partial \tau_{xz}}{\partial z} + b_{sx} = 0 \tag{3.2a}$$

$$\frac{\partial \tau_{yx}}{\partial x} + \frac{\partial \sigma_y}{\partial y} + \frac{\partial \tau_{yz}}{\partial z} + b_{sy} = 0 \tag{3.2b}$$

$$\frac{\partial \tau_{zx}}{\partial x} + \frac{\partial \tau_{zy}}{\partial y} + \frac{\partial \sigma_z}{\partial z} + b_{sz} = 0 \tag{3.2c}$$

Eq. (3.2) can be expressed in a more compact form

$$\widetilde{\nabla}^T \boldsymbol{\sigma} + \mathbf{b}_s = 0 \tag{3.3}$$

where $\mathbf{b}_s = \gamma_s \mathbf{b}$ is the body force vector of soil skeleton, γ_s is unit weight of soil and $\mathbf{b} = \{0, 0, 1\}^T$. $\widetilde{\nabla}$ is a differential operator defined as

$$\widetilde{\nabla}^T = \begin{bmatrix} \dfrac{\partial}{\partial x} & 0 & 0 & \dfrac{\partial}{\partial y} & 0 & \dfrac{\partial}{\partial z} \\ 0 & \dfrac{\partial}{\partial y} & 0 & \dfrac{\partial}{\partial x} & \dfrac{\partial}{\partial z} & 0 \\ 0 & 0 & \dfrac{\partial}{\partial z} & 0 & \dfrac{\partial}{\partial y} & \dfrac{\partial}{\partial x} \end{bmatrix} \tag{3.4}$$

In conjunction with the principle of effective stress, $\boldsymbol{\sigma} = \boldsymbol{\sigma}' + \mathbf{1}p$, Eq. (3.3) can be expressed further as

$$\widetilde{\nabla}^T (\boldsymbol{\sigma}' + p\mathbf{1}) + \mathbf{b}_s = 0 \tag{3.5}$$

where $\boldsymbol{\sigma}' = \{\sigma'_x, \sigma'_y, \sigma'_z, \tau_{xy}, \tau_{yz}, \tau_{zx}\}^T$ is the vector of effective stress; $\mathbf{1} = \{1, 1, 1, 0, 0, 0\}^T$ is a second-order Kronecker delta in vectorial form, and $p = p^{st} + p^{ex}$ is total pore water pressure decomposed with steady state component, p^{st}, and excess component, p^{ex} (pore pressure in excess of that at steady state), respectively.

For a linear elastic solid element, the stress-strain relation is given as

$$\boldsymbol{\sigma}' = \mathbf{D}^e \boldsymbol{\varepsilon} \tag{3.6}$$

where $\boldsymbol{\varepsilon} = \{\varepsilon_x, \varepsilon_y, \varepsilon_z, \gamma_{xy}, \gamma_{yz}, \gamma_{zx}\}^T$ is the strain vector, and $\mathbf{D}^e$ is the elastic stress-strain matrix given as

$$\mathbf{D}^e = \frac{E'}{(1+\nu')(1-2\nu')} \begin{bmatrix} 1-\nu' & \nu' & \nu' & 0 & 0 & 0 \\ \nu' & 1-\nu' & \nu' & 0 & 0 & 0 \\ \nu' & \nu' & 1-\nu' & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5-\nu' & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5-\nu' & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.5-\nu' \end{bmatrix} \tag{3.7}$$

where E' is the effective Young's modulus and $\nu^{'}$ is the effective Poisson's ratio. The strain vector is related to the displacement vector in terms of

$$\boldsymbol{\varepsilon} = \mathbf{B}_u \mathbf{u}_e \tag{3.8}$$

where $\mathbf{u}_e = \{u_{x1}, u_{y1}, u_{z1}, u_{x2}, u_{y2}, u_{z2}, \ldots, u_{xm}, u_{ym}, u_{zm}\}^T$ is the vector of nodal displacement for a m-node solid element and $\mathbf{B}_u$ is the strain-displacement matrix.

Another relation between velocity and pore water pressure is given by the continuity equation. Physically, it means that the volume of fluid flowing in or out is equal to the volume change of the soil mass (if no sources or sinks are considered)

$$\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} + \frac{\partial \varepsilon_v}{\partial t} = 0 \tag{3.9}$$

where $\varepsilon_v = \varepsilon_x + \varepsilon_y + \varepsilon_z = \mathbf{1}^T \boldsymbol{\varepsilon}$ is the volumetric strain, and thus, Eq. (3.9) is expressed in a compact form as

$$div\, \mathbf{v} + \mathbf{1}^T \dot{\boldsymbol{\varepsilon}} = 0 \tag{3.10}$$

here $\mathbf{v} = \{v_x, v_y, v_z\}^T$ is the vector of superficial fluid velocity (i.e., average relative velocity of seepage measured over the total area). These components in coordinate directions can be determined by the Darcy's law

$$\left\{ \begin{array}{c} v_x \\ v_y \\ v_z \end{array} \right\} = \frac{1}{\gamma_w} \left[\begin{array}{ccc} k_{xx} & k_{xy} & k_{xz} \\ k_{yx} & k_{yy} & k_{yz} \\ k_{zx} & k_{zy} & k_{zz} \end{array} \right] \left\{ \begin{array}{c} \dfrac{\partial p}{\partial x} - b_{wx} \\ \dfrac{\partial p}{\partial y} - b_{wy} \\ \dfrac{\partial p}{\partial z} - b_{wz} \end{array} \right\} \tag{3.11}$$

or in a compact form

$$\mathbf{v} = \frac{[\mathbf{k}]}{\gamma_w} (\nabla p - \mathbf{b}_w) \tag{3.12}$$

where $\mathbf{b}_w = \gamma_w \mathbf{b}$; $[\mathbf{k}]$ is the permeability matrix, and usually, $k_{xy} = k_{xz} = k_{yz} = 0$ is assumed; γ_w is the unit weight of pore water taken as $10\, kN/m^3$ in this thesis.

Substituting Eq. (3.12) into the continuity Eq. (3.10) gives

$$div \left[\frac{[\mathbf{k}]}{\gamma_w} \Big(\nabla p - \mathbf{b}_w \Big) \right] + \mathbf{1}^T \dot{\boldsymbol{\varepsilon}} = 0 \tag{3.13}$$

Therefore, Biot's consolidation equations are composed of Eq. (3.5) and Eq. (3.13). To carry out finite element analysis of Biot's consolidation problem, it is required to discretize the consolidation equations in space domain and time domain, respectively. Usually, weighted residual Galerkin method is adopted for the discretization of space domain.

3.2.2 Spatial Discretization

Applying the weighting residual method to Eq. (3.5) results in

$$\int_V \mathbf{w}_u^T [\widetilde{\nabla}^T (\boldsymbol{\sigma}' + \mathbf{1}p) + \mathbf{b}_s] \, dV = 0 \tag{3.14}$$

where $\mathbf{w}_u = \{w_x, w_y, w_z\}^T$ is the vector of weighting functions, and usually, the choice is the Galerkin-weighted functions

$$\mathbf{w}_u = \mathbf{N}_u \delta \mathbf{u}_e \tag{3.15}$$

where $\mathbf{N}_u$ is the displacement shape function matrix for m-node solid element

$$\mathbf{N}_u = \begin{bmatrix} N_1 & 0 & 0 & N_2 & 0 & 0 & \cdots & N_m & 0 & 0 \\ 0 & N_1 & 0 & 0 & N_2 & 0 & \cdots & 0 & N_m & 0 \\ 0 & 0 & N_1 & 0 & 0 & N_2 & \cdots & 0 & 0 & N_m \end{bmatrix}_{3\times 3m} \tag{3.16}$$

and $\delta \mathbf{u}_e$ is the vector of arbitrary increment of element nodal displacements, and $\mathbf{u}_e$ is interpolated to derive the displacement at any point in the element through

$$\mathbf{u} = \mathbf{N}_u \mathbf{u}_e \tag{3.17}$$

Similarly, for a solid element coupled with fluid element, we can define that

$$p = \mathbf{N}_p \mathbf{p}_e \tag{3.18}$$

where N_p is the shape function vector for n-node fluid element as

$$\mathbf{N}_p = \{N_{p1}, N_{p2}, \ldots, N_{pn}\} \tag{3.19}$$

and $\mathbf{p}_e = \{p_1, p_2, \ldots, p_n\}^T$ is the vector of element nodal pore water pressure.

By applying Green's first identity to the first term in Eq. (3.14), we obtain

$$\int_V (\widetilde{\nabla} \mathbf{w}_u)^T (\boldsymbol{\sigma}' + \mathbf{1}p) \, dV - \int_S \mathbf{w}_u^T \mathbf{t} \, dS - \int_V \mathbf{w}_u^T \mathbf{b}_s \, dV = 0 \tag{3.20}$$

where $\mathbf{t} = \{t_x, t_y, t_z\}^T$ is the vector of surface traction force. Substituting Eq. (3.15) into Eq. (3.20) and integrating in an element domain with volume V_e and surface boundary S_e gives

$$(\delta \mathbf{u}_e)^T \left[\int_{V_e} (\widetilde{\nabla} \mathbf{N}_u)^T \boldsymbol{\sigma}' \, dV + \int_{V_e} (\widetilde{\nabla} \mathbf{N}_u)^T \mathbf{1} p \, dV - \int_{S_e} \mathbf{N}_u^T \mathbf{t} \, dS - \int_{V_e} \mathbf{N}_u^T \mathbf{b}_s \, dV \right] = 0 \tag{3.21}$$

Given an arbitrary $\delta \tilde{\mathbf{u}}$, Eq. (3.21) holds only if

$$\int_{V_e} (\widetilde{\nabla} \mathbf{N}_u)^T \boldsymbol{\sigma}' \, dV + \int_{V_e} (\widetilde{\nabla} \mathbf{N}_u)^T \mathbf{1} p \, dV - \int_{S_e} \mathbf{N}_u^T \mathbf{t} \, dS - \int_{V_e} \mathbf{N}_u^T \mathbf{b}_s \, dV = 0 \tag{3.22}$$

Due to $\mathbf{B}_u = \widetilde{\nabla}\mathbf{N}_u$ with the expression

$$\mathbf{B}_u = \begin{bmatrix} \frac{\partial N_1}{\partial x} & 0 & 0 & \frac{\partial N_2}{\partial x} & 0 & 0 & \cdots & \frac{\partial N_m}{\partial x} & 0 & 0 \\ 0 & \frac{\partial N_1}{\partial y} & 0 & 0 & \frac{\partial N_2}{\partial y} & 0 & \cdots & 0 & \frac{\partial N_m}{\partial y} & 0 \\ 0 & 0 & \frac{\partial N_1}{\partial z} & 0 & 0 & \frac{\partial N_2}{\partial z} & \cdots & 0 & 0 & \frac{\partial N_m}{\partial z} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_1}{\partial x} & 0 & \frac{\partial N_2}{\partial y} & \frac{\partial N_2}{\partial x} & 0 & \cdots & \frac{\partial N_m}{\partial y} & \frac{\partial N_m}{\partial x} & 0 \\ 0 & \frac{\partial N_1}{\partial z} & \frac{\partial N_1}{\partial y} & 0 & \frac{\partial N_2}{\partial z} & \frac{\partial N_2}{\partial y} & \cdots & 0 & \frac{\partial N_m}{\partial z} & \frac{\partial N_m}{\partial y} \\ \frac{\partial N_1}{\partial z} & 0 & \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial z} & 0 & \frac{\partial N_2}{\partial x} & \cdots & \frac{\partial N_m}{\partial z} & 0 & \frac{\partial N_m}{\partial x} \end{bmatrix}_{6\times 3m} \tag{3.23}$$

substituting Eqs. (3.6), (3.8), (3.17) and (3.18) into Eq. (3.22) leads to

$$\left(\int_{V_e} \mathbf{B}_u^T \mathbf{D}^e \mathbf{B}_u \, dV\right)\mathbf{u}_e + \left(\int_{V_e} \mathbf{B}_u^T \mathbf{1} \mathbf{N}_p \, dV\right)\mathbf{p}_e = \int_{V_e} \mathbf{N}_u^T \mathbf{b}_s \, dV + \int_{S_e} \mathbf{N}_u^T \mathbf{t} \, dS \tag{3.24}$$

or

$$\mathbf{K}_e \mathbf{u}_e + \mathbf{L}_e \mathbf{p}_e = \mathbf{f}_e \tag{3.25}$$

In the same way, applying the Galerkin-weighted function

$$w_p = \mathbf{N}_p \delta \mathbf{p}_e \tag{3.26}$$

to Eq. (3.13) gives

$$\int_V w_p^T div \left[\frac{[\mathbf{k}]}{\gamma_w}\left(\nabla p - \mathbf{b}_w\right)\right] dV + \int_V w_p^T \mathbf{1}^T \dot{\boldsymbol{\varepsilon}} \, dV = 0 \tag{3.27}$$

and applying Green's first identity to the first term of the above equation results in

$$\int_S w_p^T (\nabla p - \mathbf{b}_w)^T \frac{[\mathbf{k}]}{\gamma_w} \mathbf{n} \, dS - \int_V (\nabla w_p)^T \frac{[\mathbf{k}]}{\gamma_w} (\nabla p - \mathbf{b}_w) \, dV + \int_V w_p^T \mathbf{1}^T \dot{\boldsymbol{\varepsilon}} \, dV = 0 \tag{3.28}$$

where $\mathbf{n}$ is the unit outward normal. Integrating in an element domain and Substituting Eqs. (3.8), (3.15), (3.17), (3.18) and (3.26) into Eq. (3.28), we have

$$\begin{aligned} (\delta \mathbf{p}_e)^T \Bigg[\int_{S_e} \mathbf{N}_p^T (\mathbf{B}_p \mathbf{p}_e - \mathbf{b}_w)^T \frac{[\mathbf{k}]}{\gamma_w} \mathbf{n} \, dS - \int_{V_e} \mathbf{B}_p^T \frac{[\mathbf{k}]}{\gamma_w} \mathbf{B}_p \, dV \mathbf{p}_e + \int_{V_e} \mathbf{N}_p^T \mathbf{1}^T \mathbf{B}_u \, dV \dot{\mathbf{u}}_e \\ + \int_V \mathbf{B}_p^T \frac{[\mathbf{k}]}{\gamma_w} \mathbf{b}_w \, dV \Bigg] = 0 \end{aligned} \tag{3.29}$$

where $\mathbf{B}_p$ is defined as

$$\mathbf{B}_p = \nabla \mathbf{N}_p = \begin{bmatrix} \frac{\partial N_{p1}}{\partial x} & \frac{\partial N_{p2}}{\partial x} & \cdots & \frac{\partial N_{pn}}{\partial x} \\ \frac{\partial N_{p1}}{\partial y} & \frac{\partial N_{p2}}{\partial y} & \cdots & \frac{\partial N_{pn}}{\partial y} \\ \frac{\partial N_{p1}}{\partial z} & \frac{\partial N_{p2}}{\partial z} & \cdots & \frac{\partial N_{pn}}{\partial z} \end{bmatrix}_{3\times n} \tag{3.30}$$

Given an arbitrary $\delta\mathbf{p}_e$, Eq. (3.29) holds only if

$$\int_{V_e} \mathbf{N}_p^T \mathbf{1}^T \mathbf{B}_u \, dV \dot{\mathbf{u}}_e - \int_{V_e} \mathbf{B}_p^T \frac{[\mathbf{k}]}{\gamma_w} \mathbf{B}_p \, dV \mathbf{p}_e = -\int_V \mathbf{B}_p^T \frac{[\mathbf{k}]}{\gamma_w} \mathbf{b}_w \, dV - \int_{S_e} \mathbf{N}_p^T (\mathbf{B}_p \mathbf{p}_e - \mathbf{b}_w)^T \frac{[\mathbf{k}]}{\gamma_w} \mathbf{n} \, dS \tag{3.31}$$

or

$$\mathbf{L}_e^T \dot{\mathbf{u}}_e - \mathbf{G}_e \mathbf{p}_e = \mathbf{g}_e \tag{3.32}$$

In view of the static component of p in Eq. (3.5) and Eq. (3.13) can be eliminated, and the right side in Eq. (3.32) is usually zero, we obtain the coupled element equation

$$\left\{ \begin{array}{lcl} \mathbf{K}_e \mathbf{u}_e + \mathbf{L}_e \mathbf{p}_e^{ex} & = & \mathbf{f}_e \\ \mathbf{L}_e^T \dot{\mathbf{u}}_e - \mathbf{G}_e \mathbf{p}_e^{ex} & = & \mathbf{0} \end{array} \right. \tag{3.33}$$

Clearly, assembling Eq. (3.33) element by element gives the global coupled equation as

$$\left\{ \begin{array}{lcl} \mathbf{K}\mathbf{u} \ + \mathbf{L}\mathbf{p}^{ex} & = & \mathbf{f} \\ \mathbf{L}^T \dot{\mathbf{u}} - \mathbf{G}\mathbf{p}^{ex} & = & \mathbf{0} \end{array} \right. \tag{3.34}$$

where

$$\mathbf{K} = \sum_e \left(\int_{V_e} \mathbf{B}_u^T \mathbf{D}^e \mathbf{B}_u \, dV \right) \tag{3.35a}$$

$$\mathbf{L} = \sum_e \left(\int_{V_e} \mathbf{B}_u^T \mathbf{1} \mathbf{N}_p \, dV \right) \tag{3.35b}$$

$$\mathbf{G} = \sum_e \left(\int_{V_e} \mathbf{B}_p^T \frac{[\mathbf{k}]}{\gamma_w} \mathbf{B}_p \, dV \right) \tag{3.35c}$$

$$\mathbf{f} = \sum_e \mathbf{f}_e \tag{3.35d}$$

The Eq. (3.34) can also be expressed as matrix form as follows

$$\begin{bmatrix} 0 & 0 \\ \mathbf{L}^T & 0 \end{bmatrix} \left\{ \begin{array}{c} \dot{\mathbf{u}} \\ \dot{\mathbf{p}}^{ex} \end{array} \right\} + \begin{bmatrix} \mathbf{K} & \mathbf{L} \\ 0 & -\mathbf{G} \end{bmatrix} \left\{ \begin{array}{c} \mathbf{u} \\ \mathbf{p}^{ex} \end{array} \right\} = \left\{ \begin{array}{c} \mathbf{f} \\ \mathbf{0} \end{array} \right\} \tag{3.36}$$

In Eq. (3.34), the soil stiffness matrix $\mathbf{K}$ is defined as linear elasticity as shown in Eq. (3.35). When nonlinear elasto-plastic soil behavior is considered, the stress-strain relation is determined by the elasto-plastic stress-strain matrix $\mathbf{D}^{ep}$.

3.2.3 Time Integration

Eq. (3.34) is a first-order ordinary differential equation (ODE), it is required to integrate it with respect to time. By using the simplest θ-method which can be categorized into one-step scheme (by definition, one-step scheme only needs the information in previous time step to march the iteration, see Borja (1991); Abbo (1997) for the detail), and thus we obtain

$$\left\{\begin{array}{lcl}\theta\mathbf{K}\mathbf{u}_{n+1}+\theta\mathbf{L}\mathbf{p}^{ex}_{n+1} & = & (\theta-1)\mathbf{K}\mathbf{u}_n+(\theta-1)\mathbf{L}\mathbf{p}^{ex}_n+\mathbf{f}\\ \theta\mathbf{L}^T\mathbf{u}_{n+1}-\theta^2\Delta t\mathbf{G}\mathbf{p}^{ex}_{n+1} & = & \theta\mathbf{L}^T\mathbf{u}_n+\theta(1-\theta)\Delta t\mathbf{G}\mathbf{p}^{ex}_n\end{array}\right. \tag{3.37}$$

or

$$\begin{bmatrix}\theta\mathbf{K} & \theta\mathbf{L}\\ \theta\mathbf{L}^T & -\theta^2\Delta t\mathbf{G}\end{bmatrix}\left\{\begin{array}{c}\mathbf{u}_{n+1}\\ \mathbf{p}^{ex}_{n+1}\end{array}\right\}=\begin{bmatrix}(\theta-1)\mathbf{K} & (\theta-1)\mathbf{L}\\ \theta\mathbf{L}^T & \theta(1-\theta)\Delta t\mathbf{G}\end{bmatrix}\left\{\begin{array}{c}\mathbf{u}_n\\ \mathbf{p}^{ex}_n\end{array}\right\}+\left\{\begin{array}{c}\mathbf{f}\\ \mathbf{0}\end{array}\right\} \tag{3.38}$$

where θ is a time integrating parameter. The choice of $\theta = 1/2$ leads to the Crank-Nicolson approximation method, however, oscillatory results may be incurred by this approximation, and thus, the fully implicit method by choosing $\theta = 1$ is often used (e.g., Smith and Griffiths, 1998, 2004). It is worth noting that for the second equation of Eq. (3.37), integrating with respect to time is processed before or after a θ is multiplied in order to preserve symmetry.

Eq. (3.38) is more often used in finite element analysis based on linear elastic soil model. For the soil with nonlinear behavior, an incremental formulation of Biot's consolidation equations is preferred. It is natural that an incremental version of first equation in Eq. (3.34) is derived as

$$\mathbf{K}\Delta\mathbf{u}+\mathbf{L}\Delta\mathbf{p}^{ex}=\Delta\mathbf{f} \tag{3.39}$$

For the second equation in Eq. (3.34), it can be written as the equations at two time steps, respectively

$$\left\{\begin{array}{lcl}\mathbf{L}^T\dot{\mathbf{u}}_{n+1}-\mathbf{G}\mathbf{p}^{ex}_{n+1} & = & \mathbf{0}\\ \mathbf{L}^T\dot{\mathbf{u}}_n-\mathbf{G}\mathbf{p}^{ex}_n & = & \mathbf{0}\end{array}\right. \tag{3.40}$$

By making use of the linear interpolation in time

$$\frac{\Delta\mathbf{u}}{\Delta t}=\theta\dot{\mathbf{u}}_{n+1}+(1-\theta)\dot{\mathbf{u}}_n \tag{3.41}$$

we can obtain

$$\mathbf{L}^T\Delta\mathbf{u}-\Delta t\mathbf{G}(\theta\Delta\mathbf{p}^{ex}+\mathbf{p}^{ex}_n)=\mathbf{0} \tag{3.42}$$

Therefore, in terms of Eqs. (3.39) and (3.42), the incremental formulation of Biot's consolidation equation can be written as

$$\begin{bmatrix}\mathbf{K} & \mathbf{L}\\ \mathbf{L}^T & -\theta\Delta t\mathbf{G}\end{bmatrix}\left\{\begin{array}{c}\Delta\mathbf{u}\\ \Delta\mathbf{p}^{ex}\end{array}\right\}=\left\{\begin{array}{c}\Delta\mathbf{f}\\ \Delta t\mathbf{G}\mathbf{p}^{ex}_n\end{array}\right\} \tag{3.43}$$

Note that this incremental formulation has been mentioned as the *d*-form implementation because only displacement and pore pressure are required to march the iteration (e.g. Borja, 1991). No matter whether Eq. (3.38) or the incremental formulation (3.43) is used, a symmetric indefinite linear system has to be solved in each time step. In a more general form, the linear system can be written as

$$\begin{bmatrix} K & B \\ B^T & -C \end{bmatrix} \begin{Bmatrix} x \\ y \end{Bmatrix} = \begin{Bmatrix} f \\ g \end{Bmatrix} \quad \text{with} \quad A = \begin{bmatrix} K & B \\ B^T & -C \end{bmatrix} \tag{3.44}$$

Here $A \in \mathbb{R}^{N \times N}$ is a sparse 2×2 block symmetric indefinite matrix; $K = \mathbf{K} \in \mathbb{R}^{m \times m}$ corresponding to soil stiffness is symmetric positive definite (SPD) , $B = \mathbf{L} \in \mathbb{R}^{m \times n}$ is the connection matrix and has full column rank, and $C = \theta \Delta t \mathbf{G} \in \mathbb{R}^{n \times n}$ corresponding to fluid stiffness is symmetric positive semi-definite.

When employing the Sylvester's inertia theorem, the congruence transform

$$A = \begin{bmatrix} K & B \\ B^T & -C \end{bmatrix} = \begin{bmatrix} I & 0 \\ B^T K^{-1} & I \end{bmatrix} \begin{bmatrix} K & 0 \\ 0 & -S \end{bmatrix} \begin{bmatrix} I & K^{-1}B \\ 0 & I \end{bmatrix} \tag{3.45}$$

indicates the indefiniteness because A has m positive eigenvalues and n negative eigenvalues (e.g. Wathen and Silvester, 1993; Demmel, 1997; Elman *et al.*, 1997). Figure 3.1 shows the eigenvalue distribution of the coefficient matrix of a small Biot's linear system. $S = C + B^T K^{-1} B$ in Eq. (3.45) is called *Schur complement* matrix, which arises in many applications.

3.3 Partitioned Iterative Methods

If we rewrite Eq. (3.44) in a more general form

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{Bmatrix} x \\ y \end{Bmatrix} = \begin{Bmatrix} f \\ g \end{Bmatrix} \tag{3.46}$$

then block Gauss elimination of Eq. (3.46) leads to

$$\begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} - A_{21} A_{11}^{-1} A_{12} \end{bmatrix} \begin{Bmatrix} x \\ y \end{Bmatrix} = \begin{Bmatrix} f \\ g - A_{21} A_{11}^{-1} f \end{Bmatrix} \tag{3.47}$$

Where, $S = A_{11} - A_{21} A_{11}^{-1} A_{12}$, is the corresponding Schur complement matrix. One solution method based on this decoupled systems in Eq. (3.47) has been given by Algorithm 3.1.

ALGORITHM 3.1 *Algorithm for the solution of* 2×2 *block linear system of Eq. (3.46).*

```
Constructing Schur complement matrix,
$S = [s_1, s_2, \cdots, s_n] \in \mathbb{R}^{n \times n}$,
for $j = 1$ to $n$ do
    Solve $A_{11}t = A_{12}e_j$
    Form $s_j = A_{22}e_j - A_{21}t$
end for
Solve the block unknowns, $y$
    Solve $A_{11}h = f$
    Solve $Sy = g - A_{21}h$
Solve the block unknowns, $x$
    Solve $A_{11}x = f - A_{12}y$
```

Where, t and h are temporary vectors, while $e_j = \{0, \ldots, \underbrace{1}_{j-th}, \ldots, 0\}^T$ is used to exact the j-th column of a matrix. However, the obvious disadvantage of this method is that exact Schur complement matrix has to be constructed and computational cost during this constructing process is very expensive due to a series of solution of linear systems with coefficient matrix A_{11}. Therefore, many partitioned iterative solution methods have been proposed to avoid this explicit construction.

3.3.1 Stationary Partitioned Iterative Methods

3.3.1.1 Preconditioned Inexact Uzawa Methods

Recently, preconditioned inexact Uzawa methods attract some attentions because inexact Uzawa methods are simple, does not need inner products involved in each iteration and have low memory requirements. Inexact Uzawa methods have been used in the engineering community for large-scale scientific applications. (e.g. Chen, 1998; Cao, 2003). Preconditioned inexact Uzawa method for solving Eq. (3.44) can be written as

$$\left\{ \begin{array}{lcl} x_{k+1} & = & x_k + G^{-1}(f - Kx_k - By_k) \\ y_{k+1} & = & y_k + H^{-1}(B^T x_{k+1} - Cy_k - g) \end{array} \right. \tag{3.48}$$

where $G \in \mathbb{R}^{m \times m}$ and $H \in \mathbb{R}^{n \times n}$ are symmetric positive definite preconditioners that should satisfy certain conditions to ensure convergence of Eq. (3.48).

3.3.1.2 SOR-like Iterative Methods

A class of SOR-like iterative method has been proposed by Golub *et al.* (2001) for the solution of Eq. (3.44) with a zero (2, 2) block, i.e. $C = 0 \in \mathbb{R}^{n\times n}$. This solution procedure can be readily generalized to the linear systems shown in Eq. (3.44). When the original 2×2 coefficient matrix in Eq. (3.44) is decomposed into

$$A = \mathcal{L} + \mathcal{D} + \mathcal{U} \tag{3.49}$$

where $\mathcal{D} = \begin{bmatrix} K & 0 \\ 0 & -\widehat{S} \end{bmatrix}$, $\mathcal{L} = \begin{bmatrix} 0 & 0 \\ B^T & 0 \end{bmatrix}$, $\mathcal{U} = \begin{bmatrix} 0 & B \\ 0 & Q \end{bmatrix}$ and $\widehat{S} = C + Q$ is an approximate Schur complement matrix. Next, over-relaxation is implemented based on the following splitting: $\omega A = (\mathcal{D} + \omega\mathcal{L}) - [(1-\omega)\mathcal{D} - \omega\mathcal{U}]$, which leads to the following iterative scheme

$$\left\{ \begin{array}{c} x_{k+1} \\ y_{k+1} \end{array} \right\} = \mathcal{M}_\omega \left\{ \begin{array}{c} x_k \\ y_k \end{array} \right\} + \omega(\mathcal{D} + \omega\mathcal{L})^{-1} \left\{ \begin{array}{c} f \\ g \end{array} \right\} \tag{3.50}$$

with

$$\mathcal{M}_\omega = (\mathcal{D}+\omega\mathcal{L})^{-1}[(1-\omega)\mathcal{D}-\omega\mathcal{U}] = \begin{bmatrix} (1-\omega)I & -\omega K^{-1}B \\ \omega(1-\omega)\widehat{S}^{-1}B^T & -\omega\widehat{S}^{-1}(\omega B^T K^{-1}B + C) + I \end{bmatrix} \tag{3.51}$$

From Eq. (3.50), the SOR-like algorithm is derived as

$$\left\{ \begin{array}{rcl} x_{k+1} & = & (1-\omega)x_k + \omega K^{-1}(f - By_k) \\ y_{k+1} & = & y_k + \omega\widehat{S}^{-1}(B^T x_{k+1} - Cy_k - g) \end{array} \right. \tag{3.52}$$

Clearly, in each iteration of this SOR-like method, there are two solves about K and $\widehat{S}$, respectively.

3.3.1.3 A More General Iteration Method

In fact, many partitioned iterative methods including preconditioned inexact Uzawa and SOR-like method fall into one class for which Richardson's iteration is applied to unknowns x (displacement) and y (excess pore pressure), respectively. Note that applying block Gauss elimination to Eq. (3.44) leads to the decoupled linear systems

$$\left\{ \begin{array}{rcl} Kx & = & f - By \\ Sy & = & B^T K^{-1} f - g \quad \text{with } S = C + B^T K^{-1} B \end{array} \right. \tag{3.53}$$

Therefore, applying Richardson's iteration $u_{k+1} = u_k + \omega r_k$ (e.g. Saad, 1996; Meurant, 1999; Saad and van der Vorst, 2000) to the above decoupled linear systems results in

$$\left\{ \begin{array}{lcl} x_{k+1} & = & x_k + \omega_x r_k \\ y_{k+1} & = & y_k + \omega_y \tilde{r}_k \end{array} \right. \tag{3.54}$$

where r_k and $\tilde{r}_k$ are the residual vectors of the corresponding linear system in Eq. (3.53). One natural extension of Eq. (3.54) is to choose ω_x and ω_y in terms of appropriate corresponding preconditioning matrices with suitable relaxation parameter ξ and η, i.e.

$$\left\{ \begin{array}{lcl} x_{k+1} & = & x_k + \xi G^{-1} r_k \\ y_{k+1} & = & y_k + \eta H^{-1} \tilde{r}_k \end{array} \right. \tag{3.55}$$

or

$$\left\{ \begin{array}{lcl} x_{k+1} & = & x_k + \xi G^{-1}(f - Kx_k - By_k) \\ y_{k+1} & = & y_k + \eta H^{-1}(B^T x_{k+1} - Cy_k - g) \end{array} \right. \tag{3.56}$$

Here, G and H are the preconditioning matrices as shown in Eq. (3.48). The convergence behavior for linear systems with zero (2, 2) block has been studied by Cui (2004). Obviously, when relaxation parameters $\xi = \eta = 1.0$, Eq. (3.56) reduces to preconditioned inexact Uzawa method, and when $\xi = \eta = \omega$, $G = K$ and $H = \widehat{S}$, Eq. (3.56) reduces to SOR-like methods given in Eq. (3.52).

A more general partitioned iterative method derived from Eq. (3.55) was suggested by Hu and Zou (2001), but the relaxation parameters ξ and η varied in each iteration. Their method exhibits some similarity to the CG method.

3.3.2 Nonstationary Partitioned Iterative Methods

It is obvious that the method suggested by Hu and Zou (2001) does not follow the stencil of stationary iterative method, and can be expected to converge faster with the choice of optimal relaxation parameters. Another nonstationary partitioned iterative method was proposed by Prevost (1997). Prevost's PCG iterative procedure can be readily applied to the two decoupled linear systems given in Eq. (3.53), while the outer PCG iteration is applied to the Schur complement system. The Schur complement matrix need not be formed explicitly. More specifically, the Prevost's PCG iterative scheme is applied in terms of the following steps: (1) Outer PCG method is applied to Schur complement linear system $Sy = B^T K^{-1} f - g$; (2) When implementing the first step, only one matrix-vector product with Schur complement S is involved, that is, $q_k = S\,p_k$, which is decomposed into several steps without forming S explicitly by carrying out the matrix-vector products of $q_k = S\,p_k = (C + B^T K^{-1} B)p_k$ separately.

The algorithm of Prevost's PCG iterative procedure applied to Eq. (3.53) is provided by the following Algorithm 3.2.

Algorithm 3.2 *Prevost's inner-outer PCG algorithm for the solution of* 2×2 *block linear system of Eq. (3.44).*

```
Choose $y_0$, solve $Kx_0 = f - By_0$,
set $\tilde{r}_0 = B^T x_0 - g - Cy_0$,
$z_0 = H^{-1}\tilde{r}_0$, $p_0 = z_0$.
for $k = 0$ to max_it do
    Solve $Kt_k = Bp_k$ by inner PCG
    $q_k = Cp_k + B^T t_k \quad (i.e., q_k = S\,p_k)$
    $\alpha_k = (\tilde{r}_k, z_k)/(q_k, p_k)$
    $y_{k+1} = y_k + \alpha_k p_k$
    $\tilde{r}_{k+1} = \tilde{r}_k - \alpha_k q_k$
    Check convergence
    $z_{k+1} = H^{-1}\tilde{r}_{k+1}$
    $\beta_{k+1} = (\tilde{r}_{k+1}, z_{k+1})/(r_k, z_k)$
    $p_{k+1} = z_{k+1} + \beta_{k+1} p_k$
end for
```

Remark 3.1 *Prevost's PCG iterative procedure is applied to Schur complement system and the excess pore pressure unknown y is obtained when residual $\tilde{r}_{k+1}$ is assumed small enough. There are two obvious methods to solve for the displacement unknown x: (1) one natural way is to solve $Kx = f - By_{k+1}$ after the solution y_{k+1} is solved by Algorithm 3.2; (2) The displacement unknown can be updated at the same time with pore pressure by $x_{k+1} = x_k - \alpha_k t_k$ (e.g. Prevost, 1997; Gambolati et al., 2002).*

Remark 3.2 *There is one linear system of K in each iteration. Thus, an inner PCG can be used for such a linear system. However, $Kt_k = Bp_k$ must be solved with sufficient precision (that is, this inner linear system of K must be solved by PCG method with a very small tolerance such as 10^{-12}) so that the resultant solution x_{k+1} is accurate enough. Otherwise, x_{k+1} may not be accurate even though convergence criterion on the residual $\tilde{r}_{k+1}$ is satisfied. In this study, the inner linear system is solved by SSOR preconditioned*

PCG method with so-called Eisenstat trick, and the algorithm is given in Algorithm 3.3 (e.g., Eisenstat, 1981; Saad, 1996; Meurant, 1999; van der Vorst, 2003).

The standard SSOR preconditioner based on $K = L_K + D_K + L_K^T$ for solving the linear system $Kx = b$ can be written as

$$P_{SSOR} = (L_k + D_k)D_k^{-1}(D_k + L_k^T) \tag{3.57}$$

where L_K, D_K are strictly lower triangular part and diagonal part of K, respectively.

Algorithm 3.3 *SSOR($\omega = 1.0$) preconditioned PCG algorithm with Eisenstat trick for the solution of inner linear system.*

```
Choose $x_0$ as initial guess, set $z_0 = 0$,
Compute $r_0 = (L_K + D_K)^{-1}b$, $s_k = D_K r_k$, $p_0 = s_0$.
for $k = 0$ to max_it do
    $t_k = \widetilde{K}p_k$    carried out by Procedure PMatvec of Eq. (3.60)
    $\alpha_k = (r_k, s_k)/(t_k, p_k)$
    $z_{k+1} = z_k + \alpha_k p_k$
    $r_{k+1} = r_k - \alpha_k t_k$
    $s_{k+1} = D_K r_{k+1}$
    Check convergence, if converged, set $x_{k+1} = x_0 + (D_K + L_K^T)^{-1} z_{k+1}$
    $\beta_{k+1} = (r_{k+1}, s_{k+1})/(r_k, s_k)$
    $p_{k+1} = s_{k+1} + \beta_{k+1} p_k$
end for
```

Remark 3.3 *Notice that when applying SSOR preconditioned PCG method with Eisenstat trick, the preconditioned matrix must be symmetric positive definite so that PCG can work. Therefore, SSOR preconditioner is used separately so that preconditioned matrix (which is still symmetric positive definite) is,*

$$\widetilde{K} = (L_K + D_K)^{-1} K (L_K^T + D_K)^{-1} \tag{3.58}$$

furthermore,

$$\widetilde{K} = (L_K + D_K)^{-1}[(L_K + D_K) - D_K + (L_K^T + D_K)](L_K^T + D_K)^{-1} \tag{3.59}$$

but with a right preconditioner D_K, which does not need to be inverted in this situation. This efficient SSOR preconditioned PCG algorithm has been mentioned by Meurant (1999). Eisenstat trick takes effect when the preconditioned matrix $\widetilde{K}$ is multiplied by a *vector p_k, thus leading to the following four steps:*

$$\begin{array}{ll} \underline{\textbf{Procedure PMatvec}}\ : & \\ (1) & f = (L_K^T + D_K)^{-1} p_k \\ (2) & g = D_K f \\ (3) & h = (L_K + D_K)^{-1}(p_k - g) \\ (4) & t_k = f + h \end{array} \tag{3.60}$$

3.4 Global Krylov Subspace Iterative Methods

By applying suitable global Krylov subspace iterative methods for symmetric indefinite Biot's linear systems, problems can be solved directly instead of decoupling it into two SPD linear systems. In Section 3.1, the brief review of iterative solution methods for symmetric indefinite linear systems shows that theoretically, only SQMR can be used with an indefinite preconditioner for Biot's linear systems. However, some recent practical applications and analyses show that breakdown may rarely occur when PCG is used for indefinite problems, and MINRES can not be used, without risk, with symmetric preconditioner unless the preconditioner is positive definite (Dongarra *et al.*, 1998). We attempt to use PCG and MINRES though the restrictions related to PCG and MINRES may be avoided. Therefore, all three iterative methods preconditioned by positive definite and indefinite preconditioners are tested, respectively. The algorithms for PCG and SQMR are given in Appendix A. There exist two versions of MINRES. One makes use of Lanczos algorithm and Givens Rotations (e.g. Paige and Saunders, 1975; Greenbaum, 1997; Dongarra *et al.*, 1998). The second version has been analyzed by Neytcheva and Vassilevski (1998) or Wang (2004). In this study, the preconditioned MINRES method given by Wang (2004) is used, but a small change is made to reduce two matrix-vector products to one in each iteration. The resultant preconditioned MINRES method with one matrix-vector product and one preconditining solve is given by Algorithm 3.4

ALGORITHM 3.4 *Preconditioned MINRES Algorithm for the solution of linear system of Eq. (3.44).*

```
choose an initial guess $x_0$,
set $r_0 = b - Ax_0$, $p_0 = z_0 = M^{-1}r_0$,
$q_0 = Ap_0$, $t_0 = Az_0$ $(= q_0)$.
for $k = 0$ to max_it do
    Compute
    $u_k = M^{-1}q_k$
    $\alpha_k = (z_k, q_k)/(q_k, u_k)$
    $x_{k+1} = x_k + \alpha_k p_k$
    $r_{k+1} = r_k - \alpha_k q_k$
    $z_{k+1} = z_k - \alpha_k u_k$
    $t_{k+1} = Az_{k+1}$
    $\beta_{k+1} = (z_{k+1}, t_{k+1})/(z_k, t_k)$
    $p_{k+1} = z_{k+1} + \beta_{k+1}p_k$
    $q_{k+1} = t_{k+1} + \beta_{k+1}q_k$
end for
```

3.5 Preconditioning Strategies

As emphasized in Section 2.2, preconditioning is always an essential issue when using iterative methods. For the above discussed partitioned iterative methods, two preconditioning matrices need to be chosen for the two decoupled SPD linear systems. For Prevost's inner-outer PCG procedure, only preconditioning matrix, H, for Schur complement matrix is needed. Therefore, both in stationary partitioned iterative method given by Eq. (3.56) and in Prevost's PCG algorithm, only two simple schemes for preconditioning matrix H are studied:

Scheme (1): Diagonal preconditioning $H = \beta\, diag(\widehat{S}) = \beta\, diag\big(C + B^T diag(K)^{-1}B\big)$;

Scheme (2): SSOR preconditioning based on $H = \beta\, \widehat{S} = \beta\,\big(C + B^T diag(K)^{-1}B\big)$.

Here, the scalar $\beta = 4$ is chosen to enhance the preconditioning matrix. There are also many choices for the other preconditioning matrix G when implementing stationary partitioned iterative method, but only SSOR preconditioning $G = (L_K + D_K)D_K^{-1}(L_K^T + D_K)$ is considered for computational efficiency.

As for global Krylov subspace iterative methods such as MINRES and SQMR, all three methods preconditioned by GJ preconditioner are compared. It is recognized that we are violating theoretical restrictions, namely, PCG is meant for SPD linear systems and MINRES should be combined with SPD preconditioners. GJ preconditioner was proposed by Phoon *et al.* (2002) as a diagonal scaling with the form,

$$P_{GJ} = \begin{bmatrix} diag(K) & 0 \\ 0 & \alpha\, diag\big(C + B^T diag(K)^{-1} B\big) \end{bmatrix} \tag{3.61}$$

and $\alpha < -1$ was recommended. The use of indefinite GJ preconditioner was found to be more suitable. The recommendation agrees with some recent observations that an indefinite preconditioner may be more effective than a definite one for the problems arising from mixed finite element approximations (e.g. Lukšan and Vlček, 1998; Perugia and Simoncini, 2000). In practical implementation, the above global form of P_{GJ} is proposed with an element-by-element (EBE) form constructed from the following pseudo-code:

for $i = 1$ to m, **do**

$$\tilde{p}_{ii} = K(i,i) \tag{3.62}$$

for $j = 1$ to n, **do**

$$\begin{aligned} \tilde{p}_{m+j,m+j} &= \alpha \left[\left(\sum_{i=1}^{m} \frac{B(i,j)^2}{K(i,i)} \right) + C(j,j) \right] \\ &= \alpha \left[\left(\sum_{i=1}^{m} \frac{\left(\sum_{e}^{ne} B(i,j)_e \right)^2}{K(i,i)} \right) + C(j,j) \right] \\ &\approx \alpha \left[\left(\sum_{i=1}^{m} \frac{\sum_{e}^{ne} (B(i,j)_e)^2}{K(i,i)} \right) + C(j,j) \right] \end{aligned} \tag{3.63}$$

where $B(i,j)_e$ is the entry in the B-block of the eth finite element referenced globally and ne is the total number of elements in the finite element mesh. In addition, only the storage of one vector $\{\tilde{p}_{ii}\}_{1\leq i\leq N}$ for P_{GJ} preconditioner is required. For sparse iterative solutions, the construction procedure for diagonal GJ preconditioner is provided in Algorithm 3.5.

Algorithm 3.5 *Constructing GJ preconditioner with a vector* $\{\tilde{a}_{ii}\}_{1\leq i\leq N}$.

```
*Set* $\mathcal{S}$ *as the nonzero sparsity structure of (1, 2) block* $B \in \mathbb{R}^{m\times n}$
**for** $i = 1$ *to* $m$ **do**
    $\tilde{a}_{ii} = K(i,i)$
**end for**
**for** $j = 1$ *to* $n$ **do**
    *Set* $\tilde{a}_{m+j,m+j} = C(j,j)$
    **for** $i = 1$ *to* $m$ **do**
        **If** $(i,j) \in \mathcal{S}$
            *Set* $\tilde{a}_{m+j,m+j} = \tilde{a}_{m+j,m+j} + B(i,j)^2 \tilde{a}_{ii}^{-1}$
        **end if**
    **end for**
    $\tilde{a}_{m+j,m+j} = \alpha\tilde{a}_{m+j,m+j}$
**end for**
```

To compare the behaviors of CG, MINRES and SQMR preconditioned by GJ preconditioner, GJ preconditioner is made to be positive-definite or indefinite by changing the sign of α.

3.6 Numerical Examples

3.6.1 Convergence Criteria

An iterative method can be terminated when the approximate solution is deemed sufficiently accurate. In this study, the following relative residual convergence criterion with zero initial guess is adopted:

$$\text{Iteration stops when } \; k \geq max\_it \; \text{ or } \; R_b = \frac{\|r_k\|_2}{\|r_0\|_2} = \frac{\|b - Ax_k\|_2}{\|b\|_2} \leq stop\_tol \qquad (3.64)$$

where $\|\cdot\|_2$ denotes the 2-norm, max_it and $stop\_tol$ are the maximum iteration number and stopping tolerance, respectively. In this study, $max\_it = 20000$ and $stop\_tol = 10^{-6}$ are designated for all the iterative methods discussed above. For the inner linear system in Prevost's method, a more stringent stopping tolerance, $stop\_tol(in) = 10^{-12}$ is required.

3.6.2 Problem Descriptions

The studied finite element mesh sizes range from $8 \times 8 \times 8$ to $20 \times 20 \times 20$ basing on one flexible square footing problem with uniform vertical pressure of 0.1 MPa. Figure 3.2 shows the largest $20\times20\times20$ finite element mesh, and the symmetry property allows only a quadrant of the footing to be considered. The adopted finite elements are 20-node solid quadrilateral elements coupled to 8-node fluid elements. Therefore, each 3-D element consists of 60 displacement degrees of freedom and 8 excess pore pressure degrees of freedom, which explains why the total number of DOFs of displacements would be much larger than that of excess pore pressure, usually, the ratio is larger than 10. The details of those 2×2 stiffness matrices for the meshes from $8 \times 8 \times 8$ to $20 \times 20 \times 20$ has been given in Table 3.1.

The ground water table is assumed to be at the ground surface and is in hydrostatic condition at the initial stage. The base of the mesh is assumed to be fixed in all directions and impermeable, side face boundaries are constrained in the transverse direction, but free in in-plane directions (both displacement and water flux). The top surface is free in all direction and free-draining with pore pressures assumed to be zero. The soil material is assumed to be isotropic, homogeneous and linear elastic with constant effective Poisson's ratio $\nu^{'} = 0.3$. In this study, two homogeneous soil profiles with significantly different effective Young's modulus E' and hydraulic permeability parameter k are defined as:

Soil profile 1: homogeneous soft clay with $E' = 1$ MPa and $k = 10^{-9}$ m/s;

Soil profile 2: homogeneous dense sand with $E' = 100$ MPa and $k = 10^{-5}$ m/s.

The 0.1 MPa uniform footing load is applied "instantaneously" over the first time step of 1s. Time increment is taken as $t = 1$ s.

3.6.3 Numerical Results

The performances of iterative methods or preconditioning methods are evaluated over a range of mesh sizes and material properties based on the following indicators:

(a) RAM usage during iteration.

(b) Number of iterations required to achieve a residual norm below 10^{-6} (iteration count).

(c) CPU time spent prior to execution of iterative solver (overhead) includes time spent on computation of element stiffness matrices, assembly of global stiffness matrix and construction of preconditioners.

(d) CPU time spent within iterative solver (iteration time), which is a function of the iteration count and the time consumed per iteration.

Compressed sparse storage schemes are used for the 2×2 coefficient matrix. More specifically, for partitioned iterative methods, the blocks, upper triangular part of K, B and upper triangular part of C, are stored separately, but for global Krylov subspace iterative methods, the upper triangular part of coupled matrix A is stored. The number of nonzero entries in each block has been provided in Table 3.1. For more details about compressed sparse storages, refer to Barrett *et al.* (1994) and Saad (1996). Table 3.2 shows the performance of stationary partitioned iteration given by Eq. (3.56) for the footing problems. The relaxation parameters have such a standard choice, $\xi = 1.0$ and $\eta = 1.0$, though optimal choices could be possible to obtain a better convergence rate. A rigorous study of the parameters choice for partitioned stationary iterative methods can be derived by Golub *et al.* (2001) for the special case where the $C = 0$. It can be seen that the scheme 2 for H may not lead to less iteration than the scheme 1 for the soil profile 1, but for the soil profile 2, the scheme 2 for H can lead to less iteration than the scheme 1. However, due to the cheap diagonal implementation, scheme 1 may need less computer runtime.

Table 3.3 provides the numerical results of Prevost's inner-outer PCG method with two preconditioning schemes for H. The provided iteration counts are outer PCG iteration counts and the average iteration counts required by inner PCG are given in bracket. Similar to stationary partitioned iterative methods, in Prevost's iterative method, scheme 1 for H leads to faster convergence than scheme 2, while for the soil profile 2, preconditioning scheme 2 for H results in faster convergence. A common phenomenon in Table 3.2 and Table 3.3 is that the performance (including iteration count and runtime) of the partitioned iterative methods is not consistent for the same preconditioning choice, but it is related closely to soil properties. Therefore, one suitable preconditioning matrix should be selected for different soil property. This inconsistency of performance of scheme 1 or 2 could be explained as: the effect of $B^T diag(K)^{-1}B$ in $\widehat{S} = C + B^T diag(K)^{-1}B$ plays an important role for a preconditioning scheme because matrix $B(L)$ is constant as shown in Eq. (3.35c). Specifically speaking, for soil profile 1, the diagonal part of $B^T diag(K)^{-1}B$ dominates the property of $\widehat{S}$, while for soil profile 2, the approximation quality of $\widehat{S}$ is dominated by both C and $B^T diag(K)^{-1}B$. From this observation, it is known that for partitioned iterative methods, care should be taken when selecting a preconditioning scheme of H for different soil conditions.

For the 2×2 block symmetric indefinite linear systems given by Eq. (3.44), the performances of PCG, MINRES and SQMR methods are compared, and numerical results are given in Table 3.4 for positive definite GJ preconditioner and in Table 3.5 for indefinite GJ preconditioner, respectively. It must be emphasized that though it is widely thought that PCG should be used for SPD linear systems and MINRES method should be combined with symmetric positive definite preconditioner, we still take a risk to use them for Biot's symmetric indefinite linear systems. As shown in Tables 3.4 and 3.5, scaling factor $\alpha = +4.0$ and $\alpha = -4.0$ are used in GJ to make this preconditioner positive definite and indefinite, respectively. When comparing Prevost's PCG procedure with GJ preconditioned methods, It can be seen that GJ preconditioned methods do not show obvious advantage unless an indefinite GJ preconditioner, that is $\alpha = -4$, is selected. This effect is obvious for all three methods because the selection of the parameter $\alpha = -4$ can significantly improve the performance of GJ preconditioned iterative method with $\alpha = +4$. More specifically, the selection of $\alpha = -4$ in GJ preconditioner can lead to about 75% saving of computer runtime based on the studied problems. This observation can be verified by the suggestions of Phoon *et al.* (2002) and Toh *et al.* (2004) to choose a negative α to obtain faster convergence when a preconditioner does not select the exact K as its (1, 1) block.

In terms of Figure 3.3, Tables 3.4 and 3.5, it is interesting to note that the iterations required by PCG and SQMR are same and convergence behaviors of two methods also coincide, but due to the cheaper iteration, PCG may spend a little less computer runtime than SQMR method. This similarity could be explained that on the indefinite problems, the two methods may be closely related by simplified Bi-CG method (e.g. Rozložník and Simoncini, 2002). However, using indefinite preconditioned CG for indefinite linear systems may not always be guaranteed theoretically.

3.7 Conclusions

To solve symmetric indefinite linear systems stemming from Biot's consolidation problems, two basic iterative solution strategies are studied, one is partitioned iterative methods and the other one is coupled iterative methods. The following observations can be summarized:

(a) With suitable choice of a preconditioner such as an indefinite preconditioner, preconditioned coupled iterative methods are superior to partitioned iterative methods.

(b) Though it is widely accepted that PCG is used for SPD linear systems and MINRES should be used with a positive definite preconditioner, numerical results show that PCG and MINRES can be used successfully even though the theoretical restrictions are violated.

(c) MINRES shows a better convergence rate than PCG and SQMR when the same preconditioner is chosen, but at the cost of two additional recurrence relations. PCG and SQMR have the same convergence behaviors, which may be explained from the observation that they are closely connected by the simplified Bi-CG method.

(d) From the above studies, it is clear that given a symmetric preconditioner, there are no obvious distinctions (on iteration counts and computing cost) between MINRES, PCG and SQMR. But when choosing an iterative method, reliability without numerical breakdown is still a considering factor. Thus, it is recommended to choose preconditioned SQMR method for further fast solution strategies of symmetric indefinite Biot's linear systems.

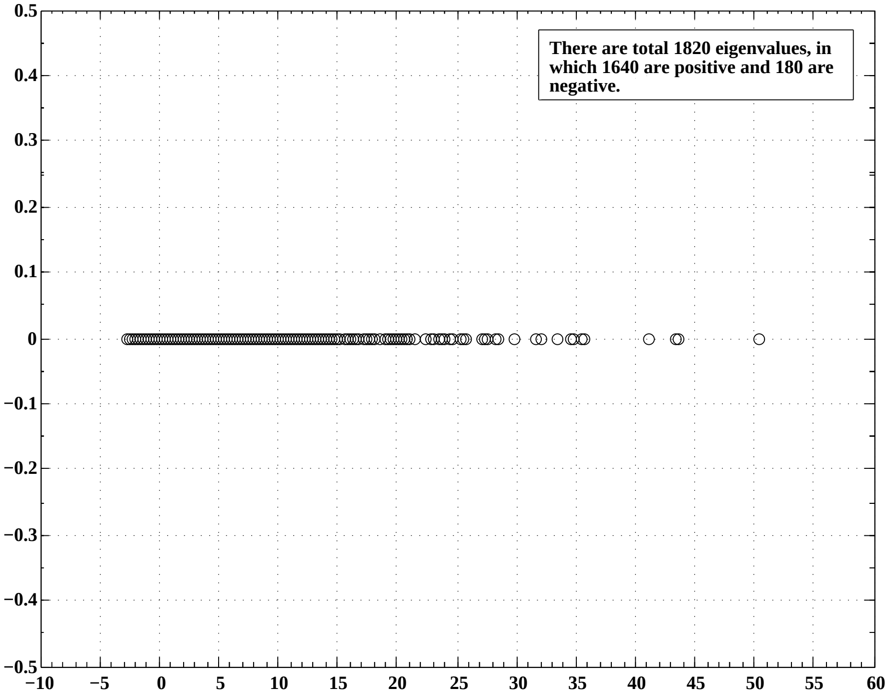


Figure 3.1: Eigenvalue distribution of stiffness matrix A ($m = 1640, n = 180$) of Biot's linear system.

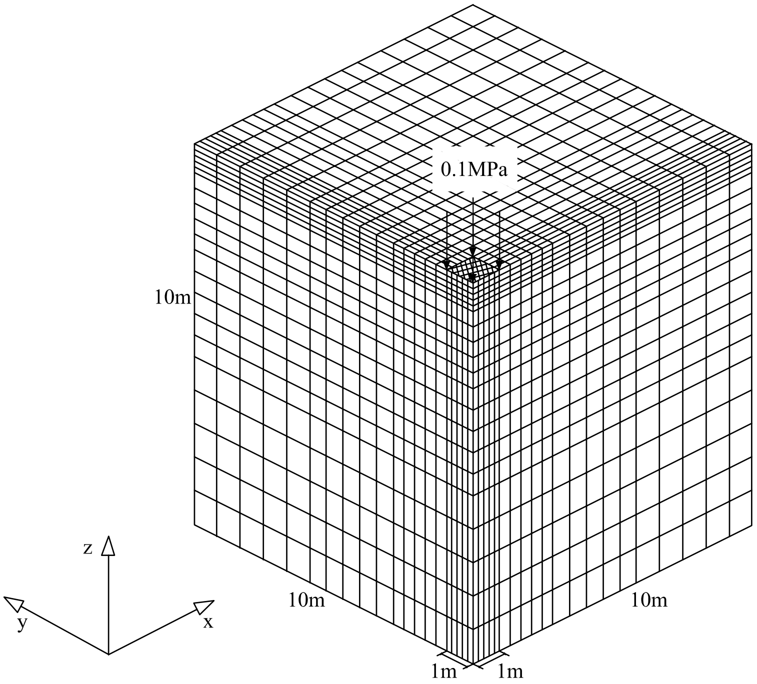


Figure 3.2: 20 × 20 × 20 finite element mesh of a quadrant symmetric shallow foundation problem.

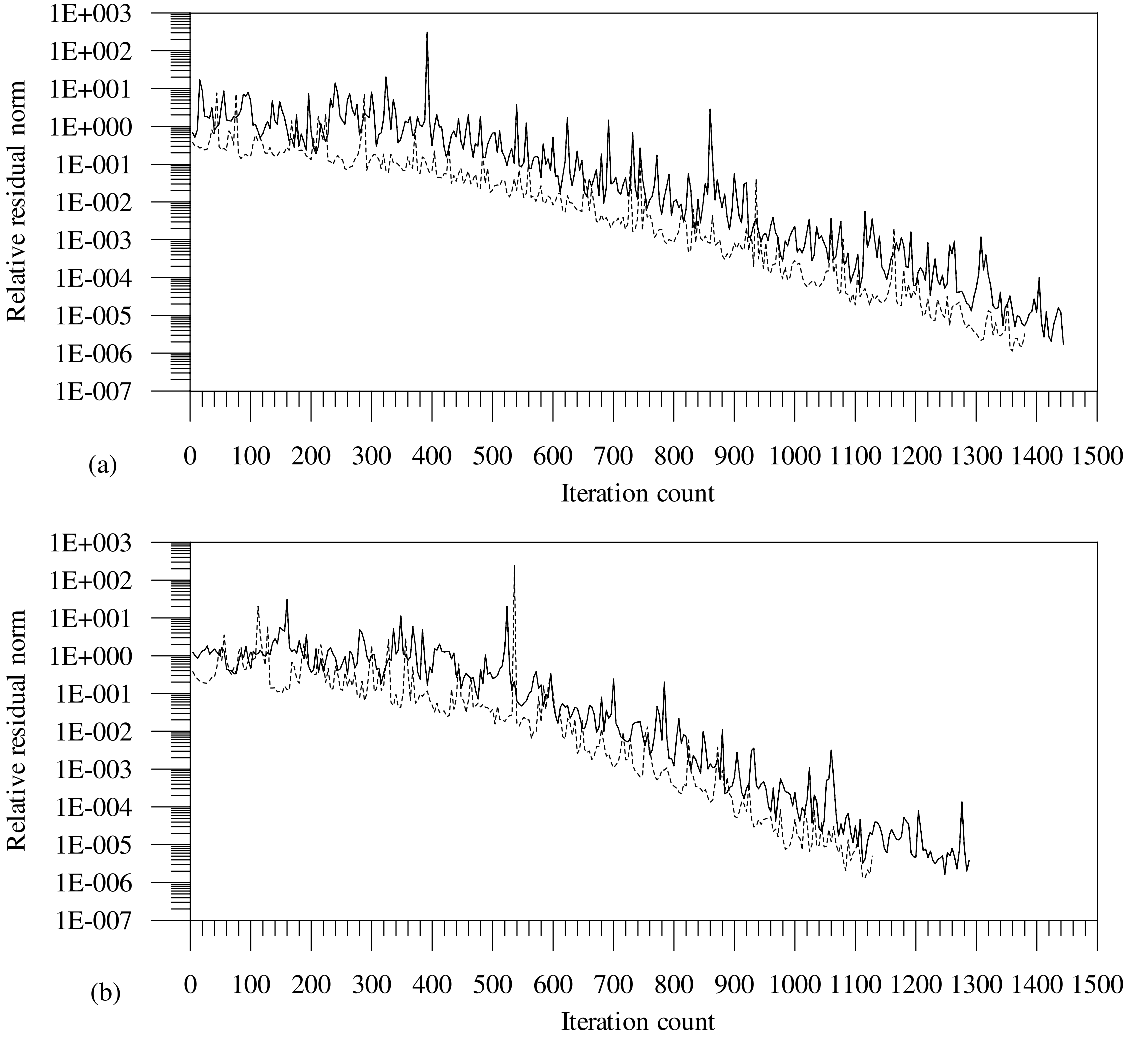


Figure 3.3: Convergence history of GJ($\alpha = -4.0$) preconditioned coupled iterative methods, solid line is for PCG and SQMR methods, while dashed line is for MINRES method; (a). Homogeneous problem with soil profile 1; (b) Homogeneous problem with soil profile 2.

Table 3.1: 3-D finite element meshes

	Mesh size			
	$8 \times 8 \times 8$	$12 \times 12 \times 12$	$16 \times 16 \times 16$	$20 \times 20 \times 20$
Number of elements(ne)	512	1728	4096	8000
Number of nodes	2673	8281	18785	35721
$DOFs$				
Pore pressure(n)	648	2028	4624	8820
Displacement (m)	6512	21576	50656	98360
Total ($N = m + n$)	7160	23604	55280	107180
No. non-zeros (nnz)				
$nnz(triu(\mathrm{K}))$	443290	1606475	3920747	7836115
$nnz(B)$	103965	375140	907608	1812664
$nnz(triu(C))$	7199	24287	57535	112319
$nnz(\widehat{S})$	51714	187974	461834	921294
$nnz(A)/N^2$ (%)	2.15	0.72	0.32	0.17

Table 3.2: Performance of stationary partitioned iterations ($\xi = 1.0$, $\eta = 1.0$) over mesh refining.

Mesh Size	$8 \times 8 \times 8$	$12 \times 12 \times 12$	$16 \times 16 \times 16$	$20 \times 20 \times 20$
Homogeneous problem with soil profile 1				
Stationary partitioned iteration by Eq. (3.56) with Scheme 1 for H				
RAM (MB)	28.0	73.0	166.5	328.0
Iteration count	927	1899	3220	4893
Overhead time (s)	11.0	37.5	89.6	177.3
Iteration time (s)	76.6	569.6	2363.1	7217.3
Total runtime (s)	88.3	609.0	2457.1	7403.1
Stationary partitioned iteration by Eq. (3.56) with Scheme 2 for H				
RAM (MB)	28.0	73.0	167.0	330.0
Iteration count	929	1911	3242	4918
Overhead time (s)	12.3	51.6	168.2	474.3
Iteration time (s)	79.0	588.1	2442.2	7445.3
Total runtime (s)	92.0	641.6	2614.8	7928.2
Homogeneous problem with soil profile 2				
Stationary partitioned iteration by Eq. (3.56) with Scheme 1 for H				
RAM (MB)	28.0	73.0	166.5	328.0
Iteration count	937	1992	3301	5062
Overhead time (s)	11.0	37.4	89.5	176.3
Iteration time (s)	77.4	597.2	2422.5	7452.0
Total runtime (s)	89.0	636.6	2516.5	7636.8
Stationary partitioned iteration by Eq. (3.56) with Scheme 2 for H				
RAM (MB)	28.0	73.0	167.0	330.0
Iteration count	939	1930	3274	4966
Overhead time (s)	12.3	51.6	168.3	474.5
Iteration time (s)	79.7	593.1	2483.3	7500.8
Total runtime (s)	92.6	646.6	2656.0	7983.8

Table 3.3: Performance of Prevost's inner-outer PCG method over mesh refining

Mesh Size	$8 \times 8 \times 8$	$12 \times 12 \times 12$	$16 \times 16 \times 16$	$20 \times 20 \times 20$
Homogeneous problem with soil profile 1				
Prevost's PCG iteration by Algorithm 3.2 with Scheme 1 for H				
RAM (MB)	28.0	73.0	166.5	328.0
Iteration count	30 (115)	28 (160)	28 (210)	28 (260)
Overhead time (s)	11.0	37.5	89.5	176.2
Iteration time (s)	140.0	675.2	2153.5	5336.8
Total runtime (s)	151.5	714.5	2247.4	5521.5
Prevost's PCG iteration by Algorithm 3.2 with Scheme 2 for H				
RAM (MB)	28.0	73.0	167.0	330.0
Iteration count	26 (115)	31 (160)	36 (210)	39 (260)
Overhead time (s)	12.3	51.5	167.8	473.7
Iteration time (s)	123.2	746.1	2762.5	7426.5
Total runtime (s)	136.1	799.5	2934.7	7908.7
Homogeneous problem with soil profile 2				
Prevost's PCG iteration by Algorithm 3.2 with Scheme 1 for H				
RAM (MB)	28.0	73.0	166.5	328.0
Iteration count	17 (85)	19 (120)	25 (160)	32 (200)
Overhead time (s)	11.0	37.5	89.5	176.1
Iteration time (s)	58.4	333.7	1403.2	4406.1
Total runtime (s)	70.0	373.1	1497.2	4590.7
Prevost's PCG iteration by Algorithm 3.2 with Scheme 2 for H				
RAM (MB)	28.0	73.0	167.0	330.0
Iteration count	17 (85)	17 (120)	17 (160)	17 (200)
Overhead time (s)	12.3	51.5	167.7	474.0
Iteration time (s)	59.7	312.9	1016.1	2542.2
Total runtime (s)	72.6	366.3	1188.3	3024.7

Table 3.4: Performance of preconditioned by GJ(+4) method over mesh refining for homogeneous problems.

Mesh Size	$8 \times 8 \times 8$	$12 \times 12 \times 12$	$16 \times 16 \times 16$	$20 \times 20 \times 20$
RAM(MB)	26.5	84.0	194.5	386.0
Homogeneous soil with soil profile 1				
SQMR				
Iteration count	1792	2947	4616	6350
Overhead (s)	10.8	37.0	88.2	173.6
Iteration time (s)	65.3	388.8	1486.0	4106.8
Total runtime (s)	76.7	427.6	1578.5	4288.6
PCG				
Iteration count	1792	2947	4616	6350
Overhead (s)	10.8	36.9	88.0	173.0
Iteration time (s)	64.9	385.1	1473.7	4070.2
Total runtime (s)	76.4	423.8	1566.1	4251.6
MINRES				
Iteration count	1676	2597	4288	5899
Overhead (s)	10.9	37.1	89.4	175.9
Iteration time (s)	62.7	348.9	1421.3	3907.2
Total runtime (s)	74.3	387.9	1515.1	4091.8
Homogeneous soil with soil profile 2				
SQMR				
Iteration count	1306	2495	4517	6655
Overhead (s)	10.8	37.0	88.2	173.7
Iteration time (s)	47.5	328.5	1454.9	4295.5
Total runtime (s)	59.0	367.4	1547.3	4477.4
PCG				
Iteration count	1306	2495	4517	6655
Overhead (s)	10.8	36.8	88.0	173.4
Iteration time (s)	47.2	325.6	1440.5	4263.5
Total runtime (s)	58.6	364.4	1532.9	4445.3
MINRES				
Iteration count	1237	2294	3991	6236
Overhead (s)	10.8	37.5	88.5	174.0
Iteration time (s)	45.8	308.0	1304.8	4095.1
Total runtime (s)	57.2	347.4	1397.6	4277.7

Table 3.5: Performance of GJ(−4) method over mesh refining for homogeneous problems.

Mesh Size	$8 \times 8 \times 8$	$12 \times 12 \times 12$	$16 \times 16 \times 16$	$20 \times 20 \times 20$
RAM(MB)	26.5	84.0	194.5	386.0
Homogeneous soil with soil profile 1				
SQMR				
Iteration count	379	700	1067	1455
Overhead (s)	10.8	37.0	88.2	173.5
Iteration time (s)	13.8	91.8	342.0	935.3
Total runtime (s)	25.2	130.7	434.4	1117.1
PCG				
Iteration count	379	700	1067	1455
Overhead (s)	10.8	36.9	87.9	173.0
Iteration time (s)	13.67	91.0	339.3	928.0
Total runtime (s)	25.1	129.9	431.6	1109.4
MINRES				
Iteration count	368	652	1002	1424
Overhead (s)	10.8	36.8	87.9	173.0
Iteration time (s)	13.5	86.7	324.8	926.1
Total runtime (s)	25.0	125.5	417.0	1107.6
Homogeneous soil with soil profile 2				
SQMR				
Iteration count	345	575	839	1251
Overhead (s)	10.8	37.0	88.2	173.7
Iteration time (s)	12.5	75.2	268.5	803.9
Total runtime (s)	23.9	114.1	360.9	985.8
PCG				
Iteration count	345	575	839	1251
Overhead (s)	10.8	36.9	87.9	172.9
Iteration time (s)	12.4	74.6	266.1	797.8
Total runtime (s)	23.8	113.4	358.4	979.2
MINRES				
Iteration count	323	559	807	1251
Overhead (s)	10.8	36.9	87.9	173.3
Iteration time (s)	11.9	74.2	261.0	813.3
Total runtime (s)	23.3	113.0	353.3	995.1

CHAPTER 4

BLOCK CONSTRAINED VERSUS GENERALIZED JACOBI PRECONDITIONERS

4.1 Introduction

The finite-element discretization of Biot's consolidation equations typically give rises to a large symmetric indefinite linear system of Eq. (3.43) or Eq. (3.44). A more detailed description of Biot's consolidation equations is given in Section 3.2. It suffices to note here that the number of displacement degrees of freedom (DOFs) m and the number of pore pressure DOFs n has a ratio of about 10 (see Table 4.1 for examples). The matrix A is generally sparse. Figure 2.3(a) illustrates the sparsity pattern of A with block ordering ($iord = 2$) from a $5 \times 5 \times 5$ meshed footing problem. A wide class of constrained problems involving mixed finite element formulations also produces the above 2×2 block matrix structure (e.g. Zienkiewicz *et al.*, 1985).

As shown in the study of Chapter 3, there are several possible symmetric iterative methods for solving symmetric indefinite Eq. (3.44), such as PCG, MINRES, and SQMR. Though numerical results provided in Chapter 3 have shown that no breakdowns occurred for PCG and MINRES methods in the iteration process when the restrictions related to CG or MINRES are violated, SQMR is still preferred as the iterative solver for Eq. (3.44) in view of the fact that theoretically, it can admit an arbitrary symmetric preconditioner.

For the linear system arising from Biot's consolidation equations, one preconditioner has been demonstrated to be effective in terms of time and storage for solution of large-scale 3-D problems on a modest PC platform. This preconditioner is the Generalized Jacobi (GJ) diagonal preconditioner with the form given by Eq. (3.61), and the EBE-based construction of GJ is provided by the pseudo-code in Eqs. (3.62) and (3.63), and the sparse construction is given by Algorithm 3.5. The motivation for the construction of the above GJ preconditioner comes from a theoretical eigenvalue clustering result developed by Murphy *et al.* (2000) for a linear system of the form given by Eq. (3.44) but with $C = 0 \in \mathbb{R}^{n \times n}$. The constant α is chosen to be -4 and the motivation comes from the theoretical result developed by Phoon *et al.* (2002). In recent years, explicit approximate inverses have been derived as preconditioners and used in conjunction with explicit preconditioned conjugate gradient methods for parallel computing (e.g. Lipitakis and Gravvanis, 1995; Huckle, 1999; Gravvanis, 1999). However, such preconditioners are designed for implementation on parallel computers and are not suitable for the PC environment assumed in this thesis.

Although the GJ preconditioned SQMR method has been demonstrated to have very good performance in that it converges in a moderate number of iterations compared to the dimension of the linear system, it is highly desirable to find other more sophisticated preconditioners that can cut down the iteration count further while at the same time keeping the cost of each preconditioning step at a reasonable level. To design such a preconditioner, it is useful to keep in mind the three basic criteria given in Section 2.2.

Toh *et al.* (2004) systematically analyzed three types of block preconditioners to solve the linear system given by Eq. (3.44) and evaluated their practical effectiveness using the first two criteria. It was assumed that sufficient memory is available to store the entire global coefficient matrix A in random access memory (RAM). On a limited set of numerical experiments on problems with DOFs less than 24,000 (so that global A can be stored), the block constrained preconditioner (P_c) considered in that paper was demonstrated to have superior performance (time-wise) compared to the block diagonal and block triangular preconditioners. However, the feasibility of implementing P_c to satisfy criterion (3) was not addressed. This issue is of paramount practical importance if the block constrained preconditioner is to be applicable for solving very large linear system of equations on PC platforms commonly found in most engineering design offices.

The first objective of this Chapter is to address the efficient implementation and memory management of P_c. The second objective is to push the envelope of problem sizes studied by Toh *et al.* (2004) to ascertain the generality of the comparative advantages P_c has over GJ on significantly larger linear systems of equations. The last objective is to explain semi-empirically why a P_c-preconditioned system is expected to converge faster than one that is preconditioned by GJ based on eigenvalue distributions.

4.2 Block Constrained Preconditioners

4.2.1 Overview

Recall that if no threshold is applied to block B, the block constrained preconditioner introduced in Section 2.2.3.2 is a 2×2 block matrix of the form:

$$P_c = \begin{bmatrix} \widehat{K} & B \\ B^T & -C \end{bmatrix} \tag{4.1}$$

where $\widehat{K}$ is a symmetric positive definite approximation of K. To solve a very large linear system of equations, the only practical choice for $\widehat{K}$ at the moment is $\widehat{K} = diag(K)$. Approximations that are based on incomplete Cholesky factorizations of K would be extremely expensive to compute (in terms of storage and time) because K needs to be assembled and stored explicitly. Throughout this thesis, the approximation $\widehat{K} = diag(K)$ is used.

To apply the preconditioner P_c within an iterative solver, it is not necessary to compute its LU factorization explicitly - it can be applied efficiently by observing that its inverse has the following analytical form:

$$P_c^{-1} = \begin{bmatrix} \widehat{K}^{-1} - \widehat{K}^{-1}B\widehat{S}^{-1}B^T\widehat{K}^{-1} & \widehat{K}^{-1}B\widehat{S}^{-1} \\ \widehat{S}^{-1}B^T\widehat{K}^{-1} & -\widehat{S}^{-1} \end{bmatrix} \tag{4.2}$$

where $\widehat{S} = C + B^T\widehat{K}^{-1}B$ is the Schur complement matrix associated with P_c. With the expression in Eq. (4.2), the preconditioning step in each iterative step can be implemented efficiently via the following pseudo-code (e.g. Toh *et al.*, 2004):

$$\begin{cases} \textbf{Given} & [u; v] \\ \textbf{Compute} & w = \widehat{K}^{-1}u \\ \textbf{Compute} & z = \widehat{S}^{-1}(B^T w - v) \\ \textbf{Compute} & P_c^{-1}[u; v] = [\widehat{K}^{-1}(u - Bz); z] \end{cases} \tag{4.3}$$

Assuming that the sparse Cholesky factorization $\widehat{S} = LL^T$ has been computed, It can be readily seen that one preconditioning step involves two multiplications of $\widehat{K}^{-1}$ with a vector, the multiplications of the sparse matrices B and B^T with a vector, and the solution of two triangular linear systems. The multiplication of $\widehat{K}^{-1}$ with a vector can be done efficiently since it is a diagonal matrix, and this involves only m scalar multiplications. In the next subsection, the implementation details on the construction of $\widehat{S}$ as well as operations involving the sparse matrices B, B^T, and C will be presented. Given the small number of non-zeros in B and $\widehat{S}$ (see Table 4.1 for examples), sparsity has to be exploited to maximize computational speed and minimize memory usage.

4.2.2 Implementation Details

For large finite element problems, the data related to the linear system can be stored in an unassembled form (e.g. van der Vorst, 2003; Saad, 1996, 2003), and the matrix-vector multiplications involved in the solution process can be carried out using an element-by-element (EBE) implementation. For the GJ preconditioner, the multiplication of A with a vector in each iterative step of SQMR is implemented at the element level as follows:

$$Av = \left(\sum_{i=1}^{ne} J_i^T A_i^e J_i \right) v \tag{4.4}$$

where A is the global coefficient matrix, $J_i \in \mathbb{R}^{p \times N}$ is a connectivity matrix transforming local to global DOFs (e.g. Dayde *et al.*, 1997), p is the number of DOFs in one element, ne is the number of elements, and $A_i^e \in \mathbb{R}^{p \times p}$ is the ith element stiffness matrix, which can be expressed further as:

$$A_i^e = \begin{bmatrix} K_i^e & B_i^e \\ B_i^{eT} & -C_i^e \end{bmatrix} \tag{4.5}$$

In the preconditioning step described in Eq. (4.3), it would be too expensive from the perspective of CPU time if $B^T w$ and Bz are implemented at the EBE level because this requires scanning through all the element stiffness matrices for assembly. Given the relatively smaller dimensions of B, B^T and C relative to K (i.e., $n \ll m$), it is more reasonable to assemble these sparse matrices in global form prior to applying the SQMR solver. In addition, the availability of the global matrices B, B^T and C allows the computation of Av to be done efficiently through the following procedure:

$$\begin{cases} \textbf{Given} & [u;v] \\ \textbf{Compute} & z_1 = Bv, z_2 = B^T u, z_3 = Cv \\ \textbf{Compute} & w = Ku \, \text{at EBE level} \\ \textbf{Compute} & A[u;v] = [w + z_1; z_2 - z_3] \end{cases} \tag{4.6}$$

where only the multiplication of Ku is done at the EBE level in a manner analogous to Eq. (4.4). From Eqs.(4.3) and (4.6), It can be seen that in each SQMR step, there are two pairs of sparse matrix-vector multiplications involving B and B^T.

In assembling the sparse global matrices B, B^T, and C, careful memory management is a must to avoid incurring excessive additional memory allocation. In implementation, one first stores all the nonzero entries of B at the element level into three vectors, where the first and second store the global row and global column indices, and the last stores the nonzero element-level value. Next, one sorts the three vectors by the row and column indices, and then add up the values that have the same row and column indices. With the sorted vectors, a final step is performed to store the sparse matrix B in a Compressed Sparse Row (CSR) format with associated 1-D arrays `icsrb(m+1)`, `jcsrb(bnz)`, `csrb(bnz)`, where *bnz* is the total number of nonzero entries in sparse B matrix (e.g. Barrett *et al.*, 1994; Saad, 1996, 2003). The Compressed Sparse Column (CSC) format of B can be obtained readily from the CSR format (via SPARSKIT at http://www-users.cs.umn.edu/∼saad/software/home.html). The arrays associated with the CSC format of B are denoted by `jcscb(n+1)`, `icscb(bnz)`, `cscb(bnz)`. Given the CSR and CSC format of B, the multiplications $B^T w$ and Bz can then be efficiently computed via Algorithm 4 and 5 described in the Appendix B.2. In our implementation, the global sparse matrices B and B^T are stored implicitly as the CSC and CSR format of B, respectively, and C is stored in the CSC format. Note that storing these global sparse matrices in the CSC or CSR format requires less RAM than storing their unassembled element versions because overlapping degrees of freedom and zero entries within element matrices are eliminated.

In constructing P_c, the next issue that needs to be addressed is the efficient computation of the Schur complement $\widehat{S} = C + B^T\widehat{K}^{-1}B$. The way it is done is shown in the following algorithm.

Algorithm 4.1 *Computing $\widehat{S} = G = C + B^T diag(K)^{-1}B$ given the CSC and CSR format of B, and the CSC format of C. Let d be the diagonal of $diag(K)^{-1}$.*

```
Allocate integer arrays rowG, colG and double array nzG.
Set $w = zeros(m, 1)$, $z = zeros(n, 1)$;
for $j = 1$ to $n$ do
    (1) initialize $w = zeros(m, 1)$, $z = zeros(n, 1)$;
    (2) extract the $j$th column of $B$ from the CSC format and put it in $w$;
    (3) for $i = 1$ to $m$ do; $w(i) = w(i) \times d(i)$; end for;
    (4) compute $z = B^T w$;
    (5) scan through the vector $z$ to add the $j$th column of $C$,
    and at the same time store the row and columns indices
    and the values of the values of the non-zero elements
    of $z$ in the arrays rowG, colG, nzG.
end for
```

Once the $n \times n$ matrix $\widehat{S}$ is computed, it is required to compute its sparse Cholesky factorization so as to compute $\widehat{S}^{-1}v$ for any given vector v. It is well known that directly factorizing a sparse matrix may lead to excessive fill-ins and hence uses up a large amount of memory space. To avoid excessive fill-ins, the matrix is usually re-ordered by the reverse Cuthill-McKee (RCM) or the Multiple Minimal Degree (MMD) algorithms (see George and Liu, 1981, 1989) prior to applying the Cholesky factorization. In this thesis, the MMD method is adopted, MMD is an effective reordering algorithm that usually leads to a sparser factor than other reordering algorithms. For readers who are not familiar with the solution process of a sparse symmetric positive definite linear system, $Hx = b$, it is noted that the sparse Cholesky factorization process is generally divided into four stages (e.g. George and Liu, 1981; Lee *et al.*, 1999):

(a) **Reordering**: Permutate symmetrically the columns and rows of matrix H using one reordering method. Suppose the permutation matrix is $\mathcal{P}$.

(b) **Symbolic factorization**: Set up a data structure for the Cholesky factor L of $\mathcal{P}H\mathcal{P}^T$.

(c) **Numerical factorization**: Perform row reductions to find L so that $\mathcal{P}H\mathcal{P}^T = LL^T$.

(d) **Triangular solution**: Solve $LL^T\mathcal{P}x = \mathcal{P}b$ for $\mathcal{P}x$ by solving two triangular linear systems. Then recover x from $\mathcal{P}x$.

In applying the P_c preconditioner in the SQMR solver, the reordering and Cholesky factorization of $\widehat{S}$ are performed only once before calling the SQMR solver. However, in the preconditioning step within the SQMR solver, the triangular solves must be repeated for each iterative step. One may use the sparse Cholesky factorization subroutine from the SparseM package (http://cran.r-project.org/). The algorithm was developed by Ng and Peyton (1993).

4.3 Numerical Examples

4.3.1 Convergence Criteria

In this study, the relative residual norm convergence criteria given by Eq. (3.64) is used, but with $stop\_tol = 10^{-6}$ and $max\_it = 2000$. More details about various stopping criteria can be found in Appendix A.2.

4.3.2 Problem Descriptions

Figure 4.1 shows a sample finite element mesh of a flexible square footing resting on homogeneous soil subjected to a uniform vertical pressure of 0.1 MPa. Symmetry consideration allows a quadrant of the footing to be analysed. Mesh sizes ranging from $12 \times 12 \times 12$ to $24 \times 24 \times 24$ were studied. These meshes result in linear systems of equations with DOFs ranging from about 20,000 to 180,000, respectively. The largest problem studied by Toh *et al.* (2004) contains only about 24,000 DOFs. Similar to the problems studied in Chapter 3, Twenty-node brick elements were used. Each brick element consists of 60 displacement degrees of freedom (8 corner nodes and 12 mid-side nodes with 3 spatial degrees of freedom per node) and 8 excess pore pressure degrees of freedom (8 corner nodes with 1 degree of freedom per node). Details of the 3-D finite element meshes are given in Table 4.1. Other problem details are same as those given in Chapter 3. All the numerical studies are conducted using a Pentium IV, 2.0 GHz desktop

PC with a physical memory of 1 GB.

4.3.3 Comparison Between GJ and P_c

Figure 4.2(a) shows that both GJ and P_c are very efficient for this class of problem from an iteration count point of view. For practical applications, it is worth noting that these preconditioners actually become even more efficient when the problem size increases. For example, iteration count for GJ decreases from about 3% of the problem dimension for the smaller $12 \times 12 \times 12$ footing problem to only 1% of the problem dimension for the $24 \times 24 \times 24$ footing problem. Despite this efficiency, P_c is able to out-perform GJ on two counts. First, the iteration count for P_c is almost the same for the two material types studied (Figure 4.2(a)). This implies better scaling with respect to the effective Young's modulus and hydraulic permeability. Second, iteration count for P_c is less than half of that for GJ and more significantly, this ratio *decreases* with problem size as shown in Figure 4.2(b). The ratio for the largest problem studied is only about one-third.

Reductions in iteration count do not translate in a straightforward way to savings in total runtime. Overhead and CPU time consumed per iteration are additional factors to consider. For the former, it is not surprising that P_c is more expensive given the fairly elaborate steps discussed in Section 4.2.2 (Figure 4.3(a)). Computation of the $n \times n$ Schur complement $\widehat{S}$ and the ensuing sparse Cholesky factorization are significant contributors to this overhead expense. Nevertheless, the increase in overhead with DOFs could have been more onerous than power of 1.27 if sparsity of B were not exploited for computation of $\widehat{S} = C + B^T \widehat{K}^{-1} B$ and Cholesky factorization. For fully dense matrices, one expects the overhead to grow with DOFs to the power of 3. As for time per iteration, one preconditioning P_c step involves two multiplications of $\widehat{K}^{-1}$ with a vector, the multiplications of the sparse matrices B and B^T with a vector, and the solution of two triangular linear systems. The corresponding GJ step only involves multiplication of P_{GJ} with a vector, which can be done efficiently in $(m+n)$ scalar multiplications since P_{GJ} is a diagonal matrix. Nevertheless, P_c is only marginally more expensive within each SQMR iteration because available sparse matrix-vector multiplications and triangular solves are very efficient (growing with DOFs to the power of 1.11 as shown in Figure 4.3(b)).

Although overhead and time per iteration are more costly for P_c, it is still possible to achieve total runtime savings over GJ as shown in Figure 4.3(c). This may be illustrated using the $24 \times 24 \times 24$ footing problem with 184,296 DOFs for soil profile 1. The reduction in iteration count is about 32% (Figure 4.2(b)). However, time per iteration and total runtime/iteration runtime for P_c are about 1.42 (Tables 4.2 and 4.3) and 1.18 (Figure 4.3(b)) that of GJ, respectively. Hence, it can be deduced that the reduction in total runtime would be $0.32 \times 1.42 \times 1.18 = 0.54$, which reproduces the result shown in Figure 4.3(c). An interesting question is how much savings in total runtime is achievable (if any) when the DOFs increase by one order of magnitude (i.e., in the millions). It is not unreasonable to extrapolate from Figure 4.2(b), 4.3(b) and 4.3(d) that iteration count, time per iteration and total runtime/iteration runtime for P_c are about 0.3, 1.8 ($1.42 \times 10^{0.11}$), and 1.25 that of GJ, respectively, in this case. These crude extrapolated data imply a reduction in total runtime to be about two-thirds.

The final practical aspect that needs to be considered is to compare RAM usage between P_c and GJ as shown in Figure 4.4. For the largest $24 \times 24 \times 24$ footing problem, P_c requires about 20% more RAM than GJ (Tables 4.2 and 4.3). For a problem with one order of magnitude larger DOFs, P_c would probably require about $1.2 \times 10^{0.08} \approx 1.45$ times the RAM required by GJ. This requirement may be reasonable when PCs with few GB RAM become more commonly available in engineering design offices.

4.3.4 Eigenvalue Distribution of Preconditioned Matrices and Convergence Rates

Figure 4.5 shows the convergence history of the relative residual norm $\|r_k\|_2/\|r_0\|_2$ as a function of iteration number k for the solution of $5 \times 5 \times 5$ meshed problem using SQMR preconditioned by GJ and P_c. It is clear that the P_c-preconditioned method converges much faster than the GJ-preconditioned counterpart, with the former terminating at 105 steps while the latter at 192 steps. It is possible to explain semi-empirically this difference on the convergence rate if one looks at the eigenvalue distributions of the preconditioned matrices. Figure 4.6 shows these eigenvalue distributions. It is noted that the real part of the eigenvalues of the GJ preconditioned matrix are contained in the interval [0.0132, 5.4476], while those of the P_c-preconditioned matrix are contained in the interval [0.0131, 5.0270]. Thus the real part of the eigenvalues of both preconditioned matrices has very similar distributions. But there is an important difference when one look at the imaginary part of the eigenvalues. While the P_c-preconditioned matrix has only real eigenvalues (follows from Theorem 2 given by Toh *et al.* (2004), the GJ-preconditioned matrix has 288 eigenvalues that have imaginary parts (appear as "wings" near the origin in Figure 4.6). Based on the eigenvalue distribution, it is possible to get an estimate of the asymptotic convergence rate of the SQMR method when applied to the preconditioned linear system. For the P_c-preconditioned system, the convergence rate is roughly given by:

$$\rho_{p_c} = \frac{\sqrt{k}-1}{\sqrt{k}+1} = 0.903 \tag{4.7}$$

where k(condition number) $= 5.0270/0.01321 = 383.741$.

The estimation of the convergence rate associated with the eigenvalue distribution for the GJ-preconditioned system is less straightforward, but is achievable with the help of Schwarz-Christoffel mapping for polygonal region. The idea is to approximately enclose the eigenvalues in a polygonal region, and find the Schwaz-Christoffel map Φ that maps the exterior of the polygonal region to the unit disk. Then the convergence rate for an eigenvalue distribution that has eigenvalues densely distributed throughout the polygon is given by:

$$\rho_{GJ} = \Phi(0) \tag{4.8}$$

The used polygonal region is shown in Figure 4.7, where the extreme left vertex of the polygon coincides with the eigenvalue 0.0132. Based on that polygonal approximation, the estimated convergence rate is $\rho_{GJ} = 0.958$. To compute the mapping Φ, the highly user friendly MATLAB software package developed by Driscoll (1996) is used. One can use the estimated asymptotic convergence rates to predict how the number of required preconditioned SQMR steps would differ. Recall that for a method with convergence rate ρ to achieve:

$$\frac{\|r_k\|_2}{\|r_0\|_2} < 10^{-6} \tag{4.9}$$

The number of steps k required is given by:

$$k \approx \frac{-6}{\log_{10}\rho} \tag{4.10}$$

Thus the estimated ratio of the number of steps required for convergence for the GJ and P_c-preconditioned iterations is given by:

$$\frac{\log_{10}\rho_{P_c}}{\log_{10}\rho_{GJ}} = 2.38 \tag{4.11}$$

The actual ratio obtained is $192/105 = 1.83$. Based on the asymptotic convergence rates estimated, there is a strong reason to assume that the significant reductions in iteration count shown in Figure 4.2(b) are applicable to problem sizes beyond those presented in this thesis.

4.4 Conclusions

This chapter compares the performance of the Generalized Jacobi (GJ) preconditioner and the block constrained preconditioner (P_c) for solving large linear systems produced by finite element discretization of Biot's consolidation equations on the commonly available PC platform. The GJ preconditioner is very cheap to construct and apply in each iteration because it is a diagonal matrix. The P_c preconditioner uses the same 2×2 block structure of the coefficient matrix but its (1, 1) block is replaced by a diagonal approximation. Due to its non-diagonal nature, it is obviously more costly to construct and apply in each iteration. However, it is able to out-perform GJ in total runtime primarily because of significant reductions in iteration count. Note that GJ is already very efficient from an iteration count point of view. Numerical studies indicate that iteration count for GJ decreases from about 3% of the problem dimension for the smaller $12 \times 12 \times 12$ footing problem to only 1% of the problem dimension for the $24 \times 24 \times 24$ footing problem. The P_c preconditioner reduces these iteration counts by a further 50% and 35% for the small and large footing problem, respectively. In other words, iteration count for P_c only constitutes 1.3% and 0.3% of the problem dimension for the small and large footing problem, respectively. Based on the asymptotic convergence rates estimated from eigenvalues of both preconditioned matrices, there is a strong reason to assume that these significant reductions in iteration count are applicable to problem sizes beyond those presented in this thesis. In addition, the iteration count for P_c is almost the same for the two material types studied, indicating better scaling with respect to the effective Young's modulus and hydraulic permeability. The key disadvantage to the P_c preconditioner is the additional RAM required for implementation. This chapter presents the application of the Compressed Sparse Column (CSC) and Compressed Sparse Row (CSR) formats for efficient storage of global sparse matrices appearing in the construction of P_c and pseudo-codes for sparse matrix-vector multiplications and computation of the sparse Schur complement. Using these sparse formats, numerical results show that overhead costs (construction of Schur complement and sparse Cholesky factorization), time per iteration (triangular solves in preconditioning step), and RAM usage only grow at a power of 1.27, 1.11, and 1.08, respectively with DOFs over the range from about 24,000 to 180,000. A crude extrapolation to problem dimensions with one order of magnitude larger DOFs indicates that P_c is still practical and preferable over GJ.

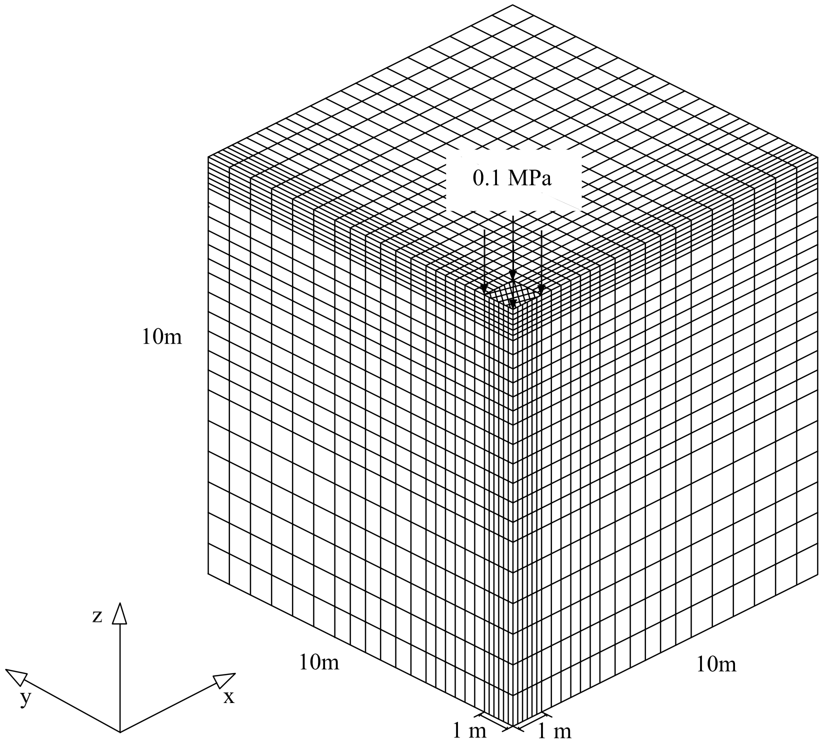


Figure 4.1: 24 × 24 × 24 finite element mesh of a quadrant symmetric shallow foundation problem.

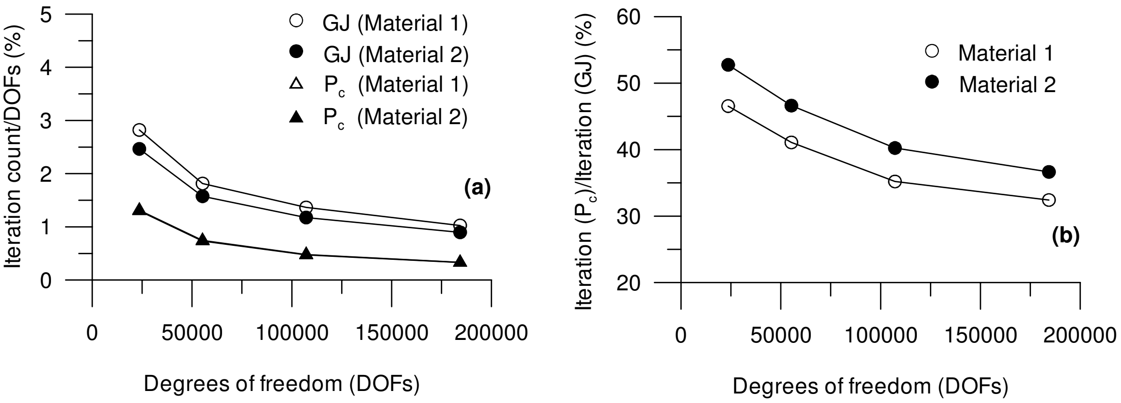


Figure 4.2: (a) Iteration count as a percentage of DOFs, and (b) Comparison of iteration count between GJ and P_c "Material 1" and"Material 2" refers to soft clays and dense sands, respectively).

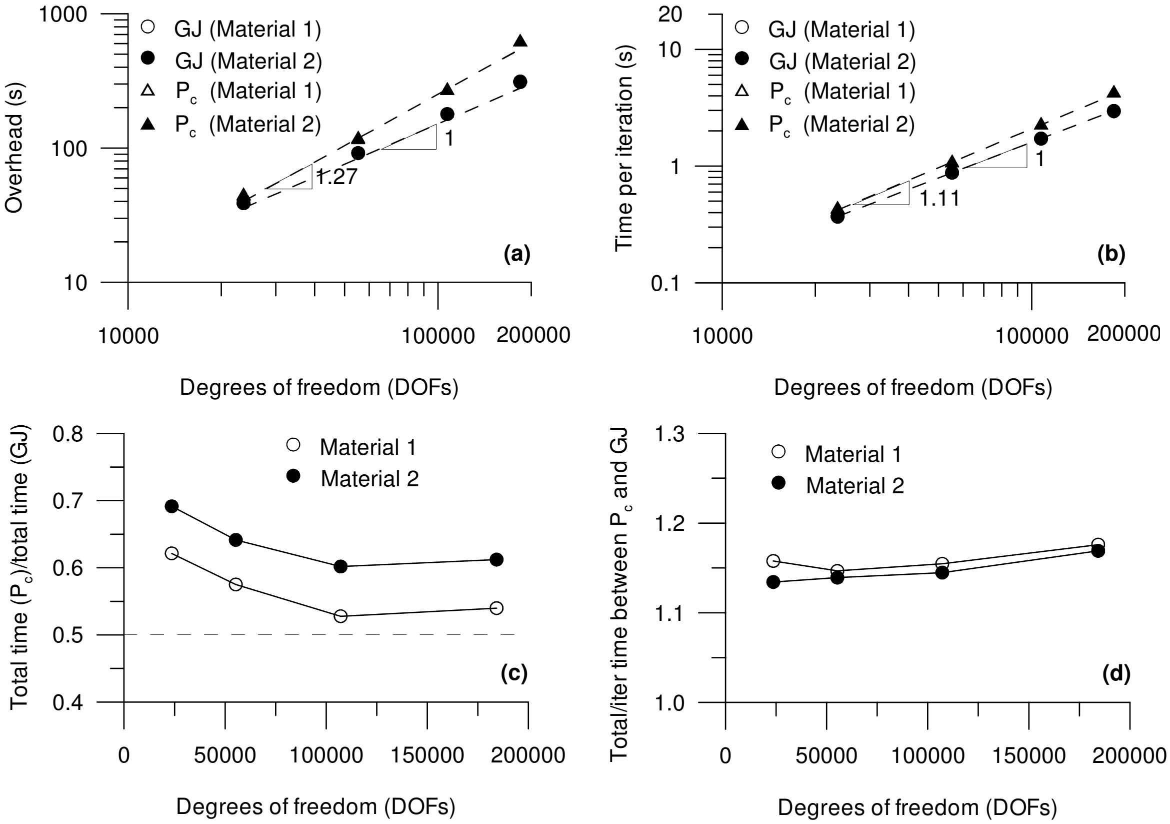


Figure 4.3: (a) Rate of increase in overhead with DOFs, (b) Rate of increase in time/iteration with DOFs, (c) Total runtime ratio between P_c and GJ, and (d) Total/iteration time ratio between P_c and GJ ("Material 1" and "Material 2" refers to soft clays and dense sands, respectively).

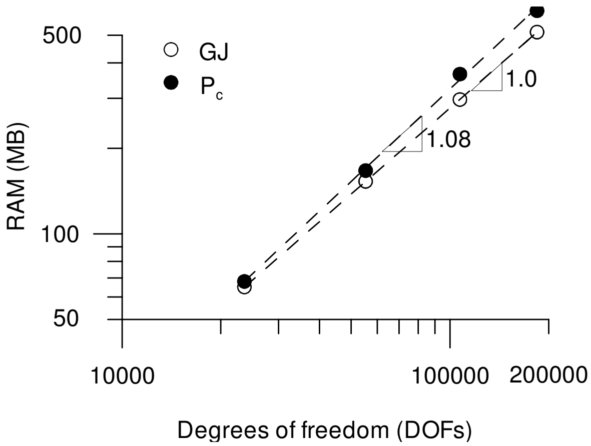


Figure 4.4: Rate of increase in RAM usage during iteration with DOFs.

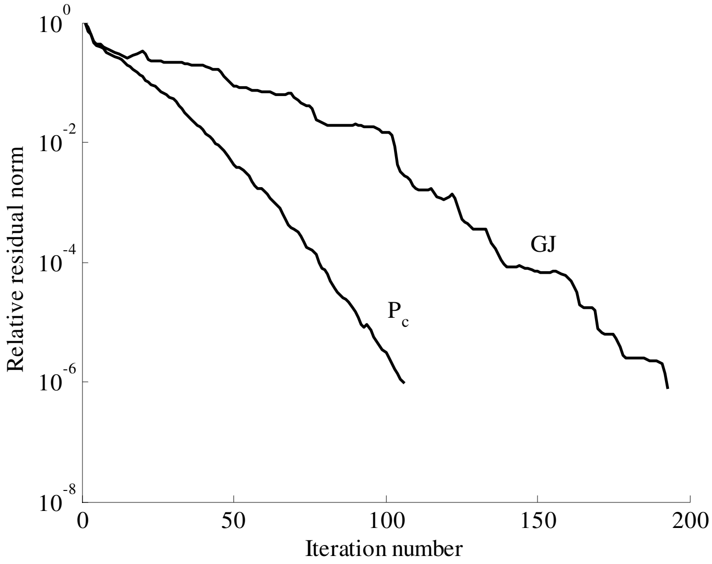


Figure 4.5: Convergence history of relative residual norms for SQMR solution of $5 \times 5 \times 5$ meshed footing problem

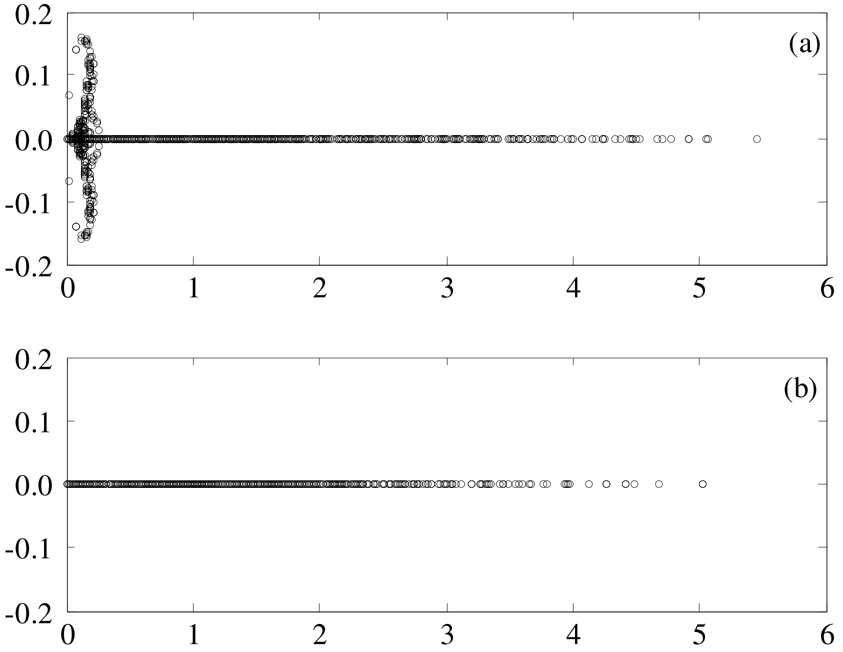


Figure 4.6: Eigenvalue distribution of: (a) GJ-precondtioned matrix and (b) P_c-precondtioned matrix.

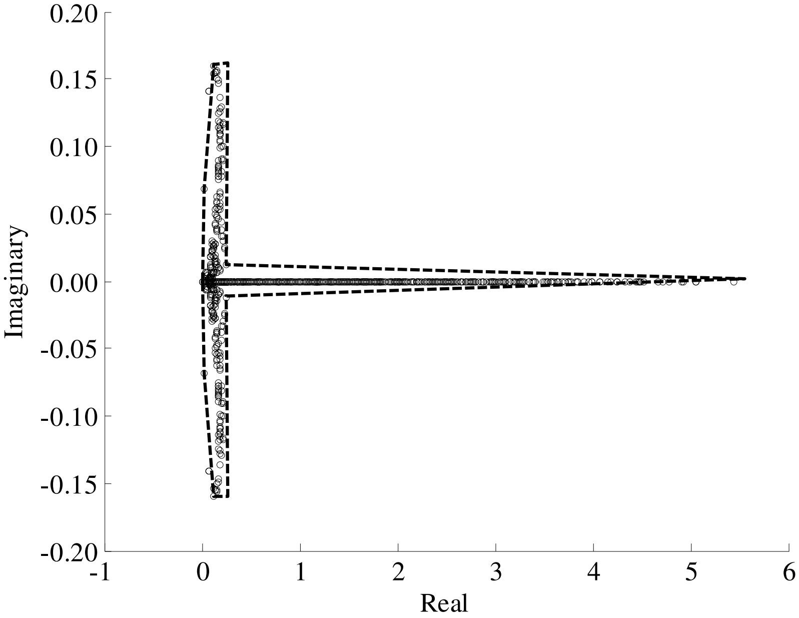


Figure 4.7: A polygonal region that approximately contains the eigenvalues of the GJ-precondtioned matrix.

Table 4.1: 3-D finite element meshes

	Mesh size				
	$5 \times 5 \times 5$	$12 \times 12 \times 12$	$16 \times 16 \times 16$	$20 \times 20 \times 20$	$24 \times 24 \times 24$
Number of elements(ne)	125	1728	4096	8000	13824
Number of nodes	756	8281	18785	35721	60625
$DOFs$					
Pore pressure(n)	180	2028	4624	8820	15000
Displacement(m)	1640	21576	50656	98360	169296
Total ($N = m + n$)	1820	23604	55280	107180	184296
n/m (%)	11.0	9.4	9.1	9.0	8.9
No. non-zeros (nnz)					
$nnz(B)$	22786	375140	907608	1812664	3174027
$nnz(B)/(nm)$ (%)	7.72	0.86	0.39	0.21	0.125
$nnz(C)$	3328	46546	110446	215818	373030
$nnz(C)/(n^2)$ (%)	10.27	1.13	0.52	0.28	0.17
$nnz(\hat{S})$	10944	187974	461834	921294	1614354
$nnz(\hat{S})/(n^2)$ (%)	33.78	4.57	2.16	1.18	0.72

Table 4.2: Performance of the GJ preconditioner over different mesh sizes and soil properties

Mesh size	$12 \times 12 \times 12$	$16 \times 16 \times 16$	$20 \times 20 \times 20$	$24 \times 24 \times 24$
RAM (MB)	65	153	297	513
Material 1: $E' = 1$ MPa, $k = 10^{-9}$ m/s (typical of soft clays)				
Iteration count	666	1003	1463	1891
Overhead (s)	38.8	91.6	178.9	312.9
Iteration time (s)	247.2	879.4	2519.7	5620.5
Total runtime (s)	288.5	976.8	2709.8	5952.8
Total/iteration time	1.16	1.10	1.07	1.06
Material 2: $E' = 100$ MPa, $k = 10^{-6}$ m/s (typical of dense sands)				
Iteration count	582	871	1260	1654
Overhead (s)	38.8	91.8	179.0	309.7
Iteration time (s)	214.2	764.9	2161.4	4877.8
Total runtime (s)	255.5	862.6	2351.7	5206.9
Total/iteration time	1.18	1.12	1.08	1.06

Table 4.3: Performance of the P_c preconditioner over different mesh sizes and soil properties

Mesh size	$12 \times 12 \times 12$	$16 \times 16 \times 16$	$20 \times 20 \times 20$	$24 \times 24 \times 24$
RAM (MB)	68	167	365	610
Material 1: $E' = 1$ MPa, $k = 10^{-9}$ m/s (typical of soft clays)				
Iteration count	310	412	515	613
Overhead (s)	44.2	115.1	267.5	614.9
Iteration time (s)	132.7	441.1	1151.9	2580.3
Total runtime (s)	179.3	561.8	1430.3	3213.9
Total/iteration time	1.33	1.26	1.23	1.24
Material 2: $E' = 100$ MPa, $k = 10^{-6}$ m/s (typical of dense sands)				
Iteration count	307	406	507	606
Overhead (s)	43.7	117.0	268.3	614.3
Iteration time (s)	130.6	430.7	1136.4	2554.6
Total runtime (s)	176.7	553.3	1415.5	3187.8
Total/iteration time	1.33	1.27	1.24	1.25

CHAPTER 5

A MODIFIED SSOR PRECONDITIONER

5.1 Introduction

In Chapter 4, a block constrained preconditioner P_c proposed by Toh *et al.* (2004) was investigated, and some crucial performances between P_c and GJ have been compared. Although P_c is significantly more complicated than P_{GJ}, each preconditioning step $P_c^{-1}[u; v]$ can be computed efficiently by solving 2 triangular linear systems of equations whose coefficient matrices involve the sparse Cholesky factor of the Schur complement matrix $\widehat{S}$. The detailed pseudocode for constructing $\widehat{S}$ and the implementation procedure for sparse Cholesky factorization have been given by Toh *et al.* (2004) as well as in Chapter 4. It has been demonstrated that P_c preconditioned SQMR method can result in faster convergence rate than the GJ preconditioned counterpart. On some fairly large linear systems arising from FE discretization of Biot's equations, the saving in CPU time can be up to 40%.

However, in pragmatic geotechnical problems, homogeneous soil is rarely encountered and more problems involve soil-structure interaction or heterogeneous soil profiles with significantly different soil properties. In these situations, the diagonal GJ preconditioner and P_c preconditioner cannot lead to satisfactory convergence rates of SQMR method. This observation can be explained that the diagonal form of GJ preconditioner and the cheap diagonal choice of (1, 1) block for P_c preconditioner cannot provide good approximation for the soil stiffness matrix, K. In this chapter, a variant of SSOR preconditioner is proposed, and this preconditioner can address this problem very well.

Standard SSOR preconditioner has been used by Mroueh and Shahrour (1999) to solve soil-structure interaction problems and they recommended left SSOR preconditioning. However, nonsymmetric iterative methods such as Bi-CGSTAB and QMR-CGSTAB were used for elastic-perfectly plastic problems with associated flow rule, even though SQMR method would be more efficient (in terms of CPU time) based on our experience (e.g. Toh *et al.*, 2005). Therefore, not only the preconditioning methods but also the iterative methods themselves have not been exploited to the greatest efficiency. For linear systems arising from pore-water-soil-structure interaction problems, the behavior of the standard SSOR preconditioner may be seriously affected by small negative diagonal elements corresponding to the pore pressure DOFs and as a result, the convergence of the standard SSOR preconditioned iterative method may be exceedingly slow. In this chapter, based on an exact factorization form of the coefficient matrix A, a new modified block SSOR preconditioner is proposed. The modified block SSOR preconditioner is expensive to construct. However, it serves as a useful theoretical basis to develop a simpler pointwise variant (named MSSOR from hereon) that can exploit the so-called Eisenstat trick (e.g. Eisenstat, 1981; Chan and van der Vorst, 1994; Freund and Nachtigal, 1995; Saad, 1996; Dongarra *et al.*, 1998; van der Vorst, 2003) for computational efficiency. It should be noted that the modified SSOR preconditioner can readily be extended to nonsymmetric 2×2 block systems.

To benchmark the proposed modified SSOR preconditioner, it is necessary to briefly describe the GJ and P_c preconditioners for clarity although the forms of GJ and P_c has been introduced in previous chapters.

5.1.1 The GJ Preconditioner

Recall that the GJ preconditioner developed by Phoon *et al.* (2002) has the form:

$$P_{GJ} = \begin{bmatrix} diag(K) & 0 \\ 0 & \alpha diag(\widehat{S}) \end{bmatrix} \tag{5.1}$$

where $\widehat{S} = C + B^T diag(K)^{-1} B$ is an approximate Schur complement matrix; α is a real scalar, and it is recommended by Phoon *et al.* (2002) to choose $\alpha \leq -1$ for a reasonable convergence rate.

5.1.2 The P_c Preconditioner

Recall that the block constrained preconditioner P_c was derived from A by replacing K by $diag(K)$ as (see Toh *et al.*, 2004):

$$P_c = \begin{bmatrix} diag(K) & B \\ B^T & -C \end{bmatrix} \tag{5.2}$$

It should be noted that in 3-D consolidation problems, the dimension of block K is significantly larger than that of block C, that is, $m \gg n$. The cheap diagonal approximation for (1, 1) block inevitably neglects the influence of varied soil properties embodied in K matrix.

5.2 Modified SSOR preconditioner

The SSOR iteration is a symmetric version of the well-known SOR iteration. When the SSOR method is used as a preconditioning technique, it is in fact equivalent to a single iteration of the standard SSOR method with zero initial guess (e.g. Axelsson, 1994; Mroueh and Shahrour, 1999). In addition, the SSOR preconditioner, like the Jacobi preconditioner, can be constructed from the coefficient matrix without any extra work.

For a symmetric matrix A with the decomposition, $A = L + D + L^T$, where L and D are the strictly lower triangular and diagonal parts of A (pointwise factorization), or the corresponding block parts if A is a block matrix (block factorization), the standard SSOR preconditioner has the following factorized form (e.g. Eisenstat, 1981; Chan and van der Vorst, 1994; Freund and Nachtigal, 1995)

$$P_{SSOR} = (L + D)D^{-1}(L^T + D) \tag{5.3}$$

It is obvious that the factorization form of Eq. (5.3) follows the stencils for LDU, so SSOR can also be regarded as an incomplete factorization preconditioner. For symmetric indefinite linear system with zero diagonal subblock, direct application of Eq. (5.3) is impossible. To avoid this difficulty, Freund *et al.* (1995, 1997) proposed applying permutation $\mathcal{P}$ to the original matrix A, that is, $\bar{A} = \mathcal{P}A\mathcal{P}^T = \bar{L} + \bar{D} + \bar{L}^T$ where $\bar{L}$

and $\bar{D}$ are resultant strictly lower triangular and block diagonal parts of $\bar{A}$, respectively. The objective of the permutation $\mathcal{P}$ is to obtain a trivially invertible and in some sense "as large as possible" diagonal $\bar{D}$ with a series of 1 × 1 and 2 × 2 blocks. Therefore, the corresponding modified SSOR preconditioner to combine with SQMR solver can be obtained from the permuted matrix as

$$\bar{P}_{SSOR} = \mathcal{P}^T(\bar{L} + \bar{D})\bar{D}^{-1}(\bar{L}^T + \bar{D})\mathcal{P} \tag{5.4}$$

However, it may be difficult to find a permutation that totally avoids zero or small diagonal elements in the reordered matrix. In addition, incomplete factorization methods such as the SSOR preconditioner are sensitive to permutations, and thus may incur a larger number of iterations than the same incomplete factorization applied to the original matrix (e.g. Duff and Meurant, 1989; Chow and Saad, 1997; Eijkhout, 1999).

5.2.1 Derivation of a New Modified Block SSOR Preconditioner

For the 2 × 2 block symmetric indefinite linear system (3.44) arising from Biot's consolidation problem, the standard pointwise SSOR preconditioner (5.3) has small negative diagonal elements corresponding to pore pressure DOFs, and they may cause the iterative solver to stagnate, diverge or even breakdown. The permutation approach has limitations as noted above. This motivated us to consider a different approach involving the modification of D without the need for choosing a permutation. First, observe that the 2 × 2 block matrix of Eq. (3.44) can be factorized as

$$\begin{bmatrix} K & B \\ B^T & -C \end{bmatrix} = \begin{bmatrix} K & 0 \\ B^T & -S \end{bmatrix} \begin{bmatrix} K & 0 \\ 0 & -S \end{bmatrix}^{-1} \begin{bmatrix} K & B \\ 0 & -S \end{bmatrix} \tag{5.5}$$

Here $S = C + B^T K^{-1} B$ is the exact Schur complement matrix. The factorization shown in Eq. (5.5) has been mentioned as block LDU factorization by Chan and van der Vorst (1994), or generalized SSOR form by Chow and Heroux (1998). In this chapter, a new modified block SSOR (MBSSOR) preconditioner can be derived from Eq. (5.5), that is,

$$P_{MBSSOR} = \begin{bmatrix} \widehat{K} & 0 \\ B^T & -\widehat{S} \end{bmatrix} \begin{bmatrix} \widehat{K} & 0 \\ 0 & -\widehat{S} \end{bmatrix}^{-1} \begin{bmatrix} \widehat{K} & B \\ 0 & -\widehat{S} \end{bmatrix} \tag{5.6}$$

Here, $\widehat{K}$ and $\widehat{S}$ are the approximations of K and S, respectively, and the approximate matrix $\widehat{K}$ may differ from the approximation of K appearing in $\widehat{S}$. Many variants can be obtained from this modified block SSOR preconditioner. It is interesting to note that the factorized form in Eq. (5.6) contains all three block preconditioners discussed by Toh *et al.* (2004), and the P_c preconditioner in Eq. (5.2) is the exact product form of Eq. (5.6) with $\widehat{K} = diag(K)$ and $\widehat{S} = C + B^T \widehat{K}^{-1} B$. The eigenvalue analysis of P_{MBSSOR} preconditioned matrix is similar to that for the P_c preconditioner given in Toh *et al.* (2004), and thus shall not be repeated here.

5.2.2 A New Modified Pointwise SSOR Preconditioner

A complicated near-exact preconditioner is not always the most efficient, because the cost of applying the preconditioner at each iteration could obviate the reduction in iteration count. The pragmatic goal is to reduce total runtime and simpler variants could potentially achieve this even though iteration counts might be higher. With this pragmatic goal in mind, a parameterized pointwise variant of P_{MBSSOR} is proposed as

$$P_{MSSOR} = \left(L + \frac{\widehat{D}}{\omega} \right) \left(\frac{\widehat{D}}{\omega} \right)^{-1} \left(L^T + \frac{\widehat{D}}{\omega} \right) = (L + \widetilde{D})(\widetilde{D})^{-1}(L^T + \widetilde{D}) \qquad (5.7)$$

where L is the strictly lower triangular part of A, $\widehat{D}$ is a diagonal matrix that is to be chosen, $\widetilde{D} = \widehat{D}/\omega$, and $\omega \in [1, 2]$ is a relaxation parameter. The choice of $\widehat{D}$ is crucial. Pommerell and Fichtner (1991) and Pommerell (1992) proposed a *D-ILU* preconditioner, for which the diagonal matrix $\widehat{D} = \{\tilde{a}_{ii}\}$ is chosen from the ILU(0) factorization procedure as follows:

Algorithm 5.1 *Algorithm of constructing diagonal $\widehat{D}$ of D-ILU Preconditioner, $\mathcal{S}$ is the non-zero sparsity structure of coefficient matrix A (e.g. Barrett et al., 1994).*

```
for $i = 1$ to $N$ do
    Set $\tilde{a}_{ii} = a_{ii}$
end for
for $i = 1$ to $N$ do
    for $j = i + 1$ to $N$ do
        if $(i, j) \in \mathcal{S}$ and $(j, i) \in \mathcal{S}$
            Set $\tilde{a}_{jj} = \tilde{a}_{jj} - a_{ji}\tilde{a}_{ii}^{-1}a_{ij}$
        end if
    end for
end for
```

However, our numerical experiences indicate that SQMR is unable to converge when the linear systems stemming from Biot's consolidation equations are preconditioned by *D-ILU*.

The GJ preconditioner is a more natural candidate for $\widehat{D}$. Phoon *et al.* (2002) has already demonstrated that P_{GJ} is a good approximation to Murphy's preconditioner (2000), which is the un-inverted block diagonal matrix in Eq. (5.6). Hence, Eq. (5.7) would be studied based on choosing $\widehat{D} = P_{GJ}$ from hereon. Another important parameter in Eq. (5.7) is the relaxation parameter ω. The optimal choice of ω is usually expensive to estimate, but practical experiences suggest that choosing ω slightly greater than 1 usually result in faster convergence than the choice $\omega = 1$ (where SSOR preconditioner reduces to the symmetric Gauss-Seidel preconditioner). It should be mentioned that picking ω too far away from 1 can sometimes result in significant deterioration in the convergence rate. Thus, it is advisable not to pick ω larger than 1.5 unless the optimal choice of ω is known. Rewriting the expression for P_{MSSOR} in Eq. (5.7) as $P_{MSSOR} = (L+\widetilde{D})(\widetilde{D})^{-1}(L^T+\widetilde{D}) = A+L\widetilde{D}^{-1}+\widetilde{D}-D$, it is quite obvious that P_{MSSOR} can better approximate A when the error matrix $E = L\widetilde{D}^{-1}L^T + \widetilde{D} - D$ is small.

Note that for the matrix A in Eq. (3.44), it is easy to see that the (1, 1) block of P_{MSSOR} is equal to

$$(L_K + D_K/\omega)(D_K/\omega)^{-1}(L_K + D_K/\omega)^T$$

where L_K and D_K are the strictly lower and diagonal parts of K. Thus the (1,1) block K in A is approximated by its SSOR matrix.

5.2.3 Combining With Eisenstat Trick

Eisenstat (1981) exploited the fact that some preconditioners such as generalized SSOR (1972), ICCG(0) (1977) and MICCG(0) (1978) contain the same off-diagonal parts of the original coefficient matrix to combine the preconditioning and matrix-vector multiplication step into one single efficient step. This trick is obviously applicable to any preconditioner of the form shown in Eq. (5.7).

The pseudo-code for the SQMR algorithm coupled with the MSSOR preconditioner is provided as follows (e.g. Freund and Nachtigal, 1995; Freund and Jarre, 1996):

Algorithm 5.2 *MSSOR Preconditioned SQMR Method*

```
Start: choose an initial guess $x_0$, then set $z_0 = 0$,
$s_0 = (L + \widetilde{D})^{-1}(b - Ax_0)$, $v_0 = s_0$, $w_0 = \widetilde{D}v_0$,
$\tau_0 = s_0^T s_0$, and $\rho_0 = s_0^T w_0$
set $d_0 = 0 \in \mathbb{R}^N$, and $\vartheta_0 = 0 \in \mathbb{R}$.
for $k = 1$ to max_it do
    Compute
    $t_{k-1} = \widetilde{A}v_{k-1}$ (Procedure PMatvec).
    $\sigma_{k-1} = w_{k-1}^T t_{k-1}$, if $\sigma_{k-1} = 0$, then stop.
    $\alpha_{k-1} = \frac{\rho_{k-1}}{\sigma_{k-1}}$ and $s_k = s_{k-1} - \alpha_{k-1}t_{k-1}$
    Compute
    $\vartheta_k = \frac{s_k^T s_j}{\tau_{k-1}}$, $c_k = \frac{1}{1 + \vartheta_k}$, $\tau_k = \tau_{k-1}\vartheta_k c_k$,
    $d_k = c_k\vartheta_{k-1}d_{k-1} + c_k\alpha_{k-1}v_{k-1}$.
    Set $z_k = z_{k-1} + d_k$.
    Check convergence
    every fifth step, compute $r_k = (L + \widetilde{D})s_k$.
    if converged, then set $x_k = x_0 + (L^T + \widetilde{D})^{-1}\widetilde{D}z_k$, then stop.
    Compute
    $\rho_k = s_k^T \widetilde{D}s_k$, $\beta_k = \frac{\rho_k}{\rho_{k-1}}$,
    $v_k = s_k + \beta_k v_{k-1}$ and $w_k = \widetilde{D}v_k$.
end for
```

In Algorithm 5.2, the Eisenstat trick is applied to the left-right preconditioned matrix of the form:

$$\widetilde{A} = (L + \widetilde{D})^{-1}A(L^T + \widetilde{D})^{-1}\widetilde{D} \tag{5.8}$$

Following Chan's implementation (e.g. Eisenstat, 1981; Chan and van der Vorst, 1994), the preconditioned matrix can be written as

$$\widetilde{A} = (L + \widetilde{D})^{-1}[(L + \widetilde{D}) + (D - 2\widetilde{D}) + (L^T + \widetilde{D})](L^T + \widetilde{D})^{-1}\widetilde{D} \tag{5.9}$$

Thus, given a vector v_{k-1}, the product $t_{k-1} = \widetilde{A}v_{k-1}$ in the SQMR algorithm can be computed from the following procedure:

<u>**Procedure PMatvec**</u> :

$$\begin{array}{l}
(1)\ f = (L^T + \widetilde{D})^{-1}w_{k-1} \quad \text{where} \quad w_{k-1} = \widetilde{D}v_{k-1} \\
(2)\ g = (D - 2\widetilde{D})^{-1}f + w_{k-1} \\
(3)\ h = (L + \widetilde{D})^{-1}g \\
(4)\ t_{k-1} = f + h
\end{array} \tag{5.10}$$

When applying the above PMatvec procedure to the P_c preconditioned matrix , we get the resultant vector,

$$\begin{aligned} t_{k-1} &= \begin{bmatrix} t_{k-1}^{(1)} \\ t_{k-1}^{(2)} \end{bmatrix} = \widetilde{A} \begin{bmatrix} v_{k-1}^{(1)} \\ v_{k-1}^{(2)} \end{bmatrix} \\ &= \begin{bmatrix} \widehat{K}^{-1}Kv_{k-1}^{(1)} - \widehat{K}^{-1}K\widehat{K}^{-1}Bv_{k-1}^{(2)} + \widehat{K}^{-1}Bv_{k-1}^{(2)} \\ \widehat{S}^{-1}B^T(\widehat{K}^{-1}Kv_{k-1}^{(1)} - \widehat{K}^{-1}K\widehat{K}^{-1}Bv_{k-1}^{(2)} + \widehat{K}^{-1}Bv_{k-1}^{(2)} - v_{k-1}^{(1)}) + v_{k-1}^{(2)} \end{bmatrix} \end{aligned} \tag{5.11}$$

This computation can be carried out by the following steps:

$$\begin{cases} \textbf{Compute} & u = \widehat{K}^{-1}Bv_{k-1}^{(2)} \\ \textbf{Compute} & s = \widehat{K}^{-1}K(v_{k-1}^{(1)} - u) + u \\ \textbf{Compute and set} & t_{k-1} = [s; \widehat{S}^{-1}B^T(s - v_{k-1}^{(1)}) + v_{k-1}^{(2)}] \end{cases} \tag{5.12}$$

Thus the computational cost of each P_c preconditioned SQMR step combined with the Eisenstat trick can broadly be summarized as: 1 K; 1 B; 1 B^T; 2 $\widehat{K}$; 2 $\widehat{S}$; 1 $\widehat{S}^{-1}$; 2 $\widehat{K}^{-1}$.

On the other hand, for P_c preconditioned SQMR method without applying the Eisenstat trick, the matrix-vector product and preconditioning steps at each iteration are separately carried out in terms of Eqs. (4.3) and (4.6) (The difference is that no element-level implementation is involved any more). Thus the main computational cost for each P_c preconditioned SQMR iteration without using the Eisenstat trick can be summarized as: 1 K; 1 C; 2 B; 2 B^T; 1 $\widehat{S}^{-1}$; 2 $\widehat{K}^{-1}$. Since the number of nonzero elements of B is only about 10% that of K, it is clear that the implementation of P_c combined with the Eisenstat trick is only marginally cheaper than the efficient implementation described in Chapter 4 for P_c without using the Eisenstat trick.

In contrast to the modified block SSOR-type preconditioners such as factorized P_c, the pointwise MSSOR preconditioner proposed in Eq. (5.7) is very promising in that it heavily exploits the Eisenstat trick. Each step of the MSSOR-preconditioned iterative method is only marginally more expensive than that of the original matrix-vector product because the strictly off-diagonal parts of P_{MSSOR} and A canceled one another and only two triangular solves are involved at each iteration. Eq. (5.8) or Eq. (5.9) demonstrated the Eisenstat trick in left-right form; in fact, it is also applicable in the left or right form for nonsymmetric solvers (e.g. Pommerell and Fichtner, 1991; Pommerell, 1992).

5.2.4 Other Implementation Issues of GJ, P_c and P_{MSSOR}

In practical finite element programming, the displacement unknowns and pore pressure unknowns can be arranged in different order. Gambolati *et al.* (2001) have studied the effect of three different nodal orderings on ILU-type preconditioned Bi-CGSTAB method, and these nodal orderings have been presented in Section 2.2.4. In this study, GJ and MSSOR preconditioners are applied to linear systems based on $iord1$, while P_c is applied to linear systems based on $iord2$. Numerical results discussed in the next section support the above choices.

Because of the symmetry in A, only the upper triangular part of A needs to be stored. In our implementation of GJ or MSSOR-preconditioned SQMR methods, the upper triangular part of A is stored in the CSC (Compressed Sparse Column) format (`icsc`, `jcsc`, `csca`) (refer to Barrett *et al.*, 1994; Saad, 1996, 2003, for sparse storage formats). Clearly, the CSC storage of the upper triangular part of A is also the CSR (Compressed Sparse Row) storage of the lower triangular part for A. Both the "forward solve step" and "backward solve step" at each iteration of the SQMR method can be executed quite rapidly from the CSC storage. The pseudo-code of the forward and backward solves are given as follows:

Algorithm 5.3 *Forward substitution ($1 \rightarrow N$) with CSC storage (*`icsc`*,* `jcsc`*,* `csca`*) of upper triangular part of A. The modified diagonal is stored in a vector $\{\tilde{a}_{jj}\}_{1\le j\le N}$, Compute $x = (L+\widetilde{D})^{-1}y$ as follows:*

```
$x(1) = y(1)/\tilde{a}_{11}$
for $j = 2$ to $N$ do
    $k1 = jcsc(j)$; $k2 = jcsc(j+1) - 1$;
    for $i = k1$ to $k2 - 1$ do
        $y(j) = y(j) - csca(i) \times x(icsc(i))$
    end for
    $x(j) = y(j)/\tilde{a}_{jj}$
end for
```

Algorithm 5.4 *Backward substitution ($N \rightarrow 1$) with CSC storage (*`icsc`*,* `jcsc`*,* `csca`*) of upper triangular part of A. The modified diagonal is stored in a vector $\{\tilde{a}_{jj}\}_{1\le j\le N}$, Compute $x = (L^T+\widetilde{D})^{-1}y$ as follows:*

```
for $j = N$ to $2$ do
    $x(j) = y(j)/\tilde{a}_{jj}$
    $k1 = jcsc(j); k2 = jcsc(j+1) - 2;$
    for $i = k1$ to $k2$ do
        $y(icsc(i)) = y(icsc(i)) - csca(i) \times x(j)$
    end for
end for
$x(1) = y(1)/\tilde{a}_{11}$
```

Other than the CSC storage for the upper triangular part of A, only a few additional vectors are required to store the original and modified diagonal elements in MSSOR. In the case of P_c, it is natural to store K, B and C separately in order to obtain the approximate Schur complement matrix and its sparse Cholesky factor. The detailed implementation is described in Chapter 4.

5.3 Numerical Experiments

5.3.1 Convergence Criteria

An iterative solver typically produces increasingly accurate solutions with iteration count, and one can terminate the solver when the approximate solution is deemed sufficiently accurate. A standard measure of accuracy is based on the relative residual norm. Suppose x_i is the approximate solution at the i-th iterative step, then $r_i = b - Ax_i$ is the corresponding residual vector. In Algorithm 5.5, the true residual with respect to the original linear system can be obtained from the preconditioned residual.

Algorithm 5.5 *Compute $r = (L + \widetilde{D})s$ given the CSC storage (*`icsc`*,* `jcsc`*,* `csca`*) of upper triangular part of A, and the modified diagonal is stored in a vector $\{\tilde{a}_{jj}\}_{1 \leq N}$.*

```
$r = zeros(N, 1)$
$r(1) = r(1) + \tilde{a}_{11} \times s(1)$
for $j = 2$ to $N$ do
    $k1 = jcsc(j);\ k2 = jcsc(j+1) - 1;$
    for $k = k1$ to $k2 - 1$ do
        $r(j) = r(j) + csca(k) \times s(icsc(k))$
    end for
    $r(j) = r(j) + \tilde{a}_{jj} \times s(j)$
end for
```

For the purpose of comparing with other preconditioned SQMR methods, the true relative residual for the SQMR method preconditioned by MSSOR or P_c combined with the Eisenstat trick is returned at every fifth iteration. Note that the true relative residual (modulo rounding errors) is easily computed from the equation $b - Ax_k = P_L(\tilde{b} - \widetilde{A}\tilde{x}_k) = (L + \widetilde{D})(\tilde{b} - \widetilde{A}\tilde{x}_k)$.

Given an initial guess x_0 (usually zero initial guess), an accuracy tolerance *stop_tol*, and the maximum number *max_it* of iterative steps allowed, the iterative process will be stopped if the relative residual based on true residuals given by Eq. (3.64) is satisfied. In this study, the initial guess x_0 is taken to be the zero vector, and $stop\_tol = 10^{-6}$, $max\_it = 5000$. More details on various stopping criteria can be found in Barrett *et al.* (1994) van der Vorst (2003).

5.3.2 Problem Descriptions

Figure 5.1 shows a 7 × 7 × 7 finite element mesh for a flexible square footing resting on homogeneous soil subjected to a uniform vertical pressure of 0.1 MPa. The sparsity pattern of the global matrix A with natural ordering obtained from the 7 × 7 × 7 meshed footing problem is also shown in the figure. Figure 5.2, a 20 × 20 × 20 mesh for the same square footing problem resting on a layered soil is plotted. Symmetry consideration allows a quadrant of the footing to be analyzed. Mesh sizes ranging from 8 × 8 × 8 to 20 × 20 × 20 were studied. These meshes result in linear systems of equations with DOFs ranging from about 7,000 to 100,000, respectively. In our FE discretization of the time-dependent Biot's consolidation problems, the twenty-node hexahedral solid elements coupled with eight-node fluid elements were used. Therefore, each hexahedral element consists of 60 displacement degrees of freedom (8 corner nodes and 12 mid-side nodes with 3 spatial degrees of freedom per node) and 8 excess pore pressure degrees of freedom (8 corner nodes with 1 degree of freedom per node). Thus the total number of DOFs of displacements is much larger than that of excess pore pressure, the ratio is usually larger than ten. Details of these 3-D finite element meshes are provided in Table 5.1.

In summary, the footing problem resting on the following 3 soil profiles are studied:

(a) Soil profile 1: homogeneous soft clay, $E' = 1MPa$, $k = 10^{-9}m/s$;

(b) Soil profile 2: homogeneous dense sand, $E' = 100MPa$, $k = 10^{-5}m/s$;

(c) Soil profile 3: heterogeneous soil consisting of alternate soft clay and dense sand soil layers with parameters $E' = 1MPa$, $k = 10^{-9}m/s$ and $E' = 100MPa$, $k = 10^{-5}m/s$, respectively.

The other descriptions on the footing problem are given Chapter 3.

5.3.3 Choice of Parameters in GJ(MSSOR) and Eigenvalue Distributions of GJ(MSSOR) Preconditioned Matrices

Figure 5.3 shows the eigenvalue distributions of two GJ preconditioned matrices, for the $7 \times 7 \times 7$ meshed problem on soil profile 1, corresponding to the parameters $\alpha = -4$, and $\alpha = -20$, respectively. It is interesting to observe that the "imaginary" wing is considerably compressed when $|\alpha|$ is larger. Thus one would expect the GJ preconditioned matrix associated with a larger $|\alpha|$ to have a faster asymptotic convergence rate. Unfortunately, the minimum real part of the eigenvalues also decreases with $|\alpha|$ and the resultant effect is to slow down the asymptotic convergence rate. Therefore, to obtain the optimal asymptotic convergence rate, a balance has to be maintained between the above two opposing effects: reducing the span of the imaginary wing, and having eigenvalues closer to the origin. With the help of Schwarz-Christoffel mapping for polygonal region, the asymptotic convergence rates for both eigenvalue distributions are estimated to be 0.98 for $\alpha = -1$ and 0.95 for $\alpha = -20$. These estimated convergence rates indicate that it is beneficial to use GJ with $\alpha < -1$. For completeness, the eigenvalue distribution of the P_c preconditioned matrix is also shown in Figure 5.3, and the asymptotic convergence rate associated with the spectrum is 0.92. Interestingly, the eigenvalue distribution of the P_c preconditioned matrix (which can be proven to have only real eigenvalues) is almost the same as the real part of the eigenvalue distribution of the GJ preconditioned matrix with $\alpha = -1$. This could partially be attributed to the fact that both GJ and P_c preconditioners use the same diagonal approximation to the (1, 1) block K.

In Section 5.2.4, it has been mentioned that the performance of MSSOR preconditioner can be influenced by different ordering scheme. This is shown by the results in Table 5.2. In this table, the performance of MSSOR with $\omega = 1.0$ and $\alpha = -4$ is evaluated under three different ordering schemes ($iord1$, $iord2$ and $iord3$). Homogeneous and layered soil profiles are considered. The performance indicators are: (1) iteration count ($iters$), (2) overhead time (t_o) which covers formation of sparse coefficient matrix and construction of the preconditioner, (3) iteration time (t_i), and (4) total runtime for one time step (t_t). Regardless of the mesh size and soil profile, it can be seen that the natural ordering scheme always lead to less iteration count and less total runtime. Hence, only MSSOR for the natural ordering scheme in the numerical examples discussed below is implemented.

It is interesting to also investigate how the choice of α and ω would affect the asymptotic convergence rate of the MSSOR preconditioned matrix. It must be emphasized that the intention here is not to search for the optimal pair of parameters but to use the example problems to investigate the benefit of taking $\widehat{D}$ to be the GJ preconditioner with $|\alpha|$ different from 4 recommended by Toh (2003). For MSSOR($\omega = 1.0$) preconditioned SQMR solver, the effect of varying α over the range from -1 to -80 is shown in Table 5.3 for the $7 \times 7 \times 7$ and $16 \times 16 \times 16$ meshed problems over different soil conditions. It is clear that the optimal α value is problem-dependent. It appears that $|\alpha|$ should be larger for a larger mesh. The iteration count under $\alpha = -4$ is at most 1.5 times the iteration count under the optimal α in the problems studied.

Next, the effect of varying the relaxation parameter ω in the MSSOR preconditioner for $\alpha = -50$ is investigated. The results for ω ranging from 1.0 to 1.8 for the $7 \times 7 \times 7$ and $16 \times 16 \times 16$ meshed problems under different soil conditions are shown in Table 5.4. It is obvious that there exists optimal values of ω for different soil conditions, and appropriate choices of ω may lead to smaller iteration counts and shorter total runtimes. However, as mentioned previously, the optimal value of ω is expensive to determine and usually can only be obtained through numerical experiments. It seems that the performance of MSSOR preconditioned SQMR method is not very sensitive to ω compared with the effect of varying α. Based on the numerical results provided in Table 5.4, a reasonably good choice for ω is located in the interval [1.2, 1.4].

Table 5.4 also gives the performance of standard SSOR preconditioned SQMR method, it is clear that for soil profile 1 and soil profile 3, standard SSOR preconditioner may breakdown (more exactly near-breakdown). The breakdown is likely to be the consequence of unstable triangular solves produced by the irregular distribution of small negative entries in the leading diagonal under natural ordering. To avoid this breakdown, block ordering strategy can be used; moreover, block ordering can significantly improve the performance of standard SSOR preconditioner for soil profile 2, but the performance of block ordering for SSOR preconditioned SQMR method is still inferior to that achieved by using the MSSOR preconditioned version.

Figure 5.4 shows the eigenvalue distributions of three MSSOR preconditioned matrices corresponding to the pair of parameters ($\alpha = -4$, $\omega = 1.0$), ($\alpha = -20$, $\omega = 1.0$) and ($\alpha = -20$, $\omega = 1.3$), respectively. Again, these estimated convergence rates indicate that it is beneficial to use $\alpha < -1$. A larger $|\alpha|$ value has a compression effect on the imaginary part of the eigenvalues. By taking $\omega > 1.0$, the interval containing the real part of the eigenvalues may be enlarged, but the eigenvalues with nonzero imaginary parts are moved further way from the origin. Observe that the eigenvalue distributions of the MSSOR preconditioned matrices have more compressed real parts compared to the GJ preconditioned counterparts (Fig. 5.3b). The MSSOR preconditioner contracts the range of the real part of eigenvalues by adopting the SSOR approximation to the (1, 1) block. From the compactness of the eigenvalue distributions, it is quite clear that the MSSOR preconditioned matrices would have faster convergence rates than the GJ preconditioned ones.

5.3.4 Performance of MSSOR versus GJ and P_c

Table 5.5 and 5.6 give the numerical results of the SQMR method preconditioned by GJ, P_c and MSSOR in homogeneous soft clay and dense sand, respectively. Numerical experiments to compare the performance of these preconditioners in the layered soil profile are also performed, and the results are tabulated in Table 5.7. The P_c preconditioner is implemented with and without the Eisenstat trick. It is clear that the Eisenstat trick works in principle on block SSOR preconditioners, but has little practical impact on the runtime. In contrast, the simple pointwise variant (see Eq. (5.7)) is able to exploit the Eisenstat trick intensively in terms of runtime.

Figure 5.5(a) displays the convergence histories of SQMR solver preconditioned by GJ, P_c and MSSOR preconditioners, respectively for the $7 \times 7 \times 7$ meshed footing problem on homogeneous soil profiles. It can be seen that the MSSOR preconditioner leads to faster convergence rate than GJ and P_c. The convergence histories of these preconditioned methods for the layered soil profile problem are shown in Figure 5.5(b). The iteration counts are larger, but the MSSOR preconditioner still out-performs the others.

Figure 5.6 shows the iteration counts versus DOFs. The basic trend for all the 3 preconditioned SQMR methods is that the iteration counts increase sub-linearly, but each with a different growth rate with GJ preconditioned method being the fastest and MSSOR preconditioned method being the slowest. In fact, for more difficult layered soil problem, the iteration count for the MSSOR preconditioned method only increases 2.1 times when the DOFs increases 15 times from 7160 to 107180 (corresponding to the $8 \times 8 \times 8$ and $20 \times 20 \times 20$ problems). Figure 5.7 compares the cost of each iteration of the various preconditioned methods. MSSOR is only marginally more expensive than GJ. This result is very significant because the cost per iteration in GJ is almost minimal (it is just a simple diagonal preconditioner). It is obvious that the Eisenstat trick is very effective when exploited in the right way.

Figure 5.8 gives the same example provided by Smith and Griffiths (2004), in this example, consolidation analysis was carried out with $20 \times 20 \times 20$ uniform finite element mesh, and the ramp load varies from 0 KPa to 1 KPa over 10 time steps. The numerical results for displacements and excess pore pressures have been provided in Figure 5.9. It also can be seen that even for the "ideal"[1] consolidation problem, the SQMR method preconditioned by the proposed MSSOR preconditioner shows fast computing capacity.

[1] Here, "ideal" means the soil condition is homogeneous without varied layers

5.4 Conclusions

The main contribution in this work is to propose a cheap modified SSOR preconditioner (MSSOR) which is applicable across a wide class of soil profiles. The modification is carefully selected to exploit the Eisenstat trick intensively in the terms of runtime. Specifically, each MSSOR preconditioned SQMR step is only marginally more expensive than a single matrix-vector product step. This is close to the minimum achievable from simple Jacobi type preconditioners, where the preconditioner application step is almost cost free. For the diagonal matrix forming part of the MSSOR preconditioner, the GJ preconditioner is adopted to overcome the numerical difficulties encountered by the standard SSOR preconditioner for matrices having very small (or even zero) diagonal entries. These matrices occur in many geomechanics applications involving solution of large-scale symmetric indefinite linear systems arising from FE discretized Biot's consolidation equations. The eigenvalue distributions of the MSSOR preconditioned matrices are analyzed empirically and it can be concluded that the associated asymptotic convergence rates to be better than those of GJ or P_c preconditioned matrices. Numerical experiments based on several footing problems including the more difficult layered soil profile problem show that our MSSOR preconditioner not only perform (in terms of total runtime) far better than the standard SSOR preconditioner, but it is also much better than some of the effective preconditioners such as GJ and P_c published very recently. Moreover, with increasing problem size, the performance of our MSSOR preconditioner against GJ or P_c is expected to get better. Another major advantage is that with a better approximation to the (1,1) K block, iteration count increases more moderately from homogeneous to layered soil profiles. Nevertheless, it is acknowledged that improvements are possible and should be further studied because one should ideally construct a preconditioner that is minimally affected by the type of soil profile or mesh size. It is worth noting that the proposed MSSOR preconditioners can be generalized readily to nonsymmetric cases or problems with a zero (2, 2) block.

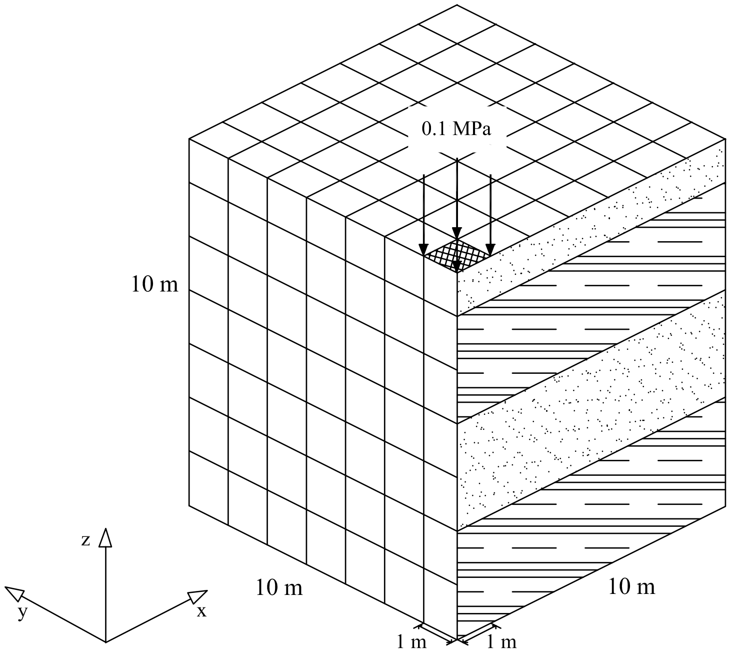


Figure 5.1: $7 \times 7 \times 7$ finite element mesh for simple footing problem.

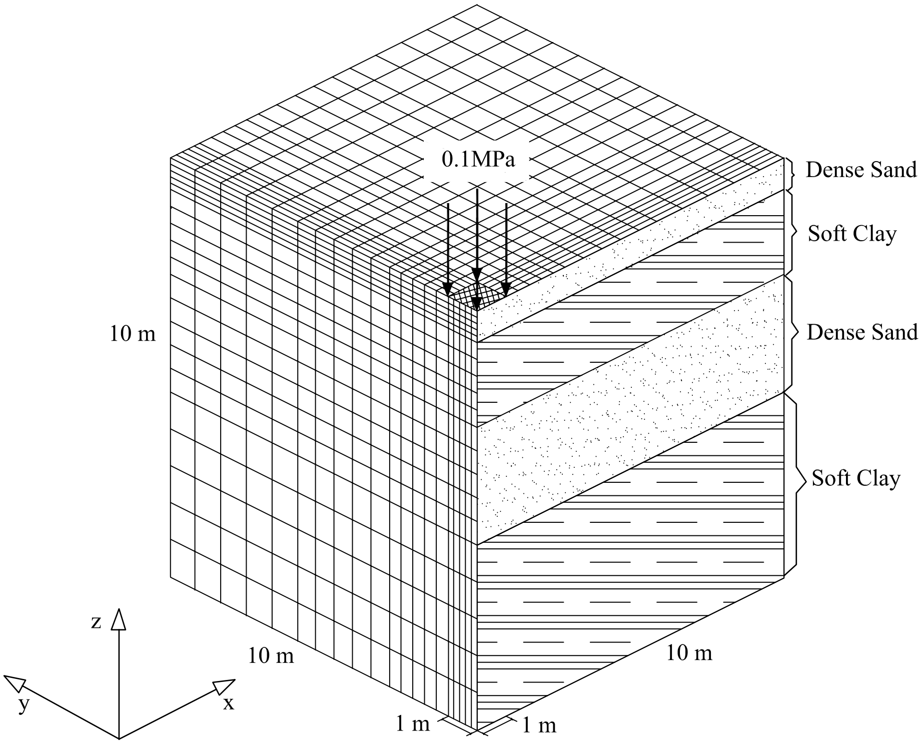


Figure 5.2: 20 × 20 × 20 finite element mesh and soil layer profile of a quadrant symmetric shallow foundation problem.

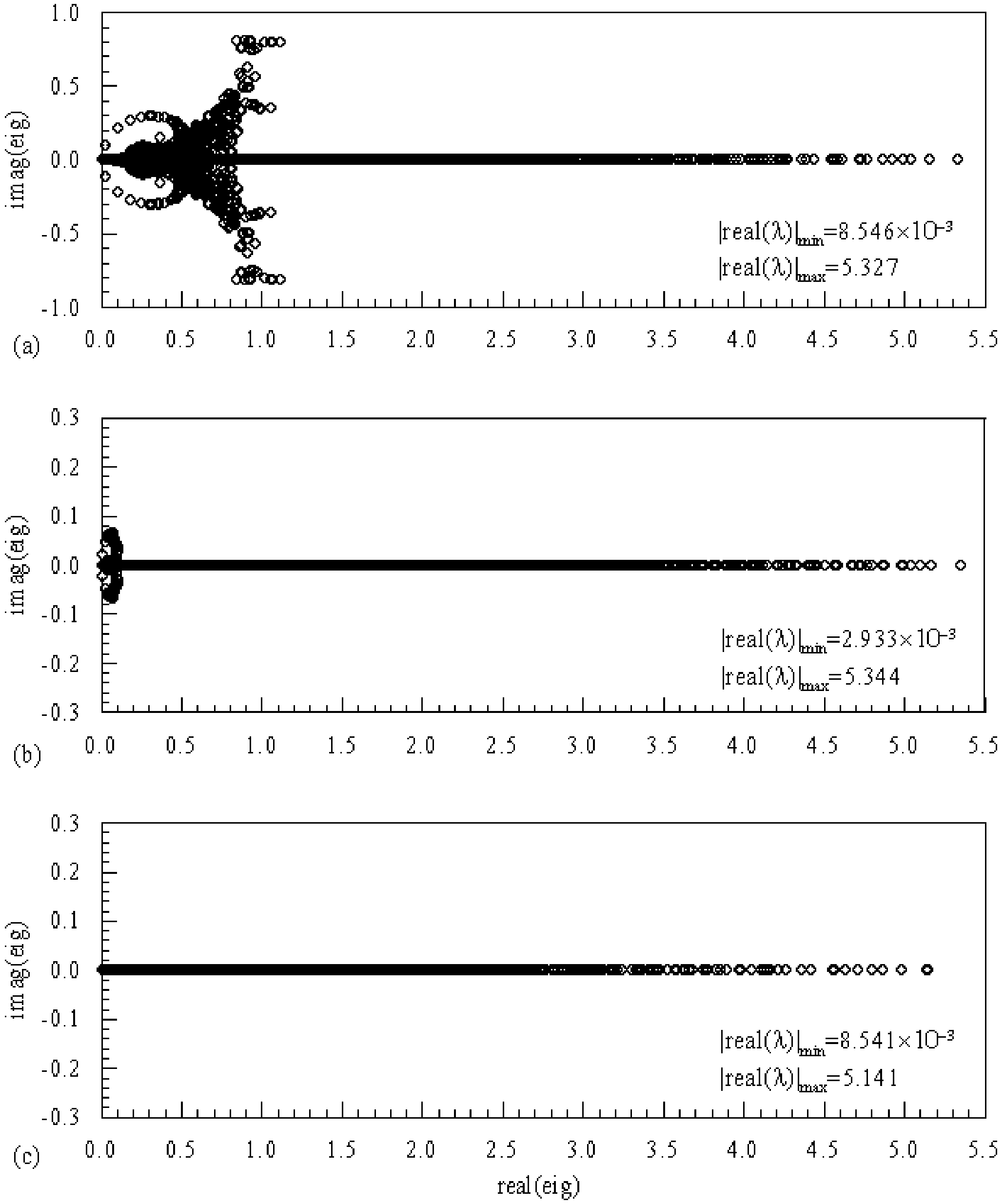


Figure 5.3: Eigenvalue distributions in complex plane (a) of GJ ($\alpha = -4$) preconditioned matrix; (b) of GJ ($\alpha = -20$) preconditioned matrix; (c) of P_c preconditioned matrix, for $7 \times 7 \times 7$ FE mesh with soil profile 1.

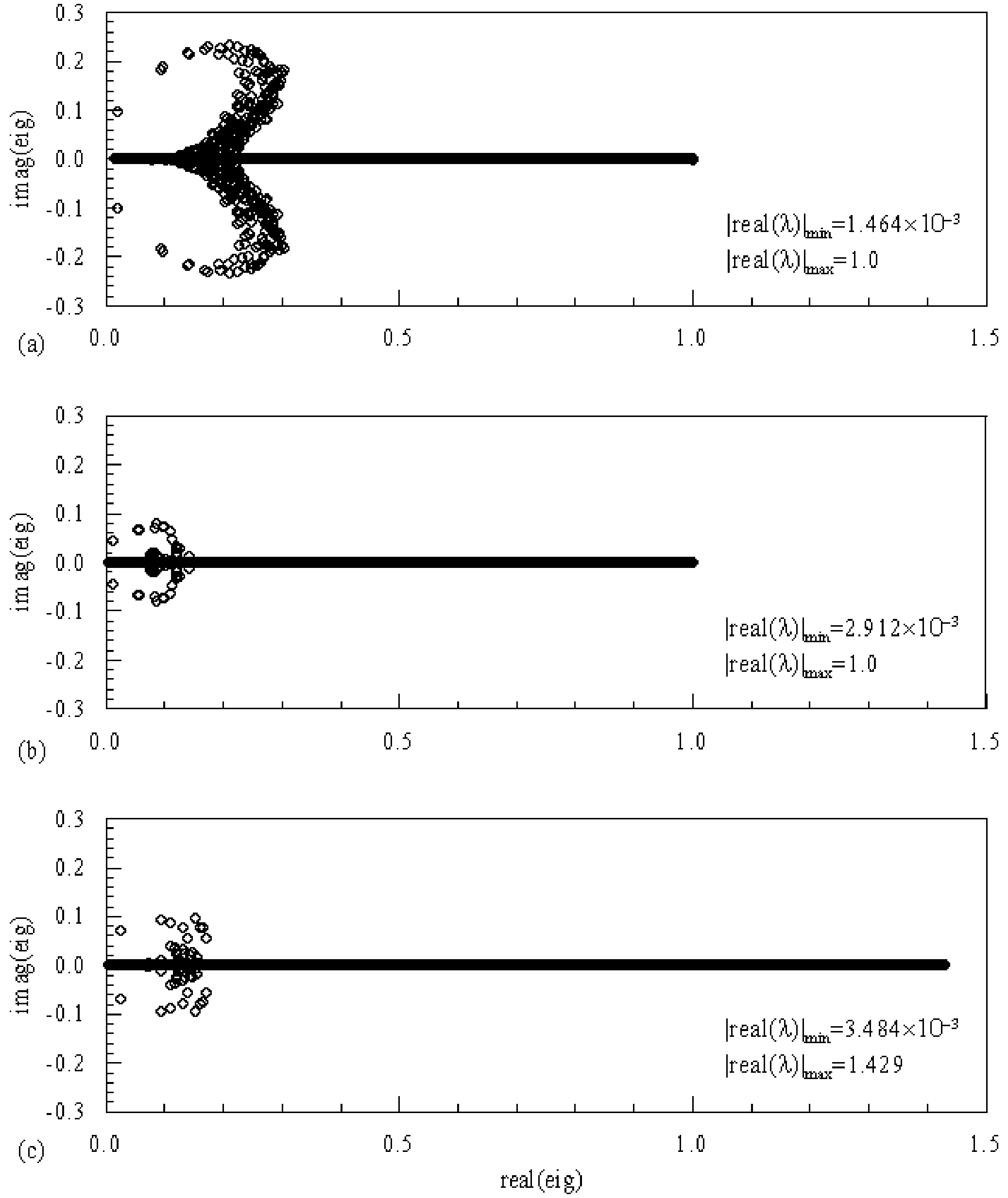


Figure 5.4: Eigenvalue distributions in complex plane of MSSOR preconditioned matrices (a) MSSOR ($\alpha = -4$, $\omega = 1.0$); (b) MSSOR ($\alpha = -20$, $\omega = 1.0$); (c) MSSOR ($\alpha = -20$, $\omega = 1.3$).

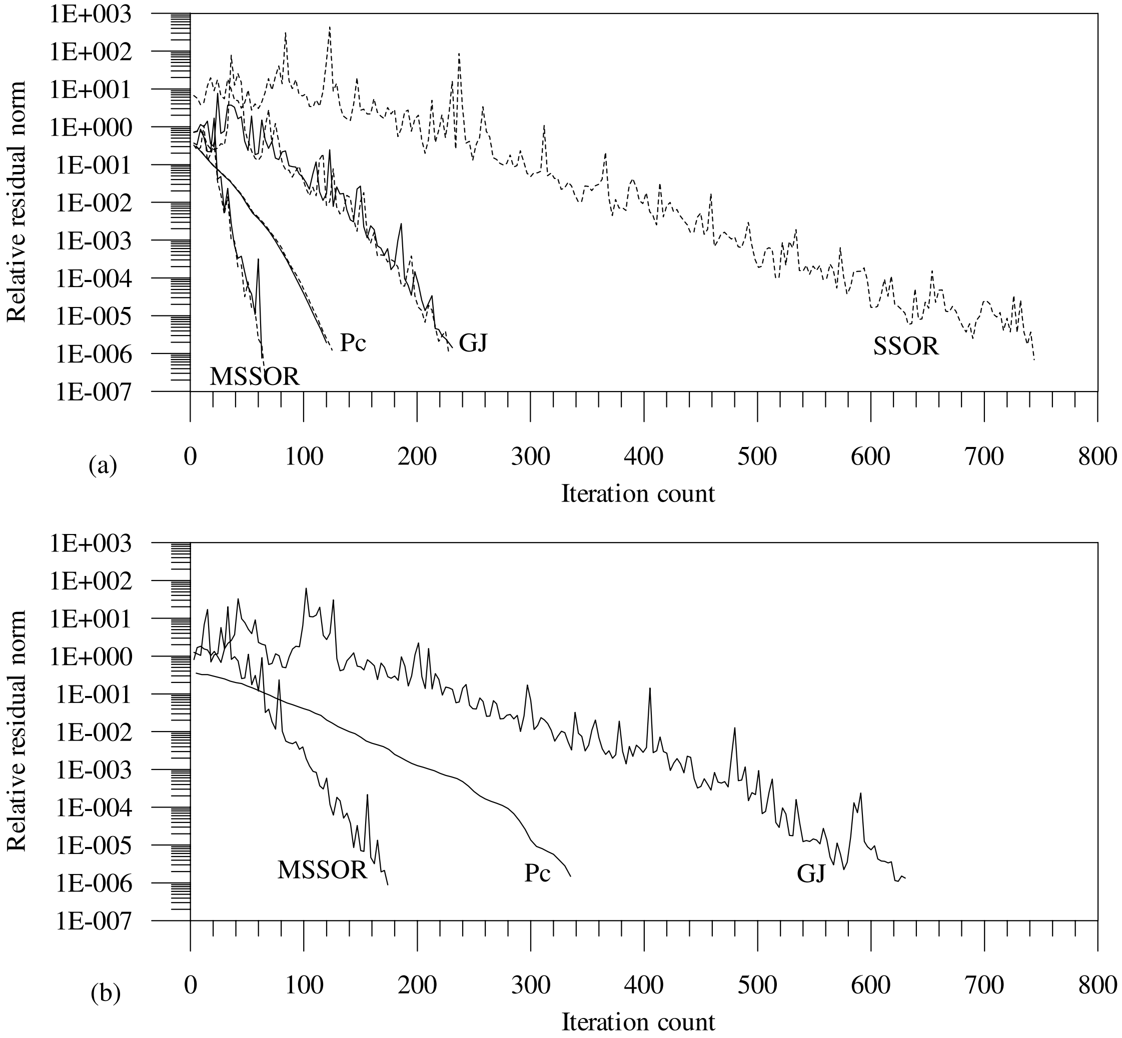


Figure 5.5: Convergence history of SQMR method preconditioned by GJ ($\alpha = -4$), P_c, SSOR ($\omega = 1.0$) and MSSOR ($\alpha = -4$, $\omega = 1.0$), respectively, for $7 \times 7 \times 7$ finite element mesh (a) with soil profile 1 (solid line) and with soil profile 2 (dashed line), respectively; (b) with soil profile 3.

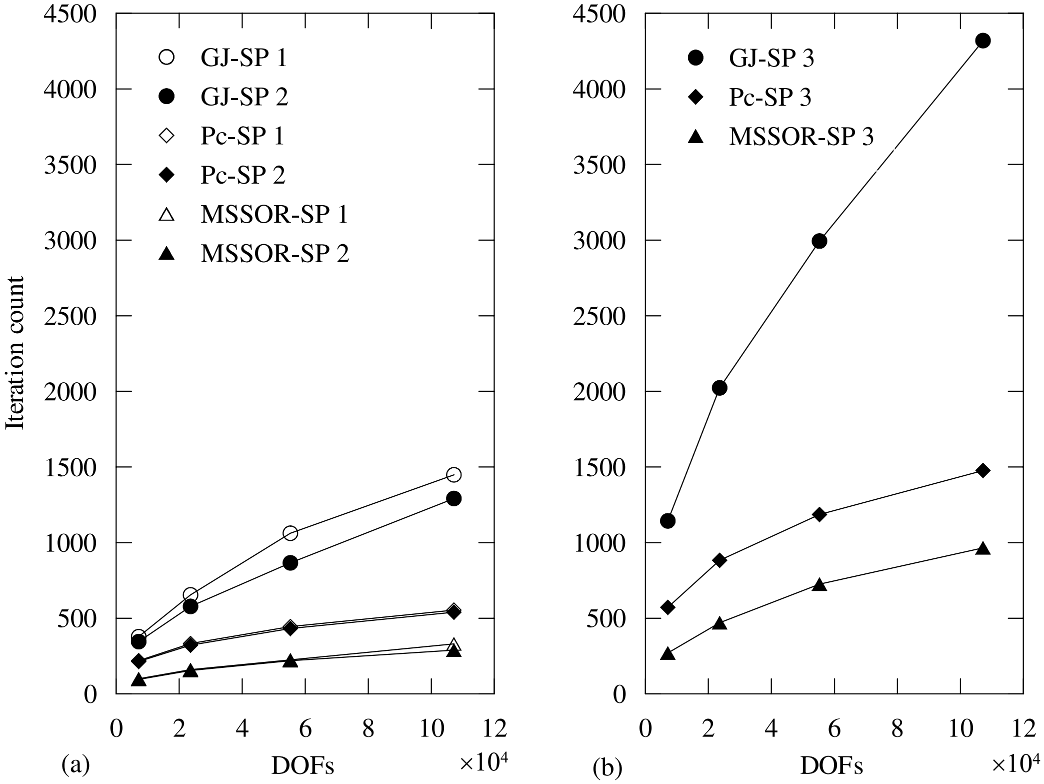


Figure 5.6: Iteration count versus DOFs for SQMR method preconditioned by GJ ($\alpha = -4$), P_c and MSSOR ($\alpha = -4$, $\omega = 1.0$), respectively for three soil profile (SP) cases.

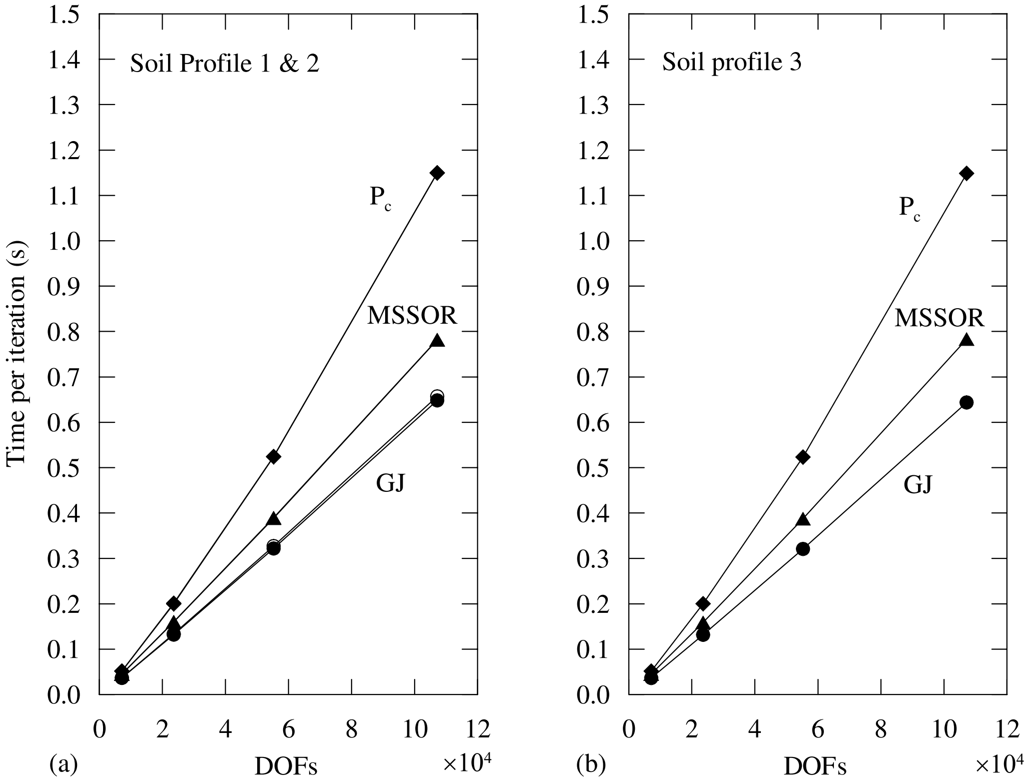
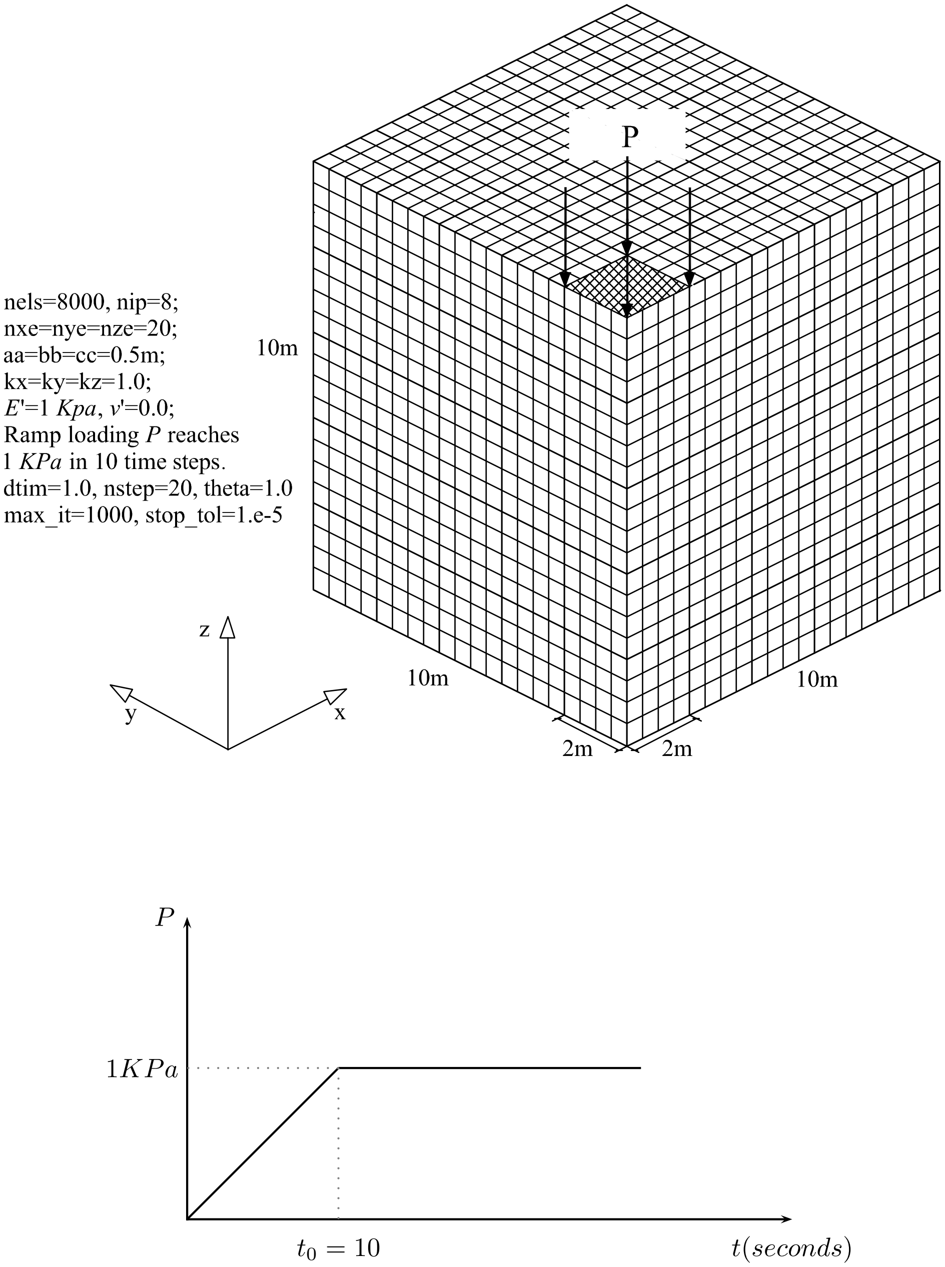


Figure 5.7: Time per iteration versus DOFs for SQMR method preconditioned by GJ ($\alpha = -4$), P_c and MSSOR ($\alpha = -4$, $\omega = 1.0$) for three soil profile cases.



nels=8000, nip=8;
nxe=nye=nze=20;
aa=bb=cc=0.5m;
kx=ky=kz=1.0;
E'=1 *Kpa*, *v*'=0.0;
Ramp loading *P* reaches
1 *KPa* in 10 time steps.
dtim=1.0, nstep=20, theta=1.0
max_it=1000, stop_tol=1.e-5

Figure 5.8: Uniform 20 × 20 × 20 finite element mesh with ramp loading

```
This job ran on 32 (16) processors:
There are 35721 nodes, 28862 restrained and 107180 equations
The current time is 0.1000E+01
(SJ preconditioned) CG took 360 iterations to converge
The nodal displacements and porepressures are :
-0.2591E+00 -0.5907E-02 -0.2582E+00 -0.1201E-01
The Gauss Point effective stresses for element 1 are :
Point 1
-0.2061E-01 -0.2061E-01 -0.9442E-01 0.6251E-03 -0.2071E-03 -
0.2071E-03
The current time is 0.2000E+01
(SJ preconditioned) CG took 262 iterations to converge
The nodal displacements and porepressures are :
-0.5495E+00 -0.1402E-01 -0.5478E+00 -0.2875E-01
The Gauss Point effective stresses for element 1 are :
Point 1
-0.5057E-01 -0.5057E-01 -0.1929E+00 0.3545E-02 -0.1009E-02 -
0.1009E-02
⋮
The current time is 0.2000E+02
(SJ preconditioned) CG took 313 iterations to converge
The nodal displacements and porepressures are :
-0.3638E+01 -0.1177E+00 -0.3623E+01 -0.2354E+00
The Gauss Point effective stresses for element 1 are :
Point 1
-0.4145E+00 -0.4145E+00 -0.99533E+00 0.5245E-02 0.6459E-02
0.6459E-02
This analysis took : 255.71 (538.00) seconds.
```

```
This job ran on desktop PC platform (single processor):
There are 35721 nodes, 28862 restrained and 107180 equations
The current time is 0.1000E+01
MSSOR preconditioned SQMR took 121 iterations to converge
The nodal displacements and porepressures are :
-0.2591E+00 -0.5906E-02 -0.2582E+00 -0.1201E-01
The Gauss Point effective stresses for element 1 are :
Point 1
-0.2060E-01 -0.2060E-01 -0.9442E-01 0.6283E-03 -0.2098E-03 -
0.2101E-03
The current time is 0.2000E+01
MSSOR preconditioned SQMR took 19 iterations to converge
The nodal displacements and porepressures are :
-0.5312E+00 -0.1351E-01 -0.5293E+00 -0.2732E-01
The Gauss Point effective stresses for element 1 are :
Point 1
-0.4698E-01 -0.4669E-01 -0.1920E+00 0.1466E-02 0.1462E-03
0.4161E-05
⋮
The current time is 0.2000E+02
MSSOR preconditioned SQMR took 73 iterations to converge
The nodal displacements and porepressures are :
-0.3547E+01 -0.1156E+00 -0.3533E+01 -0.2323E+00
The Gauss Point effective stresses for element 1 are :
Point 1
-0.4126E+00 -0.4123E+00 -0.1002E+01 0.1083E-01 0.1542E-02
0.1403E-02
This analysis took : 1935.67 seconds.
```

Figure 5.9: Results for parallel computing versus desktop PC

Table 5.1: 3-D finite element meshes

Mesh size		$8 \times 8 \times 8$	$12 \times 12 \times 12$	$16 \times 16 \times 16$	$20 \times 20 \times 20$
Number of elements(ne)		512	1728	4096	8000
Number of nodes		2673	8281	18785	35721
DOFs	Pore pressure(n)	648	2028	4624	8820
	Displacement (m)	6512	21576	50656	98360
	Total (m+n)	7160	23604	55280	107180
nnz	$nnz(triu(K))$	443290	1606475	3920747	7836115
	$nnz(B)$	103965	375140	907608	1812664
	$nnz(triu(C))$	7199	24287	57535	112319
	$nnz(\tilde{S})$	51714	187974	461834	921294
	$nnz(A)/(m+n)^2$, %	2.15	0.72	0.32	0.17
Total memory required(MB)		6.6	24.1	58.6	117.1

Table 5.2: Effect of ordering on MSSOR($\omega = 1$, $\alpha = -4$) preconditioned SQMR method.

		Soil Profile 1			Soil Profile 2			Soil Profile 3		
Mesh size		$iord1$	$iord2$	$iord3$	$iord1$	$iord2$	$iord3$	$iord1$	$iord2$	$iord3$
$8 \times 8 \times 8$	$iters$	100	110	105	95	105	105	270	300	310
	t_o(s)	10.9	11.0	11.2	10.8	11.1	11.3	11.1	11.3	11.5
	t_i(s)	4.5	5.0	4.8	4.3	4.8	4.8	12.2	13.4	13.8
	t_t(s)	15.5	16.2	16.1	15.3	16.0	16.2	23.8	25.4	25.9
$12 \times 12 \times 12$	$iters$	160	175	165	155	165	170	470	520	500
	t_o(s)	37.0	38.0	38.7	37.0	38.0	38.6	37.5	38.4	39.1
	t_i(s)	25.7	28.1	26.4	24.8	26.4	27.1	74.7	82.7	79.1
	t_t(s)	63.0	6.3	65.3	62.1	64.7	65.9	114.2	123.1	120.2
$16 \times 16 \times 16$	$iters$	225	270	265	220	250	240	725	790	740
	t_o(s)	88.3	90.7	92.5	88.3	90.7	92.5	89.4	91.9	93.6
	t_i(s)	87.6	105.5	103.5	85.5	97.6	93.8	280.7	307.5	288.0
	t_t(s)	176.2	196.6	196.3	174.1	188.6	186.7	374.5	403.7	386.0
$20 \times 20 \times 20$	$iters$	330	365	360	290	315	330	965	1045	1170
	t_o(s)	173.7	178.8	182.5	173.6	178.7	182.4	177.8	181.4	185.0
	t_i(s)	258.0	286.2	283.1	226.6	247.4	259.6	755.9	814.0	914.1
	t_t(s)	432.3	465.6	466.1	400.9	426.7	442.6	942.2	1003.9	1107.6

Table 5.3: Effect of α on iterative count of MSSOR($\omega = 1.0$) preconditioned SQMR method

$-\alpha$	1	4	10	20	30	40	50	60	70	$\frac{It_{\alpha=-4}}{It_{min}}$
Mesh size: $7 \times 7 \times 7$										
Soil profile 1	70	65	90	120	145	145	165	165	165	1.00
Soil profile 2	65	65	65	85	95	100	100	110	115	1.00
Soil profile 3	245	175	150	140	155	165	180	180	210	1.25
Mesh size: $16 \times 16 \times 16$										
Soil profile 1	405	225	215	205	200	210	210	225	235	1.13
Soil profile 2	215	220	200	180	165	155	155	165	170	1.42
Soil profile 3	1540	725	600	550	505	500	490	485	495	1.49

Table 5.4: Effect of ω on iterative count of standard SSOR and MSSOR($\alpha = -50$) preconditioned SQMR methods, respectively.

$-\alpha$		1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8
Mesh size: $7 \times 7 \times 7$										
Soil profile 1	SSOR	*	*	*	*	*	*	*	*	*
	MSSOR	165	155	145	155	160	165	180	205	245
Soil profile 2	SSOR	745	1830	–	–	–	–	–	–	–
	MSSOR	100	105	110	110	115	120	135	140	180
Soil profile 3	SSOR	*	*	*	*	*	*	*	*	*
	MSSOR	180	175	185	190	195	220	240	270	285
Mesh size: $16 \times 16 \times 16$										
Soil profile 1	SSOR	*	*	*	*	*	*	*	*	*
	MSSOR	210	205	185	190	190	210	220	260	335
Soil profile 2	SSOR	380	480	720	3600	–	–	–	–	–
	MSSOR	155	155	145	140	155	165	175	200	240
Soil profile 3	SSOR	*	*	*	*	*	*	*	*	*
	MSSOR	490	460	445	420	415	415	440	485	570

(*) means ‘Breakdown’; (–) means no convergence.

Table 5.5: Performance of several preconditioners over different mesh sizes for soil profile 1 with homogeneous soft clay, $E' = 1$ MPa, $\nu' = 0.3$, $k = 10^{-9}$ m/s.

Mesh size	$8 \times 8 \times 8$	$12 \times 12 \times 12$	$16 \times 16 \times 16$	$20 \times 20 \times 20$
GJ ($\alpha = -4$)				
RAM(MB)	26.5	84.0	194.5	386.0
Iteration count	378	654	1062	1448
Overhead (s)	11.1	37.8	90.1	177.2
Iteration time (s)	14.0	87.5	347.5	951.2
Total runtime (s)	25.7	127.3	442.0	1136.9
P_c				
RAM (MB)	27.5	84.5	196	387
Iteration count	220	333	444	554
Overhead (s)	11.4	41.0	106.7	247.5
Iteration time (s)	11.4	66.9	232.9	636.8
Total runtime (s)	23.4	109.8	343.9	892.8
P_c with Eisenstat trick				
RAM (MB)	27.5	84.5	196	387
Iteration count	220	335	445	560
Overhead (s)	11.1	39.5	102.3	238.1
Iteration time (s)	11.0	65.6	227.3	629.5
Total runtime (s)	22.7	107.0	334.0	876.1
MSSOR ($\omega = 1.0$, $\alpha = -4$)				
RAM (MB)	26.5	84.5	195.0	387.0
Iteration count	100	160	225	330
Overhead (s)	10.9	37.0	88.3	173.7
Iteration time (s)	4.5	25.7	87.6	258.0
Total runtime (s)	15.5	63.0	176.2	432.3
MSSOR ($\omega = 1.3$, $\alpha = -50$)				
RAM (MB)	26.5	84.5	195.0	387.0
Iteration count	205	185	190	215
Overhead (s)	10.9	37.1	88.3	173.6
Iteration time (s)	9.2	29.6	73.8	168.3
Total runtime (s)	20.1	66.9	162.5	342.5

Table 5.6: Performance of several preconditioners over different mesh sizes for soil profile 2 with homogeneous dense sand, $E' = 100$ MPa, $\nu^{'} = 0.3$, $k = 10^{-5}$ m/s.

Mesh size	$8 \times 8 \times 8$	$12 \times 12 \times 12$	$16 \times 16 \times 16$	$20 \times 20 \times 20$
GJ ($\alpha = -4$)				
RAM(MB)	26.5	84.0	194.5	386.0
Iteration count	346	578	866	1292
Overhead (s)	10.9	37.3	89.0	174.9
Iteration time (s)	12.7	76.3	278.7	837.4
Total runtime (s)	24.2	115.5	371.9	1020.8
P_c				
RAM (MB)	27.5	84.5	196	387
Iteration count	215	322	432	540
Overhead (s)	11.4	40.9	106.7	247.1
Iteration time (s)	11.1	64.4	226.5	620.8
Total runtime (s)	23.1	107.3	337.5	876.3
P_c *with Eisenstat trick*				
RAM (MB)	27.5	84.5	196	387
Iteration count	215	325	435	540
Overhead (s)	11.0	39.5	102.3	237.5
Iteration time (s)	10.8	63.5	222.5	606.6
Total runtime (s)	22.4	105.0	329.1	852.7
MSSOR ($\omega = 1.0$, $\alpha = -4$)				
RAM (MB)	26.5	84.0	194.5	386
Iteration count	95	155	220	290
Overhead (s)	10.8	37.0	88.3	173.6
Iteration time (s)	4.3	24.8	85.5	226.6
Total runtime (s)	15.3	62.1	174.1	400.9
MSSOR ($\omega = 1.3$, $\alpha = -50$)				
RAM (MB)	26.5	84.0	194.5	386
Iteration count	115	115	140	185
Overhead (s)	10.9	37.1	88.3	173.6
Iteration time (s)	5.2	18.4	54.5	144.7
Total runtime (s)	16.2	55.8	143.2	319.0

Table 5.7: Performance of several preconditioners over different mesh sizes for soil profile 3 with alternative soil properties $E' = 100$ MPa, $\nu^{'} = 0.3$, $k = 10^{-5}$ m/s and $E' = 1$ MPa, $\nu^{'} = 0.3$, $k = 10^{-9}$ m/s

Mesh size	$8 \times 8 \times 8$	$12 \times 12 \times 12$	$16 \times 16 \times 16$	$20 \times 20 \times 20$
GJ ($\alpha = -4$)				
RAM(MB)	26.5	84	194.5	386
Iteration count	1143	2023	2994	4318
Overhead (s)	10.9	37.0	88.6	174.2
Iteration time (s)	41.6	266.2	961.3	2779.5
Total runtime (s)	53.0	305.0	1054.2	2962.1
P_c				
RAM (MB)	27.5	84.5	196	387
Iteration count	572	883	1186	1477
Overhead (s)	11.4	41.1	107.1	247.2
Iteration time (s)	29.5	176.7	620.6	1696.3
Total runtime (s)	41.6	219.7	732.0	1952.0
P_c with Eisenstat trick				
RAM (MB)	27.5	84.5	196	387
Iteration count	575	880	1190	1480
Overhead (s)	11.0	39.2	101.8	236.8
Iteration time (s)	28.8	172.1	607.9	1662.0
Total runtime (s)	40.4	213.2	714.1	1907.2
MSSOR ($\omega = 1.0$, $\alpha = -4$)				
RAM (MB)	26.5	84.0	194.5	386
Iteration count	270	470	725	965
Overhead (s)	11.1	37.5	89.4	177.8
Iteration time (s)	12.2	74.7	280.7	755.9
Total runtime (s)	23.8	114.2	374.5	942.2
MSSOR ($\omega = 1.3$, $\alpha = -50$)				
RAM (MB)	26.5	84.0	194.5	386
Iteration count	240	330	420	515
Overhead (s)	10.9	37.3	88.9	174.9
Iteration time (s)	10.6	52.4	162.3	397.8
Total runtime (s)	22.2	91.7	255.5	581.2

CHAPTER 6

NEWTON-KRYLOV ITERATIVE METHODS FOR LARGE-SCALE NONLINEAR CONSOLIDATION

6.1 Introduction

Three-dimensional Biot's consolidation equations have been widely used to simulate soil consolidation problems in geotechnical engineering (e.g. Biot, 1941; Lewis and Schrefler, 1998; Zienkiewicz *et al.*, 1998). Nonlinearity is quite commonly incorporated into finite element analysis to obtain more realistic answers. Hence, it is necessary to consider nonlinear behavior of soil skeleton or fluid flow (e.g. Siriwardane and Desai, 1981). The nonlinearity in a general consolidation problem may arise from two sources: one is soil material nonlinearity which is dominated by the constitutive relation of soil model and the other is geometrical nonlinearity which may arise due to large deformation of soil skeleton. In this chapter, only material nonlinearity is considered. For nonlinear finite element equations involving elasto-plastic problems, there are well-known iterative schemes including Newton method, modified Newton methods and so-called initial stiffness method. Newton method is popular because of its asymptotically quadratic convergence rate. When applying Newton method to finite element analysis, it is necessary to form and factorize the tangent stiffness matrix in each iteration, and this process could be expensive. In some applications, modified Newton method may be preferred. Newton iteration have been exploited due to its fast convergence rate in some elasto-plastic finite element analysis. However, this quadratic convergence rate may lose due to some reasons. Simo and Taylor (1985) introduced the notion of the consistency between integration algorithm and tangent stiffness operator and suggested the use of a consistent tangent operator which plays a crucial role in preserving the quadratic convergence behavior of global Newton iteration. This consistency of integration and tangent stiffness operator has been emphasized by many researches (e.g. Borja and Lee, 1990; Borja, 1991; Crisfield, 1991; Hashash and Whittle, 1992; Jeremić and Sture, 1997; Hickman and Gutierrez, 2005).

Linearizing the nonlinear problems may always produce many linear equations for which direct or iterative methods can be adopted. Recently, Newton-Krylov iterative methods have received some attention in engineering problems because they are believed to provide robust solution for large-scale nonlinear problems. By definition, the term “Newton-Krylov” means that Newton-type iterative scheme is used for linearizing nonlinear problems, and Krylov subspace iterative method is adopted for the linearized linear systems. Instead of using direct solution methods which are based on Gauss elimination algorithm, Krylov subspace iterative methods are particularly valuable because iteration can be stopped when the desired precision is acquired. Wang (1996) also studied the characteristics of PCG in elasto-plastic analysis and concluded that although the PCG method was about 30% slower than the direct method for plastic analysis at that time, but he was very confident that with some improvements, PCG can be made faster in plastic analysis in the future. For nonlinear Biot’s consolidation problems, Borja (1991) compared the composite Newton-PCG method to Newton-Gauss method and quasi-Newton method (rank-two BFGS inverse update). Numerical results which were based on both the ideal von Mises elasto-plastic model and the more realistic modified Cam-Clay model showed that the composite Newton-PCG method possesses promising potentials for large-scale computations. Based on the square footing problems, Axelsson *et al.* (1997) suggested using efficient preconditioned linear solver in inexact Newton solvers for 3-D elasto-plastic problems. Jones and Woodward (2001) used multigrid Newton-Krylov solvers in saturated flow simulations because one obvious advantage is that matrix-vector products in

Krylov subspace method can be approximated by the finite difference method.

In view of the recent rapid developments on fast preconditioners and robust linear solvers for large consolidation problems (e.g. Chan *et al.*, 2001; Phoon *et al.*, 2002; Toh *et al.*, 2004), it is worthwhile to investigate the numerical performance of fast preconditioned Newton-Krylov methods in large-scale computations of nonlinear consolidations. In this chapter, modified Cam clay soil model and ideal von Mises soil model are adopted, and fully implicit backward Euler return algorithm is used to integrate the stress-strain relation. Explicit tangent operator is derived to maintain consistency with the integration algorithm and preserve fast convergence of global Newton iteration. For each weakly nonsymmetric indefinite linear system in each global Newton iteration (e.g. Borja, 1991), symmetric quasi-minimal residual (SQMR) is used in conjunction with recently developed robust preconditioners such as generalized Jacobi (GJ) and modified SSOR (MSSOR) preconditioners as well as the preconditioner proposed by Borja (1991). In chapter 4, numerical results have shown that P_c preconditioned SQMR with proposed implementations can be faster than GJ preconditioned SQMR method. However, GJ is chosen for nonlinear consolidation problems for two reasons: one reason is that GJ is very simple to implement and almost cost-free to construct, but P_c needs one more complicated sparse Cholesky solve in each iteration. The other reason is that a higher convergence tolerance (such as 10^{-3} or 10^{-4}) may be needed for nonlinear consolidations so that iteration time savings may not overwhelm the cost of constructing the P_c preconditioner, and thus, P_c will not be considered in this chapter for nonlinear consolidation problems.

6.2 Nonlinearity of FE Biot's Consolidation Equations

Biot's consolidation equations are composed of the equilibrium equations and the continuity equation as follows (e.g. Abbo, 1997)

$$\left\{ \begin{array}{rcl} \widetilde{\nabla}^T(\boldsymbol{\sigma}' + \beta p \mathbf{1}) + \mathbf{b} & = & 0 \\ div\, \mathbf{v} + \mathbf{1}^T \dot{\boldsymbol{\varepsilon}} & = & 0 \end{array} \right. \tag{6.1}$$

where $\widetilde{\nabla}$ is a differential operator; $\boldsymbol{\sigma}' = \{\sigma'_x, \sigma'_y, \sigma'_z, \tau_{xy}, \tau_{yz}, \tau_{zx}\}^T$ is the vector of effective stress, and β is the Biot coefficient which is often assumed to be unity; $\mathbf{1} = \{1, 1, 1, 0, 0, 0\}^T$ is the second-order Kronecker delta in vectorial form; $\mathbf{b} = \{b_x, b_y, b_z\}^T$ is the vector of body force; $\dot{\boldsymbol{\varepsilon}}$ is time differentiation of strain vector $\boldsymbol{\varepsilon} = \{\varepsilon_x, \varepsilon_y, \varepsilon_z, \gamma_{xy}, \gamma_{yz}, \gamma_{zx}\}^T$;

$p = p^s + p^e$ is total pore water pressure decomposed with steady state component p^s, and excess component p^e (pore pressure in excess of that at steady state), respectively. $\mathbf{v} = \{v_x, v_y, v_z\}^T$ is the vector of superficial fluid velocity (i.e., average relative velocity of seepage measured over the total area). The components of the velocity vector can be determined by Darcy's law as follows,

$$\mathbf{v} = \frac{[\mathbf{k}]}{\gamma_w}(\nabla p + \mathbf{b}_w) \tag{6.2}$$

where $[\mathbf{k}]$ is the permeability matrix, and usually, $k_{xy} = k_{xz} = k_{yz} = 0$ is assumed; γ_w is the unit weight of pore water; $\mathbf{b}_w = \gamma_w \mathbf{b}$.

Spatial discretization and time integration result in the incremental formulation of Biot's consolidation equation for which only displacement and pore pressure are required to march the iteration,

$$\begin{bmatrix} \mathbf{K} & \mathbf{L} \\ \mathbf{L}^T & -\theta\Delta t\mathbf{G} \end{bmatrix} \begin{Bmatrix} \Delta\mathbf{u} \\ \Delta\mathbf{p}^e \end{Bmatrix} = \begin{Bmatrix} \Delta\mathbf{f} \\ \Delta t\mathbf{G}\mathbf{p}_n^e \end{Bmatrix} \tag{6.3}$$

where $\mathbf{L}$ is solid-fluid connection matrix; $\mathbf{G}$ is the fluid stiffness matrix; θ is time integration parameter, the choice $\theta = 1$ corresponds to the fully implicit method (e.g. Smith and Griffiths, 1998); Δt is time increment, $\Delta\mathbf{u}$ is the displacement increment, $\Delta\mathbf{p}^e$ is the increment of excess pore water pressure and Δf is applied force increment; $\mathbf{K}$ is a stress-dependent solid stiffness matrix

$$\mathbf{K} = \sum_e \left(\int_{V_e} \mathbf{B}_u^T \mathbf{D}^{ep} \mathbf{B}_u \, dV \right) \tag{6.4}$$

in which, $\mathbf{B}_u$ is the strain-displacement matrix and $\mathbf{D}^{ep}$ is elasto-plastic constitutive matrix. Obviously, if only material nonlinearity is considered, the nonlinear stress-strain relation is one source of the nonlinearity, while the other source, which is not considered in this chapter, may arise from $\mathbf{G}$ or more accurately, the permeability matrix $[\mathbf{k}]$ due to the compaction of soil skeleton. Because $\mathbf{G}$ is assumed to be positive definite, and thus, the symmetry of the coupled stiffness matrix in Eq. (6.3) is obviously determined by the stress-strain matrix $\mathbf{D}^{ep}$.

6.3 Incremental Stress-Strain Relations

In computational elasto-plastic analysis, it is convenient to distinguish the stress states in terms of elastic and plastic components by introducing a "yield" function (or surface)

$f(\boldsymbol{\sigma}, \mathbf{q})$, where $\boldsymbol{\sigma} = \{\sigma_x, \sigma_y, \sigma_z, \tau_{xy}, \tau_{yz}, \tau_{zx}\}^T$ is the effective stress vector (here, prime symbol is omitted for clarity) and $\mathbf{q} = \{q_1, q_2, \ldots, q_n\}^T$ is the vector with n internal variables. Whether a stress state is elastic or plastic depends on its the current location. It is said that the a stress state goes into plasticity from elasticity when it changes from the inside of the yield surface onto the the yield surface, and a stress state lying outside of the yield surface is inadmissable.

Once the elastic limit is exceeded in an elasto-plastic analysis, it is assumed that the total strain increment, $\mathrm{d}\boldsymbol{\varepsilon}$, is composed of two components: elastic strain increment and plastic strain increment,

$$\mathrm{d}\boldsymbol{\varepsilon} = \mathrm{d}\boldsymbol{\varepsilon}^e + \mathrm{d}\boldsymbol{\varepsilon}^p \tag{6.5}$$

and the total strain increment is related to the stress increment by the relation

$$\mathrm{d}\boldsymbol{\sigma} = \mathbf{D}^{ep}\mathrm{d}\boldsymbol{\varepsilon} \tag{6.6}$$

where

$$\mathbf{D}^{ep} = \mathbf{D}^e - \mathbf{D}^p \tag{6.7}$$

Because the stress increment is only generated by the elastic strain increment, we have

$$\mathrm{d}\boldsymbol{\sigma} = \mathbf{D}^e\mathrm{d}\boldsymbol{\varepsilon}^e = \mathbf{D}^e(\mathrm{d}\boldsymbol{\varepsilon} - \mathrm{d}\boldsymbol{\varepsilon}^p) \tag{6.8}$$

The plastic strain increment, which is normal to the plastic potential surface denoted by $g(\boldsymbol{\sigma}, \mathbf{q})$, is determined by the plastic flow rule

$$\mathrm{d}\boldsymbol{\varepsilon}^p = \mathrm{d}\lambda\left\{\frac{\partial g}{\partial \boldsymbol{\sigma}}\right\} \tag{6.9}$$

where $\mathrm{d}\lambda$ is the change of the plastic multiplier, $\Delta\lambda$. If the strain increment is purely elastic, i.e., $\Delta\varepsilon_v^p = 0$, no hardening occurs. In more details, it can be observed from the following Karush-Kuhn-Tucker form

$$f(\boldsymbol{\sigma}, \mathbf{q}) \leq 0 \tag{6.10a}$$

$$\mathrm{d}\lambda \geq 0 \tag{6.10b}$$

$$f\mathrm{d}\lambda = 0 \tag{6.10c}$$

Eqs. (6.10b)$\sim$ (6.10c) must be satisfied simultaneously in any loading process. By making use of Karush-Kuhn-Tucker form, elastic behavior can be identified with $f < 0$ and

$\mathrm{d}\lambda = 0$, and plastic yielding can be characterized by $\mathrm{d}\lambda > 0$ (e.g. Jeremić and Sture, 1997; Hickman and Gutierrez, 2005).

Once a stress state reaches the yield surface, it remains on the surface. This process is dictated by the consistency condition,

$$\mathrm{d}f = \left\{\frac{\partial f}{\partial \boldsymbol{\sigma}}\right\}^T \mathrm{d}\boldsymbol{\sigma} + \left\{\frac{\partial f}{\partial \mathbf{q}}\right\}^T \mathrm{d}\mathbf{q} = 0 \tag{6.11}$$

with the assumed relation,

$$\mathrm{d}\mathbf{q} = \mathrm{d}\lambda \mathbf{h}(\boldsymbol{\sigma}, \mathbf{q}) \tag{6.12}$$

Substituting Eqs. (6.8), (6.9) and (6.12) into Eq. (6.11), we get the plastic multiplier,

$$\mathrm{d}\lambda = \frac{\left\{\dfrac{\partial f}{\partial \boldsymbol{\sigma}}\right\}^T \mathbf{D}^e \,\mathrm{d}\boldsymbol{\varepsilon}}{\left\{\dfrac{\partial f}{\partial \boldsymbol{\sigma}}\right\}^T \mathbf{D}^e \left\{\dfrac{\partial g}{\partial \boldsymbol{\sigma}}\right\} - \mathbf{c}^T\mathbf{h}} \tag{6.13}$$

Thus, substituting Eqs. (6.9) and (6.13) into Eq. (6.8) leads to

$$\mathrm{d}\boldsymbol{\sigma} = \mathbf{D}^{ep}\mathrm{d}\boldsymbol{\varepsilon} = (\mathbf{D}^e - \mathbf{D}^p)\mathrm{d}\boldsymbol{\varepsilon} = \left(\mathbf{D}^e - \frac{\mathbf{D}^e\mathbf{b}\mathbf{a}^T\mathbf{D}^e}{\mathbf{a}^T\mathbf{D}^e\mathbf{b} - \mathbf{c}^T\mathbf{h}}\right)\mathrm{d}\boldsymbol{\varepsilon} \tag{6.14}$$

where

$$\mathbf{a} = \left\{\frac{\partial f}{\partial \boldsymbol{\sigma}}\right\} = \left\{\frac{\partial f}{\partial \sigma_x}, \frac{\partial f}{\partial \sigma_y}, \frac{\partial f}{\partial \sigma_z}, \frac{\partial f}{\partial \tau_{xy}}, \frac{\partial f}{\partial \tau_{yz}}, \frac{\partial f}{\partial \tau_{zx}}\right\}^T \tag{6.15a}$$

$$\mathbf{b} = \left\{\frac{\partial g}{\partial \boldsymbol{\sigma}}\right\} = \left\{\frac{\partial g}{\partial \sigma_x}, \frac{\partial g}{\partial \sigma_y}, \frac{\partial g}{\partial \sigma_z}, \frac{\partial g}{\partial \tau_{xy}}, \frac{\partial g}{\partial \tau_{yz}}, \frac{\partial g}{\partial \tau_{zx}}\right\}^T \tag{6.15b}$$

$$\mathbf{c} = \left\{\frac{\partial g}{\partial \mathbf{h}}\right\} = \left\{\frac{\partial g}{\partial h_1}, \frac{\partial g}{\partial h_2}, \ldots, \frac{\partial g}{\partial h_n}\right\}^T \tag{6.15c}$$

It is worth noting that in Eq. (6.14), the denominator is a scalar, and thus, the symmetry of $\mathbf{D}^p$ or $\mathbf{D}^{ep}$ depends on the product of $\mathbf{b}\mathbf{a}^T$. In the physical sense, $\mathbf{b} = \mathbf{a}$ leads to an associated plastic flow rule (that is, the vector of plastic strain increment should be normal to the yield surface) and a symmetric elasto-plastic stress-strain matrix, while $\mathbf{b} \neq \mathbf{a}$ results in non-associated plastic flow rule and nonsymmetric elasto-plastic stress-strain matrix.

In Eq. (6.14), $\mathbf{D}^{ep}$ is known as standard or continuum tangent matrix though it is not required to be formed explicitly. A widely adopted consistent tangent stiffness operator with backward Euler integration was proposed by Simo and Taylor (1985). This consistent tangent stiffness operator can be derived by making use of the complete expression of $\mathrm{d}\boldsymbol{\varepsilon}^p$ which incorporates the change of plastic flow direction,

$$\mathrm{d}\boldsymbol{\varepsilon}^p = \mathrm{d}\lambda\mathbf{b} + \Delta\lambda\left[\frac{\partial \mathbf{b}}{\partial \boldsymbol{\sigma}}\right]\mathrm{d}\boldsymbol{\sigma} + \Delta\lambda\left[\frac{\partial \mathbf{b}}{\partial \mathbf{q}}\right]\mathrm{d}\mathbf{q} \tag{6.16}$$

Substituting the complete expression of plastic strain into Eq. (6.8) gives

$$\mathrm{d}\boldsymbol{\sigma} = \mathbf{R}\mathrm{d}\boldsymbol{\varepsilon} - \mathrm{d}\lambda\mathbf{R}\mathbf{t} \tag{6.17}$$

where $\mathbf{Q} = \mathbf{I} + \Delta\lambda\mathbf{D}^e\left[\frac{\partial\mathbf{b}}{\partial\boldsymbol{\sigma}}\right]$, $\mathbf{R} = \mathbf{Q}^{-1}\mathbf{D}^e$ and $\mathbf{t} = \mathbf{b} + \Delta\lambda\left[\frac{\partial\mathbf{b}}{\partial\mathbf{q}}\right]\mathbf{h}$. Further substituting Eqs. (6.12) and (6.17) into Eq. (6.11) results in

$$\mathrm{d}\lambda = \frac{\mathbf{a}^T\mathbf{R}\,\mathrm{d}\boldsymbol{\varepsilon}}{\mathbf{a}^T\mathbf{R}\mathbf{t} - \mathbf{c}^T\mathbf{h}} \tag{6.18}$$

Thus, Eq. (6.17) and Eq. (6.18) give the stress-strain relation as

$$\mathrm{d}\boldsymbol{\sigma} = \left(\mathbf{R} - \frac{\mathbf{R}\mathbf{t}\mathbf{a}^T\mathbf{R}}{\mathbf{a}^T\mathbf{R}\mathbf{t} - \mathbf{c}^T\mathbf{h}}\right)\mathrm{d}\boldsymbol{\varepsilon} = \mathbf{D}^{epc}d\boldsymbol{\varepsilon} \tag{6.19}$$

where $\mathbf{D}^{epc}$ is the consistent tangent matrix. By incorporating second-derivative term of plastic potential function which are ignored in standard elasto-plastic stress-strain relations, the consistent tangent matrix technique can recover a good convergence rate of global Newton iteration. It can been seen that the symmetry of $\mathbf{D}^{epc}$ depends on $\left[\frac{\partial\mathbf{b}}{\partial\boldsymbol{\sigma}}\right]$ as well as $\mathbf{t}\mathbf{a}^T$. However, it is important to distinguish strong nonsymmetry and weak nonsymmetry because symmetric iterative solver can be employed for weakly nonsymmetric linear system. Strong nonsymmetry may arise from the non-associated plastic flow rule, while weak nonsymmetry may arise from the associated plastic flow rule with exact linearization (e.g. Borja, 1991).

6.4 Backward Euler Return Algorithm

In a strain-driven finite element analysis, the stress state at previous time step and current strain increment are known. The new stress state can be obtained by using the backward-Euler return algorithm without the need of computing the intersection point between the stress path and the yield surface which is required in substepping schemes of elasto-plastic stress-strain integration (e.g. Sloan, 1987; Abbo, 1997; Potts and Zdravković, 1999). The new stress state with backward Euler return can be expressed as

$$\boldsymbol{\sigma}_{n+1} = (\boldsymbol{\sigma}_n + \Delta\boldsymbol{\sigma}^{tr}_{n+1}) - \Delta\lambda\mathbf{D}^e\mathbf{b}_{n+1} = \boldsymbol{\sigma}^{tr}_{n+1} - \Delta\lambda\mathbf{D}^e\mathbf{b}_{n+1} \tag{6.20}$$

where $\Delta\boldsymbol{\sigma}^{tr}_{n+1}$ is the stress increment by assuming the input strain increment is purely elastic. If the stress increment causes a change from elasticity to plasticity, plastic relaxation is required, and this process is named as elastic estimation with a plastic corrector (e.g. Ortiz and Simo, 1986; Crisfield, 1991). The objective of this plastic correction as shown in Eq. (6.20) is to minimize $f(\boldsymbol{\sigma}_{n+1}, \mathbf{q}_{n+1})$. When considering first order Taylor expansion at $\boldsymbol{\sigma}^{tr}_{n+1}$, we get

$$\begin{aligned} f(\boldsymbol{\sigma}_{n+1}, \mathbf{q}_{n+1}) &\approx f(\boldsymbol{\sigma}^{tr}_{n+1}, \mathbf{q}_n) + \left\{\frac{\partial f}{\partial \boldsymbol{\sigma}}\right\}^T \Delta\boldsymbol{\sigma} + \left\{\frac{\partial f}{\partial \mathbf{q}}\right\}^T \Delta\mathbf{q} \\ &= f(\boldsymbol{\sigma}^{tr}_{n+1}, \mathbf{q}_n) - \Delta\lambda \mathbf{a}^T \mathbf{D}^e \mathbf{b} + \Delta\lambda \mathbf{c}^T \mathbf{h} \end{aligned} \tag{6.21}$$

As a result, $\Delta\lambda$ can be obtained as follows

$$\Delta\lambda = \frac{f(\boldsymbol{\sigma}^{tr}_{n+1}, \mathbf{q}_n)}{\mathbf{a}^T \mathbf{D}^e \mathbf{b} - \mathbf{c}^T \mathbf{h}} \tag{6.22}$$

and the stress state and internal variables can be updated as follows

$$\left\{ \begin{array}{lcl} \boldsymbol{\sigma}^{st}_{n+1} & = & \boldsymbol{\sigma}^{tr}_{n+1} - \Delta\lambda \mathbf{D}^e \mathbf{b} \\ \mathbf{q}^{st}_{n+1} & = & \mathbf{q}_n + \Delta\lambda \mathbf{h} \\ \Delta\lambda^{st} & = & \Delta\lambda \end{array} \right. \tag{6.23}$$

This start state, $(\boldsymbol{\sigma}^{st}_{n+1}, \mathbf{q}^{st}_{n+1}, \Delta\lambda^{st})$, is called semi backward Euler starting point (e.g. Jeremić and Sture, 1997). By using this starting point to approximate $\boldsymbol{\sigma}_{n+1}$, the difference between the two stress states (or stress residual) can be written as

$$\mathbf{r}_\sigma = \boldsymbol{\sigma}_{n+1} - (\boldsymbol{\sigma}^{tr}_{n+1} - \Delta\lambda \mathbf{D}^e \mathbf{b}_{n+1}) \tag{6.24}$$

The stress residual for $j + 1$ iteration can be computed in terms of truncated first-order Tayler expansion,

$$(\mathbf{r}_\sigma)_{j+1} = (\mathbf{r}_\sigma)_j + \delta\boldsymbol{\sigma} + \delta\Delta\lambda \mathbf{D}^e \mathbf{b} + \Delta\lambda \mathbf{D}^e \left[\frac{\partial \mathbf{b}}{\partial \boldsymbol{\sigma}}\right] \delta\boldsymbol{\sigma} + \Delta\lambda \mathbf{D}^e \left[\frac{\partial \mathbf{b}}{\partial \mathbf{q}}\right] \delta\mathbf{q} \tag{6.25}$$

Because the objective of this iterative process is to minimize $(\mathbf{r}_\sigma)_{j+1}$ and $\delta\mathbf{q}$ can be given as

$$\delta\mathbf{q} = \delta\Delta\lambda \mathbf{h} \tag{6.26}$$

we get

$$\delta\boldsymbol{\sigma} = -\mathbf{Q}^{-1}[(\mathbf{r}_\sigma)_j + \delta\Delta\lambda \mathbf{D}^e \mathbf{t}] \tag{6.27}$$

in which, the expressions of $\mathbf{t}$ and $\mathbf{Q}$ have been given in Eq. (6.17). Parameter $\delta\Delta\lambda$ can be derived from the consistency condition

$$f_{j+1} = f_j + \mathbf{a}^T \delta\boldsymbol{\sigma} + \mathbf{c}^T \delta\mathbf{q} = f_j + \mathbf{a}^T \delta\boldsymbol{\sigma} + \delta\Delta\lambda \mathbf{c}^T \mathbf{h} \tag{6.28}$$

With the objective to minimize f_{j+1}, substituting Eq. (6.27) into Eq. (6.28) leads to

$$\delta\Delta\lambda = \frac{f_j - \mathbf{a}^T\mathbf{Q}^{-1}(\mathbf{r}_\sigma)_j}{\mathbf{a}^T\mathbf{R}\mathbf{t} - \mathbf{c}^T\mathbf{h}} \tag{6.29}$$

where the expression of $\mathbf{R}$ has been given in Eq. (6.17). Updating stress state and internal variables with $\delta\boldsymbol{\sigma}$ and $\delta\Delta\lambda$ as follows

$$\left\{\begin{array}{rcl} \boldsymbol{\sigma}_{n+1,j+1} & = & \boldsymbol{\sigma}_{n+1,j} + \delta\boldsymbol{\sigma} \\ \mathbf{q}_{n+1,j+1} & = & \mathbf{q}_{n+1,j} + \delta\mathbf{q} \\ \Delta\lambda_{n+1,j+1} & = & \Delta\lambda_{n+1,j} + \delta\Delta\lambda \end{array}\right. \quad \text{with} \quad \left\{\begin{array}{rcl} \boldsymbol{\sigma}_{n+1,0} & = & \boldsymbol{\sigma}^{st}_{n+1} \\ \mathbf{q}_{n+1,0} & = & \mathbf{q}^{st}_{n+1} \\ \Delta\lambda_{n+1,0} & = & \Delta\lambda^{st}_{n+1} \end{array}\right. \quad \text{and} \quad 0 \le j \le i \tag{6.30}$$

This iterative process is terminated until the final stress is “corrected” within a preset tolerance to the yield surface.

6.5 Modified Cam Clay Model

In three-dimensional stress space, the two-invariant yield function of the modified Cam clay model is given as

$$f = \frac{q^2}{M^2} + p'(p' - p'_c) \tag{6.31}$$

where M is the slope of the critical state line; p'_c, the so-called preconsolidation pressure, is the diameter of ellipsoid of modified Cam clay model and it is also the hardening parameter which controls the size of yield locus; p' is the mean effective stress and q is the deviatoric stress expressed, respectively, as

$$p' = -\frac{1}{3}\,tr(\boldsymbol{\sigma}) \tag{6.32a}$$

$$q = \sqrt{\frac{3}{2}}\,\|\mathbf{s}\| \quad \text{where} \quad \mathbf{s} = \boldsymbol{\sigma} + p'\mathbf{1} \tag{6.32b}$$

in which $\mathbf{s}$ is the stress deviator.

For modified Cam clay model with an associated plastic flow ($f = g$), we have

$$\mathbf{a} = \mathbf{b} = \frac{\partial f}{\partial \boldsymbol{\sigma}} = \frac{\partial f}{\partial p'}\frac{\partial p'}{\partial \boldsymbol{\sigma}} + \frac{\partial f}{\partial q}\frac{\partial q}{\partial \boldsymbol{\sigma}} \tag{6.33}$$

where

$$\frac{\partial f}{\partial p'} = 2p' - p'_c \tag{6.34a}$$

$$\frac{\partial f}{\partial q} = \frac{2q}{M^2} \tag{6.34b}$$

$$\frac{\partial f}{\partial p'_c} = -p' \tag{6.34c}$$

and $\dfrac{\partial p'}{\partial \boldsymbol{\sigma}}$, $\dfrac{\partial q}{\partial \boldsymbol{\sigma}}$ can be given as

$$\begin{aligned} \frac{\partial p'}{\partial \boldsymbol{\sigma}} &= -\frac{1}{3}\mathbf{1} && (6.35a) \\ \frac{\partial q}{\partial \boldsymbol{\sigma}} &= \sqrt{\frac{3}{2}}\,\hat{\mathbf{n}} = \frac{3}{2q}\mathbf{s} \quad \text{where} \quad \hat{\mathbf{n}} = \frac{\mathbf{s}}{\|\mathbf{s}\|} && (6.35b) \end{aligned}$$

Thus, Eq. (6.33) can be derived as

$$\mathbf{a} = \mathbf{b} = -\frac{1}{3}(2p' - p'_c)\,\mathbf{1} + \frac{3}{M^2}\,\mathbf{s} \tag{6.36}$$

Thus, the second derivative of f about stress is

$$\frac{\partial \mathbf{b}}{\partial \boldsymbol{\sigma}} = \frac{\partial^2 f}{\partial \boldsymbol{\sigma}^2} = -\frac{2}{3}\mathbf{1} \otimes \frac{\partial p'}{\partial \boldsymbol{\sigma}} + \frac{3}{M^2}\frac{\partial \mathbf{s}}{\partial \boldsymbol{\sigma}} = \frac{2}{9}\mathbf{1} \otimes \mathbf{1} + \frac{3}{M^2}\,\mathbb{I}_d \tag{6.37}$$

where $\mathbb{I}_d = \mathbf{I} - \mathbb{I}_v$ with $\mathbb{I}_v = \frac{1}{3}\mathbf{1} \otimes \mathbf{1}$, and the symbol "$\otimes$" denotes tensor product. In critical state soil mechanics, it is more convenient to distinguish the two components of plastic strain increment: the plastic volumetric strain increment $\mathrm{d}\varepsilon_v^p$ which governs isotropic hardening behavior and the plastic deviatoric strain increment $\mathrm{d}\varepsilon_d^p$ which governs deviatoric or kinematic hardening behavior. When only isotropic hardening plasticity is considered in modified Cam clay model, the hardening process dominated by the increasing p'_c is related to the plastic volumetric strain increment as follows

$$\mathrm{d}p'_c = \frac{p'_c v}{(\chi - \kappa)}\mathrm{d}\varepsilon_v^p = \mathrm{d}\lambda \vartheta p'_c \frac{\partial g}{\partial p'} \tag{6.38}$$

where $\vartheta = \dfrac{v}{\chi - \kappa}$, in which χ is the virgin compression index, κ is the swelling or recompression index and $v = 1 + e$ is the specific volume, while e is the void ratio of soil. The specific volume can be calculated from the following equation which is appropriate for normally consolidated soils under isotropic or anisotropic condition,

$$v = v_\chi - \chi \ln p'_c + \kappa \ln(p'_c/p') \tag{6.39}$$

Here, v_χ is the corresponding value of v at $\ln p' = 0$ or $p' = 1KPa$. Comparing Eq. (6.38) to Eq. (6.12), we have

$$\mathbf{h}(\boldsymbol{\sigma}, \mathbf{q}) = h = \vartheta p'_c(2p' - p'_c) \tag{6.40}$$

and thus, the expression of $\mathbf{t}$ in Eq. (6.17) can be

$$\mathbf{t} = \mathbf{b} + \frac{1}{3}\Delta\lambda h\,\mathbf{1} \tag{6.41}$$

It should be more convenient to formulate the elastic stress-strain matrix as

$$\mathbf{D}^e = K'\,\mathbf{1}\otimes\mathbf{1} + 2G'\mathbb{I}_d \tag{6.42}$$

where, for conventional soil models such as Mohr-Coulomb or von Mises soil models, the elastic bulk modulus K' and the elastic shear modulus G' can be given as

$$K' = \frac{E'}{3(1-2\nu')} \tag{6.43a}$$

$$G' = \frac{3(1-2\nu^{'})K'}{2(1+\nu^{'})} \tag{6.43b}$$

in which, E' is the effective Young's modulus and $\nu^{'}$ is the effective Poisson's ratio. However, for pressure-dependent soil models such as critical state soil models, the elastic stress-strain matrix is nonlinear with the elastic bulk modulus $K' = K'(p')$. For the second elastic parameter, there are several different choices (e.g. Potts and Zdravković, 1999; Hickman and Gutierrez, 2005), if the Poisson's ratio $\nu^{'}$ is assumed to be constant, the elastic bulk and shear modulus can be given as

$$\hat{K} = \frac{\mathrm{d}p'}{\mathrm{d}\varepsilon_v^e} = \frac{vp'}{\kappa} \tag{6.44a}$$

$$\hat{G} = \frac{3(1-2\nu^{'})\hat{K}}{2(1+\nu^{'})} \tag{6.44b}$$

in which $\mathrm{d}\varepsilon_v^e$ is the elastic volumetric strain increment. Eq. (6.44) gives tangent bulk and shear modulus, and integrating Eq. (6.44) results in the so-called secant bulk and shear modulus

$$\bar{K} = \frac{\Delta p'}{\Delta\varepsilon_v^e} = \frac{p_n'}{\Delta\varepsilon_v^e}\left[\exp\left(\frac{v}{\kappa}\Delta\varepsilon_v^e\right) - 1\right] \tag{6.45a}$$

$$\bar{G} = \frac{3(1-2\nu^{'})\bar{K}}{2(1+\nu^{'})} \tag{6.45b}$$

Therefore, if the secant bulk modulus are used to describe the elastic nonlinearity in critical state soil models, the elastic stress-strain matrix should be replaced with the secant elastic stress-strain matrix $\bar{\mathbf{D}}^e = \mathbf{D}^e(\bar{K},\bar{G})$, and the corresponding elastic predictor can be computed from

$$\Delta\boldsymbol{\sigma}_{n+1}^{tr} = \bar{\mathbf{D}}^e\Delta\boldsymbol{\varepsilon} \tag{6.46}$$

6.6 von Mises Model

The von Mises model without hardening can be expressed as

$$f = \sqrt{\frac{3}{2}}\|\mathbf{s}\| - \sigma_Y = q - q_Y \tag{6.47}$$

where without hardening, $\sigma_Y = q_Y$ is fixed, the expression of $\mathbf{s}$ has been shown in Eq. (6.32b). For associated plastic flow rule, the plastic flow can be obtained as

$$\mathbf{a} = \mathbf{b} = \frac{\partial q}{\partial \boldsymbol{\sigma}} = \sqrt{\frac{3}{2}}\,\hat{\mathbf{n}} = \frac{3}{2q}\mathbf{s} \quad \text{where} \quad \hat{\mathbf{n}} = \frac{\mathbf{s}}{\|\mathbf{s}\|} \tag{6.48}$$

and the correspondent second derivative of f about stress is

$$\frac{\partial \mathbf{b}}{\partial \boldsymbol{\sigma}} = \frac{\partial^2 f}{\partial \boldsymbol{\sigma}^2} = \frac{3}{2}\frac{q\left[\frac{\partial \mathbf{s}}{\partial \boldsymbol{\sigma}}\right] - \mathbf{s} \otimes \left\{\frac{\partial q}{\partial \boldsymbol{\sigma}}\right\}}{q^2} = \frac{3}{2q}\left(\mathbb{I}_d - \frac{3}{2q^2}\mathbf{s} \otimes \mathbf{s}\right) \tag{6.49}$$

It should be mentioned that without hardening, $\mathbf{t} = \mathbf{b}$, the consistent tangent matrix as shown in Eq. (6.19) is strictly symmetric.

6.7 Global Newton-Krylov Iteration in Finite Element Implementation

In strain-driven finite element analysis, it is required to calculate iteratively the displacement increment for each load increment or each time step. In each load increment, a series of linear systems can be derived from Newton linearization, the linear systems with the solid-fluid coupled stiffness matrices have the form

$$\mathbf{A}_{n+1,\,i} = \begin{bmatrix} \mathbf{K}_{n+1,\,i} & \mathbf{L} \\ \mathbf{L}^T & -\theta \Delta t \mathbf{G} \end{bmatrix} \tag{6.50}$$

in which $\mathbf{K}_{n+1,\,i}$ has been defined by Eq. (6.4), and to obtain good convergence rate, the consistent tangent matrix given in Eq. (6.19) is used, while the subscript i denotes the converged iteration counter of Newton method in each load increment. In Eq. (6.3), $\Delta\mathbf{f}$ is the residual vector or the out-of-balance force which can be defined as

$$\Delta\mathbf{f}_{n+1,i} = \sum_e \left(\int_V \mathbf{B}_u^T \Delta\boldsymbol{\sigma}^p_{n+1,i} dV\right) \tag{6.51}$$

in which $\Delta\boldsymbol{\sigma}^p$ defined at Gauss point can be computed from

$$\Delta\boldsymbol{\sigma}^p_{n+1,i} = (\boldsymbol{\sigma}_{n,i} + \Delta\boldsymbol{\sigma}^{tr}_{n+1,i}) - \boldsymbol{\sigma}_{n+1,i} \tag{6.52}$$

It has been recognized that there are two types of sources of out-of-balance forces: one is due to the yielding and stress correction, this out-of-balance force is normal and permitted. The other unacceptable source may be due to numerical problems such as very stiff structures locating in soft soil or inappropriate time step used for consolidation analysis (e.g. Woods, 1999).

In a more general form without bold symbol, the linear system as shown in Eq. (6.3) can be rewritten as

$$Ax = b \quad \text{with} \quad A = \begin{bmatrix} K & B \\ B^T & -C \end{bmatrix} \tag{6.53}$$

Here $K \in \mathbb{R}^{m\times m}$, $B \in \mathbb{R}^{m\times n}$ and $C \in \mathbb{R}^{n\times n}$ are correspondent to $\mathbf{K}$, $\mathbf{L}$ and $\theta\Delta t\mathbf{G}$, respectively. To solve such symmetric or weakly nonsymmetric indefinite linear system, Krylov subspace iterative method, SQMR, developed by Freund and Nachtigal (1994) is adopted because it is believed to be more appropriate for symmetric and possible indefinite linear systems. However, an efficient preconditioner is crucial. To combine with Newton-PCG method, Borja (1991) proposed to use the stiffness matrix at the first iteration of each time step as a preconditioner, that is,

$$P_B = P_{n+1,i} = A_{n+1,1} \tag{6.54}$$

and thus, there is one factorization in each time step and this factorization is used in the preconditioning steps of the following plastic iterations. This preconditioner was believed to be effective for symmetric or weakly nonsymmetric systems. However, a direct linear solver (In this chapter, sparse direct solution subroutine MA47 developed by Duff and Reid (1995) is adopted because the subroutine is well suited for saddle point matrices (e.g. Benzi *et al.*, 2005)) is needed to obtain the factorization, and for large-scale problems, it may be expensive to use direct solution method from the perspectives of memory requirement and computer runtime. A cheap and efficient diagonal generalized Jacobi (GJ) preconditioner was proposed by Phoon *et al.* (2002) for 3-D linear elastic soil consolidation with the form given in Eq. (5.1). Toh *et al.* (2004) and in Chapter 4 also proposed a so-called block constrained preconditioner which has the same block structure as that of original block matrix, and the preconditioner can lead to faster convergence and saving in computing time. Derived from block *LDU* factorization, Chapter 5 developed a modified SSOR preconditioner with the form given in Eq. 5.7. It should be noted that the MSSOR preconditioner falls into the class which can be used in conjunction with Eisenstat trick, the computing time can be reduced about half in each iteration (e.g. Eisenstat, 1981; Saad, 1996).

6.8 Numerical Examples

6.8.1 Convergence Criteria

In the following numerical examples, the relative residual convergence criterion is adopted for the linear iterative solver and the convergence criterion with zero initial guess is given as

$$\text{Iteration stops when } \ k \geq max\_it \quad \text{or} \quad \frac{\|r_k\|_2}{\|r_0\|_2} = \frac{\|b - Ax_k\|_2}{\|b\|_2} \leq stop\_tol \tag{6.55}$$

where $\|\cdot\|_2$ denotes the 2-norm, *max_it* and *stop_tol* are the maximum iteration number and stopping tolerance, respectively. In this study, $max\_it = 500$ and $stop\_tol = 10^{-2}$ are designated for SQMR iterative method. For nonlinear Newton iterative method, the convergence criterion is still based on the relative residual (unbalanced force) norm

$$\frac{\|\Delta\mathbf{f}_{n+1,i}\|_2}{\|\Delta\mathbf{f}_{n+1,1}\|_2} < nonlin\_tol \tag{6.56}$$

$nonlin\_tol = 10^{-2}$ is the corresponding stopping tolerance for the Newton iteration. Another tolerance, yield function tolerance to stop trial stress return, is set to be $Ytol = 10^{-6}$.

6.8.2 Problem Descriptions

Figure 6.1 shows a 8 × 8 × 8 finite element mesh of a quadrant symmetric shallow foundation (due to the symmetry property of the analyzed footing problem) applied by a ramp loading, it simulates a flexible square footing resting on homogeneous soil and a vertical ramp loading varies 0.01 MPa from 1st time step to 10th time step. In our 3-D finite element analysis, the twenty-node hexahedral solid elements coupled with eight-node fluid elements were used, and as a result, the dimension of each element stiffness matrix is 68 and the total number of DOFs of displacements is much larger than that of excess pore pressure.

The ground water table is assumed to be at the ground surface and is in hydrostatic condition at the initial stage. The base of the mesh is assumed to be fixed in all directions and impermeable, side face boundaries are constrained in the transverse direction, but free in in-plane directions (both displacement and water flux). The top surface is free in all direction and free-draining with pore pressures assumed to be zero. Two different soil models, normally consolidated modified Cam clay model and ideal von Mises model, are adopted for the nonlinear consolidation analysis. In fact, von Mises model in geotechnical analysis is applicable to "undrained clay" and only applicable in the context of total stress analysis, but here it is adopted just to test Newton-Krylov subspace iterative methods from a numerical viewpoint. The parameters required by the two models are listed in Table 6.1 and Table 6.2, respectively. The coefficient of permeability are same for two models as $k_x = k_y = k_z = 10^{-7} m/s$, and for the two models, the initial stress state is $\sigma'_x = \sigma'_y = \sigma'_z = 0.1MPa$ and $\tau_{xy} = \tau_{yz} = \tau_{zx} = 0$.

6.8.3 Numerical Results

Based on the $8 \times 8 \times 8$ finite element mesh, the performance of Newton-SQMR method preconditioned by P_B, P_{GJ} and P_{MSSOR}, respectively, are compared to Newton-MA47 method. The nonlinear iteration counts (Nonlinear ITs), the average iteration counts per nonlinear iteration and the total computer runtime for the whole nonlinear consolidation analysis are provided in Table 6.3 for modified Cam clay model and Table 6.4 for von Mises model. The "0" average iteration count per nonlinear iteration for Newton-MA47 means that sparse direct solution method, MA47, does not require iterations to solve a linear system. P_B preconditioner can be more efficient than P_{GJ} and P_{MSSOR} preconditioners from the perspective of reducing iteration counts of SQMR because the stiffness matrix at first plastic iteration is a good approximation to the stiffness matrices in following plastic iterations. However, as mentioned previously, the direct solver, MA47, is still used to obtain the factorization which could be prohibitive for large problems, and as a result, P_{GJ} and P_{MSSOR} can outperform P_B in total computer runtime. For a small problem such as this $8 \times 8 \times 8$ problem, P_B preconditioner still can compete with P_{GJ} or even P_{MSSOR} because the computing time spent on the factorization can be amortized by the fast convergence of SQMR methods, but the computation of factorization would become a bottleneck for P_B when problem size increases. A possible remedy for this problem is to use incomplete factorization as a preconditioner such as ILU(0) or ILUT (e.g. Saad, 1996) at the price of slowing down the convergence rate. Another obvious disadvantage for P_B preconditioner as well as ILU-type preconditioners is that in the following plastic iterations, the physical RAM should have the capability to store L, U factors and the stiffness matrix concurrently. For $8 \times 8 \times 8$ finite element mesh, the memory requirement for Newton-SQMR method preconditioned by P_B is 134 MBytes and the memory requirement for Newton-MA47 is a little less than 130 MBytes, but the memory requirement for P_{GJ} or P_{MSSOR} preconditioned methods is only 32 MBytes. For a larger $12 \times 12 \times 12$ as shown in Table 6.5 and 6.6, the computing is already impossible and the error "out of memory" may occur for P_B preconditioned SQMR method, and the memory requirement for Newton-MA47 method increases significantly to 670 MBytes, but the required memory for P_{GJ} and P_{MSSOR} preconditioned methods is only 102 MBytes. For the two different size problems, it can be seen that SQMR methods preconditioned by P_{GJ} and P_{MSSOR} should be more efficient as problem size increases.

It should be mentioned that when using Newton-Krylov methods, it is not necessary to set a stringent tolerance for *stop_tol* because the results provided by a loose tolerance are very close to those by direct solution method. This can be verified in Figure 6.2 by plotting settlements computed by different strategies.

6.9 Conclusions

Newton-Krylov methods in conjunction with efficient preconditioners have been used to large-scale nonlinear consolidation problems. In this thesis, three different preconditioners have been used and compared in nonlinear consolidation problems based on both modified Cam clay model and ideal von Mises model, and numerical examples show that Newton-Krylov methods have several advantages such as the less computer runtime and memory requirement than Newton-Gauss type method. It should be emphasized that a robust preconditioner is very crucial for Newton-Krylov methods, P_{GJ} and P_{MSSOR} perform far better than that suggested by Borja (1991), and thus, they are recommended for large-scale computations.

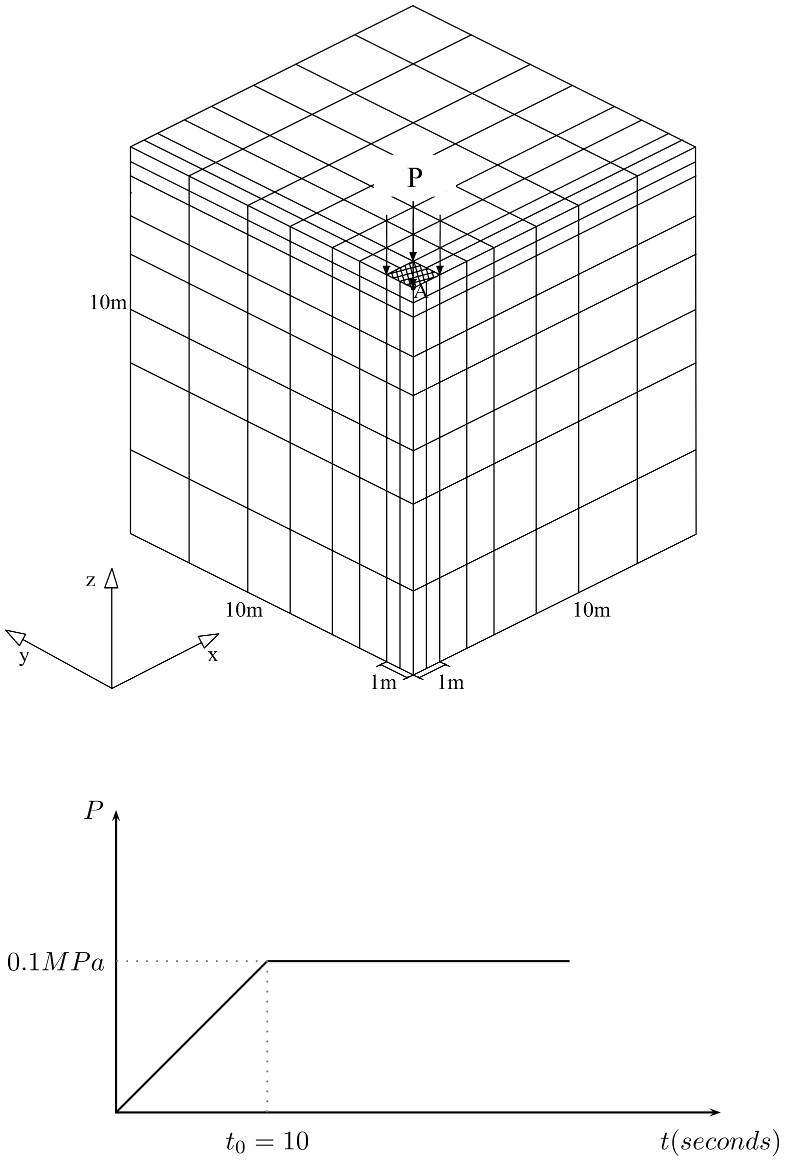
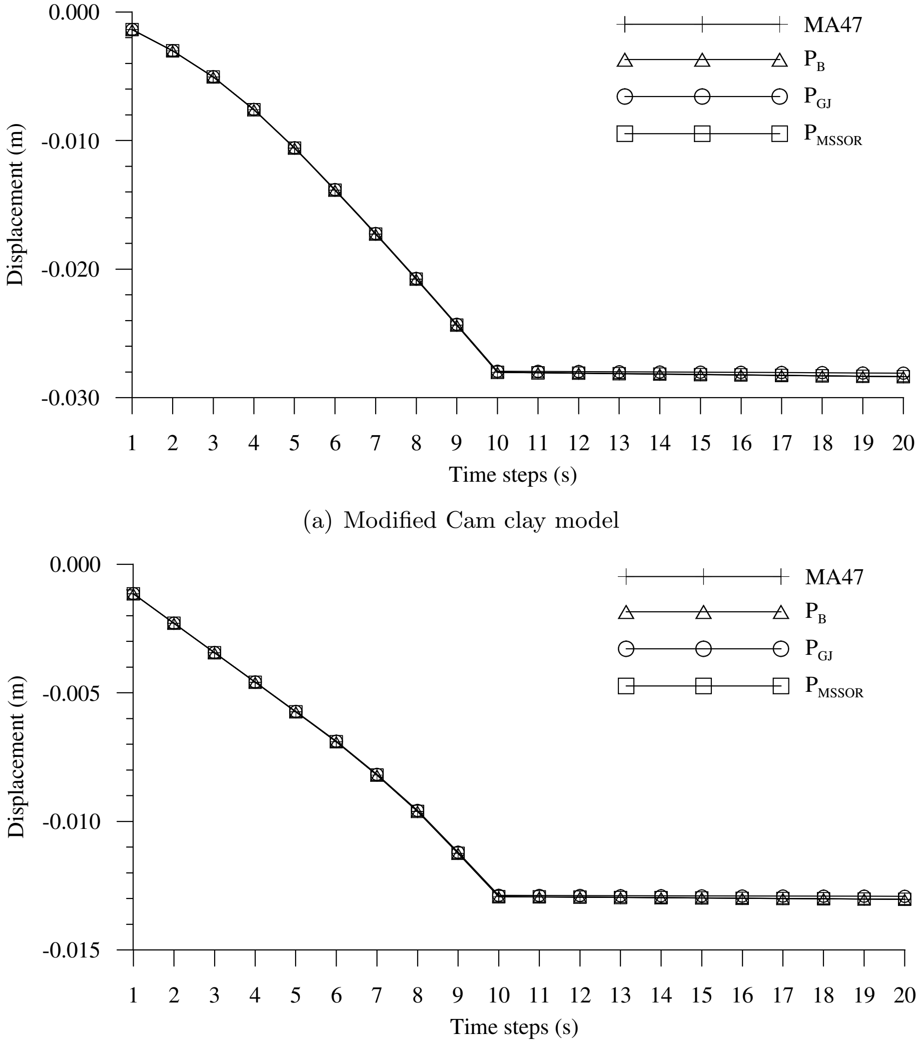


Figure 6.1: 3-D $8 \times 8 \times 8$ finite element mesh of a quadrant symmetric shallow foundation with ramp loading P.



(a) Modified Cam clay model

(b) von Mises model

Figure 6.2: Vertical settlements at point "A" with time steps.

Table 6.1: Parameters of modified Cam clay model.

Parameter	M	χ	κ	v_χ	$\nu^{'}$
Value	1.20	0.16	0.05	3.176	0.25

Table 6.2: Parameters of von Mises model.

Parameter	E'	$\nu^{'}$	σ_Y
Value	10 MPa	0.0	0.05 MPa

Table 6.3: Nonlinear consolidation analysis based on $8 \times 8 \times 8$ finite element mesh (DOFs = 7160) and modified Cam clay model.

		Average iteration counts per nonlinear iteration			
Time step	Nonlinear ITs	MA47	SQMR + P_B	SQMR + P_{GJ}	SQMR + P_{MSSOR}
1	5	0	2	151	42
2	6	0	2	152	44
3	8	0	2	95	26
4	9	0	2	91	24
5	10	0	2	139	23
6	11	0	2	139	24
7	12	0	2	137	24
8	13	0	2	150	24
9	13	0	2	140	26
10	14	0	2	141	24
Total runtime (s)		6924.4	1887.5	1827.8	1259.8

Table 6.4: Nonlinear consolidation analysis based on $8\times8\times8$ finite element mesh (DOFs = 7160) and von Mises model.

		Average iteration counts per nonlinear iteration			
Time step	Nonlinear ITs	MA47	SQMR + P_B	SQMR + P_{GJ}	SQMR + P_{MSSOR}
1	1	0	0	181	50
2	1	0	0	180	50
3	1	0	0	182	50
4	1	0	0	183	50
5	1	0	0	183	50
6	2	0	3	124	35
7	3	0	2.5	117	48
8	3	0	3	138	55
9	2	0	3	159	35
10	2	0	2	140	35
Total runtime (s)		1280.7	927.3	273.8	201.9

Table 6.5: Nonlinear consolidation analysis based on $12\times12\times12$ finite element mesh (DOFs = 23604) and modified Cam clay model.

		Average iteration counts per nonlinear iteration			
Time step	Nonlinear ITs	MA47	SQMR + P_B	SQMR + P_{GJ}	SQMR + P_{MSSOR}
1	5	0	–	214	69
2	6	0	–	195	40
3	8	0	–	112	39
4	9	0	–	153	34
5	11	0	–	177	32
6	11	0	–	172	57
7	12	0	–	176	58
8	13	0	–	170	58
9	13	0	–	174	51
10	14	0	–	205	36
Total runtime (s)		106283.0	–	6439.7	4653.2

Table 6.6: Nonlinear consolidation analysis based on 12 × 12 × 12 finite element mesh (DOFs = 23604) and von Mises model.

		Average iteration counts per nonlinear iteration			
Time step	Nonlinear ITs	MA47	SQMR + P_B	SQMR + P_{GJ}	SQMR + P_{MSSOR}
1	1	0	–	309	80
2	1	0	–	329	80
3	1	0	–	337	80
4	1	0	–	318	95
5	1	0	–	318	100
6	2	0	–	204	55
7	2	0	–	192	58
8	3	0	–	220	83
9	3	0	–	234	95
10	3	0	–	214	75
Total runtime (s)		21983.0	–	1273.2	813.5

CHAPTER 7

CONCLUSIONS & FUTURE WORK

7.1 Summary and Conclusions

The solution of the linear systems of equations is the most time-consuming part in large-scale engineering problems. For example, finite element discretization of time-dependent or elastic-plastic problems can lead to a large number of linear systems with hundreds of thousands of DOFs. The developments of efficient iterative solution methods as well as efficient preconditioning methods are therefore of importance to large-scale finite element analysis. In this thesis, efficient preconditioned iterative methods were investigated and used for large-scale linear systems arising from 3-D Biot's consolidation equations. The linear system up to about 200,000 DOFs were solved successfully on a modest personal computer. To accelerate the convergence of SQMR method, two recently developed preconditioners, GJ and P_c, were compared in detail and then, a robust preconditioning technique named as MSSOR was proposed for large symmetric indefinite linear systems arising from 3-D Biot's consolidation equations. One important and noteworthy practical advantage of MSSOR is that it can handle heterogeneous soils better than previous preconditioners.

This thesis is organized as follows. In Chapter 1, the advances of preconditioned iterative methods in geotechnical engineering were described, and then the objective and significance of this thesis was given. In Chapter 2, a brief review of iterative methods, preconditioning techniques and data storage schemes were introduced from a historical perspective. In Chapter 3, several iterative solution strategies including partitioned iterative methods and global Krylov subspace iterative methods were investigated and compared after FE discretization and time integration of 3-D Biot's consolidation. In Chapter 4, block constrained preconditioner, P_c, which was already studied by Toh *et al.* (2004), was investigated and then compared with the recently developed GJ preconditioner. The numerical results show that P_c can further lead to computer runtime saving. From the observation of a theoretical block factorization, a modified block SSOR preconditioner was proposed in Chapter 5. However, a cheap pointwise modified SSOR preconditioner, P_{MSSOR}, was strongly recommended for practical applications. Both theoretical and numerical results demonstrated that P_{MSSOR} is very robust when combining with Eisenstat trick. In Chapter 6, the proposed preconditioners were suggested in conjunction with Newton-Krylov methods for nonlinear consolidation problems based on the modified Cam clay soil model and the von Mises soil model, respectively. Numerical studies showed that the proposed preconditioned iterative methods far outperformed some popular solution methods. Based on the above studies, some concluding remarks were given in this concluding chapter.

In more details, these useful concluding remarks can be summarized as follows:

(a) Iterative methods only need matrix-vector products and inner products which make them quite suitable for large, especially sparse, linear systems arising from partial differential equations by finite element method discretization. For large-scale linear systems of equations, iterative methods should be more efficient than direct solution methods in terms of computer runtime and memory usage. It is not easy to decide on a threshold DOF limit beyond which iterative method is more efficient than direct solution method, because there are many efficient direct solution methods and preconditioned iterative methods and one method may achieve fast solution (runtime efficiency) but may not be memory efficient. However, based on the author's experiences on linear solvers for 3-D consolidation problems, it is recommended that direct solution method can be used when total DOFs may fall below 10,000, otherwise, preconditioned iterative methods should be used from the perspective of memory efficiency and fast runtime. As for consolidation problems, theoretical and numerical evidences show that SQMR should be used with a symmetric indefinite preconditioner. Preconditioning methods are very crucial for iterative methods to be successful. A good preconditioner has the desirable properties: the preconditioned linear system has a faster convergence rate than the original one, but the computing time incurred by the construction of the preconditioner and preconditioning steps must be compensated by the resultant faster convergence rate. Moreover, the construction of a good preconditioner should minimize the memory usage. Many numerical applications show that diagonal and SSOR-type preconditioners are very simple to implement and very efficient (in memory storage and computing time) for most cases. Thus, diagonal and SSOR-type preconditioners should be used as basic preconditioners to evaluate other more complicated preconditioners. At present, an EBE version of the MSSOR preconditioner is not available and GJ has an edge in a parallel computing environment.

(b) For an iterative method, the data storage scheme is another important factor to be considered. A brief comparison between compressed sparse storage and element-by-element storage was given in Chapter 2. Although element-by-element storage could be extremely memory efficient, the compressed sparse storage was chosen in this thesis because in symmetric case, the compressed sparse storage can be kept as almost same as that of EBE storage and matrix-vector products can be carried out more efficiently in the compressed sparse storage (See Section 2.3 and Appendix B.1.2 for the detail). It is believed that element-by-element storage may be more suitable for parallel computers than ordinary desktop computers. Sometimes, it may be difficult to decide which storage to choose. However, it should be kept in mind that in the iterative process, when the coefficient matrix is stored in compressed sparse storage, the matrix-vector products can be carried out directly to achieve fast solution time. To optimize memory efficiency (no need to assemble the global matrix) or parallel operations, the matrix-vector products should be carried out element-by-element, which is slower. Hence, there is a trade-off between memory and runtime.

(c) For symmetric indefinite linear systems with 2×2 block coefficient matrix, partitioned iterative methods or global Krylov subspace iterative methods can be used. From the investigations and comparisons given in Chapter 3, it is clear that global iterative methods can be more efficient when combining with efficient preconditioners than partitioned iterative methods. For indefinite Biot's linear systems, the indefinite preconditioners can significantly improve the convergence of an iterative method (including PCG, MINRES and SQMR) compared to the corresponding positive definite ones. For a certain preconditioner, MINRES can lead to slightly faster convergence than PCG and SQMR. It is interesting to notice that numerical results shown that PCG and MINRES can be successfully used with symmetric indefinite preconditioners for Biot's linear systems though theoretically, PCG is restricted to SPD linear systems and MINRES is restricted to combine with SPD preconditioner. If theoretical guarantee is desired, SQMR is the only iterative method which is applicable to symmetric indefinite linear systems in conjunction with an arbitrary symmetric preconditioner. Theoretical guarantee is desirable because it ensures generality. Therefore, the SQMR was adopted for symmetric indefinite linear systems arising from Biot's consolidation equations.

(d) As a further study of block constrained preconditioner which has been investigated by Toh *et al.* (2004), a P_c preconditioner with diagonal approximation to (1, 1) block was proposed and compared to the GJ preconditioner (e.g. Phoon *et al.*, 2002, 2003). When applying P_c preconditioner, it has been shown that it is more beneficial to express the inverse of P_c explicitly and then by following the efficient implementations given by Toh *et al.* (2004), each iteration of P_c preconditioned SQMR can be performed efficiently. Numerical results based on a series of finite element meshes showed that P_c can lead to about 40% computer runtime compared to GJ preconditioner.

(e) A modified SSOR (MSSOR) preconditioner was developed in this thesis. In the theoretical perspective, a modified block SSOR derived from a LDU factorization was proposed for the first time, however, this factorized form could be believed to be expensive. A cheap pointwise variant of the modified block SSOR preconditioner can be derived to combine with Eisenstat trick so that each iteration of MSSOR preconditioned method can be performed as cheap as that of unpreconditioned one. In practical application, MSSOR has been proved to be very robust for large-scale Biot's consolidation problems with highly varied soil properties, and perform much better than recently developed GJ and P_c preconditioners. This improvement can be explained that the diagonal approximation of K in GJ or P_c is a bad approximation to soil stiffness K when heterogeneous soils with highly varied soil properties are involved although it is cheap. MSSOR was thus developed to overcome this problem as well as to overcome the numerical difficulties encountered by standard SSOR preconditioner. The numerical problem with standard SSOR preconditioner may result from the small entries in the diagonal, and these small entries may lead to numerical instability and convergence difficulty. Therefore, the modified SSOR preconditioner can be expected to be efficient for large-scale computation involving multi-phases. MSSOR is as cheap as a diagonal preconditioner (cheapest possible) when preconditioning step and matrix-vector product is combined by the Eisenstat trick.

(f) For large nonlinear consolidation problems, the Newton-Krylov iterative methods has been presented and several preconditioners including GJ, MSSOR and P_B (see Borja, 1991) combined with Newton-SQMR method have been compared. Newton-Krylov methods have been chosen because they have the obvious advantage that the linear systems in each linearized iteration need not be solved with high accuracy. This conclusion can be verified by comparing the numerical results given by Newton-Gauss (Newton-MA47) method and Newton-SQMR. It has been shown that the GJ and MSSOR are also readily adapted to the plastic iterations involved in each nonlinear iteration (or time step). The two preconditioners gave excellent numerical performance compared to the P_B preconditioner.

7.2 Future Work

To give a closure of this thesis, some suggestions of future work on preconditioned iterative methods can be given for large-scale linear systems arising from Biot's consolidation equations:

(a) MSSOR preconditioner was proposed in sparse storage and sparse implementation, but the MSSOR can be explored to combine with element-by-element strategy in parallel computing. The extension is very natural because the element-by-element SSOR and GJ were both developed (e.g. Dayde *et al.*, 1997; Phoon *et al.*, 2002), and these strategies can be borrowed directly, but the possibly encountered difficulty is how to carry out triangular solves efficiently at element level. The performance of MSSOR in parallel computing should be worth investigating.

(b) The other efficient preconditioning techniques such as multigrid preconditioner or domain decomposition preconditioner can be studied and used for large linear systems of Biot's consolidation problems on parallel environment or on a serial computer.

(c) The linear systems discretized from Biot's consolidation equations were all symmetric indefinite or weakly nonsymmetric indefinite. However, they may be nonsymmetric or strongly nonsymmetric in some cases, for example, if soil model with non-associated plastic flow rule or unsaturated soil model (e.g. Sheng *et al.*, 2003) is considered, the resultant linear systems could be strongly nonsymmetric. Some efficient preconditioning techniques should be developed, and the performance of MSSOR preconditioner which can be extended naturally to nonsymmetric cases is also worth studying. As we mentioned above, the memory storage requirement can be kept almost as same as that required by EBE technique for symmetric linear system by using the symmetric compressed sparse storage strategy proposed in this thesis. Although, it can be predicted that by using the same global matrix assembly strategy, for nonsymmetric linear system but with symmetric nonzero structure, the memory requirement is about 1.5 times of that required by EBE technique, and for nonsymmetric linear with nonsymmetric nonzero structure, the memory requirement is about 2.0 times of that required by EBE technique. The practical applicability of compressed sparse storage scheme is also worth investigating for nonsymmetric linear systems.

REFERENCES

Abbo, A.J. Finite Element Algorithms for Elastoplasticity and Consolidation. Ph.D Thesis, University of Newcastle. 1997.

Ashby, S.F., T.A. Manteuffel, P.E. Saylor. A Taxonomy for Conjugate Gradient Methods, SIAM Journal on Numerical Analysis, 27(6):pp.1542–1568. 1990.

Axelsson, O. A Survey of Preconditioned Iterative Methods for Linear Systems of Algebraic Equations, BIT, 25:pp.166–187. 1985.

Axelsson, O. Iterative Solution Methods. Cambridge: Cambridge University Press. 1994.

Axelsson, O., R. Blaheta, R. Kohut. Inexact Newton Solvers in Plasticity: Theory and Experiments, Numerical Linear Algebra With Applications, 4:pp133–152. 1997.

Barrett, R., M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H.A. van der Vorst. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. Philadelphia: SIAM Press. 1994.

Beauwens, R. Iterative Solution Methods, Applied Numerical Mathematics, 51:pp.437–450. 2004.

Benzi, M., C.D. Meyer, M. Tůma. A Sparse Approximate Inverse Preconditioner for the Conjugate Gradient Method, SIAM Journal on Scientific Computing, 17:pp.1135–1149. 1996.

Benzi, M., M. Tůma. A Sparse Approximate Inverse Preconditioner for Nonsymmetric Linear Systems, SIAM Journal on Scientific Computing, 19(3):pp.968–994. 1998.

Benzi, M., M. Tůma. A Comparative Study of Sparse Approximate Inverse Preconditioners, Applied Numerical Mathematics, 30:pp.305–340. 1999.

Benzi, M., J.K. Cullum, M. Tůma. Robust Approximate Inverse Preconditioning for the Conjugate Gradient Method, SIAM Journal on Scientific Computing, 22(4):pp.1318–1332. 2000.

Benzi, M. Preconditioning Techniques for Large Linear Systems: A Survey, Journal of Computational Physics, 182:pp.418–477. 2002.

Benzi, M., G.H. Golub, J. Liesen. Numerical Solution of Saddle Point Problems, Acta Numerica, 14:pp.1–137. 2005.

Bergamaschim, L., J. Gondzio, G. Zilli. Preconditioning Indefinite Systems in Interior Point Methods for Optimization, Computational Optimization and Applications, 28:pp.149–171. 2004.

Biot, M.A. General Theory of Three-Dimensional Consolidation, Journal of Applied Physics, 12:pp.155–164. 1941.

Blaheta, R. Displacement Decomposition-Incomplete Factorization Preconditioning Techniques for Linear Elasticity, Numerical Linear Algebra With Applications, 1:pp.107–128. 1994.

Blaheta, R., R. Kohut. Solution of Large Nonlinear Systems in Elastoplastic Incremental Analysis, Journal of Computational and Applied Mathematics, 63:pp255–264. 1995.

Blaheta, R., P. Byczanski, O. Jakl, J. Starý. Space Decomposition Preconditioners and Their Application in Geomechanics, Mathematics and Computers in Simulation, 61:pp.409–420. 2003.

Boer R. de, Highlights in the Historical Development of the Porous Media Theory: Toward a Consistent Macroscopic Theory, Applied Mechanics Reviews, 49:pp.201–262. 1996.

Borja, R.I., S.R. Lee. Cam-Clay Plasticity, Part I: Implicit Integration of Elasto-Plastic Constitutive Relations, Computer Methods in Applied Mechanics and Engineering, 78:pp.49–72. 1990.

Borja, R.I. Composite Newton-PCG and Quasi-Newton Iterations for Nonlinear Consolidation, Computer Methods in Applied Mechanics and Engineering, 86:pp.27–60. 1991.

Borja, R.I. Cam-Clay Plasticity, Part II: Implicit Integration of Constitutive Equation Based on a Nonlinear Elastic Stress Predictor, Computer Methods in Applied Mechanics and Engineering, 88:pp.225–240. 1991.

Braess, D. Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics. Cambridge; New York: Cambridge University Press, 2001. 2nd ed.

Bramble, J.H. and J.E. Pasciak. Iterative Techniques for Time Dependent Stokes Problems, Computers and Mathematics with Applications, 33:pp.13–30. 1997.

Brezzi F. and M. Fortin. Mixed and Hybrid Finite Element Methods. Springer-Verlag, New York, 1991.

Briggs, W.L., V.E. Henson, S.F. McCormick. A Multigrid Tutorial. Philadelphia, PA: Society for Industrial and Applied Mathematics, c2000. 2nd ed.

Britto, A.M. and M.J. Gunn. Critical State Soil Mechanics via Finite Elements. Chichester: Ellis Horwood. 1987.

Bruaset, A.M. A Survey of Preconditioned Iterative Methods, Pitman Research Notes In Mathematics Series, Longman Scientific & Technical. 1995.

Bruaset, A.M. Krylov Subspace Iterations for Sparse Linear Systems. In:Morten Dæhlen, Aslak Tveito, (Eds) Numerical Methods and Software Tools in Industrial Mathematics. Birkhäuser, Boston, 21. 1997.

Bruaset, A.M. and H.P. Langtangen. Object-oriented Design of Preconditioned Iterative Methods in Diffpack, ACM Transactions on Mathematical Software(TOMS), 23:pp.50–80. 1997.

Cao Z.H. Fast Uzawa Algorithm for Generalized Saddle Point Problems, Applied Numerical Mathematics, 46:pp.157–171. 2003.

Chan, T.F., H.A. van der Vorst. Approximate and Incomplete Factorizations, Technical Report 871, Department of Mathematics, University of Utrecht, 1994.

Chan, T.F., W.P. Tang and W.L. Wan. Wavelet Sparse Approximate Inverse Preconditioners, BIT, 3:644–660. 1997.

Chan, T.F., E. Chow, Y. Saad, M.C. Yeung. Preserving Symmetry in Preconditioned Krylov Subspace Methods, SIAM Journal on Scientific Computing, 20:pp.568–581. 1998.

Chan, S.H. Iterative Solution for Large-scale Indefinite Linear Systems from Biot's Finite Element Formulation. Ph.D Thesis, National University of Singapore. 2002.

Chan, S.H., K.K. Phoon and F.H. Lee. A Modified Jacobi Preconditioner for Solving Ill-Conditioned Biot's Consolidation Equations Using Symmetric Quasi-Minimal Residual Method, International Journal for Numerical and Analytical Methods in Geomechanics, 25:pp.1001–1025. 2001.

Chen C.-N. Efficient and Reliable Accelerated Constant Stiffness Algorithms for the Solution of Non-linear Problems, International Journal for Numerical Methods in Engineering, 35:pp.481–490. 1992.

Chen, K. Discrete Wavelet Transforms Accelerated Sparse Preconditioners for Dense Boundary Element Systems, Electronic Transactions on Numerical Analysis, 8:pp.138–153. 1999.

Chen, K. An Analysis of Sparse Approximate Inverse Preconditioners for Boundary Integral Equations, SIAM Journal on Matrix Analysis and Applications, 22:pp.1058–1078. 2001.

Chen, X.J. On Preconditioned Uzawa Methods and SOR Methods for Saddle-Point Problems, Journal of Computational and Applied Mathematics, 100:pp.207–224. 1998.

Chow, E. Robust Preconditioning for Sparse Linear Systems. Ph.D Thesis, University of Minnesota. 1997.

Chow, E., Y. Saad. Approximate Inverse Techniques for Block-Partitioned Matrices, SIAM Journal on Scientific Computing, 18:pp.1657-1675. 1997.

Chow, E., Y. Saad. Experimental Study of ILU Preconditioners for Indefinite Matrices, Journal of Computational and Applied Mathematics, 86:pp.387–414. 1997.

Chow, E., Y. Saad. Approximate Inverse Preconditioners via Sparse-Sparse Iterations, SIAM Journal on Scientific Computing, 19(3):pp.995–1023. 1998.

Chow, E., M.A. Heroux. An Object-Oriented Framework for Block Preconditioning, ACM Transactions on Mathematical Software (TOMS), 24:pp.159–183, 1998.

Coughran W.M., R. W. Freund. Recent Advances in Krylov-Subspace Solvers for Linear Systems and Applications in Device Simulation, Proceedings of the 1997 International Conference on Simulation of Semiconductor Processes and Devices, IEEE, pp.9–16. 1997.

Crisfield, M.A. Non-Linear Finite Element Analysis of Solids And Structures, Vol.1: Essentials. New York: Wiley & Sons, 1991.

Crisfield, M.A. Non-Linear Finite Element Analysis of Solids And Structures, Vol.2: Advanced Topics. New York: Wiley & Sons, 1997.

Cui, M.R. Analysis of Iterative Algorithms of Uzawa Type for Saddle Point Problems, Applied Numerical Mathematics, 50:pp.133–146. 2004.

Davis, T.A., P. Amestoy, I.S. Duff. An Approximate Minimum Degree Ordering Algorithm, Technical Report TR-94-039, Computer and Information Science Department, University of Florida, Gainesville, FL. 1994.

Dayde, M.J., J.Y. L'Excellent, Gould N.I.M. Element-By-Element Preconditioners for Large Partially Separable Optimization Problems, SIAM Journal on Scientific Computing, 18(6):pp.1767–1787. 1997.

Dembo, R.S., S.C. Eisenstat, T. Steihaug. Inexact Newton Methods, SIAM Journal on Numerical Analysis, 19(2):pp.400–408. 1982.

Demmel, J.W. Applied Numerical Linear Algebra. Philadelphia: Society for Industrial and Applied Mathematics. c1997.

de Sturler, E. Truncation Strategies for Optimal Krylov Subspace Methods, SIAM Journal on Numerical Analysis, 36(3):pp.864–889. 1999.

Dongarra, J.J., I.S. Duff, D.C. Sorensen, H.A. van der Vorst. Numerical Linear Algebra for High-Performance Computers. Philadelphia: SIAM, 1998.

Driscoll, T.A. A MATLAB Toolbox for Schwarz-Christoffel Mapping, ACM Transactions on Mathematical Software, 22:pp.168–186. 1996.

Duff, I.S., J.K. Reid. The Multifrontal Solution of Indefinite Sparse Symmetric Linear Equations, ACM Transactions on Mathematical Software (TOMS), 9:pp.302–325. 1983.

Duff, I.S., G.A. Meurant. The Effect of Ordering on Preconditioned Conjugate Gradients, BIT, 29:pp.635–657, 1989.

Duff, I.S., J.K. Reid. MA47, A Fortran Code for Direct Solution of Sparse Symmetric Linear Systems of Equations. Report RAL-95-001, Rutherford Appleton Laboratory, Oxfordshire. 1995.

Duff, I.S., H.A. van der Vorst. Preconditioning and Parallel Preconditioning, RAL Technical Reports, RAL-TR-1998-052, 1998.

Duff, I.S., S. Pralet. Strategies for Scaling and Pivoting for Sparse Symmetric Indefinite Problems. RAL Technical Reports, RALTR 2004020, CSE, CCLRC. 2004.

Eijkhout, V. On the Existence Problem of Incomplete Factorisation Methods, Lapack Working Note 144, UT–CS–99–435. 1999.

Eijkhout, V. The ‘Weighted Modification’ Incomplete Factorisation Method, Lapack Working Note 145, UT–CS–99–436, 1999.

Eijkhout, V. A Bluffer’s Guide to Iterative Methods for Systems of Linear Equations. 2003.

Eisenstat, S.C. Efficient Implementation of A Class of Preconditioned Conjugate Gradient Methods, SIAM Journal on Scientific and Statistical Computing, 2(1):pp.1–4. 1981.

Eisenstat, S.C., H.F. Walker. Choosing the Forcing Terms in An Inexact Newton Method, SIAM Journal on Scientific Computing, 17(1):pp.16–32. 1996.

Ekevid, T.,P. Kettil, N.-E. Wiberg. Adaptive Multigrid for Finite Element Computations in Plasticity, Computers and Structures, 82(28):pp.2413–2424. 2004.

Elman, H.C., G.H. Golub. Inexact and Preconditioned Uzawa Algorithms for Saddle Point Problems, SIAM Journal on Numerical Analysis, 31(6):pp.1645–1661. 1994.

Elman, H.C., D. Silvester, A.J. Wathen. Iterative Methods for Problems in Computational Fluid Dynamics, in Iterative Methods in Scientific Computing, Chan R., Chan T.F. and Golub G. (Eds), Springer-Verlag, 1997.

Elman, H.C., D.J. Silvester, A.J. Wathen. Finite Elements and Fast Iterative Solvers, Numerical Mathematics and Scientific Computation, Oxford University Press, 2005.

Erhel, J. Iterative Solvers for Large Sparse Linear Systems. Seminar, University of Neuchâtel, Swiss, August 2001.

Ferronato, M., G. Gambolati, P. Teatini. Total Stress and Pressure Gradient Formulations in Coupled Poroelasticity Problems. In J.C. Roegiers et al. (eds.), Proceedings of the International Symposium on Coupled Phenomena in Civil, Mining, and Petroleum Engineering, 1:pp.101–109, 2000. Sanya, Hainan Island.

Ferronato, M., G. Gambolati, P. Teatini. Ill-Conditioning of Finite Element Poroelasticity Equations, International Journal of Solid and Structure, 38:pp.5995–6014. 2001.

Fischer, B. Polynomial Based Iteration Methods for Symmetric Linear Systems. Chichester; New York: Wiley; Stuttgart: Teubner , c1996.

Fischer, B., A. Ramage, D.J. Silvester, A.J. Wathen. Minimum Residual Methods for Augmented Systems, BIT, 38:pp.527–543. 1998.

Fletcher, R. Conjugate Gradient Methods for Indefinite Systems, Lecture Notes In Mathematics, Vol.506, Springer Verlag, pp.73–89. 1976.

Fokkema, D.R., G.L.G. Sleijpen, H.A. van der Vorst. Accelerated Inexact Newton Schemes for Large Systems of Nonlinear Equations, SIAM Journal on Scientific Computing, 19(2):pp.657–674. 1998.

Freund, R.W., N.M. Nachtigal. QMR: A Quasi-Minimal Residual Method for Non-Hermitian Linear Systems, Numerical Mathematics, 60:pp.315–339. 1991.

Freund, R.W., G.H. Golub, N.M. Nachtigal. Iterative Solution of Linear Systems, Acta Numerica, 1:pp.57–100. 1992.

Freund, R.W. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems, SIAM Journal on Scientific Computing, 14(2):pp.470–482. 1993.

Freund, R.W., N.M. Nachtigal. A New Krylov-Subspace Method for Symmetric Indefinite Linear System, In Proceedings of the 14th IMACS World Congress on Computational and Applied Mathematics; Atlanta, USA, pp.1253–1256. 1994.

Freund, R.W., N.M. Nachtigal. An Implementation of the QMR Method Based on Coupled Two-Term Recurrences, SIAM Journal on Scientific Computing,15(2):pp.313–337. 1994.

Freund, R.W., N.M. Nachtigal. Software for Simplified Lanczos and QMR algorithms, Applied Numerical Mathematics, 19:pp.319–341, 1995.

Freund, R.W., F. Jarre. A QMR-based Interior-Point Algorithm for Solving Linear Programs, Mathematical Programming, 76:pp.183–210. 1996.

Freund, R.W. Preconditioning of Symmetric, But Highly Indefinite Linear Systems, In Proceedings of 15th IMACS World Congress on Scientific Computation Modelling and Applied Mathematics, 25-29 August, Berlin, Germany, pp.551–556. 1997.

Gambolati, G., G. Pini, M. Ferronato. Numerical Performance of Projection Methods in Finite Element Consolidation Models, International Journal for Numerical and Analytical Methods in Geomechanics, 25:pp.1429–1447. 2001.

Gambolati, G., G. Pini, M. Ferronato. Direct, Partitioned and Projected Solution to Finite Element Consolidation Models, International Journal for Numerical and Analytical Methods in Geomechanics, 26:pp.1371–1383. 2002.

Gambolati, G., G. Pini, M. Ferronato. Scaling Improves Stability of Preconditioned CG-like Solvers for FE Consolidation Equations, International Journal for Numerical and Analytical Methods in Geomechanics, 27:pp.1043–1056. 2003.

Gens, A., D.M. Potts. Critical State Models in Computational Geomechanics, Engineering Computations, 5:pp178–197. 1988.

George, A. Nested Dissection of A Regular Finite Element Mesh, Tech. Rep. STAN-CS-208, Standard University, 1971.

George, A., J.W.H. Liu. Computer Solution of Large Sparse Positive Definite Systems. Prentice Hall, Englewood Cliffs, NJ, 1981.

George, A., J.W.H. Liu. The Evaluation of the Minimum Degree Ordering Algorithm. SIAM Review, 31(1):1–19, 1989.

Golub, G.H., C.F. Van Loan. Matrix Computations, 3rd edn. Baltimore: The John Hopkins University Press. 1996.

Golub G.H., H.A. van der Vorst, "Closer to the Solution: Iterative Linear Solvers", in: I.S. Duff and G.A. Watson (eds), The State of the Art in Numerical Analysis (Clarendon Press, Oxford), pp.63, 1997.

Golub, G.H., A.J. Wathen. An Iteration for Indefinite Systems and Its Application to the Navier-Stokes Equations, SIAM Journal on Scientific Computing, 19:pp.530–539, 1998.

Golub, G.H., X. Wu, J.Y. Yuan. SOR-like Methods for Augmented Systems, BIT, 41:pp.071–085. 2001.

Gould, N.I.M., J.A. Scott. Sparse Approximate-Inverse Preconditioners Using Norm-Minimization Techniques. SIAM Journal on Scientific Computing, 19(2):pp.605–625, 1998.

Gravvanis, G.A. Generalized Approximate Inverse Finite Element Matrix Techniques, Neural Parallel and Scientific Computations, 7:pp.487–500. 1999.

Greenbaum, A. Iterative Methods for Solving Linear Systems. Philadelphia: SIAM. 1997.

Griffiths, D.V., I.M. Smith. Numerical Methods for Engineers: A Programming Approach. Blackwell, Oxford, 1991.

Grote, M.J., T. Huckle. Parallel Preconditioning With Sparse Approximate Inverses, SIAM Journal on Scientific Computing, 18:pp.838–853. 1997.

Gustafsson, I. A Class of First Order Factorization Methods, BIT, 18:142–156. 1978.

Haber, E., U. Ascher. Preconditioned All-At-Once Methods for Large, Sparse Parameter Estimation problems, Inverse Problems, 17:pp.1847–1864. 2001.

Hackbusch W. Iterative solution of large sparse systems of equations. New York: Springer-Verlag, 1994.

Hashash, Y.M.A., A.J. Whittle. Integration of The Modified Cam-Clay Model in Non-Linear Finite Element Analysis, Computers and Geotechnics, 14:pp.59–83. 1992.

Hestenes, M.R., E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems, Journal of Research of the National Bureau of Standards, 49:pp.409–436. 1952.

Hickman, R.J., M. Gutierrez. An Internally Consistent Integration Method for Critical State Models, International Journal for Numertical and Analytical Methods in Geomechanics, 29:pp.227–248. 2005.

Hladík I. Reed M.B. Swoboda G. Robust Preconditioners for Linear Elasticity FEM Analyses, International Journal for Numerical Methods in Engineering, 40:pp.2109–2127. 1997.

Hu, Q., J. Zou. An Iterative Method with Variable Relaxation Parameters for Saddle-Point Problems, SIAM Journal on Matrix Analysis and Applications, 23:pp.317–338. 2001.

Huckle, T. Approximate Sparsity Patterns for the Inverse of A Matrix and Preconditioning, Applied Numerical Mathematics, 30:pp.291–303. 1999.

Jeremić, B., S. Sture. Implicit Integration in Elastoplastic Geotechnics, Mechanics of Cohesive-Frictional Materials, 2:pp.165–183. 1997.

Jones, J.E., C.S. Woodward. Newton-Krylov-Multigrid Solvers for Large-Scale, Highly Heterogeneous, Variably Saturated Flow Problems. Advances in Water Resources, 24:pp.763–774. 2001.

Kayupov, M.A., V.E. Bulgakov, G. Kuhn. Efficient Solution of 3-D Geomechanical Problems by Indirect BEM Using Iterative Methods, International Journal for Numerical and Analytical Methods in Geomechanics, 22:pp.983–1000. 1998.

Keller, C., Gould, N.I.M., Wathen, A.J. Constraint Preconditioning for Indefinite Linear Systems, SIAM Journal on Matrix Analysis and Applications, 21(4):pp.1300–1317. 2000.

Kelley, C.T. Iterative Methods for Linear and Nonlinear Equations, Philadelphia: Society for Industrial and Applied Mathematics, 1995.

Klawonn, A. Preconditioners for Indefinite Problems. PhD thesis, Universitat Munster. 1995.

Krabbenhøft K. Basic Computational Plasticity. Lecture Notes, 2002.

Langtangen, H.P. Computational Partial Differential Equations: Numerical Methods and Diffpack Programming. New York: Springer, 1999.

Lee, F.H., K.K. Phoon, K.C. Lim, S.H. Chan. Performance of Jacobi Preconditioning in Krylov Subspace Solution of Finite Element Equations, International Journal for Numerical and Analytical Methods in Geomechanics, 26:pp.341–372. 2002.

Lee, H.C., A.J. Wathen. On Element-by-Element Preconditioning for General Elliptic Problems, Computer Methods in Applied Mechanics and Engineering, 92:pp.215–229. 1991.

Lee, L.Q., J.G. Siek, A. Lumsdaine. Generic Graph Algorithms for Sparse Matrix Ordering, Third International Symposium ISCOPE 99, volume 1732 of Lecture Notes in Computer Science, Springer-Verlag. 1999.

Lewis, R.W., B.A. Schrefler. The Finite Element Method in the Static and Dynamic Deformation and Consolidation of Porous Media, 2nd edn. Chichester: John Wiley & Sons. 1998.

Lipitakis, E.A., G.A. Gravvanis. Explicit Preconditioned Iterative Methods for Solving Large Non-symmetric Finite Element Systems, Computing, 54:167–183. 1995.

Lipnikov, K. Numerical Methods for The Biot Model in Poroelasticity. Ph.D Thesis, The Faculty of the Department of Mathematics, University of Houston. 2002.

Little, L., Y. Saad, L. Smoch. Block LU Preconditioners for Symmetric and Nonsymmetric Saddle Point Problems, SIAM Journal on Scientific Computing, 25(2):pp.729–748. 2003.

Liu, J.W.H. Modification of the Minimum-Degree Algorithm by Multiple Elimination, ACM Transaction on Mathematical Software, 11:pp.141–153. 1985.

Liu, M.D., J.P. Carter. A Structured Cam Clay Model, Canadian Geotechnical Journal, 39:pp.1313–1332. 2002.

Lukšan, L., J. Vlček. Indefinitely Preconditioned Inexact Newton Method For Large Sparse Equality Constrained Non-linear Programming Problems, Numerical Linear Algebra with Applications, 5:pp.219–247. 1998.

Lukšan, L., J. Vlček. Conjugate gradient methods for saddle point systems, Technical Report, V-778, ICS AS CR, 1999.

Lukšan, L., J. Vlček. Numerical experience with iterative methods for equality constrained nonlinear programming problems, Technical Report, DMSIA 22/2000.

MATLAB, Software, Version 6.5 Release 13. Mathworks. 2002.

Matthies, H., G. Strang. The Solution of Nonlinear Finite Element Equations, International Journal for Numerical Methods in Engineering, 14:pp.1613–1626. 1979.

Meerbergen, K., D. Roose. The Restarted Arnoldi Method Applied to Iterative Linear System Solvers for the Computation of Rightmost Eigenvalues, SIAM Journal on Matrix Analysis and Applications, 18(1):pp.1–20, 1997.

Meijerink, J.A., H.A. van der Vorst. An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix Is a Symmetric M-Matrix, Mathematics of Computation, 31:pp.148–162. 1977.

Meijerink, J.A., H.A. van der Vorst. Guidelines for the Usage of Incomplete Decompositions in Solving Sets of Linear Equations As They Occur in Practical Problems, Journal of Computational Physics, 44:pp.134–155. 1981.

Meurant G.A. Computer Solution of Large Linear Systems. Amsterdam; New York: North-Holland: Elsevier, 1999.

Mitchell, J.A., J.N. Reddy. A Multilevel Hierarchical Preconditioner for Thin Elastic Solids, International Journal for Numerical Methods in Engineering, 43:pp.1383–1400. 1998.

Mroueh, H., I. Shahrour. Use of Sparse Iterative Methods for the Resolution of Three-dimensional Soil/structure Interaction Problems, International Journal for Numerical and Analytical Methods in Geomechanics, 23:pp.1961–1975. 1999.

Mroueh, H., I. Shahrour. A Full 3-D Finite Element Analysis of Tunneling-Adjacent Structures Interaction, Computers and Geotechnics, 30:pp.245–253. 2003.

Murphy, M.F., G.H. Golub, A.J. Wathen. A Note on Preconditioning for Indefinite Linear Systems. SIAM Journal on Scientific Computing, 21(6):1969–1972. 2000.

Nayak, G.C., O.C. Zienkiewicz. Note On The 'Alpha'-Constant Stiffness Method for The Analysis of Non-Linear Problems, International Journal for Numerical Methods in Engineering, 4:pp.579–582. 1972.

Nayak, G.C., O.C. Zienkiewicz. Elasto-Plastic Stress Analysis. A Generalization For Various Constitutive Relations Including Strain Softening, International Journal for Numerical Methods in Engineering, 5:pp.113–135. 1972.

Neytcheva, M.G., P.S. Vassilevski. Preconditioning of Indefinite and Almost Singular Finite Element Elliptic Equations, SIAM Journal on Scientific Computing, 19(5):pp.1471-1485. 1998.

Ng, E.G., BW. Peyton. Block Sparse Cholesky Algorithms on Advanced Uniprocessor Computers, SIAM Journal on Scientific Computing, 14:pp.1034–1056. 1993.

Ortiz, M., E.P. Popov. Accuracy And Stability of Integration Algorithms for Elastoplastic Constitutive Relations, International Journal for Numerical Methods in Engineering, 21:pp.1561–1576. 1985.

Ortiz, M., J.C. Simo. An Analysis of a New Class of Integration Algorithms for Elastoplastic Constitutive Relations, International Journal for Numerical Methods in Engineering, 23:pp.353–366. 1986.

Paige, C.C., M.A. Saunders. Solution of Sparse Indefinite Systems of Linear Equations, SIAM Journal on Numerical Analysis, 12(4):pp.617–629. 1975.

Payer, H.J., H.A. Mang. Iterative Strategy for Solving Systems of Linear, Algebraic Equations Arising in 3D BE-FE Analyses of Tunnel Drivings, Numerical Linear Algebra with Applications, 4(3):239–268. 1997.

Perugia, I., V. Simoncini. Block-Diagonal and Indefinite Symmetric Preconditioners for Mixed Finite Element Formulations, Numerical Linear Algebra with Applications, 7(7-8):pp.585–616. 2000.

Phoon, K.K., K.C. Toh, S.H. Chan, F.H. Lee. An Efficient Diagonal Preconditioner for Finite Element Solution of Biot's Consolidation Equations, International Journal for Numerical Methods in Engineering, 55:pp.377–400. 2002.

Phoon, K.K., S.H. Chan, K.C. Toh, F.H. Lee. Fast Iterative Solution of Large Undrained Soil-structure Interaction Problems, International Journal for Numerical and Analytical Methods in Geomechanics, 27:pp.159–181. 2003.

Phoon, K.K. Iterative Solution of Large-Scale Consolidation and Constrained Finite Element Equations for 3D Problems, Proceedings, International e-Conference on Modern Trends in Foundation Engineering: Geotechnical Challenges and Solutions, IIT Madras, India, Jan 26-30, 2004.

Pini, G., G. Gambolati. Is A Simple Diagonal Scaling the Best Preconditioner for Conjugate gradients on Supercomputers? Adv. Water Resources, Vol.13, No.3. 1990.

Pommerell C., W. Fichtner. PILS: An Iterative Linear Solver Package for Ill-conditioned Systems, Proceedings of the 1991 ACM/IEEE conference on Supercomputing, pp.588–599, November 18-22, 1991, Albuquerque, New Mexico, United States.

Pommerell, C. Solution of Large Unsymmetric Systems of Linear Equations, volume 17 of Series in Micro-electrons, Hartung-Gorre Verlag, Konstanz. 1992.

Potts, D.M., L. Zdravković. Finite Element Analysis in Geotechnical Engineering. Thomas Telford, London, 1999.

Press, W.H., B.P. Flannery, S.A. Teukolsky and W.T. Vetterling. Numerical Recipes. Cambridge: Cambridge University Press. 1986.

Prevost, J.H. Implicit-Explicit Schemes for Nonlinear Consolidation, Computer Methods in Applied Mechanics and Engineering, 39:pp.225–239. 1983.

Prevost, J.H. Partitioned Solution Procedure for Simultaneous Integration of Coupled-Field Problems, Communication in Numerical Methods in Enginnering, 13:pp.239–247. 1997.

Ribeiro, F.L.B., A.L.G.A. Coutinho. Comparison Between Element, Edge and Compressed Storage Schemes for Iterative Solutions in Finite Element Analyses, International Journal for Numerical Methods in Engineering, 63:pp.569–588. 2005.

Rozložník, M., V. Simoncini. Krylov Subspace Methods for Saddle Point Problems With Indefinite Preconditioning, SIAM Journal on Matrix Analysis and Applications, 24:pp.368–391. 2002.

Rusten, T., R. Winther. A Preconditioned Iterative Method for SaddlePoint Problems, SIAM Journal on Matrix Analysis and Applications. 13(3):pp.887–904. 1992.

Saad, Y., M.H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems, SIAM Journal on Scientific and Statistical Computing, 7:pp.856–869. 1986.

Saad, Y. SPARSKIT–A Basic Tool-Kit for Sparse Matrix Computations, Version 2, June 1994.

Saad, Y. ILUT: A Dual Threshold Incomplete LU Factorization, Numerical Linear Algebra With Applications, 1:387–402. 1994.

Saad, Y. Iterative Methods for Sparse Linear Systems. PWS Publishing Company, Boston. 1996.

Saad, Y. ILUM: A Multi-Elimination ILU Preconditioner for General Sparse Matrices, SIAM Journal on Scientific Computing, 17(4):pp.830–847. 1996.

Saad, Y., H.A. van der Vorst. Iterative Solution of Linear Systems in the 20th Century, Journal of Computational and Applied Mathematics, 123:pp.1–33. 2000.

Saad, Y. Iterative methods for sparse linear systems. 2nd ed. Philadelphia: SIAM, c2003.

Sandhu, R.S., E.L. Wilson. Finite Element Analysis of Seepage In Elastic Media, Journal for Engineering Mechanics, ASCE, 95:pp.641–652. 1969.

Sarin, V., A.H. Sameh. An Efficient Iterative Method for the Generalized Stokes problem, SIAM Journal on Scientific Computing, 19(1):pp.206–226. 1998.

Sheng, D., S.W. Sloan, H.S. Yu. Aspects of Finite Element Implementation of Critical State Models, Computational Mechanics, 26:pp.185–196. 2000.

Sheng D., S.W. Sloan. Load Stepping Schemes for Critical State Models, International Journal for Numerical Methods in Eingeering, 50:pp.67–93. 2001.

Sheng D., S.W. Sloan, A.J. Abbo. An Automatic Newton-Raphson Scheme, The International Journal of Geomechanics, 2:pp.471–502. 2002.

Sheng Daichao, S.W. Sloan, A. Gens, D.W. Smith, Finite Element Formulation and Algorithms for Unsaturated Soils. Part I: Theory. International Journal for Numerical and Analytical Methods in Geomechanics. 27:pp.745–765, 2003.

Silvester, D.J., A.J. Wathen. Fast Iterative Solution of Stabilised Stokes Systems Part II: Using General Block Preconditioners, SIAM Journal on Numerical Analysis, 31(5):pp.1352–1367. 1994.

Silvester, D.J., H. Elman, D. Kay, A.J. Wathen. Efficient Preconditioning of the Linearized Navier-Stokes Equations, Numuerical Analysis Report No.352, Manchester Centre for Computational Mathematics, 1999.

Simo, J.C., R.L. Taylor. Consistent Tangent Operators for Rate-Independent Elastoplasticity, Computer Methods in Applied Mechanics and Engineering, 48:pp.101–118. 1985.

Simoncini, V. On the Convergence of Restarted Krylov Subspace Methods, SIAM Journal on Matrix Analysis and Applications,22(2):pp.430–452. 2000.

Simoncini V., D.B. Szyld. Recent developments in Krylov Subspace Methods for linear systems. 2005.

Siriwardane, H.J., C.S. Desai. Two Numerical Schemes for Nonlinear Consolidation, International Journal for Numerical Methods in Engineering, 17:pp.405–426. 1981.

Sleijpen G.L.G., H.A. van der Vorst. Krylov Subspace Methods for Large Linear Systems of Equations. Preprint 803, Department of Mathematics, University Utrecht, 1993.

Sloan, S.W. Substepping Schemes for The Numerical Integration of Elastoplastic Stress-Strain Relations, International Journal for Numerical Methods in Engineering, 24:pp.893–911. 1987.

Sloan S.W., A.J. Abbo. Biot Consolidation Analysis With Automatic Time Stepping And Error Control. Part 1:Theory And Implementation, International Journal for Numerical and Analytical Methods in Geomechanics, 23:pp.467-492. 1999.

Sloan,S.W., D. Sheng, A.J. Abbo. Accelerated Initial Stiffness Methods for Elastoplasticity, International Journal for Numerical and Analytical Methods in Geomechanics, 24:pp.579–599. 2000.

Sloan, S.W., A.J. Abbo, D. Sheng. Refined Explicit Integration of Elastoplastic Models With Automatic Error Control, Engineering Computations, 18:pp.121–154. 2001.

Smith, I.M., D.V. Griffiths. Programming the Finite Element Method, 3rd ed., John Wiley & Sons. 1998.

Smith, I.M., A. Wang. Analysis of Piled Rafts, International Journal for Numerical and Analytical Methods in Geomechanics, 22(10):pp.777–790. 1998.

Smith, I.M. A General Purpose System for Finite Element Analyses in Parallel, Engineering Computations, 17(1):pp.75–91. 2000.

Smith, I.M., D.V. Griffiths. Programming the Finite Element Method, 4th ed., John Wiley & Sons. 2004.

Sonneveld P. CGS, A Fast Lanczos-Type Solver for Nonsymmetric Linear Systems, SIAM Journal on Scientific and Statistical Computing, 10:pp.36–52. 1989.

Sullivan, F. The Joy of Algorithms, Computing in Science and Engineering, 02(1):pp.22–79. 2000.

Thomas, J.N. An Improved Accelerated Initial Stress Procedure For Elasto-Plastic Finite Element Analysis, International Journal for Numerical and Analytical Methods in Geomechanics, 8:pp.359–379. 1984.

Toh, K.C. GMRES vs. Ideal GMRES, SIAM Journal on Matrix Analysis and Applications. 18(1):pp.30–36. 1997.

Toh, K.C., M. Kojima. Solving Some Large Scale Semidefinite Programs via the Conjugate Residual Method, SIAM Journal on Optimization, 22:pp.669–691. 2002.

Toh, K.C. Solving Large Scale Semidefinite Programs via an Iterative Solver on the Augmented Systems, SIAM Journal on Optimization, 14:pp.670–698. 2003.

Toh, K.C., K.K. Phoon, S.H. Chan. Block Preconditioners for Symmetric Indefinite Linear Systems, International Journal for Numerical Methods in Engineering, 60:pp.1361–1381. 2004.

Toh, K.C., K.K. Phoon. Further Observations on Generalized Jacobi Preconditioning for Iterative Solution of Biot's FEM Equations, Working Paper, Department of Mathematics, National University of Singapore, 2005.

Trefethen, L.N., DIII. Bau. Numerical Linear Algebra. SIAM, Philadelphia, PA, 1997.

van den Eshof, J., Sleijpen, G.L.G., van Gijzen, M.B. Iterative Linear Solvers with Approximate Matrix-Vector Products, Preprint 1293, Dep. Math., University Utrecht, 2003.

van der Vorst, H.A. 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:pp.631–644. 1992.

van der Vorst, H.A. Krylov Subspace Iteration, Computing in Science and Engineering, 2:pp.32–37. 2000.

van der Vorst, H.A. Iterative Krylov Methods for Large Linear Systems. Cambridge University Press, Cambridge, 2003.

Vermeer, P.A., R. de Borst. Non-Associated Plasticity For Soils, Concrete And Rock. Heron 29:1, 1984.

Wang, A. Three Dimensional Finite Element Analysis of Pile Groups and Piled-Rafts. Ph.D. Thesis, University of Manchester. 1996.

Wang, Y.Q. Preconditioning for The Mixed Formulation of Linear Plane Elasticity. Ph.D Thesis, Texas A&M University, 2004.

Warsa, J.S., M. Benzi, T.A. Wareing, J.E. Morel. Preconditioning A Mixed Discontinuous Finite Element Method for Radiation Diffusion, Numerical Linear Algebra With Applications, 99:pp.1–18. 2002.

Wathen, A.J. An Analysis of Some Element-by-Element Techniques, Computer Methods in Applied Mechanics and Engineering, 74:pp.271–287. 1989.

Wathen, A.J., D.J. Silvester. Fast Iterative Solution of Stabilised Stokes Systems Part I: Using Simple Diagonal Preconditioners, SIAM Journal on Numerical Analysis, 30(3):pp.630-649. 1993.

Wathen, A.J., B. Fischer, D. Silvester. The Convergence of Iterative Methods for Symmetric and Indefinite Linear Systems, in: D.F. Griffiths, D.J. Higham, G.A. Watson (Eds.), Numerical Analysis 1997, Longman Scientific, pp.230–243. 1997.

Wathen, A.J. Preconditioning and Fast Solvers for Incompressible Flow. Numerical Analysis Group Research Reports, Oxford University Computing Laboratory, 2004.

Winnicki, A. Radial Return Algorithms In Computational Plasticity. 2001.

Wood, D.M. Soil Behaviour And Critical State Soil Mechanics. Cambridge University Press: Cambridge, England, New York: 1990.

Woods, R., A. Rahim. SAGE-CRISP Technical Reference Manual. Version 4. 1999.

Young David M. Iterative Methods for Solving Partial Difference Equations of Elliptic Type. Ph.D Thesis, Harvard University. 1950.

Zaman, M., J.R. Booker, G. Gioda. (ed) Modeling in Geomechanics. pp.23–50, New York: John Wiley & Sons. 2000.

Zhang, J. On Preconditioning Schur Complement and Schur Complement Preconditioning, Electronic Transactions of Numerical Analysis, 10:115–130. 2000.

Zienkiewicz, O.C., S. Valliappan, I.P. King. Elasto-Plastic Solutions of Engineering Problems Initial Stress', Finite Element Approach, International Journal for Numerical Methods in Engineering, 1:pp.75–100. 1969.

Zienkiewicz, O.C., C. Humpheson, R.W. Lewis. Associated and Non-Associated Visco-Plasticity and Plasticity in Soil Mechanics, Geotechnique, 25:pp.671–689, 1975.

Zienkiewicz, O.C., J.P. Vilotte,S. Toyoshima, S. Nakazawa. Iterative Method for Constrained and Mixed Approximation, An Inexpensive Improvement of F.E.M. Performance, Computer Methods in Applied Mechanics and Engineering, 51:pp.3–29. 1985.

Zienkiewicz, O.C., A.H.C. Chan, B.A. Schrefler, T. Shiomi. Computational Geomechanics With Special Reference to Earthquake Engineering. Chichester: John Wiley & Sons. 1998.

Zienkiewicz, O.C., R.L. Taylor. The Finite Element Method, 5th edn, Oxford: Butterworth Heinemann. 2000.

APPENDIX A

SOME ITERATIVE ALGORITHMS AND CONVERGENCE CRITERIA

A.1 Algorithms for PCG and SQMR

Algorithm 1 Preconditioned CG Method (e.g. Erhel, 2001)

CG algorithm for symmetric positive definite system $Ax = b$, using a symmetric preconditioner M.

```
Start: choose an initial guess $x_0$,
Set $r_0 = b - Ax_0$; $z_0 = M^{-1}r_0$; $p_0 = z_0$.
for $k = 0$ to $max\_it$ do
    $q_k = Ap_k$
    $\alpha_k = (r_k, z_k)/(q_k, p_k)$
    $x_{k+1} = x_k + \alpha_k p_k$
    $r_{k+1} = r_k - \alpha_k q_k$
    Check convergence
    $z_{k+1} = M^{-1}r_{k+1}$
    $\beta_{k+1} = (r_{k+1}, z_{k+1})/(r_k, z_k)$
    $p_{k+1} = z_{k+1} + \beta_{k+1}p_k$
end for
```

Algorithm 2 Preconditioned SQMR Method (e.g. Freund and Nachtigal, 1994; Freund, 1997)

The Symmetric QMR algorithm for the symmetric system $Ax = b$, using a symmetric preconditioner $M = M_L M_R$.

```
**Start:** choose an initial guess $x_0 \in \mathbb{R}^n$, then set
$s_0 = b - Ax_0, \;\; t = M_L^{-1} s_0, \;\; q_0 = M_R^{-1} t, \; \tau_0 = \|t\|_2, \; \vartheta_0 = 0, \; \rho_0 = s_0^T q_0, \; d_0 = 0.$
**for** $k = 1$ to $max\_it$ **do**
    **Compute**
    $t = Aq_{k-1}, \;\; \sigma_{k-1} = q_{k-1}^T t, \;\; \alpha_{k-1} = \frac{\rho_{k-1}}{\sigma_{k-1}}, \;\; s_k = s_{k-1} - \alpha_{k-1} t$
    **Compute**
    $t = M_L^{-1} s_k, \;\; \vartheta_k = \frac{\|t\|_2}{\tau_{k-1}}, \; c_k = \frac{1}{\sqrt{1 + \vartheta_k^2}}, \; \tau_k = \tau_{k-1} \vartheta_k c_k,$
    $d_k = c_k^2 \vartheta_{k-1}^2 d_{k-1} + c_k^2 \alpha_{k-1} q_{k-1}$
    **Set**
    $x_k = x_{k-1} + d_k,$
    **Check convergence**
    **Compute**
    $u_k = M_R^{-1} t, \;\; \rho_k = s_k^T u_k, \;\; \beta_k = \frac{\rho_k}{\rho_{k-1}}, \;\; q_k = u_k + \beta_k q_{k-1}.$
**end for**
```

A.2 Convergence Criteria for Iterative Methods

In iterative algorithms, an essential component is convergence criterion or stopping criterion by which we can determine when to stop the iteration process with acceptable approximate solution.

In general, a convergence criterion is applied in the following iterative framework:

```
**for** $k = 1$ to $max\_it$ (user-supplied maximum iterations), **do**
    **Compute** current approximate solution, $x_k$;
    **Compute** current residual, $r_k = b - Ax_k$, if necessary;
    **Apply** convergence criteria with *stop_tol*
            (convergence tolerance) until convergence.
**end for**
```

A good convergence criterion should be capable of identifying the true error, $e_k = x - x_k$, indirectly by other quantities, such as residual vector which is inherently available for many iterative methods (However, there is no guarantee that a small residual indicates a small error, especially for very ill-conditioned system (e.g. Bruaset, 1997). Iteration process stops when it is identified as convergence and controls the maximum iteration time. There are several popular convergence criteria as follows:

(a) **Relative residual norm criterion**

Iteration continues until

$$R_r = \frac{\|r_k\|}{\|r_0\|} = \frac{\|b - Ax_k\|}{\|b - Ax_0\|} \leq stop\_tol, \quad k = 1, 2, \ldots, max\_it. \tag{A.1}$$

where x_0 is the initial guess of the solution, and $\|\cdot\|$ represents any vector norm, but 2-norm is commonly used. Because of $r_k = Ae_k$ or $e_k = A^{-1}r_k$, the criterion has the following error bound:

$$\|e_k\| \leq \|A^{-1}\| \cdot \|r_k\| \leq stop\_tol \cdot \|A^{-1}\| \cdot \|r_0\|$$

The obvious disadvantage of the above criterion is that convergence strongly depends on the initial guess, x_0. In some applications, the choice of previous step solution for initial guess may significantly reduce the total computational time. However, when solving a time-dependent problem for which the current time-step solution can be provided as the initial guess for the next time stop, the relative residual norm criterion may have to become stricter with time. An absolute criterion by testing whether the residual has dropped below a threshold. When there is no good initial guess available, the following more stricter convergence criterion (i.e., set $x_0 = 0$ for criterion R_r) can be adopted,

$$R_b = \frac{\|r_k\|_2}{\|b\|_2} = \frac{\|b - Ax_k\|_2}{\|b\|_2} \leq stop\_tol, \quad k = 1, 2, \ldots, max\_it. \tag{A.2}$$

It should be mentioned that when the "penalty" technique is used to implement the prescribed displacements or the prescribed pore water pressures, the right hand vector b has to be modified with "big" numbers. This modification makes R_r or R_b be satisfied even though $\|r_k\|_2$ is still very large. In this situation, the true residual norm $\|r_k\|_2$ should also be checked.

(b) **Relative 'improvement' norm criterion**

This convergence criterion has been used with PCG iterative method in many geotechnical applications (e.g. Smith and Griffiths, 1998; Smith and Wang, 1998;

Smith, 2000; Smith and Griffiths, 2004), in which the stopping criterion is defined as:

Iteration continues until

$$R_i = \frac{\|x_k - x_{k-1}\|_\infty}{\|x_k\|_\infty} \leq stop\_tol, \quad k = 1, 2, \ldots, max\_it. \tag{A.3}$$

where $\|\cdot\|_\infty$ represents the infinity norm.

(c) **Relative error energy-norm criterion**

Iteration continues until

$$R_e = \frac{\|e_k\|_A}{\|e_0\|_A} \leq stop\_tol, \quad k = 1, 2, \ldots, max\_it.$$

As we mentioned in section (2.1.1), error vector e_k can not be evaluated directly due to unavailable exact solution, thus direct computing of value R_e is impossible. Notice that (e.g. Mitchell and Reddy, 1998; Lee *et al.*, 2002)

$$\frac{\|e_k\|_A}{\|e_0\|_A} \leq 2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k \leq stop\_tol, \quad k = 1, 2, \ldots, max\_it.$$

where $\kappa(A)$ denotes the condition number of matrix A and is computed by

$$\kappa(A) = \|A\|_2 \cdot \|A^{-1}\|_2 = \frac{\sigma_{max}}{\sigma_{min}} \tag{A.4}$$

where σ_{max} and σ_{min} represent the maximum and the minimum singular values, respectively. When matrix A is symmetric and positive definite, its condition number is

$$\kappa(A) = \frac{\lambda_{max}}{\lambda_{min}} \tag{A.5}$$

Where λ_{max} and λ_{min} denote the maximum and the minimum eigenvalues, respectively. Clearly, It is natural to choose R_e as

$$R_e = 2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k \leq stop\_tol, \quad k = 1, 2, \ldots, max\_it. \tag{A.6}$$

(d) **R$_4$ criterion**

Iteration continues until

$$R_4 = \frac{\|r_k\|}{\|A\| \cdot \|x_k\| + \|b\|} \leq \mathit{stop\_tol}, \quad k = 1, 2, \ldots, \mathit{max\_it}. \tag{A.7}$$

This criterion yields the forward error bound

$$\|e_k\| \leq \|A^{-1}\| \cdot \|r_k\| \leq \mathit{stop\_tol} \cdot \|A^{-1}\| \cdot (\|A\| \cdot \|x_k\| + \|b\|)$$

(e) **R$_5$ criterion**

Iteration stops until

$$R_5 = \frac{\|A^{-1}\| \cdot \|r_k\|}{\|x_k\|} \leq \mathit{stop\_tol}, \quad k = 1, 2, \ldots, \mathit{max\_it}. \tag{A.8}$$

which guarantees that $\dfrac{\|e_k\|}{\|x_k\|} \leq \mathit{stop\_tol}$.

(f) **R$_6$ and R$_7$ criteria**

For CG iterative method, the following criteria are recommended as (e.g. Bruaset, 1997; van der Vorst, 2003),

R_6 convergence criterion:

$$\|r_k\|_2 \leq \lambda_n \cdot \|x_k\|_2 \cdot \frac{\varepsilon}{\varepsilon + 1} \tag{A.9}$$

Where $\varepsilon = \mathit{stop\_tol}$, and λ_n still denotes smallest eigenvalue. This criterion ensures that $\|x - x_k\|_2 / \|x_k\|_2$ is bounded by ε. If the Ritz value, $\lambda_n^{(k)}$, is computed to approximate λ_n (for the procedure of eigenvalues approximated by Ritz values (refer to Bruaset, 1997; Bruaset and Langtangen, 1997), we get the rather robust convergence criterion,

R_7 convergence criterion:

$$\|r_k\|_2 \leq \lambda_n^{(k)} \cdot \|x_k\|_2 \cdot \frac{\varepsilon}{\varepsilon + 1} \tag{A.10}$$

It is obvious that R_e and R_{4-7} all require additional estimation or computation such as eigenvalue (singular number) or matrix norm. Thus they are not used widely in FEM programming like R_r and R_i criteria. The behaviors of relative residual, relative improvement and relative energy error criteria (R_r, R_i and R_e) have been compared and analyzed in several geotechnical applications by Lee *et al.* (2002). Another problem one may encounter is that for some problems such as consolidation problem, there are two or more types of degree-of-freedoms with significantly different magnitude. The set of degrees-of-freedom with large numerical values may overpower the set with smaller numerical values so that the convergence check is dominated by the large numerical values. Therefore, in a practical FEM analysis, it is recommended to use two or more convergence criteria to mitigate this undesirable situation. (see reference Barrett *et al.*, 1994; Langtangen, 1999, for more details) on convergence criteria of iterative methods.

APPENDIX B

SPARSE MATRIX TECHNIQUES

Standard discretization of partial differential equations (PDEs) by finite element or finite difference method typically leads to large sparse stiffness matrices because each node is only connected with a small number of neighboring nodes which indicates that there are a small number of entries in each row of stiffness matrix. Therefore, for large linear systems discretized from PDEs of 3-D problems, sparsity should be exploited for preconditioned iterative methods. By storing only nonzero entries of a matrix, the required storage and arithmetic operations can both be reduced more or less depending on the sparsity ratio.

B.1 Storage Schemes for Sparse Matrix

B.1.1 Some Popular Storages

Sparse storage schemes allocate continuous storage space in memory for the nonzero entries and perhaps a limited number of zeros. Saad (1996) and Barrett *et al.* (1994) described many methods for storing the data, for example, Compressed Sparse Row (CSR) storage, Compressed Sparse Column (CSC) storage, Modified Sparse Row (MSR) storage scheme, and so on. Here, we only give some matrix-vector operations by CSR and CSC storage schemes as demonstrated in Figure B.1.

Given an example matrix

$$A = \begin{bmatrix} 11.0 & 0 & 1.0 & 0 & 3.3 \\ 0 & 6.0 & 0 & 0 & 0.9 \\ 1.0 & 0 & 5.0 & 0.2 & 0 \\ 0 & 0 & 0.2 & 9.2 & 0 \\ 3.3 & 0.9 & 0 & 0 & 4.8 \end{bmatrix} \tag{B.1}$$

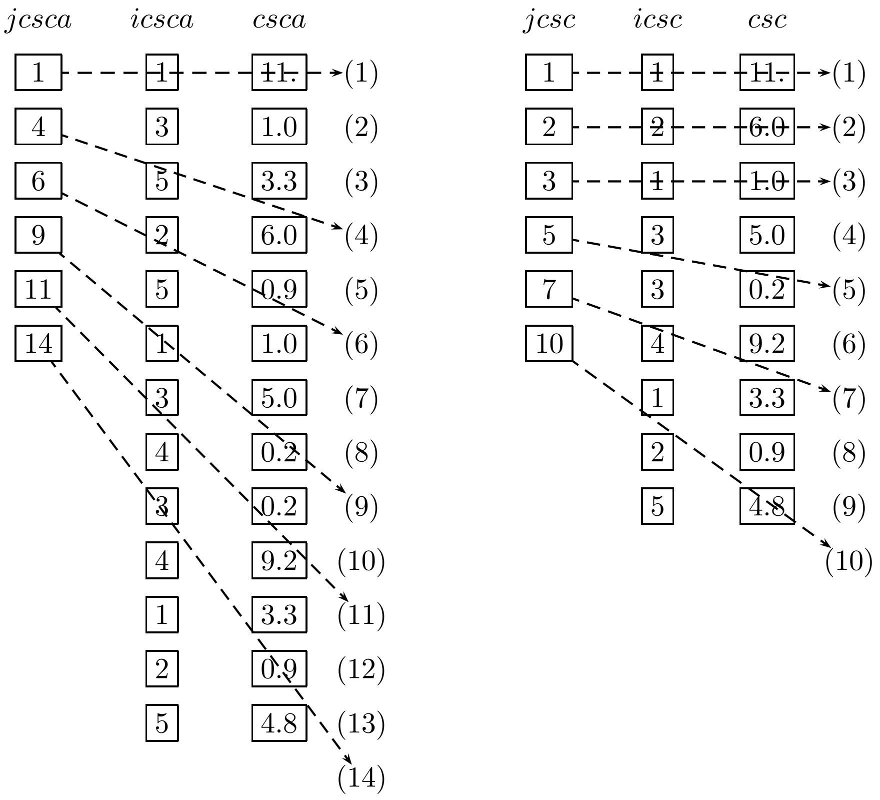


Figure B.1: CSC storage of matrix A and CSC storage of its upper triangular part.

B.1.2 Demonstration on How to Form Symmetric Compressed Sparse Storage

Assume that there are three symmetric element stiffness matrices in a finite element analysis

$$\text{Element 1:} \quad \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}, \quad a_{ij} = a_{ji}\ (i, j = 1, \ldots, 4)$$

$$\text{Element 2:} \quad \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}, \quad b_{ij} = b_{ji} \; (i, j = 1, \ldots, 4)$$

$$\text{Element 3:} \quad \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}, \quad c_{ij} = c_{ji} \; (i, j = 1, \ldots, 4)$$

Therefore, in this example, the dimension of element stiffness matrix is $n_{el} = 4$ and the total number of elements is $n_e = 3$. Then, assume that the final assembled global matrix with dimension $n = 8$ is given as

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & 0 & 0 & 0 & 0 \\ & a_{22} & a_{23} & a_{24} & 0 & 0 & 0 & 0 \\ & & \underline{a_{33} + b_{11}} & \underline{a_{34} + b_{12}} & b_{13} & b_{14} & 0 & 0 \\ & & & \underline{a_{44} + b_{22}} & b_{23} & b_{24} & 0 & 0 \\ & & & & \underline{b_{33} + c_{11}} & \underline{b_{34} + c_{12}} & c_{13} & c_{14} \\ & & symmetry & & & \underline{b_{44} + c_{22}} & c_{23} & c_{24} \\ & & & & & & c_{33} & c_{34} \\ & & & & & & & c_{44} \end{bmatrix}_{8\times 8} \tag{B.2}$$

The compressed sparse storage scheme tries to assemble and stores only the nonzero entries in the above global stiffness matrix A. The compressed sparse storage scheme demonstrated here uses only three vectors, namely *iebe*, *jebe* and *ebea*, respectively, and these three vectors collect the global row numbers and the global column numbers of nonzero element entries in the upper triangular part element by element. When allocating the storage of the three vectors, i.e., *iebe*, *jebe* and *ebea*, the dimension (i.e., the length of the three vectors) can be defined as

$$nzebe = \frac{n_{el}^2 + n_{el}}{2} \times n_e \tag{B.3}$$

Thus, the allocated storage for each vector is about $nzebe = \dfrac{4^2 + 4}{2} \times 3 = 30$. When using these three vectors to collect the nonzero element entries in the upper part in column-wise and element by element (notice that in each element, the collection can be carried out in column-wise or in row-wise), we can get these three vectors demonstrated as follows,

iebe	*jebe*	*ebea*
1	1	a_{11}
1	2	a_{12}
2	2	a_{22}
1	3	a_{13}
2	3	a_{23}
3	3	a_{33}
1	4	a_{14}
2	4	a_{24}
3	4	a_{34}
4	4	a_{44}
3	3	b_{11}
3	4	b_{12}
4	4	b_{22}
3	5	b_{13}
4	5	b_{23}
5	5	b_{33}
3	6	b_{14}
4	6	b_{24}
5	6	b_{34}
6	6	b_{44}
5	5	c_{11}
5	6	c_{12}
6	6	c_{22}
5	7	c_{13}
6	7	c_{23}
7	7	c_{33}
5	8	c_{14}
6	8	c_{24}
7	8	c_{34}
8	8	c_{44}

With these three vectors which store nonzero element entries, we sort the first vector (*iebe*) which stores the global row number for each nonzero element entry, at the same time, the other two vectors change their orders correspondingly. After the first vector is sorted in the ascending order, we can sort the second vector (*jebe*) which stores the global column number, at the same time, the vector *ebea* changes its order correspondingly. Clearly, by summing up the values in vector *ebea* with both the same global row number and the same global column number, the assembly can be carried out. This is the CSR storage assembly scheme. For the CSC storage assembly scheme, the only difference is that we need to sort *jebe* first and then to sort *iebe*. Given the CSC storage assembly scheme for an example, firstly, we sort *jebe* in the ascending order as follows

First step:			⇒	Second step:			⇒	Assembly step:		
iebe	*jebe*	*ebea*		*iebe*	*jebe*	*ebea*		*iebe*	*jebe*	*ebea*
1	1	a_{11}		1	1	a_{11}		1	1	a_{11}
1	2	a_{12}		1	2	a_{12}		1	2	a_{12}
2	2	a_{22}		2	2	a_{22}		2	2	a_{22}
1	3	a_{13}		1	3	a_{13}		1	3	a_{13}
2	3	a_{23}		2	3	a_{23}		2	3	a_{23}
3	3	a_{33}		3	3	a_{33}		3	3	$a_{33} + b_{11}$
3 ↑	3 ↑	b_{11} ↑		3	3	b_{11}		1	4	a_{14}
1	4	a_{14}		1	4	a_{14}		2	4	a_{24}
2	4	a_{24}		2	4	a_{24}		3	4	$a_{34} + b_{12}$
3	4	a_{34}		3	4	a_{34}		4	4	$a_{44} + b_{22}$
4	4	a_{44}		3 ↑	4	b_{12} ↑		3	5	b_{13}
3	4	b_{12}		4	4	a_{44}		4	5	b_{23}
4	4	b_{22}		4	4	b_{22}		5	5	$b_{33} + c_{11}$
3	5	b_{13}		3	5	b_{13}		3	6	b_{14}
4	5	b_{23}		4	5	b_{23}		4	6	b_{24}
5	5	b_{33}		5	5	b_{33}		5	6	$b_{34} + c_{12}$
5 ↑	5 ↑	c_{11} ↑		5	5	c_{11}		6	6	$b_{44} + c_{22}$
3	6	b_{14}		3	6	b_{14}		5	7	c_{13}
4	6	b_{24}		4	6	b_{24}		6	7	c_{23}
5	6	b_{34}		5	6	b_{34}		7	7	c_{33}
6	6	b_{44}		5 ↑	6	c_{12} ↑		5	8	c_{14}
5	6	c_{12}		6	6	b_{44}		6	8	c_{24}
6	6	c_{22}		6	6	c_{22}		7	8	c_{34}
5	7	c_{13}		5	7	c_{13}		8	8	c_{44}
6	7	c_{23}		6	7	c_{23}				
7	7	c_{33}		7	7	c_{33}				
5	8	c_{14}		5	8	c_{14}				
6	8	c_{24}		6	8	c_{24}				
7	8	c_{34}		7	8	c_{34}				
8	8	c_{44}		8	8	c_{44}				

In the second step, we sort the segments (denoted in the same color) in vector *iebe* with the same global column number to make all numbers in their segments increase in the ascending order. In the assembly stage, we add up all the numbers with both the same global row number and the same global column number. It should be noted that the assembled compressed sparse storage can over write the data of these three vectors in the second step for memory efficiency. In addition, the spare storage after the assembly can be reused for other purposes, and in this example, the spare storage length is 6. This symmetric compressed sparse storage scheme can be verified by the assembled 8×8 matrix in Eq. B.2. For a detailed comparison between EBE storage scheme and compressed sparse storage scheme, see Section 2.3.

B.1.3 Adjacency Structure for Sparse Matrix

To implement the sparse Cholesky routines of SparsM package introduced in section 1.4, the adjacency structure (*XADJ, ADJNCY*) for a sparse matrix is required to represent the the graph of the matrix, the array pair (*XADJ, ADJNCY*) is used in ordering algorithm such as MMD algorithm and also in the symbolic factorization process (e.g., Gorge and Liu, 1981). The array pair (*XADJ, ADJNCY*) for the matrix in Equation B.1 can be illustrated as follows:

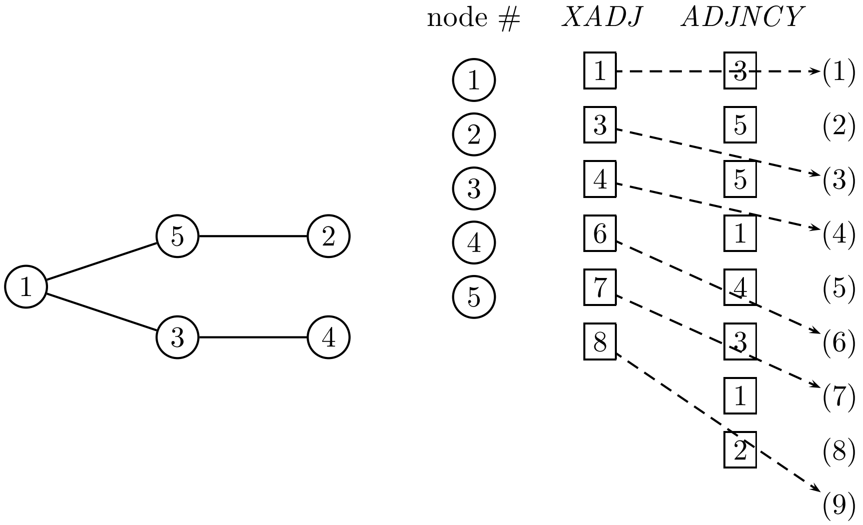


Figure B.2: Adjacency structure for matrix A of Equation (B.1)

It is clear that the adjacency structure (*XADJ, ADJNCY*) is the the index arrays of CSR storage for $A - diag(A)$.

B.2 Basic Sparse Matrix Operations

Algorithm 3 $y = Bx$ with CSR format B

Computation of $y = Bx$, given the CSR format of B, `icsrb(m+1)`, `jcsrb(bnz)`, `csrb(bnz)` (`bnz` is the number of nonzero entries in matrix B). Suppose m and n are the number of rows and columns of B, respectively.

```
$y = zeros(n, 1)$
for $i = 1$ to $m$ do
  if $x(i) \neq 0$ then
    for $k = icsrb(i)$ to $icsrb(i+1) - 1$ do
      $r = jcsrb(k)$
      $y(r) = y(r) + x(i) * csrb(k)$
    end for
  end if
end for
```

Algorithm 4 $y = B^T x$ with CSR format B

Computation of $y = B^T x$, given the CSR format of B, `icsrb(m+1)`, `jcsrb(bnz)`, `csrb(bnz)` (`bnz` is the number of nonzero entries in matrix B). Suppose m and n are the number of rows and columns of B, respectively.

```
$y = zeros(n, 1)$
for $i = 1$ to $m$ do
  if $x(i) \neq 0$ then
    for $k = icsrb(i)$ to $icsrb(i+1) - 1$ do
      $r = jcsrb(k)$
      $y(r) = y(r) + x(i) * csrb(k)$
    end for
  end if
end for
```

Algorithm 5 $y = Bx$ with CSC format B

Computation of $y = Bx$, given the CSC format of B, `jcscb(n+1)`, `icscb(bnz)`, `cscb(bnz)` (`bnz` is the number of nonzero entries in matrix B). Suppose m and n are the number of rows and columns of B respectively.

```
$y = zeros(m, 1)$
for $j = 1$ to $n$ do
  if $x(j) \neq 0$ then
    for $k = jcscb(j)$ to $jcscb(j+1) - 1$ do
      $r = icscb(k)$
      $y(r) = y(r) + x(j) * cscb(k)$
    end for
  end if
end for
```

Algorithm 6 $y = B^T x$ with CSC format B

Computation of $y = Bx$, given the CSC format of B, `jcscb(n+1)`, `icscb(bnz)`, `cscb(bnz)` (`bnz` is the number of nonzero entries in matrix B). Suppose m and n are the number of rows and columns of B respectively.

```
$y = zeros(n, 1)$
for $j = 1$ to $n$ do
  for $k = jcscb(j)$ to $jcscb(j+1) - 1$ do
    if $x(icscb(k)) \neq 0$ then
      $y(j) = y(j) + cscb(k) * x(icscb(k))$
    end if
  end for
end for
```

APPENDIX C

SOURCE CODES IN FORTRAN 90

C.1 Main Program for 3-D Biot's Consolidation FEM Analysis

This program is a 3-D extension of program 9.1 or program 9.2 of book "Programming the Finite Element Method" (called PFEM book from hereon) written by Smith and Griffiths (1998). For 3-D linear elastic FEM analysis of Biot's consolidation problems, 20-node solid brick elements coupled to 8-node fluid elements are used. Thus, the dimension of element "stiffness" matrices are $20 \times 3 + 8 = 68$. Sparse preconditioned iterative solver is used for the discretized linear system in each time step. Apart from standard Jacobi and standard SSOR preconditioners, the newly developed GJ or MSSOR preconditioner can be chosen to combine with SQMR, PCG and MINRES solvers. Furthermore, two different convergence criteria, relative "improvement" norm criterion and relative residual norm criterion, are available. To use sparse preconditioned SQMR method, a new `sparse_lib` library was developed and several subroutines were added in the available `new_library` library.

Some descriptions of using sparse iterative methods for 3-D Biot's consolidation problem are given as follows:

(a) **Input:**
`nels,nxe,nye,nze,nn,nip,permx,permy,permz,e,v,dtim,nstep,theta,maxit,`

`tol,coef,omega,isolver,icho,ipre,icc,iinc`

Apart from the parameters used in original main program of PFEM book, the required new parameters are listed as: `maxit,tol,omega,isolver,icho,ipre,icc,iinc`.

(b) **Call** `nfinfo(nn,nodof,nxe,nye,nze,nf,neq,sneq,fneq)`

This subroutine generates the array `nf` for 3-D Biot's consolidation models studied in this thesis and outputs the numbers, `neq,sneq(sn),fneq(fn)`.

(c) **Input:** `loadl, nmesh, unipval`

These inputs are provided for the following subroutine `loadid`, then loaded node number is calculated from `loaded_nodes = (nmesh*2+1)*(nmesh+1)+(nmesh+1)*nmesh`.

`loadl:` length (m) of loaded area in each direction;

`nmesh:` mesh (element) number of loaded area in each direction, and these meshes should be uniform in the loaded area;

`unipval:` value (MPa) of uniform pressure.

(d) **Call**

```
loadid(nxe,nye,nze,nf,nels,nn,nodof,ntot,loadl,nmesh,unipval,&
loaded_nodes,nodnum,load_val,id,ebeanz)
```

This subroutine is designed for uniform pressure loaded area with uniform meshes for the examples in the thesis. The loaded node number and equivalent concentrated load can be input directly if the subroutine is not adopted.

`nodnum:` array of loaded node number;

`load_val:` array of load value corresponding to `lnn` array;

`id:` identifier array for displacement or pore pressure DOF.

(e) **Call** `formspars(ntot,g,ke,iebe,jebe,ebea,ebeanz)`

This subroutine is embedded in `elements_2` loop, and collects all nonzero entries element by element (only upper triangular part is scanned).

(f) **Call**

```
sortadd(ebeanz,jebe(1:ebeanz),iebe(1:ebeanz),ebea(1:ebeanz),neq+1,&
uanz,jcsc,icsc,csca)
```

This subroutine sorts `iebe,jebe,ebea` into compresses sparse (column) storage.

(g) **Call** `formda(n,icsc,jcsc,csca,icho,ipre,coef,omega,id,d,da,da1)`

This subroutine constructs standard or modified diagonal (by GJ algorithm) for Jacobi or SSOR preconditioners.

(h) **Call** `kpu(icsc,jcsc,csca,theta,id,loads(1:),ans(1:))`

This subroutine generates the part correspondent to pore water pressure of right-hand-side vector for incremental formulation of 3-D Biot's consolidation equations.

`loads(1:)`: current solution;

`ans(1:)`: returned right-hand-side vector (incremental formulation), only the part correspondent to pore water pressure is generated.

(i) **Applied load**

This part can be combined with the subroutine `kpu`, but the codes are simple, thus can be written into main program directly. After this part, the true right-hand-side vector of incremental formulation is generated. In addition, the load can be applied in first time step or in a "ramp" loading form.

(j) **Call**
`psolver(neq,icsc,jcsc,csca,diag,diaga,diaga1,ans(1:),maxit,tol,&`

`isolver,ipre,icho,icc,iinc,iters,resi)`
This subroutine solves the generated linear system in each time step by standard (or generalized) Jacobi or standard (or modified) SSOR preconditioned iterative method.

(k) **Input file '`p3dbiot.dat`' of a small example for a $5 \times 5 \times 5$ problem**

Here, MSSOR preconditioned SQMR method is chosen.

```
nels nxe nye nze nn nip permx permy permz e v dtim nstep theta ⟵
125 5 5 5 756 27 1.e-7 1.e-7 1.e-7 1.  0.3 1.  1 1.

maxit tol coef omega isolver icho ipre icc iinc ⟵
1000 1.e-6 -4.  1.0 1 2 2 2 5

widthx(i), i = 1, nxe + 1  ⟵
.0 1.0 3.25 5.5 7.75 10.0

widthy(i), i = 1, nye + 1  ⟵
.0 1.0 3.25 5.5 7.75 10.0

depth(i), i = 1, nze + 1  ⟵
.0 -1.0 -3.25 -5.50 -7.75 -10.0

loadl nmesh unipval ⟵
1.  1 0.1
```

(l) **Output file ‘p3dbiot.res’ for the above problem**

```
**********************************************
The current time is  0.1000E+01
  ---> MSSOR preconditioned SQMR solver
  with relative residual norm criterion!
**********************************************
SQMR converges to user-defined tolerance.
Psolver took   65  iterations to converge to   0.5120E-06
The nodal displacements and porepressures are  :
     1      0.00000E+00  0.00000E+00 -0.14503E+00  0.00000E+00
     2     -0.70824E-02  0.00000E+00 -0.14199E+00  0.00000E+00
     3     -0.13808E-01  0.00000E+00 -0.90871E-01  0.00000E+00
     4     -0.70046E-02  0.00000E+00 -0.18951E-01  0.00000E+00
     5      0.12224E-02  0.00000E+00 -0.11831E-01  0.00000E+00
     6      0.51799E-03  0.00000E+00 -0.28245E-02  0.00000E+00
     7     -0.24022E-03  0.00000E+00  0.75205E-03  0.00000E+00
     8      0.15517E-03  0.00000E+00  0.29346E-02  0.00000E+00
     9      0.40951E-03  0.00000E+00  0.37435E-02  0.00000E+00
    10      0.23098E-03  0.00000E+00  0.43290E-02  0.00000E+00
    11      0.00000E+00  0.00000E+00  0.45353E-02  0.00000E+00
    12      0.00000E+00  0.00000E+00 -0.11212E+00  0.00000E+00
    13      0.13639E-01  0.00000E+00 -0.77092E-01  0.00000E+00
    14      0.25257E-02  0.00000E+00 -0.10092E-01  0.00000E+00
    15      0.14592E-02  0.00000E+00  0.69487E-03  0.00000E+00
     . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
   750      0.00000E+00  0.00000E+00  0.00000E+00 -0.47693E-03
   751      0.00000E+00  0.00000E+00  0.00000E+00  0.00000E+00
   752      0.00000E+00  0.00000E+00  0.00000E+00 -0.33150E-03
   753      0.00000E+00  0.00000E+00  0.00000E+00  0.00000E+00
   754      0.00000E+00  0.00000E+00  0.00000E+00 -0.21429E-03
   755      0.00000E+00  0.00000E+00  0.00000E+00  0.00000E+00
   756      0.00000E+00  0.00000E+00  0.00000E+00 -0.16963E-03
                      Overhead time is:    5.26800012588501
Iterative time (the last time step) is:    1.70199990272522
  Total runtime for the FEM program is     7.34100008010864  seconds.
```

The main program for 3-D Biot’s consolidation problem is given as:

```
!-----------------------------------------------------------------------
program p3dbiot ! 3-D Biot Consolidation Analysis
!-----------------------------------------------------------------------
!   Program for 3-D consolidation analysis using 20-node solid brick
!   elements coupled to 8-node fluid elements, incremental formulation.
!   Iterative solver is used for solving the linear system in each time
!   step, GJ preconditioner and MSSOR preconditioner are available.
!    isolver=1 for SQMR;    isolver=2 for PCG;   isolver=3 for MINRES.
!-----------------------------------------------------------------------
  use new_library ; use geometry_lib;  use sparse_lib; use dfport ;
  implicit none
  integer::i,j,k,l,nn,nels,nxe,nye,nze,nip,nodof=4,nod=20,nodf=8,nst=6,&
           ndim=3,nodofs=3,ntot,ndof,iel,ns,nstep,inc,loaded_nodes,neq,&
           nband,sneq,fneq,nmesh,ebeanz,uanz,maxit,iters,isolver, icho,&
           ipre,icc,iinc

  real(8)::permx,permy,permz,e,v, det, dtim, theta,ttime,loadl,unipval,&
           tol,coef,omega,resi,ot1,ot2,it1,it2,tt1,tt2

  logical:: converged
  character (len=15):: element = 'hexahedron'
!----------------------- dynamic arrays---------------------------------
  real(8),allocatable ::dee(:,:), points(:,:), coord(:,:), derivf(:,:),&
          jac(:,:),kay(:,:),der(:,:),deriv(:,:),weights(:), derf(:,:), &
          funf(:),coordf(:,:),bee(:,:),km(:,:),eld(:),sigma(:),kp(:,:),&
          ke(:,:), g_coord(:,:), kd(:,:), fun(:), c(:,:), bk(:),vol(:),&
          volf(:,:),widthx(:),widthy(:),depth(:), load_val(:),loads(:),&
          ans(:),ebea(:),csca(:),diag(:),diaga(:),diaga1(:)

  integer,allocatable:: nf(:,:), g(:), num(:), g_num(:,:), g_g(:,:),   &
                        nodnum(:),iebe(:),jebe(:),icsc(:),jcsc(:),id(:)
!---------------------- input and initialization ---------------------
  open (10,file='p3dbiot.dat',status=    'old',action='read')
  open (11,file='p3dbiot.res',status='replace',action='write')
    print*," Preconditioned Iterative Solution Method  "
    write(11,*)" Preconditioned Iterative Solution Method  "
    print*," The program is running, please wait..................... "

  read (10,*) nels,nxe,nye,nze,nn,nip,permx, permy,permz,e,v,dtim, &
            nstep,theta,maxit,tol,coef,omega,isolver,icho,ipre,icc,iinc

  ndof=nod*3; ntot=ndof+nodf ;
  allocate(dee(nst,nst),points(nip,ndim),coord(nod,ndim),jac(ndim,ndim),&
         derivf(ndim,nodf),kay(ndim,ndim),der(ndim,nod),deriv(ndim,nod),&
         derf(ndim,nodf), funf(nodf), coordf(nodf,ndim), bee(nst,ndof), &
         km(ndof,ndof),eld(ndof),sigma(nst),kp(nodf,nodf),ke(ntot,ntot),&
         g_g(ntot,nels), fun(nod), c(ndof,nodf), vol(ndof),nf(nodof,nn),&
         g(ntot),volf(ndof,nodf),g_coord(ndim,nn),num(nod),weights(nip),&
         g_num(nod,nels),widthx(nxe+1),widthy(nye+1),depth(nze+1) )
  !
  kay=0.0; kay(1,1)=permx; kay(2,2)=permy; kay(3,3)=permz
  read (10,*) widthx, widthy, depth
  call nfinfo(nn,nodof,nxe,nye,nze,nf,neq,sneq,fneq)
  !----------------------------------------------------
  call deemat (dee,e,v); call sample(element,points,weights)
!------------- loop the elements to set up global arrays----------------
        tt1=rtc()
  elements_1: do iel = 1, nels
        call geometry_20bxz(iel,nxe,nze,widthx,widthy,depth,coord,num)
        inc=0 ;
        do i=1,20; do k=1,3; inc=inc+1;g(inc)=nf(k,num(i));end do;end do
        do i=1,7,2; inc=inc+1;g(inc)=nf(4,num(i)); end do
        do i=13,19,2; inc=inc+1;g(inc)=nf(4,num(i)); end do
        g_num(:,iel)=num;g_coord(:,num)=transpose(coord);g_g(:,iel)= g
        if(nband<bandwidth(g))nband=bandwidth(g) ;
  end do elements_1
    write(11,'(a)') "Global coordinates "
    do k=1,nn;
      write(11,'(a,i7,a,3e12.4)')"Node",k,"       ",g_coord(:,k);
    end do
    write(11,'(a)') "Global node numbers "
    do k = 1  , nels;
      write(11,'(a,i6,a,20i7)') "Element ",k,"        ",g_num(:,k);
    end do
  write(11,'(2(a,i5))')                                                &
      "There are  ",neq, "  equations and the half-bandwidth is   ",nband
  !----------------------------------------------------
   read(10,*) loadl, nmesh, unipval
  loaded_nodes=(nmesh*2+1)*(nmesh+1)+(nmesh+1)*nmesh ;
  allocate(nodnum(loaded_nodes),load_val(loaded_nodes),id(1:neq))
  call loadid(nxe,nye,nze,nf,nels,nn,nodof,ntot,loadl,nmesh,unipval,  &
              loaded_nodes,nodnum,load_val,id,ebeanz)
  !----------------------------------------------------
  allocate( iebe(ebeanz),jebe(ebeanz), ebea(ebeanz)  )
!------------- element stiffness integration and assembly --------------
           ebeanz=0  ! used for counting the true number
           ot1=rtc() ;
  elements_2:  do iel = 1 , nels
        num = g_num(: , iel );  coord=transpose(g_coord(:,num))
        g = g_g( : , iel ) ;  coordf(1 : 4 , : ) = coord(1 : 7 : 2, : )
        coordf(5 : 8 , : ) = coord(13 : 19 : 2, : )
        km = .0; c = .0; kp = .0
      gauss_points_1: do i = 1 , nip
          call shape_der(der,points,i);  jac = matmul(der,coord)
          det = determinant(jac );  call invert(jac);
          deriv = matmul(jac,der);  call beemat(bee,deriv);
          vol(:)=bee(1,:)+bee(2,:)+bee(3,:)
         km=km + matmul(matmul(transpose(bee),dee),bee) *det* weights(i)
!-----------------------now the fluid contribution----------------------
          call shape_fun(funf,points,i); call shape_der(derf,points,i) ;
          derivf=matmul(jac,derf)
kp=kp+matmul(matmul(transpose(derivf),kay),derivf)*det*weights(i)*dtim ;
          do l=1,nodf; volf(:,l)=vol(:)*funf(l); end do
          c= c+volf*det*weights(i)
       end do gauss_points_1
          call formke(km,kp,c,ke,theta) ! for incremental formula
          !---collect nonzero entries from element stiffness matrices---
          call formspars(ntot,g,ke,iebe,jebe,ebea,ebeanz)
  end do elements_2
!-----------------------------------------------------------------------
    uanz = int(neq - nband/2)*nband  ! This is a conservative estimation
  ! uanz = int(0.58*ebeanz)
  ! 0.58 is the estimated ratio of assembled to element-level nonzero.

  allocate( icsc(uanz), jcsc(neq+1), csca(uanz), diag(neq), diaga(neq),&
            diaga1(neq) )
  call sortadd(ebeanz,jebe(1:ebeanz),iebe(1:ebeanz),ebea(1:ebeanz),  &
               neq+1,uanz,jcsc,icsc,csca)
  deallocate(iebe,jebe,ebea)
  ! form GJ preconditioner or modified diagonal for MSSOR preconditioner
call
formda(neq,icsc,jcsc,csca,icho,ipre,coef,omega,id,diag,diaga,diaga1)
  ot2=rtc() ; ot2=ot2-ot1
! --------------------- enter the time-stepping loop--------------------
  allocate(loads(0:neq), ans(0:neq) )
      ttime = .0; loads = .0
  time_steps:  do ns = 1 , nstep
    write(*, '(a, i5,a)') '  Current Time Step is No.', ns , ' Step. '
    write(11,*)'********************************************** '
   ttime=ttime+dtim; write(11,'(a,e12.4)')" The current time is",ttime
   ans=.0; call kpu(icsc,jcsc,csca,theta,id,loads(1:),ans(1:));ans(0)=.0
! ----------------------- Apply Constant Loading -----------------------
        if(ns <= 10) then        ! the Load is applied at the first step.
          do i=1,loaded_nodes
            ans(nf(3, nodnum(i))) = - load_val(i)*0.1
          end do
        end if
!----------------- Preconditioned Iterative Solver ---------------------
     it1=rtc()
     call psolver(neq,icsc,jcsc,csca,diag,diaga,diaga1,ans(1:),maxit,  &
                  tol,isolver,icho,ipre,icc,iinc,iters,resi); ans(0)=.0 ;
     it2=rtc() ; it2=it2-it1
     write(11,'(a,i5,a,e12.4)')" Psolver took", iters,"  iterations to &
                                converge to ",resi
     loads=loads+ans
     write(11,'(a)') " The nodal displacements and porepressures are  :"
     do k=1,nn; write(11,'(i7,a,4e13.5)')k,"    ",loads(nf(:,k)); end do
!-------------------recover stresses at  Gauss-points-------------------
    elements_5 :  do iel = 1 , nels
      num = g_num(:,iel); coord=transpose(g_coord(:,num))
      g = g_g( : , iel );  eld = loads( g ( 1 : ndof ) )
      ! print*,"The Gauss Point effective stresses for element",iel,"are"
      gauss_points_2: do i = 1,nip
          call shape_der (der,points,i);  jac= matmul(der,coord)
          call invert ( jac );    deriv= matmul(jac,der)
          bee= 0.;call beemat(bee,deriv);
          sigma= matmul(dee,matmul(bee,eld))
          !  print*,"Point    ",i       ;!  print*,sigma
      end do gauss_points_2
    end do elements_5
  end do time_steps
   tt2=rtc() ; tt2=tt2-tt1
   write(11,*) "                      Overhead time is: ",ot2
   write(11,*) "Iterative time (the last time step) is: ",it2
   write(11,*) "  Total runtime for the FEM program is  ",tt2, "seconds."
 end program p3dbiot
!------------------------------------------------------------------------
```

C.2 New Subroutines for Module new_library

```
module new_library
 contains
!-----------------------------------------------------------------------
subroutine nfinfo(nn,nodof,nx,ny,nz,nf,neq,sn,fn)
  ! This subroutine generates nf array with only 0 and 1 integer value
  !                 for 3-D Biot's consolidation problems, and this
  !                 subroutine is restricted to the geometric model
  !                 discussed in this thesis.
  !             nn: total number of nodes;
  !          nodof: number of freedoms per node;
  !     nx, ny, nz: i.e. nxe, nye, nze(element number in each direction)
  !             nf: generated nodal freedom array,
  !                 nf(:,:) = 1, free DOF;
  !                 nf(:,:) = 0, restricted DOF.
  !            neq: number of DOFs in the mesh (or number of equations);
  !             sn: number of DOFs corresponding to displacement;
  !             fn: number of DOFs corresponding to pore pressure;
  !                 and there exists "neq = sn + fn".
    integer::i,j,k,nf(:,:),nn,nodof,nx,ny,nz,nodplane,nodbetwplane,neq,&
                       sn,fn
    !-----------------
   nodplane=(2*nx+1)*(nz+1)+(nx+1)*nz ; ! node number for each x-z plane
   nodbetwplane=(nx+1)*(nz+1)       ! node number between two x-z planes
    !
    nf = 1 ;                        ! initialize all nodes unrestricted
    xdirection1: do i=0, ny
      xloop1: do j=0, nz
              nf(1,i*(nodplane+nodbetwplane)+2*nx+1+j*(3*nx+2))=0 ;
              !
              nf(1,i*(nodplane+nodbetwplane)+1+j*(3*nx+2))=0 ;
      end do xloop1
      !
      xloop2: do j=1, nz
              nf(1,i*(nodplane+nodbetwplane)+j*(3*nx+2))=0 ;
              !
              nf(1,i*(nodplane+nodbetwplane)+j*(3*nx+2)-nx)=0 ;
      end do xloop2
    end do xdirection1
    !
    xdirection2: do i=1, ny
      xloop3: do j=1, nz+1
              nf(1,i*nodplane+(i-1)*nodbetwplane+j*(nx+1))=0 ;
      end do xloop3
      !
      xloop33: do j=0, nz
              nf(1,i*nodplane+(i-1)*nodbetwplane+j*(nx+1)+1)=0 ;
      end do xloop33
    end do xdirection2
    !--------------------------
    ydirection1: do i=1, nodplane
              nf(2,ny*(nodplane+nodbetwplane)+i)=0
              !
              nf(2,i)=0
    end do ydirection1
    !--------------------------
    xyzdirection1: do i=0, ny
      xyzloop1: do j=1,2*nx+1
              nf(1,i*(nodplane+nodbetwplane)+nodplane-2*nx-1+j)=0 ;
              nf(2,i*(nodplane+nodbetwplane)+nodplane-2*nx-1+j)=0 ;
              nf(3,i*(nodplane+nodbetwplane)+nodplane-2*nx-1+j)=0 ;
      end do xyzloop1
    end do xyzdirection1
    !
    xyzdirection2: do i=1, ny
```

```
      xyzloop2: do j=1,nx+1
              nf(1,i*(nodplane+nodbetwplane)-(nx+1)+j)=0 ;
              nf(2,i*(nodplane+nodbetwplane)-(nx+1)+j)=0 ;
              nf(3,i*(nodplane+nodbetwplane)-(nx+1)+j)=0 ;
      end do xyzloop2
    end do xyzdirection2
    !----------------------------
    pp1: do i=0,ny
      ploop1: do j=0, nz
        ploop2: do k=1, nx
              nf(4, i*(nodplane+nodbetwplane)+j*(3*nx+2)+2*k)=0 ;
        end do ploop2
      end do ploop1
      !
      ploop3: do j=1, nz
        ploop4: do k=1, nx+1
          nf(4, i*(nodplane+nodbetwplane)+j*(2*nx+1)+(j-1)*(nx+1)+k)=0 ;
        end do ploop4
      end do ploop3
    end do pp1
    !
    pp2: do i=1,ny
      ploop5: do j=1,nodbetwplane
              nf(4, i*nodplane+(i-1)*nodbetwplane+j)=0 ;
      end do ploop5
    end do pp2
    !
    pp3: do i=0, ny
      ploop6: do j=1, 2*nx+1
              nf(4, i*(nodplane+nodbetwplane)+j)=0 ;
      end do ploop6
    end do pp3
    !----------------------------
    fn=0; do i=1, nn; fn = fn + nf(4,i);  end do ;
    call formnf(nf); neq=maxval(nf);  sn = neq - fn ;
    !
    return
end subroutine nfinfo
!-------------------------------------------------------------------------
subroutine loadid(nx,ny,nz,nf,nels,nn,nodof,ntot,loadl,nmesh,unipval, &
                  nlnod,lnn,lnv,id,ebenz)
  ! This subroutine computes nodal loads from uniform load pressure.
  ! In this subroutine
  !  nx, ny, nz: i.e. nxe, nye, nze(element number in each direction);
  !        nels: total element number;
  !       nodof: DOF number per node;
  !       loadl: length (m) of loaded area in each direction;
  !       nmesh: mesh (element) number of loaded area in each direction;
  !              these meshes should be uniform in the loaded area.
  !     unipval: value (MPa) of uniform pressure;
  !         lnn: array of loaded node number
  !         lnv: array of load value correspondent to lnn array.
  !          id: identifier array for displacement or porepressure DOF;
  !              id(i) = 1, displacement DOF;
  !              id(i) = 0, pore pressure DOF.
  !       elenz: estimated number of total nonzero entries of all element
  !              stiffness matrices.
    integer:: i,j,k,l,m, nx, ny, nz, nf(:,:),nels,nn,nodof,ntot,nmesh, &
              nlnod,nodplane,nodbetwplane,lnn(:),id(:),ebenz
    real(8):: loadl, unipval,lnv(:)
    !
    sval=unipval*(loadl/nmesh)**2/12.             ! the special value
    nodplane=(2*nx+1)*(nz+1)+(nx+1)*nz ; ! node number for each x-z plane
    nodbetwplane=(nx+1)*(nz+1) ;     ! node number between two x-z planes
    !
```

```
        m=0 ;
    do i=1, nmesh
      k=(i-1)*(nodplane+nodbetwplane)+1 ;
      if(i==1)then
        do j=0, 2*nmesh
          if(mod(j, 2)==0)then
             m=m+1; lnn(m)=k+j ; lnv(m)=-2*sval ;
          else
             m=m+1; lnn(m)=k+j ; lnv(m)=4*sval ;
          end if
        end do
      else
        do j=0, 2*nmesh
          if(mod(j,2)==0)then
            m=m+1; lnn(m)=k+j ; lnv(m)=-4*sval ;
          else
            m=m+1; lnn(m)=k+j ; lnv(m)=8*sval ;
          end if
        end do
      end if
      !
      k=i*nodplane+(i-1)*nodbetwplane+1 ;
      do j=0, nmesh
        if(j==0.or.j==nmesh)then
          m=m+1; lnn(m)=k+j; lnv(m)=4*sval ;
        else
          m=m+1; lnn(m)=k+j; lnv(m)=8*sval ;
        end if
      end do
    end do
    !----------------------
    k=nmesh*(nodplane+nodbetwplane)+1
    do j=0,2*nmesh
      if(mod(j,2)==0)then
        m=m+1; lnn(m)=k+j ; lnv(m)=-2*sval ;
      else
        m=m+1; lnn(m)=k+j ; lnv(m)=4*sval ;
      end if
    end do
    !--Modify the two sides--
    do i=1,nmesh+1
      l=(i-1)*(3*nmesh+2)+1; lnv(l)=-2*sval ;
      l=(i-1)*(3*nmesh+2)+2*nmesh+1; lnv(l)=-2*sval ;
    end do
      lnv(1)=-sval; lnv(2*nmesh+1)=-sval ;
      lnv(nlnod)=-sval; lnv(nlnod-2*nmesh)=-sval ;
    !------------------------
    id(:)=1; do i=1,nn; if(nf(nodof,i)/=0) id(nf(nodof,i))=0 ; end do;
    !------------------------
    ebenz = ntot*int(ntot/2)*nels ;
    !------------------------
    return
end subroutine loadid
!-----------------------------------------------------------------------

end module new_library
```

C.3 A New Module sparse_lib

For sparse GJ and MSSOR-preconditioned iterative solvers for 3-D Biot's consolidation problems, a sparse_lib module is built. The Fortran 90 source code of sparse_lib is provided as following.

```
module sparse_lib
 contains
!----------------------------BASIC INFORMATION--------------------------
!-----------------------------------------------------------------------
   !
   subroutine snfn(nf,nn,nodof,ndim,nels,ntot,sn,fn,ebenz)
     ! This subroutine computes the displacement DOFs (sn)
     !     and pore pressure DOFs (fn).
     !     Then it gives an estimation (ebenz).
       implicit none
           integer:: i,nf(:,:),nn,nodof,ntot,ndim,nels,sn,fn,ebenz
           sn=0 ; fn = 0;
           fn = sum(nf(nodof,:)) ;
           do i=1,ndim ; sn = sn + sum(nf(i,:)) ; end do
           ebenz = ntot*int(ntot/2)*nels ; ! Estimated number
            return
    end subroutine snfn
    !
    subroutine form_id(nf,nn,nodof,id)
     ! This subroutine for the identifier array, "id".
     !     nf(:,:) is the original input node freedom array.
     !     nn - total node number.
     !     nodof - number of freedoms per node.
       implicit none
           integer:: i,nf(:,:),nn,nodof,id(:)
            id(:) = 1;
            do i = 1,nn ;
               if(nf(nodof,i)/=0)id(nf(nodof,i))=0;
            end do;
            !
            return
     end subroutine form_id
!---------------------------SORTING SUBROUTINES-------------------------
!-----------------------------------------------------------------------
subroutine quicksort(uanz,arr,brr,crr)
  ! This subroutine
  ! Quicksort - sorts arr into ascending order, brr and crr change
  !             correspondingly.
  ! quicksort chooses a "pivot" in the set, and explores the array
  ! from both ends, looking for a value > pivot with the increasing
  ! index, for a value <= pivot with the decreasing index, and
  ! swapping them when it has found one of each. !  The array is then
  ! subdivided in 2 ([3]) subsets: { values <= pivot} {pivot}
  ! {values > pivot}. One then call recursively the program to sort
  ! each subset.  When the size of the subarray is small enough, one
  ! uses an insertion sort that is faster for very small sets.
  ! Sorting an array arr(1:n) into ascending order with quicksort,
  ! while making the corresponding rarrangements of arrays brr(1:n)
  ! and crr(1:n).
  ! (Revised from ORDERPACK codes)
  !     uanz: the nonzero number of arr (or brr, crr).
  !--------------------------------
    implicit none
    real(8):: crr(:)
    integer::uanz,arr(:),brr(:)
     !
     call subsort(arr,brr,crr,1, uanz) ;
     call inssor(arr,brr,crr,uanz) ;
     !
     return
end subroutine quicksort
!-----------------------------------------------------------------------
Recursive subroutine subsort(arr,brr,crr, ideb1, ifin1)
  !  This subroutine sorts arr from ideb1 to ifin1
      real(8)::  crr(:), zwrk
      integer, intent (in) :: ideb1, ifin1
      integer :: arr(:), brr(:), icrs, ideb, idcr, ifin, imil, xpiv, &
                 xwrk, ywrk, nins = 16  ! Max for insertion sort
        ideb = ideb1
        ifin = ifin1
  !  if we don’t have enough values to make it worth while, we leave
  !  them unsorted, and the final insertion sort will take care of them.
      if ((ifin - ideb) > nins) Then
         imil = (ideb+ifin) / 2
  !  One chooses a pivot, median of 1st, last, and middle values
  !
         if (arr(imil) < arr(ideb)) Then
    xwrk = arr(ideb) ;      ywrk=brr(ideb);         zwrk=crr(ideb)
    arr(ideb) = arr(imil); brr(ideb) = brr(imil); crr(ideb) = crr(imil)
    arr(imil) = xwrk;      brr(imil) = ywrk;      crr(imil) = zwrk
         end if
         !
         if (arr(imil) > arr(ifin)) Then
    xwrk = arr(ifin);       ywrk = brr(ifin);       zwrk = crr(ifin)
    arr(ifin) = arr(imil); brr(ifin) = brr(imil); crr(ifin) = crr(imil)
    arr(imil) = xwrk;      brr(imil) = ywrk;      crr(imil) = zwrk
            if (arr(imil) < arr(ideb)) Then
    xwrk = arr(ideb);       ywrk = brr(ideb);       zwrk = crr(ideb)
    arr(ideb) = arr(imil); brr(ideb) = brr(imil); crr(ideb) = crr(imil)
    arr(imil) = xwrk;      brr(imil) = ywrk;      crr(imil) = zwrk
            end if
         end if
         xpiv = arr(imil)
  !
  !  One exchanges values to put those > pivot in the end and
  !  those <= pivot at the beginning
  !
         icrs = ideb
         idcr = ifin
          ech2: do         !---------------------------
            do
               icrs = icrs + 1
               if (icrs >= idcr) Then
  !
  !  the first > pivot is idcr
  !  the last <= pivot is icrs-1
  !  Note: if one arrives here on the first iteration, then
  !        the pivot is the maximum of the set, the last value is equal
  !        to it, and one can reduce by one the size of the set to
  !        process, as if arr (ifin) > xpiv
  !
                  exit ech2
  !
               end if
               if (arr(icrs) > xpiv) exit
            end do
            !
            do
               if (arr(idcr) <= xpiv) exit
               idcr = idcr - 1
               if (icrs >= idcr) Then
  !
  !  The last value < pivot is always icrs-1
  !
                  exit ech2
               end if
            end do
  !
            xwrk = arr(idcr);      ywrk = brr(idcr);  zwrk = crr(idcr)
            arr(idcr)=arr(icrs); brr(idcr)=brr(icrs); crr(idcr)=crr(icrs)
            arr(icrs)=xwrk;      brr(icrs)=ywrk;      crr(icrs)=zwrk
         end do ech2         !---------------------------
  !
  !  One now sorts each of the two sub-intervals
  !
         call subsort (arr,brr,crr, ideb1, icrs-1)
         call subsort (arr,brr,crr, idcr, ifin1)
      end if
      !
      return
end subroutine subsort
!-----------------------------------------------------------------------
subroutine inssor(arr,brr,crr,uanz)
  ! This subroutine sorts arr into increasing order (Insertion sort)
    integer :: arr(:),brr(:),uanz, icrs, idcr, xwrk,ywrk
    real(8) :: crr(:),zwrk
  !
    do icrs = 2, uanz
      xwrk = arr(icrs);     ywrk = brr(icrs);   zwrk = crr(icrs);
      if (xwrk >= arr(icrs-1)) cycle
       arr(icrs)=arr(icrs-1); brr(icrs)=brr(icrs-1); crr(icrs)=crr(icrs-1)
       do idcr = icrs - 2, 1, - 1
         if (xwrk >= arr(idcr)) exit
         arr (idcr+1) = arr (idcr)
         brr (idcr+1) = brr (idcr)
         crr (idcr+1) = crr (idcr)
       end do
       arr(idcr+1) = xwrk;   brr(idcr+1) = ywrk;    crr(idcr+1) = zwrk
      end do
      !
      return
end subroutine inssor
!-----------------------------------------------------------------------
subroutine sortadd(uanz,arr,brr,crr,ni,nnz,ia,ja,aa)
  ! For the same arr index, subsort brr, and at the same time, crr
  ! changes correspondingly with brr. After this work, adding up all crr
  ! components with the same (arr, brr) or (brr, arr) index, and the
  ! zero-value crr entry will be removed. Finally forming the Compressed
  ! Sparse Row (CSR) format or Compressed Sparse Column (CSC) format
  ! given by (ia,ja,aa).
  !         uanz: the nonzero number of arr (or brr, crr).
  !  arr,brr,crr: three vectors required to be sorted.
  !           ni: = n + 1 (n is dimension of A)
  !          nnz: the nonzero number of aa.
  !     ia,ja,aa: the CSR or CSC storage of A.
    integer:: i,j,k,k1,k2,m,arr(:),brr(:),uanz,nnz,ni,ia(:),ja(:)
    integer, allocatable:: itep(:)
    real(8):: crr(:) ,aa(:)
    allocate (itep(ni))
    call quicksort(uanz,arr,brr,crr) ;          ! sorting three vectors
    k=1;  itep(1)=1
    do i=2, uanz
       if(arr(i)/=arr(i-1)) then
        k=k+1 ;  itep(k)=i
       end if
    end do
    itep(k+1)=uanz+1
  !-----------------------------
```

```
   do i=1, k
       k1=itep(i);  k2=itep(i+1)-1
       j=k2-k1+1
       if(j<=16) then       ! sub-brr sorting by Insertion sort if j <= 16.
         call subbrr2(brr(k1:k2),crr(k1:k2),j)
      else                  ! quick sorting when j is larger (>16).
       call quicksort2(j,brr(k1:k2),crr(k1:k2))
     end if
   end do
  !----------------------------
   m=0; aa=.0
   do i=1, k
     k1=itep(i);  k2=itep(i+1)-1  ;  m=m+1;   ia(i)=m ;  ja(m)= brr(k1)
     do j=k1, k2-1
       aa(m)=aa(m)+crr(j)
       if(brr(j+1)/=brr(j) ) then
         if(aa(m)/=.0) then
           m=m+1 ; ja(m)= brr(j+1)   !  aa(m) is removed when it is zero.
         else
           ja(m)= brr(j+1)
         end if
       end if
     end do
     aa(m)=aa(m)+crr(k2)
     if(aa(m)==.0) m=m-1
   end do
   ia(k+1)=m+1;  nnz=m
   !
   return
end subroutine sortadd
!-----------------------------------------------------------------------
subroutine quicksort2(uanz,arr,crr)
  ! This subroutine Quicksort2
  !        - sorts arr into ascending order, crr changes correspondingly.
  !  Sorts arr into ascending order - Quicksort
  !  Quicksort chooses a "pivot" in the set, and explores the
  !  array from both ends, looking for a value > pivot with the
  !  increasing index, for a value <= pivot with the decreasing
  !  index, and swapping them when it has found one of each.
  !  The array is then subdivided in 2 ([3]) subsets:
  !  { values <= pivot} {pivot} {values > pivot}
  !  One then call recursively the program to sort each subset.
  !  When the size of the subarray is small enough, one uses an
  !  insertion sort that is faster for very small sets.
  !  Sorting an array arr(1:n) into ascending order with quicksort,
  !  while making the corresponding rarrangements of arrays crr(1:n).
  ! (Revised from ORDERPACK codes)
  !-------------------------------
     real(8):: crr(:)
     integer::uanz,arr(:)
     !
     call subsort2(arr,crr,1, uanz)
     call inssor2(arr,crr,uanz)
     !
     return
end subroutine quicksort2
!-----------------------------------------------------------------------
Recursive subroutine subsort2(arr,crr, ideb1, ifin1)
  ! This subroutine sorts arr from ideb1 to ifin1
     real(8)::  crr(:), zwrk
     integer, intent (in) :: ideb1, ifin1
     integer :: arr(:),  icrs, ideb, idcr, ifin, imil, xpiv, &
                xwrk,  nins = 16  ! Max for insertion sort
        ideb = ideb1
        ifin = ifin1
  ! if we don’t have enough values to make it worth while, we leave
  ! them unsorted, and the final insertion sort will take care of them
    if ((ifin - ideb) > nins) Then
         imil = (ideb+ifin) / 2
  ! One chooses a pivot, median of 1st, last, and middle values
  !
         if (arr(imil) < arr(ideb)) Then
    xwrk = arr(ideb) ;               zwrk=crr(ideb)
    arr(ideb) = arr(imil);  crr(ideb) = crr(imil)
    arr(imil) = xwrk;        crr(imil) = zwrk
         end if
         if (arr(imil) > arr(ifin)) Then
    xwrk = arr(ifin);               zwrk = crr(ifin)
    arr(ifin) = arr(imil);  crr(ifin) = crr(imil)
    arr(imil) = xwrk;       crr(imil) = zwrk
            if (arr(imil) < arr(ideb)) Then
    xwrk = arr(ideb);               zwrk = crr(ideb)
    arr(ideb) = arr(imil);  crr(ideb) = crr(imil)
    arr(imil) = xwrk;       crr(imil) = zwrk
            end if
         end if
         xpiv = arr(imil)
  !
  !  One exchanges values to put those > pivot in the end and
  !  those <= pivot at the beginning
  !
         icrs = ideb
         idcr = ifin
         ech2: do        !-------------------------
           do
               icrs = icrs + 1
               if (icrs >= idcr) Then
  !
  !  the first  >  pivot is idcr
  !  the last   <= pivot is icrs-1
  !  Note: if one arrives here on the first iteration, then
  !        the pivot is the maximum of the set, the last value is equal
  !        to it, and one can reduce by one the size of the set to
  !        process, as if arr(ifin) > xpiv
  !
                  exit ech2
  !
               end if
               if (arr(icrs) > xpiv) exit
            end do
            !
            do
               if (arr(idcr) <= xpiv) exit
               idcr = idcr - 1
               if (icrs >= idcr) Then
  !
  !  The last value < pivot is always icrs-1
  !
                  exit ech2
               end if
            end do
            !
            xwrk = arr(idcr);               zwrk = crr(idcr)
            arr(idcr) = arr(icrs);   crr(idcr) = crr(icrs)
            arr(icrs) = xwrk;        crr(icrs) = zwrk
         end do ech2       !---------------------------
  !
  !  One now sorts each of the two sub-intervals
  !
         call subsort2 (arr,crr, ideb1, icrs-1)
         call subsort2 (arr,crr, idcr, ifin1)
      end if
      !
      return
```

```
end subroutine subsort2
!-----------------------------------------------------------------------
subroutine inssor2(arr,crr,uanz)
  ! This subroutine sorts arr into increasing order (Insertion sort)
     integer :: arr(:),uanz, icrs, idcr, xwrk
     real(8) :: crr(:),zwrk
  !
     do icrs = 2, uanz
        xwrk = arr (icrs);         zwrk = crr (icrs);
        if (xwrk >= arr(icrs-1)) cycle
          arr (icrs) = arr (icrs-1);  crr (icrs) = crr (icrs-1)
          do idcr = icrs - 2, 1, - 1
            if (xwrk >= arr(idcr)) exit
            arr (idcr+1) = arr (idcr)
            crr (idcr+1) = crr (idcr)
          end do
          arr (idcr+1) = xwrk;       crr (idcr+1) = zwrk
     end do
     !
     return
end subroutine inssor2
!-----------------------------------------------------------------------
subroutine subbrr2(br,cr,n)
  ! For the same arr index, subsort brr, and at the same time, crr
  ! changes correspondingly with brr. Because of small number of sub-brr,
  ! Insertion sort should be faster.
     integer:: br(n), icrs,idcr,n,ywrk
     real(8):: cr(n), zwrk
     !
     do icrs = 2, n
        ywrk = br (icrs);       zwrk = cr (icrs)
        if (ywrk >= br(icrs-1)) Cycle
          br (icrs) = br (icrs-1); cr (icrs) = cr (icrs-1)
          do idcr = icrs - 2, 1, - 1
            if (ywrk >= br(idcr)) exit
            br (idcr+1) = br (idcr)
            cr (idcr+1) = cr (idcr)
          end do
          br (idcr+1) = ywrk;    cr (idcr+1) = zwrk
     end do
end subroutine subbrr2
!-----------------------------------------------------------------------
subroutine formspars(ntot,g,ke,iebea,jebea,ebea,ebeanz)
  ! This subroutine collect non-zero entries for each new generated    &
  !                 element stiffness matrix, forming the element-level&
  !                 three vectors which store nonzero entries of upper &
  !                 triangular part of A.
  !           ntot: total freedoms per element;
  !              g: element steering vector;
  !             ke: element "stiffness" matrix;
  !          iebea: global row index;
  !          jebea: global column index;
  !           ebea: correspondent value of the nonzero element stiffness&
  !                 entry;
  !         ebeanz: returned true total number of nonzero element-level
  !                 entries, not estimated number any more when returned.
  implicit none
  real(8):: ke(:,:),ebea(:)
  integer::i,j,ntot,ebeanz,g(:),iebea(:),jebea(:)
  !--- Storing upper triangle of element stiffness column by column ---
         do j=1, ntot
           do i=1, j
             if(g(i)/=0.and.g(j)/=0) then
               if(ke(i,j)/=.0)then
                 if(g(i)<=g(j) )then
                   ebeanz=ebeanz+1 ;  iebea(ebeanz)=g(i)
                   jebea(ebeanz)=g(j) ; ebea(ebeanz)=ke(i,j)
                 else
                   ebeanz=ebeanz+1 ;  iebea(ebeanz)=g(j)
                   jebea(ebeanz)=g(i) ; ebea(ebeanz)=ke(i,j)
                 end if
               end if
             end if
           end do
         end do
         !
         return
end subroutine formspars
!-----------------------------------------------------------------------

!---- SUBROUTINES FOR MATRIX-VECTOR PRODUCTS AND TRIANGULAR SOLVERS ----
!-----------------------------------------------------------------------
subroutine cscbx(icscb,jcscb,cscb,x,y)
  ! Compute y=B*x, and B is stored in CSC (icscb,jcscb,cscb) format.
  ! x: input vector
  ! y: output vector
    implicit none
    real(8):: cscb(:),x(:),y(:)
    integer::i,j,n,k1,k2,icscb(:),jcscb(:)
      n=ubound(jcscb,1)-1 ; y=.0 ;
      do i=1, n
        if(x(i)/=.0) then
          k1=jcscb(i); k2=jcscb(i+1)-1
          do j=k1, k2
            y(icscb(j))=y(icscb(j))+cscb(j)*x(i)
          end do
        end if
      end do
end subroutine cscbx
!-----------------------------------------------------------------------
subroutine cscbtx(icscb,jcscb,cscb,x,y)
  ! Compute y=B'*x, and B is stored in CSC (icscb,jcscb,cscb) format.
  ! x: input vector
  ! y: output vector
    implicit none
    real(8):: cscb(:),x(:),y(:)
    integer::i,j,n,k1,k2,icscb(:),jcscb(:)
      n=ubound(jcscb,1)-1 ; y=.0 ;
      do j=1, n
        k1=jcscb(j); k2=jcscb(j+1)-1
        do i=k1, k2
          y(j)=y(j)+cscb(i)*x(icscb(i))
        end do
      end do
end subroutine cscbtx
!-----------------------------------------------------------------------
subroutine csrbx(icsrb,jcsrb,csrb,x,y)
  ! Compute y=B*x, and B is stored in CSR (icsrb,jcsrb,csrb) format.
  ! x: input vector
  ! y: output vector
    implicit none
    real(8):: csrb(:),x(:),y(:)
    integer::i,j,k1,k2,n,icsrb(:),jcsrb(:)
       n=ubound(icsrb,1)-1 ; y=.0 ;
       do i=1, n
         k1=icsrb(i); k2=icsrb(i+1)-1
         do j=k1, k2
           y(i)=y(i)+csrb(j)*x(jcsrb(j))
         end do
       end do
end subroutine csrbx
!-----------------------------------------------------------------------
```

```
subroutine csrbtx(icsrb,jcsrb,csrb,x,y)
  ! Compute y=B'*x, and B is stored in CSR (icsrb,jcsrb,csrb) format.
  ! x: input vector
  ! y: output vector
    implicit none
    real(8):: csrb(:),x(:),y(:)
    integer::i,j,n,k1,k2,icsrb(:),jcsrb(:)
      n=ubound(icsrb,1)-1 ;  y=.0 ;
      do i=1, n
        if(x(i)/=.0) then
          k1=icsrb(i); k2=icsrb(i+1)-1
          do j=k1, k2
            y(jcsrb(j))=y(jcsrb(j))+csrb(j)*x(i)
          end do
        end if
      end do
   end subroutine csrbtx
!-----------------------------------------------------------------------
subroutine csrax(icsr,jcsr,csra,x,y)
  ! Compute y = Ax with A is symmetric and square, only upper triangular&
  !                 part is stored.
  !              n: dimension of coefficient matrix A;
  ! icsr,jcsr,csra: CSR storage of upper triangular part of matrix A;
  !              x: is input vector;
  !              y: is output vector.
   implicit none
   real(8):: csra(:),x(:),y(:)
   integer::i,j,k1,k2,n,icsr(:),jcsr(:)
     n = ubound(icsr,1)-1 ; y=.0 ;
     do i=1, n
       k1=icsr(i); k2=icsr(i+1)-1
       do j=k1, k2
         y(i)=y(i)+csra(j)*x(jcsr(j))
       end do
       !
       if(x(i)/=.0) then
        do j=k1+1, k2
         y(jcsr(j))=y(jcsr(j))+csra(j)*x(i)
        end do
       end if
     end do
end subroutine csrax
!-----------------------------------------------------------------------
subroutine cscax(icsc,jcsc,csca,x,y)
  ! Compute y = Ax with A is symmetric and square, only upper triangular&
  !                 part is stored.
  !              n: dimension of coefficient matrix A;
  ! icsc,jcsc,csca: CSC storage of upper triangular part of matrix A;
  !              x: is input vector;
  !              y: is output vector.
  implicit none
  real(8):: csca(:),x(:),y(:),tmp
  integer::j,k,r,k1,k2,n,jcsc(:),icsc(:)
    n = ubound(jcsc,1)-1 ; y=.0
    do j=1, n
      if(x(j)/=.0)then
        k1=jcsc(j); k2=jcsc(j+1)-1 ;
        do k=k1, k2
          r=icsc(k) ;
          y(r)=y(r)+csca(k)*x(j) ;
        end do
      end if
      !
      tmp=.0 ; k1=jcsc(j); k2=jcsc(j+1)-2 ;
      do k=k1, k2
```

```
        r=icsc(k) ;
        tmp = tmp+x(r)*csca(k) ;
      end do
      y(j) = y(j)+tmp ;
    end do
    !
    return
end subroutine cscax
!-----------------------------------------------------------------------
subroutine lsolve(n, da1,icsc,jcsc,csca,b, x)
  !  This subroutine performs forward solve of MSSOR, that is,
  !                 (L+DA)x = b (provided for sqmrmssor subroutine).
  !              n: dimension of coefficient matrix A;
  ! icsc,jcsc,csca: CSC storage of upper triangular part of matrix A;
  !            da1: inverse of da (da: modified diagonal for MSSOR);
  !              b: it is right hand vector b;
  implicit none
  real(8):: da1(:), csca(:), b(:), x(:), tmp
  integer::i, j, k1, k2, n, icsc(:), jcsc(:)
  ! -------- forward substitution --------
     x(1)=b(1)*da1(1);
     do j=2, n
        k1=jcsc(j); k2=jcsc(j+1)-1 ; tmp=.0
        do i=k1, k2-1
          tmp = tmp + csca(i) * x(icsc(i)) ;
        end do
          tmp = b(j) - tmp
          x(j) = tmp*da1(j) ;
     end do
     !
     return
end subroutine lsolve
!-----------------------------------------------------------------------
subroutine usolve(n, da1,icsc,jcsc,csca,b, x)
  !  This subroutine performs backward solve of MSSOR, that is,
  !                 (DA+U)x = b (provided for sqmrmssor subroutine).
  !              n: dimension of coefficient matrix A;
  ! icsc,jcsc,csca: CSC storage of upper triangular part of matrix A;
  !            da1: inverse of da (da: modified diagonal for MSSOR);
  !              b: it is right hand vector b;
  implicit none
  real(8):: da1(:),csca(:),b(:),x(:)
  real(8),allocatable:: tmp(:)
  integer::j,r,k,k1,k2,n,icsc(:), jcsc(:)
  allocate(tmp(n) )
  ! ----- backward substitution -----
     tmp = b ;
     do k = n, 2, -1
       x(k) = tmp(k)*da1(k)
       do j = jcsc(k), jcsc(k+1)-2
          r = icsc(j)
          tmp(r) = tmp(r) - x(k)*csca(j)
       end do
     end do
     x(1) = tmp(1)*da1(1)
     !
     return
end subroutine usolve
!-----------------------------------------------------------------------
subroutine gtor(n,icsc,jcsc,csca,da,g,r)
  !  This subroutine performs lower triangular product, that is,
  !                 (L+DA)g = r (provided for sqmrmssor subroutine).
  !              n: dimension of coefficient matrix A;
  ! icsc,jcsc,csca: CSC storage of upper triangular part of matrix A;
  !             da: the modified diagonal for MSSOR;
  !              g: it is input vector(preconditioned residual);
  !              r: output vector (returned true residual).
  implicit none
  real(8):: da(:),csca(:),g(:),r(:)
  integer::i,j,n,k1,k2,icsc(:), jcsc(:)
     r=.0 ; r(1)=r(1)+da(1)*g(1);
     do j=2, n
       k1=jcsc(j); k2=jcsc(j+1)-2
       do i= k1, k2
         r(j)=r(j)+csca(i)*g(icsc(i))
       end do
       r(j)=r(j)+da(j)*g(j);
     end do
     !
     return
end subroutine gtor
!-----------------------------------------------------------------------
subroutine kpu(icsc,jcsc,csca,theta,id,x,y)
  ! this subroutine performs the KP*u product for Biot's incremental
  !                 formula.
  ! icsc,jcsc,csca: CSC storage of upper triangular part of A.
  !          theta: implicit time-stepping parameter [0.5, 1].
  !             id: identifier array for displacement or porepressure DOF;
  !              id(i) = 1, displacement DOF;
  !              id(i) = 0, pore pressure DOF.
  !              x: x = u is the current excess pore pressure.
    implicit none
    real(8):: theta, csca(:), x(:), y(:)
    integer::j,k,r,k1,k2,n,jcsc(:),icsc(:),id(:)
      !
      n=ubound(jcsc,1)-1 ;  y =.0
      do j=1, n
        if( id(j)==0 ) then            ! pore pressure DOF
          if( x(j)/=.0)then
            k1=jcsc(j);  k2=jcsc(j+1)-1
            do k = k1, k2
              if(id(icsc(k))==0) y(icsc(k))=y(icsc(k)) + csca(k)*x(j)
            end do
          end if
          !
          k1=jcsc(j);  k2=jcsc(j+1)-2
          do k = k1, k2
            if(id(icsc(k))==0) y(j)=y(j) + csca(k)*x(icsc(k))
          end do
        end if
      end do
      !
      do j=1, n;
        if(id(j)==0) y(j)=-y(j)/theta ;
      end do
      !
      return
end subroutine kpu

!-----------------------------------------------------------------------
!-------------- PRECONDITIONED SPARSE SOLVER
SUBROUTINES----------------
!-----------------------------------------------------------------------
subroutine formda(n,icsc,jcsc,csca,icho,ipre,coef,omega,id,d,da,da1)
  ! This subroutine forms GJ diagonal vector - da (d and da1);
  ! In this routine:
  !              n: dimension of coefficient matrix A;
  ! icsc,jcsc,csca: CSC storage of upper triangular part of matrix A;
  !           icho: choose standard or modified preconditioenr.
  !               =1: standard preconditioner.
  !               =2: generalized or modified preconditioner.
```

```fortran
  !          ipre: choose preconditioner,
  !               =1: Jacobi preconditioner.
  !               =2: SSOR preconditioner (omega should be applied).
  !           coef: the scaling factor for GJ diagonal vector.
  !          omega: relaxation parameter, which is applied to MSSOR.
  !             id: a vector to indicate the type of current DOF,
  !                 id(j)= 0 for pore water pressure DOF;
  !                 id(j)= 1 for displacement DOF.
  !              d: diagonal of A;
  !             da: modified diagonal for MSSOR preconditioner;
  !            da1: inverse of da;
    implicit none
    real(8):: coef,omega,d(:),da(:),da1(:),csca(:),absv, maxabs,minabs
    integer::n,j,r,k,k1,k2,icho,ipre,id(:),icsc(:), jcsc(:)
      !
      do j=1, n;
        r=jcsc(j+1)-1 ; da(j) = csca(r);
      end do
      !
      if(ipre==2) d = da ; ! Transfer diagonal of A from da to d;
      if(icho == 2)then     ! For generalized or modified preconditioner
        !
        do j=2, n
          k1=jcsc(j) ;  k2=jcsc(j+1)-2
          if(id(j)==1) then
            do k=k1 , k2
              if(id(icsc(k))==0 ) then
                da(icsc(k))=da(icsc(k))-csca(k)**2/da(j) ;
              end if
            end do
          else                     ! id(j)==0
            do k=k1 , k2
              if(id(icsc(k))== 1 ) then
                da(j)=da(j)-csca(k)**2/da(icsc(k)) ;
              end if
            end do
          end if
        end do
        !
        coef = coef/omega ;
        do j=1, n          ! coef-scaling factor (negative is preferred)
          if(id(j)==0)then ! modified diagonal with relaxation parameter.
            da(j)=coef*abs(da(j))
          else             ! id(j)==0
            da(j)=da(j)/omega
          end if
        end do
      end if
      da1 = 1./da ;
      !
      return
end subroutine formda
!-----------------------------------------------------------------------
  subroutine sqmrmssor(n,icsc,jcsc,csca,d,da,da1,rhs,maxit,tol,icc, &
                       iinc,qmriters,relres)
  ! Modified SSOR preconditioned SQMR for symmetric Ax=b linear system.
  ! Combining with Eisenstat trick.
  ! In this routine:
  !              n: dimension of coefficient matrix A;
  ! icsc,jcsc,csca: CSC storage of upper triangular part of matrix A;
  !              d: diagonal of A;
  !             da: modified diagonal for MSSOR preconditioner;
  !            da1: inverse of da;
  !            rhs: at input, it is right hand vector b;
  !                 at output,it is returned approximate solution x;
  !          maxit: user-defined maximum iteration count;
  !            tol: it is the user-defined stopping tolerance;
  !            icc: choice for convergence criterion;
  !                 = 1, relative improvement norm criterion.
  !                 = 2, relative residual norm criterion (x0=.0)
  !           iinc: Check convergence every 'iinc' iteration.
  !       qmriters: the iterative count when SQMR converges;
  !         relres: the relative residual when SQMR converges.
  implicit none
  integer::i,n,maxit,icc,iinc,qmriters,ic,icsc(:),jcsc(:)
  real(8)::tol,relres,tao,theta0,theta,rho0,rho,phi,nrmb,nrmr,sigma,   &
           kesa,beta,rhs(:),csca(:),d(:),da(:),da1(:)
  real(8),allocatable::x(:), xold(:), z(:), r(:), g(:),v(:),w(:),c(:), &
                       t1(:),t2(:),t(:),p(:),q(:)
  allocate(x(n),xold(n),z(n),r(n),g(n),v(n),w(n),c(n),t1(n),t2(n),t(n),&
           p(n),q(n) )
  !---------- initialize vectors ------------
      x = .0 ; z = .0                           ! initial guess
      r=rhs;
      call lsolve(n,da1,icsc,jcsc,csca,r, g); ! preconditioned residual
      v =g;   w=da*v;
      tao = dot_product(g, g);
      rho0 = dot_product(g, w)
      c=.0 ; theta0=.0 ;
      nrmb=sqrt(dot_product(r, r))*tol ;
      p=d-2.0*da  ;                       ! p is used in each iteration
      ic = 0;
  !---------- SQMR iteration ----------
  iteration: do i=1, maxit
        !----- matrix-vector product with Eisenstat trick -----
              call usolve(n, da1,icsc,jcsc,csca,w, t1);
              t2 = p*t1+w ;
              call lsolve(n, da1,icsc,jcsc,csca,t2, t);
              t =  t1 + t;
         !-----------------------------------------------------
         sigma=dot_product(w, t);
         if(sigma==.0) then
           write(11,*) 'SQMR stops due to Sigma=0 ';  stop
         end if
         kesa=rho0/sigma ; g = g - kesa*t ;
         theta=dot_product(g, g)/tao ;
         phi=1./(1.+theta) ;
         tao=tao*theta*phi ;
         c=phi*(theta0*c+kesa*v) ;
         z = z + c ;
         select case (icc)
           case (1)
             call ccri(n,i,icsc,jcsc,csca,da,da1,z,iinc,tol, xold, &
                       rhs,ic,qmriters,relres)
           case (2)
             call ccrb(n,i,icsc,jcsc,csca,da,da1,g,z,iinc,tol,nrmb,&
                       rhs,ic,qmriters,nrmr,relres)
         end select
         if(ic==1) return ;
         if(rho0==.0)then
           write(11,*) 'SQMR stops due to rho0=0 ' ; stop
         end if
         q = da * g ;
         rho=dot_product(g, q) ;
         beta = rho/rho0 ;  v= g + beta*v;
         w= da*v;
         theta0=theta;  rho0= rho
  end do iteration
  !--------- End iteration --------
```

```
     write(11,*)'********************************************** '
     write(11,*) 'SQMR does not converge to user-defined tolerance. '
        z = da * z;
        call usolve(n, da1,icsc,jcsc,csca,z, x)
        if(icc==1) relres = nrmr*tol/nrmb ;
        qmriters = maxit ;  rhs = x ;
  return
end subroutine sqmrmssor
!-----------------------------------------------------------------------
subroutine ccrb(n,i,icsc,jcsc,csca,da,da1,g,z,iinc,tol,nrmb,rhs,ic, &
                iters,nrmr,relres)
  ! This subroutine performs convergence check in terms of relative
  !                 residual with x0=.0 is chosen,
  !              n: i.e. neq - number of total DOFs or equations;
  !              i: current iteration # ;
  ! icsc,jcsc,csca: CSC storage of upper triangular part of matrix A;
  !             da: modified diagonal for MSSOR preconditioner;
  !            da1: inverse of da;
  !              g: preconditioned residual;
  !              z: "preconditioned" solution;
  !           iinc: Check convergence every 'iinc' iteration.
  !            tol: it is the user-defined stopping tolerance;
  !           nrmb: computed initial residual (b) norm multiplied by tol;
  !            rhs: at input, it is right hand vector b;
  !                 at convcergence,it is returned approximate solution x;
  !             ic: = 1 converged (ic is a identifier);
  !                 = 0 doesn't satisfy the convergence criterion;
  !          iters: returned iteration count when converged;
  !           nrmr: norm of true residual.
  !         relres: returned relative residual when converged.
    implicit none
    integer:: i,j,n,iinc,iters,ic,icsc(:),jcsc(:)
    real(8):: tol,nrmb,nrmr,relres,csca(:),da(:),da1(:),g(:),z(:),rhs(:)
    real(8),allocatable:: r(:)
    allocate( r(n) )
    !
      if(mod(i, iinc)==0)then       !  per iinc steps, check convergence
        call gtor(n,icsc,jcsc,csca,da,g,r) ; ! true residual is computed
        nrmr=sqrt(dot_product(r, r)) ;
        if(nrmr < nrmb)then          !  solver converged
          write(11,*)'********************************************** '
          write(11,*) 'SQMR converges to user-defined tolerance. '
          z = da * z ;
          call usolve(n,da1,icsc,jcsc,csca,z,rhs) ! rhs is the solution.
          ic =1 ; iters = i ; relres = nrmr*tol/nrmb ;
          return
        end if
      end if
      !
      return
  end subroutine ccrb
!-----------------------------------------------------------------------
subroutine ccri(n,i,icsc,jcsc,csca,da,da1,z,iinc,tol,xold,rhs,ic,iters,&
                relres)
  ! This subroutine performs convergence check in terms of rleative
  !                   'improvement' norm criterion ( when set icc = 2 )
  !                n: i.e. neq - number of total DOFs or equations;
  !                i: current iteration # ;
  ! icsc,jcsc,csca: CSC storage of upper triangular part of matrix A;
  !               da: modified diagonal for MSSOR preconditioner;
  !              da1: inverse of da;
  !                z: "preconditioned" solution;
  !             iinc: Check convergence every 'iinc' iteration.(iinc > 1)
  !              tol: it is the user-defined stopping tolerance;
  !             xold: solution of the previous iterative step;
  !            rhs: at input, it is right hand vector b;
  !                 at convcergence, it is returned solution;
  !             ic: = 1 converged (ic is a identifier);
  !                 = 0 doesn't satisfy the convergence criterion;
  !          iters: returned iteration count when converged;
  !         relres: returned relative residual when converged.
     implicit none
     integer:: i,j,n,iinc,iters,ic,icsc(:),jcsc(:)
     real(8):: big,tol,ratio,relres,csca(:),da(:),da1(:),rhs(:),z(:),&
               xold(:)
     real(8),allocatable:: t(:)
     allocate(t(n) )
     !
       if( mod(i, iinc)==0 )then      !  per iinc steps, compute a xold;
         t = da * z ;
         call usolve(n, da1,icsc,jcsc,csca,t, rhs) ;
         xold = rhs ;
         !
       else if( mod(i, iinc)==1 )then ! check convergence, closely next
                                      !             per iinc iterations;
         t = da * z ;
         call usolve(n, da1,icsc,jcsc,csca,t, rhs) ;
         big=.0 ;
         do j = 1, n; if( abs(rhs(j)) > big ) big=abs(rhs(j));  end do
         relres =.0
         do j = 1, n;
           ratio = abs(rhs(j)-xold(j))/big;
           if(ratio > relres ) relres = ratio;
         end do
         if(relres < tol) ic=1;       !  Solver converged
         if(ic==1)then
           write(11,*)'********************************************* '
           write(11,*) 'SQMR converges to user-defined tolerance. '
           iters = i
           !
           return
         end if
         !
       end if
       !
       return
end subroutine ccri
!-------------------------------------------------------------------------
subroutine psqmr(n,icsc,jcsc,csca,pr,rhs,maxit,tol,icc,qmriters,relres)
  ! This subroutine uses SQMR to solve Ax=b linear system with a right
  !                 diagonal preconditioner.
  ! In this routine:
  !              n: dimension of coefficient matrix A;
  ! icsc,jcsc,csca: CSC storage of coefficient matrix A;
  !             pr: right preconditioner (which is inverted at input);
  !            rhs: at input, it is right hand vector b;
  !                 at output,it is returned approximate solution x;
  !          maxit: user-defined maximum iteration count;
  !            tol: it is the user-defined stopping tolerance;
  !            icc: choice for convergence criterion;
  !                 = 1, relative improvement norm criterion
  !                 = 2, relative residual norm criterion (x0=.0)
  !             ic: indentifier of convergence;
  !                 = 1, solver converged;
  !                 = 0, not converge.
  !       qmriters: the iterative count when SQMR converges;
  !         relres: the relative residual when SQMR converges.
  implicit none
  integer::i,n,maxit,qmriters,icc,ic,icsc(:),jcsc(:)
  real(8)::tol,relres,tao,theta0,rho0,rho,nrmb,nrmr,sigma,alpha,beta,&
           theta,cj,rhs(:),csca(:),pr(:)
  real(8),allocatable::x(:),xold(:),r(:),t(:),q(:),d(:),u(:)
  allocate(x(n),xold(n),r(n),t(n),q(n),d(n),u(n)  )
  !------ Initial vectors of SQMR iterations ------
      x=.0                            ! assumed initial guess
      r=rhs;
      t=r;                            ! left preconditioning
      q=pr*t;                         ! right preconditioning
      tao=sqrt(dot_product(t, t) );
      theta0=.0;
      rho0=dot_product(r,q);
      nrmb=sqrt(dot_product(r, r))*tol;
      d=.0; ic=0 ; xold = x ;
  !------------ Sart SQMR iterations -------------
  iteration: do i=1, maxit
    call cscax(icsc,jcsc,csca,q,t)   !  t=A*q, Matrix-vector product.
    sigma=dot_product(q,t);
    alpha=rho0/sigma ;
    r = r-alpha*t ;
    t= r;                             ! left preconditioning
    theta=sqrt(dot_product(t, t) )/tao ;
    cj=1./sqrt(1+theta*theta);
    tao=tao*theta*cj;
    d=(cj*theta0)**2*d+(cj*cj)*alpha*q ;
    x = x + d;
    nrmr=sqrt(dot_product(r, r)) ;
    select case (icc)
      case (1)
        call pccri(n,i,xold,x,rhs,tol,ic,qmriters,relres)
        xold = x ;
      case (2)
        call pccrb(n,i,r,x,rhs,tol,nrmb,ic,qmriters,nrmr,relres)
    end select
    if(ic==1) return ;           ! SQMR converged
    u=pr*t ;                     ! right preconditioning
    rho=dot_product(r,u);
    beta=rho/rho0;  q = u + beta*q ;
    !
    rho0=rho; theta0=theta;
  end do iteration
     write(11,*)'*********************************************** '
     write(11,*) 'SQMR does not converge to user-defined tolerance. '
     relres=nrmr*tol/nrmb ;  qmriters = maxit ;  rhs=x
     !
     return
end subroutine psqmr
!------------------------------------------------------------------------
subroutine pccrb(n,i,r,x,rhs,tol,nrmb,ic,iters,nrmr,relres)
  ! This subroutine performs convergence check in terms of relative
  !             residual with x0=.0 is chosen,
  !          n: i.e. neq - number of total DOFs or equations;
  !          i: current iteration # ;
  !          r: current residual;
  !        rhs: at input, it is right hand vector b;
  !             at convcergence,it is returned approximate solution x;
  !        tol: it is the user-defined stopping tolerance;
  !       nrmb: computed initial residual (b) norm multiplied by tol;
  !         ic: = 1 converged (ic is a identifier);
  !             = 0 doesn't satisfy the convergence criterion;
  !      iters: returned iteration count when converged;
  !       nrmr: norm of true residual.
  !     relres: returned relative residual when converged.
     implicit none
     integer:: i,j,n,iters,ic
     real(8):: tol,nrmb,nrmr,relres,r(:),x(:),rhs(:)
     !
```

```
       nrmr=sqrt(dot_product(r, r)) ;
       if(nrmr < nrmb)then          !  solver converged
         write(11,*)'********************************************* '
         write(11,*) 'Psolver converges to user-defined tolerance. '
         ic =1 ; iters = i ; relres = nrmr*tol/nrmb ; rhs = x;
         !
         return
       end if
       !
       return
  end subroutine pccrb
!-----------------------------------------------------------------------
subroutine pccri(n,i,xold,x,rhs,tol,ic,iters,relres)
  ! This subroutine performs convergence check in terms of rleative
  !                 'improvement' norm criterion ( when set icc = 2 )
  !              n: i.e. neq - number of total DOFs or equations;
  !              i: current iteration # ;
  !           xold: approximate solution of the previous iterative step;
  !              x: approximate solution of current iterative step;
  !            rhs: at input, it is right hand vector b;
  !                 at convcergence, it is returned solution;
  !            tol: it is the user-defined stopping tolerance;
  !             ic: = 1 converged (ic is a identifier);
  !                 = 0 doesn't satisfy the convergence criterion;
  !          iters: returned iteration count when converged;
  !         relres: returned relative residual when converged.
     implicit none
     integer:: i,j,n,iters,ic
     real(8):: big,tol,ratio,relres,rhs(:),xold(:),x(:)
     !
         big=.0 ;
         do j = 1, n; if( abs(x(j)) > big ) big=abs(x(j));  end do
         relres =.0
         do j = 1, n;
           ratio = abs(x(j)-xold(j))/big;
           if(ratio > relres ) relres = ratio;
         end do
         if(relres < tol) ic=1;       !  Solver converged
         if(ic==1)then
           write(11,*)'********************************************* '
           write(11,*) 'Psolver converges to user-defined tolerance. '
           iters = i ; rhs = x ;
           !
           return
         end if
         !
         return
end subroutine pccri
! ----------------------------------------------------------------------
! --------- Preconditioned CG Method for Linear System Ax = b  ---------
! ----------------------------------------------------------------------
subroutine pcg(n,icsc,jcsc,csca,pr,rhs,maxit,tol,icc,iters,relres)
  ! This subroutine uses PCG to solve Ax=b linear system with a right
  !                 diagonal preconditioner.
  ! In this routine:
  !              n: dimension of coefficient matrix A;
  ! icsc,jcsc,csca: CSC storage of coefficient matrix A;
  !             pr: right preconditioner (which is inverted at input);
  !            rhs: at input, it is right hand vector b;
  !                 at output,it is returned approximate solution x;
  !          maxit: user-defined maximum iteration count;
  !            tol: it is the user-defined stopping tolerance;
  !            icc: choice for convergence criterion;
  !                 = 1, relative improvement norm criterion
  !                 = 2, relative residual norm criterion (x0=.0)
  !             ic: indentifier of convergence;
  !                  = 1, solver converged;
  !                  = 0, not converge.
  !           iters: the iterative count when PCG converges;
  !          relres: the relative residual when PCG converges.
  ! This subroutine is for preconditioned PCG iterative method.
  ! Refer to the PCG Algorithm by van der Vorst's book or Template Book.
integer:: i,n,ic,icc,icsc(:),jcsc(:),maxit,iters
real(8):: pr(:),csca(:),tol,rhs(:),nrmb,nrmr,rho,rho0,alpha,beta,relres
real(8),allocatable:: r(:),z(:),p(:),q(:),x(:),xold(:)
allocate (r(n),z(n),p(n),q(n),x(n),xold(n) )
!  b -- is RHS vector when inputting, while it is the Solution Vector
!  when returning.
   x = .0 ;  r = rhs ;              ! x0=0 is the initial solution guess
   nrmb=sqrt(dot_product(rhs, rhs))*tol;
   ic=0 ; xold = x ;
pcg_iter: do i=1, maxit
   z = pr*r ; rho = dot_product(r, z) ;
     if ( i > 1 )then               ! direction vector
        beta = rho/rho0;
        p = z + beta*p;
     else
        p = z;
     end if
   call cscax(icsc,jcsc,csca,p,q)   !  q=Ap, Matrix-vector product.
   alpha = rho/dot_product(p, q) ;
   x = x + alpha * p ;
   r = r - alpha * q ;
   nrmr=sqrt(dot_product(r, r)) ;
   select case (icc)
      case (1)
        call pccri(n,i,xold,x,rhs,tol,ic,iters,relres)
        xold = x ;
      case (2)
        call pccrb(n,i,r,x,rhs,tol,nrmb,ic,iters, nrmr,relres)

   end select
   if(ic==1)return ;          ! PCG converged
   rho0 = rho
end do pcg_iter
     write(11,*)'********************************************* '
     write(11,*) 'PCG does not converge to user-defined tolerance. '
     relres=nrmr*tol/nrmb ;  iters = maxit ;  rhs=x
    return
end subroutine pcg
! -------------------------------------------------------------------
subroutine pcgmssor(n,icsc,jcsc,csca,d,da,da1,rhs,maxit,tol,icc,iinc, &
                    iters,relres)
  ! Modified SSOR preconditioned PCG for symmetric Ax=b linear system.
  ! Combining with Eisenstat trick.
  ! In this routine:
  !               n: dimension of coefficient matrix A;
  ! icsc,jcsc,csca: CSC storage of upper triangular part of matrix A;
  !               d: diagonal of A;
  !              da: modified diagonal for MSSOR preconditioner;
  !             da1: inverse of da;
  !             rhs: at input, it is right hand vector b;
  !                  at output,it is returned approximate solution x;
  !           maxit: user-defined maximum iteration count;
  !             tol: it is the user-defined stopping tolerance;
  !             icc: choice for convergence criterion;
  !                  = 1, relative improvement norm criterion.
  !                  = 2, relative residual norm criterion (x0=.0)
  !            iinc: Check convergence every 'iinc' iteration.
  !           iters: the iterative count when PCG converges;
  !          relres: the relative residual when PCG converges.
  implicit none
  integer:: i,n,icc,ic,iinc,icsc(:),jcsc(:),maxit,iters
  real(8):: csca(:),d(:),da(:),da1(:),tol,rhs(:), nrmb, nrmr, rho,rho0,&
            alpha,beta,relres
  real(8),allocatable:: tmp(:),x(:),xold(:),z(:),t(:),t1(:),t2(:),r(:),&
            g(:),p(:),q(:)
 allocate(tmp(n),x(n),xold(n),z(n),t(n),t1(n),t2(n),r(n),g(n),p(n),q(n))
   !---------- initialize vectors ------------
   !  b is RHS vector when inputting, while it is the Solution Vector  &
   !    when returning.
   x = 0 ; z =.0;  r = rhs ;       ! x0=0 is the initial solution guess
   nrmb=sqrt(dot_product(r, r))*tol;
   call lsolve(n,da1,icsc,jcsc,csca,r, g);   ! preconditioned residual
   tmp = d - 2.0*da ; ic=0 ;
pcg_iter: do i=1, maxit
   t = da*g ; rho = dot_product(g,t) ;        !(t = z ; g =r)
   if(i==1)then
     p = t;
    else
     beta = rho/rho0 ; p = t + beta*p ;
   end if
   !  q=Ap, Matrix-vector product.
   !----- matrix-vector product with Eisenstat trick -----
     call usolve(n, da1,icsc,jcsc,csca,p, t1);
     t2 = tmp*t1 + p ;
     call lsolve(n, da1,icsc,jcsc,csca,t2, q);
     q =  t1 + q;
   !-----------------------------------------------------
   alpha = rho/dot_product(p, q) ;
   z = z + alpha * p ;
   g = g - alpha * q ;
   select case (icc)
      case (1)
         call cgccri(n,i,icsc,jcsc,csca,da,da1,z,iinc,tol, xold, &
                   rhs,ic,iters,relres)
      case (2)
         call cgccrb(n,i,icsc,jcsc,csca,da,da1,g,z,iinc,tol,nrmb,&
                   rhs,ic,iters,nrmr,relres)
   end select
   if(ic==1) return ;
   rho0 = rho
 end do pcg_iter
    !
    write(11,*)'********************************************** '
    write(11,*) 'PCG does not converge to user-defined tolerance. '
        call usolve(n, da1,icsc,jcsc,csca,z, x)
        if(icc==1) relres = nrmr*tol/nrmb ;
        iters = maxit ;  rhs = x ;
     return
end subroutine pcgmssor
!-----------------------------------------------------------------------
subroutine cgccrb(n,i,icsc,jcsc,csca,da,da1,g,z,iinc,tol,nrmb,rhs,ic, &
                iters,nrmr,relres)
  ! This subroutine performs convergence check in terms of relative
  !             residual with x0=.0 is chosen,
  !          n: i.e. neq - number of total DOFs or equations;
  !          i: current iteration # ;
  ! icsc,jcsc,csca: CSC storage of upper triangular part of matrix A;
  !         da: modified diagonal for MSSOR preconditioner;
  !        da1: inverse of da;
  !          g: preconditioned residual;
  !          z: "preconditioned" solution;
  !       iinc: Check convergence every 'iinc' iteration.
  !        tol: it is the user-defined stopping tolerance;
  !       nrmb: computed initial residual (b) norm multiplied by tol;
  !        rhs: at input, it is right hand vector b;
  !             at convcergence,it is returned approximate solution x;
  !         ic: = 1 converged (ic is a identifier);
  !             = 0 doesn't satisfy the convergence criterion;
  !      iters: returned iteration count when converged;
  !       nrmr: norm of true residual.
  !     relres: returned relative residual when converged.
    implicit none
    integer:: i,j,n,iinc,iters,ic,icsc(:),jcsc(:)
    real(8):: tol,nrmb,nrmr,relres,csca(:),da(:),da1(:),g(:),z(:),rhs(:)
    real(8),allocatable:: r(:)
    allocate( r(n) )
    !
    if(mod(i, iinc)==0)then        !  per iinc steps, check convergence
       call gtor(n,icsc,jcsc,csca,da,g,r) ; ! true residual is computed
       nrmr=sqrt(dot_product(r, r)) ;
       if(nrmr < nrmb)then          !  solver converged
          write(11,*)'********************************************* '
          write(11,*) 'PCG converges to user-defined tolerance. '
          ! z = da * z ;
          call usolve(n,da1,icsc,jcsc,csca,z,rhs) ! rhs is the solution.
          ic =1 ; iters = i ; relres = nrmr*tol/nrmb ;
          return
       end if
    end if
    !
    return
  end subroutine cgccrb
!-----------------------------------------------------------------------
subroutine cgccri(n,i,icsc,jcsc,csca,da,da1,z,iinc,tol,xold,rhs,ic, &
                  iters,relres)
  ! This subroutine performs convergence check in terms of rleative
  !                 'improvement' norm criterion ( when set icc = 2 )
  !              n: i.e. neq - number of total DOFs or equations;
  !              i: current iteration # ;
  ! icsc,jcsc,csca: CSC storage of upper triangular part of matrix A;
  !             da: modified diagonal for MSSOR preconditioner;
  !            da1: inverse of da;
  !              z: "preconditioned" solution;
  !           iinc: Check convergence every 'iinc' iteration.(iinc>1)
  !            tol: it is the user-defined stopping tolerance;
  !           xold: solution of the previous iterative step;
  !            rhs: at input, it is right hand vector b;
  !                 at convcergence, it is returned solution;
  !             ic: = 1 converged (ic is a identifier);
  !                 = 0 doesn't satisfy the convergence criterion;
  !          iters: returned iteration count when converged;
  !         relres: returned relative residual when converged.
     implicit none
     integer:: i,j,n,iinc,iters,ic,icsc(:),jcsc(:)
     real(8):: big,tol,ratio,relres,csca(:),da(:),da1(:),rhs(:),z(:), &
               xold(:)
     real(8),allocatable:: t(:)
     allocate(t(n) )
     !
       if( mod(i, iinc)==0 )then      !  per iinc steps, compute a xold;
         ! t = da * z ;
         call usolve(n, da1,icsc,jcsc,csca,z, rhs) ;
         xold = rhs ;
         !
       else if( mod(i, iinc)==1 )then ! check convergence, closely next
                                      !             per iinc iterations;
         !t = da * z ;
         call usolve(n, da1,icsc,jcsc,csca,z, rhs) ;
         big=.0 ;
         do j = 1, n; if( abs(rhs(j)) > big ) big=abs(rhs(j));  end do
         relres =.0
         do j = 1, n;
           ratio = abs(rhs(j)-xold(j))/big;
           if(ratio > relres ) relres = ratio;
         end do
         if(relres < tol) ic=1;       !  Solver converged
         if(ic==1)then
           write(11,*)'********************************************** '
           write(11,*) 'PCG converges to user-defined tolerance. '
           iters = i
           !
           return
         end if
         !
       end if
       !
       return
end subroutine cgccri
! ----------------------------------------------------------------------
! ------ MINRES method for symmetric and possible indefinite Ax=b ------
! ----------------------------------------------------------------------
subroutine minres(n,icsc,jcsc,csca,pr,rhs,maxit,tol,icc,iters,relres)
  ! This subroutine uses MINRES to solve Ax=b linear system with a right
  !                diagonal preconditioner.
  ! In this routine:
  !              n: dimension of coefficient matrix A;
  ! icsc,jcsc,csca: CSC storage of coefficient matrix A;
  !             pr: right preconditioner (which is inverted at input);
  !            rhs: at input, it is right hand vector b;
  !                 at output,it is returned approximate solution x;
  !          maxit: user-defined maximum iteration count;
  !            tol: it is the user-defined stopping tolerance;
  !            icc: choice for convergence criterion;
  !                 = 1, relative improvement norm criterion
  !                 = 2, relative residual norm criterion (x0=.0)
  !             ic: indentifier of convergence;
  !                 = 1, solver converged;
  !                 = 0, not converge.
  !          iters: the iterative count when MINRES converges;
  !         relres: the relative residual when MINRES converges.
  ! This subroutine is Jacobi preconditioned MINRES iterative method.
  ! Refer to the MINRES Algorithm. (Wang,2004)
integer:: i,n,icc,ic,icsc(:),jcsc(:),maxit,iters
real(8):: pr(:),csca(:),tol,rhs(:),nrmb,nrmr,rho,rho0,alpha,beta,relres
real(8),allocatable::r(:),u(:),p(:),q(:),x(:),xold(:),z0(:),z(:),t0(:),&
          t(:)
allocate (r(n),u(n),p(n),q(n),x(n),xold(n),z0(n),z(n),t0(n),t(n) )
   x = 0;  r = rhs ! x0=0 is the initial guess.
   p = pr*r ; z0 = p ;
   call cscax(icsc,jcsc,csca,p,q)   !  q=Ap, Matrix-vector product.
   t0 = q; nrmb = sqrt(dot_product(r, r))*tol ;
   ic=0 ; xold = x ;
mr_iter: do i=1, maxit
   u = pr * q ;
   alpha = dot_product(z0,q)/dot_product(q,u);
   x = x + alpha * p ;
   r = r - alpha * q ;
   z = z0 - alpha * u ;
   call cscax(icsc,jcsc,csca,z,t)   !  t = Az, Matrix-vector product.
   beta = dot_product(z, t)/dot_product(z0, t0) ;
   p = z + beta*p ;
   q = t + beta*q ;
   nrmr = sqrt(dot_product(r, r));
   select case (icc)
      case (1)
```

```
        call pccri(n,i,xold,x,rhs,tol,ic,iters,relres)
        xold = x ;
      case (2)
        call pccrb(n,i,r,x,rhs,tol,nrmb,ic,iters,nrmr,relres)
   end select
   if(ic==1) return ;          ! MINRES converged
   z0 = z ; t0 = t ;
end do mr_iter
     write(11,*)'********************************************** '
     write(11,*) 'MINRES does not converge to user-defined tolerance. '
     relres=nrmr*tol/nrmb ;  iters = maxit ;  rhs=x
    return
end subroutine minres
!-----------------------------------------------------------------------
subroutine mrmssor(n,icsc,jcsc,csca,d,da,da1,rhs, maxit, tol, icc,iinc,&
                   iters,relres)
   ! Modified SSOR preconditioned MINRES for symmetric Ax=b linear system.
   ! Combining with Eisenstat trick.
   ! In this routine:
   !              n: dimension of coefficient matrix A;
   ! icsc,jcsc,csca: CSC storage of coefficient matrix A;
   !             pr: right preconditioner (which is inverted at input);
   !            rhs: at input, it is right hand vector b;
   !                 at output,it is returned approximate solution x;
   !          maxit: user-defined maximum iteration count;
   !            tol: it is the user-defined stopping tolerance;
   !            icc: choice for convergence criterion;
   !                 = 1, relative improvement norm criterion
   !                 = 2, relative residual norm criterion (x0=.0)
   !             ic: indentifier of convergence;
   !                 = 1, solver converged;
   !                 = 0, not converge.
   !          iters: the iterative count when MINRES converges;
   !         relres: the relative residual when MINRES converges.
   ! This subroutine is Jacobi preconditioned MINRES iterative method.
   ! Refer to the MINRES Algorithm. (Wang,2004)
integer::i,n,icc,ic,iinc,icsc(:),jcsc(:),maxit,iters
real(8)::d(:),da(:),da1(:),csca(:),tol,rhs(:),nrmb,nrmr,rho,rho0,alpha,&
          beta,relres
real(8),allocatable::tmp(:),r(:),g(:),u(:),p(:),q(:),x(:),xold(:),y(:),&
          y0(:),z(:),t0(:),t1(:),t2(:),t(:)
allocate (tmp(n),r(n),g(n),u(n),p(n),q(n),x(n),xold(n),y(n),y0(n),z(n),&
          t0(n),t1(n),t2(n),t(n) )
     x = .0 ;  z = .0 ;  r = rhs ;
     ! x0 = 0 is the initial guess, y is the preconditioned solution.
     call lsolve(n,da1,icsc,jcsc,csca,r,g); ! g -preconditioned residual
     p = da*g ; y0 = p ; tmp = d - 2.0*da ;
     !----- matrix-vector product q=A*p with Eisenstat trick ------
     call usolve(n, da1,icsc,jcsc,csca,p,t1);
     t2 = tmp*t1+p ;
     call lsolve(n, da1,icsc,jcsc,csca,t2,q);
     q =  t1 + q;
     !-------------------------------------------------------------
     t0 = q ;  ic=0 ; nrmb = sqrt(dot_product(r, r))*tol ;
mr_iter: do i=1, maxit
   u = da * q ;
   alpha = dot_product(y0,q)/dot_product(q,u);
   z = z + alpha * p ;
   g = g - alpha * q ;
   y = y0 - alpha * u ;
   !----- matrix-vector product t=A*y with Eisenstat trick -----
     call usolve(n, da1,icsc,jcsc,csca,y, t1);
     t2 = tmp*t1+y ;
     call lsolve(n, da1,icsc,jcsc,csca,t2, t);
     t = t + t1;
   !------------------------------------------------------------
   beta = dot_product(y, t)/dot_product(y0, t0) ;
   p = y + beta*p ;
   q = t + beta*q ;
   select case (icc)
       case (1)
          call cgccri(n,i,icsc,jcsc,csca,da,da1,z,iinc,tol, xold, &
                    rhs,ic,iters,relres)
       case (2)
          call cgccrb(n,i,icsc,jcsc,csca,da,da1,g,z,iinc,tol,nrmb,&
                    rhs,ic,iters,nrmr,relres)
   end select
   if(ic==1) return ;          ! MINRES converged
   y0 = y ; t0 = t ;
end do mr_iter
    write(11,*)'********************************************** '
    write(11,*) 'MINRES does not converge to user-defined tolerance. '
        call usolve(n, da1,icsc,jcsc,csca,z, x)
        if(icc==1) relres = nrmr*tol/nrmb ;
        iters = maxit ;  rhs = x ;
     return
end subroutine mrmssor
!-----------------------------------------------------------------------
subroutine psolver(n,icsc,jcsc,csca,d,da,da1,b,maxit,tol,isolver,icho, &
                   ipre,icc,iinc,iters,relres)
   ! Choose preconditioned iterative methods:
   !             n: i.e. neq - number of total DOFs or equations;
   ! icsc,jcsc,csca: CSC storage of upper triangular part of matrix A;
   !             d: true diagonal of A;
   !            da: modified diagonal basing on GJ;
   !           da1: inverse of da;
   !             b: right hand side vector;
   !         maxit: user-defined maximal iteration number;
   !           tol: user-defined stopping tolerance;
   !       isolver: Iterative solver selection;
   !               = 1, SQMR iterative solver;
   !               = 2, PCG iterative solver;
   !               = 3, MINRES iterative solver;
   !          icho: choose standard or modified preconditioenr.
   !             =1: standard preconditioner.
   !             =2: generalized or modified preconditioner.
   !         ipre: choose preconditioned iterative solver;
   !               = 1, GJ preconditioned iterative method;
   !               = 2, MSSOR preconditioned iterative method;
   !           icc: choose convergence criterion;
   !               = 1, relative improvement norm criterion;
   !               = 2, relative residual norm criterion;
   !          iinc: check convergence every "iinc" step when ipre = 2;
   !         iters: returned iteration number when converged;
   !        relres: returned relative residual when converged;
   implicit none
   integer::n,icsc(:),jcsc(:),maxit,isolver,icho,ipre,icc,iinc,iters
   real(8)::csca(:),d(:),da(:),da1(:),b(1:),tol, relres
   !real(8),allocatable::d1(:)
   !allocate(d1(n))
   !d1(1:n) = 1./d(1:n) ;
   if(isolver==1)then          ! (SQMR Iterative Solver)
       select case (ipre)
        case(1)
         if(icho==1) write(11,*) '  ---> SJ preconditioned SQMR solver'
         if(icho==2) write(11,*) '  ---> GJ preconditioned SQMR solver'
           select case (icc)
             case (1)
               write(11,*) '   with relative improvement norm criterion!'
             case (2)
               write(11,*) '   with relative residual norm criterion!'
           end select
           !
           call psqmr(n,icsc,jcsc,csca,da1,b,maxit,tol,icc,iters,relres)
        case(2)
         if(icho==1)write(11,*) ' ---> SSOR preconditioned SQMR solver'
         if(icho==2)write(11,*) ' ---> MSSOR preconditioned SQMR solver'
           select case (icc)
             case (1)
               write(11,*) '   with relative improvement norm criterion!'
             case (2)
               write(11,*) '   with relative residual norm criterion!'
           end select
           call sqmrmssor(n,icsc,jcsc,csca,d,da,da1,b,maxit,tol,icc, &
                          iinc,iters,relres)
        case default
          write(*,*) ' No preconditioned solver is chosen, stop here! '
          Stop
       end select
    else if(isolver==2)then  ! (PCG Iterative Solver)
       select case (ipre)
        case(1)
          if(icho==1)write(11,*) ' ---> SJ preconditioned PCG solver'
          if(icho==2)write(11,*) ' ---> GJ preconditioned PCG solver'
           select case (icc)
             case (1)
               write(11,*) '   with relative improvement norm criterion!'
             case (2)
               write(11,*) '   with relative residual norm criterion!'
           end select
           !
           call pcg(n,icsc,jcsc,csca,da1,b,maxit,tol,icc,iters,relres)
        case(2)
          if(icho==1)write(11,*) ' ---> SSOR preconditioned PCG solver'
          if(icho==2)write(11,*) ' ---> MSSOR preconditioned PCG solver'
           select case (icc)
             case (1)
               write(11,*) '   with relative improvement norm criterion!'
             case (2)
               write(11,*) '   with relative residual norm criterion!'
           end select
           call pcgmssor(n,icsc,jcsc,csca,d,da,da1,b,maxit,tol,icc, &
                          iinc,iters,relres)
        case default
          write(*,*) ' No preconditioned solver is chosen, stop here! '
          Stop
       end select
    else if(isolver==3)then   ! isolver =3  ! (MINRES Iterative Solver)
       select case (ipre)
        case(1)
           if(icho==1)write(11,*) ' ---> SJ preconditioned MINRES solver'
           if(icho==2)write(11,*) ' ---> GJ preconditioned MINRES solver'
           select case (icc)
             case (1)
               write(11,*) '   with relative improvement norm criterion!'
             case (2)
               write(11,*) '   with relative residual norm criterion!'
           end select
           !
          call minres(n,icsc,jcsc,csca,da1,b,maxit,tol,icc,iters,relres)
        case(2)
         if(icho==1)write(11,*) ' ---> SSOR preconditioned MINRES solver'
         if(icho==2)write(11,*) ' --->MSSOR preconditioned MINRES solver'
           select case (icc)
             case (1)
```

```
               write(11,*) '   with relative improvement norm criterion!'
             case (2)
               write(11,*) '   with relative residual norm criterion!'
           end select
           call mrmssor(n,icsc,jcsc,csca,d,da,da1,b,maxit,tol,icc, &
                          iinc,iters,relres)
        case default
          write(*,*) '  No preconditioned solver is chosen, stop here! '
          Stop
       end select
       !
    else
       write(11,*) '  No Iterative Solver is selected, and STOP! ' ;
       stop
    end if
    !
    return
  end subroutine psolver
!-----------------------------------------------------------------------

end module sparse_lib
```

C.4 How to Use sparse_lib in FEM Package

C.4.1 Introduction

Sparse_lib library is built compatibly with the Fortran 90 programs given in the book "Programming the Finite Element Method" authored by Smith and Griffiths (1998). The library is designed to replace direct solution method or element-by-element based iterative method. It can be used for a general FEM package, but for special problems, some basic information should be provided. Many iterative solvers and preconditioners can be used for fast solutions of large-scale linear systems arising from finite element discretization, but, currently, only symmetric PCG, SQMR and MINRES iterative solvers and standard (or generalized) Jacobi and standard (or modified) SSOR preconditioners are added in. Further extension of this library with nonsymmetric iterative solvers and other preconditioning techniques is not a difficult task.

C.4.2 Three Basic Components in sparse_lib

The sparse_lib library is mainly composed of the following three components:

- **Sorting Subroutines**:

  The sorting subroutines contains quicksort and insertion sort routines, which can be obtained from ORDERPACK package. Sorting three vectors (arr, brr and crr) which stores element-level nonzero entries of global stiffness matrix and then assembling them by using the subroutine `sortadd` lead to the final CSC or CSR storage.

Subroutine `sortadd` functions as follows: sort `arr` (global row number) index in ascending order and at the same time, `brr`, `crr` are reordered correspondingly. For the same `arr` index, sub-sort `brr` (global column index), and at the same time, `crr` (corresponding nonzero entry value collected from elements) changes correspondingly with `brr`. After this work, add up all `crr` components with the same (`arr`, `brr`), and the zero-value `crr` entry will be removed. Finally forming the Compressed sparse Row (CSR) format or Compressed Sparse Column (CSC) format. The true nonzero number of the compressed sparse storage is also returned.

- **Sparse Operations Subroutines**:

  This part includes those subroutines for matrix-vector products and triangular solves, these subroutines are required by sparse preconditioned solvers. For symmetric matrix, only CSR or CSC storage of upper triangular part of this matrix is stored, and thus, the corresponding matrix-vector product is implemented with the symmetric storage. This part can be extended by including the operations with other sparse storages.

- **Sparse Preconditioned Iterative Solvers Subroutines**:

  In this library, only SQMR, PCG and MINRES iterative methods are included. To combine with any included iterative method, standard (generalized) Jacobi and standard (modified) SSOR preconditioners are proposed. It is worth mentioning that to switch generalized (or modified) preconditioner to standard one. This switch can be realized by choosing `icho`.

C.4.3 Parameters or Basic Information for `sparse_lib` Library

To use the library `sparse_lib`, some parameters or basic information should be provided. These parameters and basic information can be described as following.

C.4.3.1 Input Parameters

```
integer::  maxit,isolver,icho,ipre,icc,iinc
real(8)::  tol,coef,omega
```

`maxit`: user-defined maximum iteration number, when this number is reached, iteration is stopped no matter whether the convergence tolerance is satisfied.

`isolver`: choose a preferred iterative solver, `isolver = 1` means to select SQMR method, `isolver = 2` means to select PCG method and `isolver = 3` means to select MINRES method.

`icho`: choose a standard or modified diagonal. `icho = 1` means to select standard preconditioner, while `icho = 2` means to select modified (or generalized) preconditioner.

`ipre`: choose a preconditioner, `ipre = 1` for Jacobi preconditioner and `ipre = 2` for SSOR preconditioner. `ipre` should be used in conjunction with `icho` with the following combinations:

- `icho = 1` and `ipre = 1`, choose standard Jacobi preconditioner;
- `icho = 1` and `ipre = 2`, choose standard SSOR preconditioner;
- `icho = 2` and `ipre = 1`, choose Generalized Jacobi preconditioner;
- `icho = 2` and `ipre = 2`, choose Modified SSOR preconditioner.

`icc`: choose convergence criterion, `icc = 1` for relative 'improvement' norm criterion. `icc = 2` for relative residual norm criterion.

`iinc`: check convergence every "`iinc`" step when `ipre = 2` because the true residual can not be monitored directly. This parameter can be selected in the range `iinc`$\in$ [4, 8] if users have no experiences on this selection.

`tol`: user-defined convergence (stopping) tolerance for a selected convergence criterion. This parameter can be selected to be `tol =` 10^{-6} if users have no experiences on this selection.

`coef`: This parameter is used for the modified diagonal in GJ or MSSOR preconditioner. When using a GJ or MSSOR preconditioned iterative method, this parameter can also be made positive or negative in terms of the selected iterative method. This parameter can be selected to be `coef = -4.0` if users have no experiences on this selection, but for MINRES iterative method, a positive `coef = 4.0` can be used.

`omega`: relaxation parameter for MSSOR preconditioning methods. `omega` can be selected in the range [1.0, 1.4] if users have no experiences on this choice.

C.4.3.2 Parameters or Basic Information to Be Obtained

```
integer::neq,sneq,fneq,id(neq),ebeanz,iebe,jebe,ebea,uanz,iters
real(8)::resi
```

`neq,sneq(sn),fneq(fn)`: `neq` is total number of DOFs in the mesh (or number of equations), `sneq` is the number of DOFs corresponding to displacement and `fneq` is the number of DOFs corresponding to the pore water pressure. There exists the relation, `neq = sneq + fneq`. The computation to get `sneq` or `fneq` is not necessary if no pore water pressure is involved.

`id(neq)`: `id` is an identifier array for displacement or pore pressure DOF;

`id(i) = 1` for displacement DOF (`i=1, neq`);

`id(i) = 0` for pore pressure DOF (`i=1, neq`).

`ebeanz`: an estimated (or returned true) number for total nonzero entries collected from all element stiffness matrices; the formula for the estimated `ebeanz` is `ebeanz = ntot*int(ntot/2)*nels`, `ntot` is the dimension of element stiffness matrices as mentioned above. When no pore pressure unknowns are involved, `ntot = ndof` should be set.

`iebe,jebe,ebea`: global row index, global column index and correspondent value, respectively, collected from element matrices.

`uanz`: this value is used to estimate storage for CSC (or CSR) format of upper triangular part of coefficient matrix A. We can use half bandwidth "`nband`" to give a conservative estimation of `uanz`, i.e., `uanz = int(neq - nband/2)*nband`. Alternatively, the estimated formula is `uanz = int(0.6*ebeanz)` for large 3-D problems, here `ebeanz` is the returned true number of nonzero entries collected from element "stiffness" matrices. 0.6 is an estimated ratio. In practical implementation, the ratio can be determined based on a small-size or middle-size problem before running a very large problem because the ratio usually decreases with problem size.

`jcsc,icsc,csca`: subroutine `sortadd` sorts element based storage, `iebe,jebe,ebea`, and leads to compressed sparse column storage `jcsc,icsc,csca`.

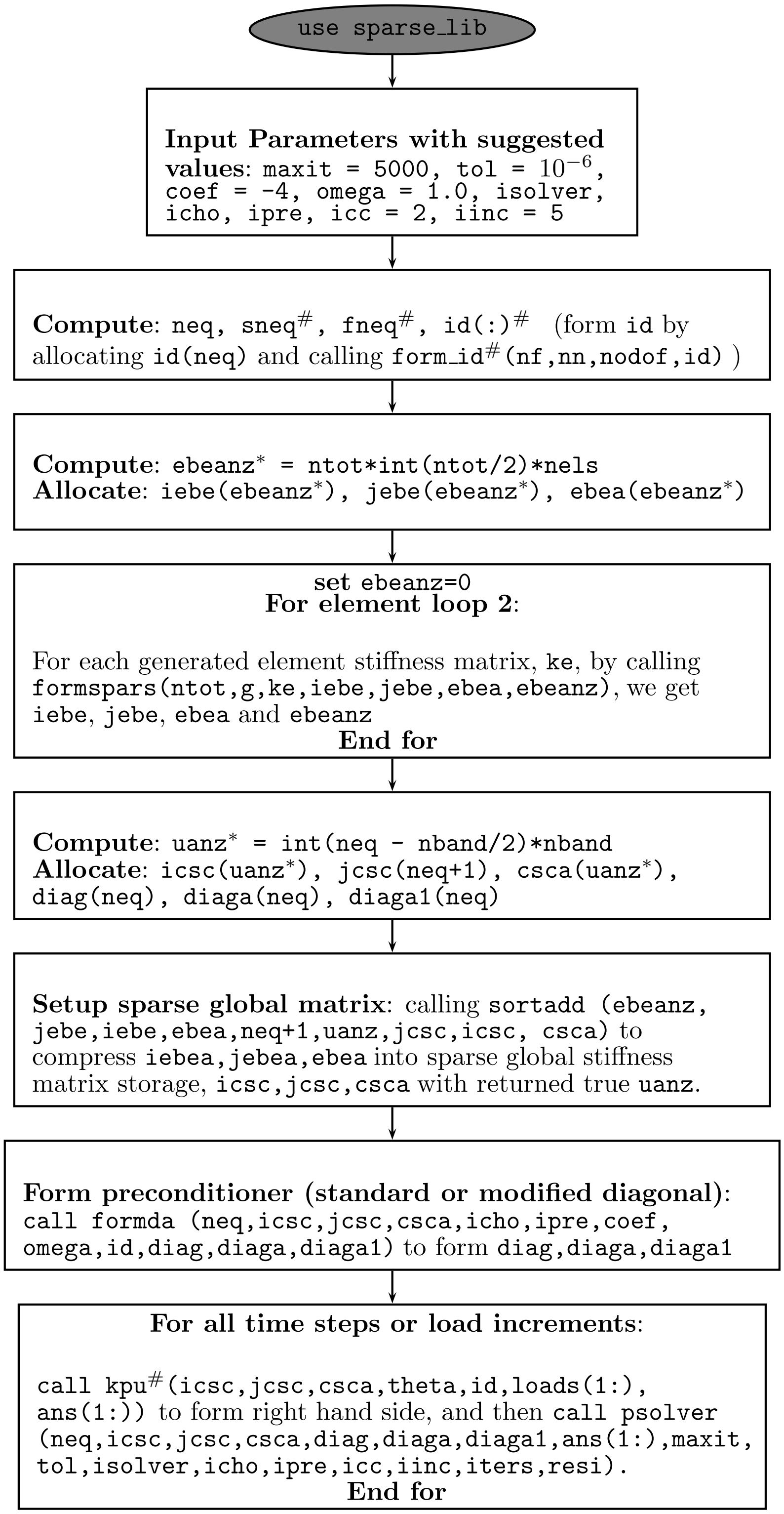
`diag,diaga,diaga1`: For GJ or MSSOR preconditioner, `diag` stores the true diagonal of coefficient matrix A, `diaga` returns modified diagonal constructed by GJ algorithm and `diaga` stores the inverse of diaga. But for standard Jacobi or SSOR preconditioner, `diaga = diag` stores the diagonal of A and `diaga1 = 1./diaga`.

`iters`: returned iteration number. When iterative solver converges, it stores the iterative number corresponding to the convergence tolerance, and it is equal to `maxit` when the iterative solver doesn't converge.

`resi`: returned residual, it has different meaning for different convergence criteria.

C.4.4 Flowchart of Using `sparse_lib` Library

The flowchart of calling `sparse_lib` subroutines is given as following.



use sparse_lib

Input Parameters with suggested values: maxit = 5000, tol = 10^{-6}, coef = -4, omega = 1.0, isolver, icho, ipre, icc = 2, iinc = 5

Compute: neq, sneq$^{\#}$, fneq$^{\#}$, id(:)$^{\#}$ (form id by allocating id(neq) and calling form_id$^{\#}$(nf,nn,nodof,id))

Compute: ebeanz* = ntot*int(ntot/2)*nels
Allocate: iebe(ebeanz*), jebe(ebeanz*), ebea(ebeanz*)

set ebeanz=0
For element loop 2:

For each generated element stiffness matrix, ke, by calling formspars(ntot,g,ke,iebe,jebe,ebea,ebeanz), we get iebe, jebe, ebea and ebeanz
End for

Compute: uanz* = int(neq - nband/2)*nband
Allocate: icsc(uanz*), jcsc(neq+1), csca(uanz*), diag(neq), diaga(neq), diaga1(neq)

Setup sparse global matrix: calling sortadd (ebeanz, jebe,iebe,ebea,neq+1,uanz,jcsc,icsc, csca) to compress iebea,jebea,ebea into sparse global stiffness matrix storage, icsc,jcsc,csca with returned true uanz.

Form preconditioner (standard or modified diagonal): call formda (neq,icsc,jcsc,csca,icho,ipre,coef, omega,id,diag,diaga,diaga1) to form diag,diaga,diaga1

For all time steps or load increments:

call kpu$^{\#}$(icsc,jcsc,csca,theta,id,loads(1:), ans(1:)) to form right hand side, and then call psolver (neq,icsc,jcsc,csca,diag,diaga,diaga1,ans(1:),maxit, tol,isolver,icho,ipre,icc,iinc,iters,resi).
End for

In the above flowchart, the number with star symbol (*) means it is an estimated number, while the numbers or subroutines with # symbol is specially for coupled problems such as consolidation problem. Therefore, when solving the drained problems, the computing or calling of the numbers and subroutines with # symbol may not be necessary. Moreover, it should be noted that `icho = 1` must be set for these cases.

C.4.5 Demonstration of Using `sparse_lib` in Program `p92`

To demonstrate the application of module `sparse_lib`, the input file, the main program and output file can be changed or generated correspondingly.

C.4.5.1 Input File `p92.dat`

In the following input file '`p92.dat`' of the test example, the new parameters required by the sparse preconditioned iterative solvers are provided in the underline part.

```
nels nxe nye nn nip ⟵ 4 1 4 23 4
permx permy e v ⟵ 1.0 1.0 1.0 .0
dtim nstep theta maxit tol coef omega isolver icho ipre icc iinc

1.0 20 1.0 1000 1.e-6 -4.0 1.0 1 2 2 2 5
width(i), i = 1, nxe + 1  ⟵ 0.0 1.0
depth(i), i = 1, nye + 1  ⟵ 0.0 -2.5 -5.0 -7.5 -10.0
nr  ⟵ 23
k, nf(:,k), k=1, nr  ⟵
1 0 1 0 2 1 1 0 3 0 1 0 4 0 1 0 5 0 1 0
6 0 1 1 7 1 1 0 8 0 1 1 9 0 1 0 10 0 1 0
11 0 1 1 12 1 1 0 13 0 1 1 14 0 1 0 15 0 1 0
16 0 1 1 17 1 1 0 18 0 1 1 19 0 1 0 20 0 1 0
21 0 0 1 22 0 0 0 23 0 0 1
```

C.4.5.2 Main Program of `P92` with Direct Solver Replaced By Sparse Preconditioned Iterative Method

```
program p92
!-----------------------------------------------------------------------
!      program 9.2 plane strain consolidation of a Biot elastic
!      solid using 8-node solid quadrilateral elements
!      coupled to 4-node fluid elements : incremental version
!      Linear systems are solved by sparse iterative methods
!-----------------------------------------------------------------------
 use new_library; use geometry_lib; use sparse_lib; implicit none !<--(1)
 integer::nels,nxe,nye,neq,nband,nn,nr,nip,nodof=3,nod=8,nodf=4,nst=3,&
        ndim=2,ndof,i,k,l,iel,ns,nstep,ntot,nodofs=2,inc,             &
        sn,fn,ebeanz,uanz,isolver,icho,ipre,icc,iinc,iters,maxit  !<--(2)
 real(8):: permx,permy,e,v,det,dtim,theta,time,                        &
```

```
           coef,omega,resi,tol                                   !<--(3)
 character(len=15):: element = 'quadrilateral'
!---------------------------- dynamic arrays---------------------------
 real(8),allocatable ::dee(:,:), points(:,:), coord(:,:), derivf(:,:),&
                    jac(:,:),kay(:,:),der(:,:),deriv(:,:),weights(:),&
                    derf(:,:),funf(:),coordf(:,:), bee(:,:), km(:,:),&
                    eld(:), sigma(:), kp(:,:), ke(:,:), g_coord(:,:),&
                    fun(:),c(:,:),width(:), depth(:),vol(:),loads(:),&
                    ans(:) ,volf(:,:),                               &
                    ! bk(:),store_kp(:,:,:),phi0(:),phi1(:)       !-->(4)
                    ebea(:),csca(:),diag(:),diaga(:),diaga1(:)    !<--(5)
 integer, allocatable :: nf(:,:),g(:),num(:),g_num(:,:), g_g(:,:),   &
                        id(:),iebe(:),jebe(:),icsc(:),jcsc(:)     !<--(6)
!--------------------------input and initialisation--------------------
  open (10,file='p92.dat',status='old',action='read')
  open (11,file='p92.res',status='replace',action='write')
  read (10,*) nels,nxe,nye,nn,nip,                                   &
             permx, permy, e,v, dtim, nstep, theta ,                 &
             maxit,tol,coef,omega,isolver,icho,ipre,icc,iinc    !<--(7)
  ndof=nod*2; ntot=ndof+nodf
  allocate ( dee(nst,nst), points(nip,ndim), coord(nod,ndim),        &
     derivf(ndim,nodf),jac(ndim,ndim),kay(ndim,ndim),der(ndim,nod),  &
     deriv(ndim,nod), derf(ndim,nodf),funf(nodf),coordf(nodf,ndim),  &
     bee(nst,ndof), km(ndof,ndof),eld(ndof),sigma(nst),kp(nodf,nodf), &
     g_g(ntot,nels),ke(ntot,ntot),fun(nod),c(ndof,nodf),width(nxe+1), &
     depth(nye+1),vol(ndof),nf(nodof,nn), g(ntot), volf(ndof,nodf),  &
     g_coord(ndim,nn),g_num(nod,nels),num(nod),weights(nip) )
     ! store_kp(nodf,nodf,nels),phi0(nodf),phi1(nodf) )          !-->(8)
     kay=0.0; kay(1,1)=permx; kay(2,2)=permy
     read (10,*)width , depth
  nf=1; read(10,*) nr ; if(nr>0) read(10,*)(k,nf(:,k),i=1,nr)
  call snfn(nf,nn,nodof,ndim,nels,ntot,sn,fn,ebeanz)             !<--(9)
  call formnf(nf); neq=maxval(nf)
  allocate(id(neq));                                             !<--(10)
  call form_id(nf,nn,nodof,id);                                  !<--(11)
  call deemat(dee,e,v); call sample(element,points,weights)
!--------- loop the elements to find nband and set up global arrays-----
  nband = 0
 elements_1: do iel = 1 , nels
        call geometry_8qxv(iel,nxe,width,depth,coord,num)
        inc=0
        do i=1,8;do k=1,2;inc=inc+1;g(inc)=nf(k,num(i));end do;end do
        do i=1,7,2;inc=inc+1;g(inc)=nf(3,num(i)); end do
        g_num(:,iel)=num;g_coord(:,num)=transpose(coord);g_g(:,iel)= g
        if(nband<bandwidth(g))nband=bandwidth(g)
 end do elements_1
  write(11,'(a)') "Global coordinates "
  do k=1,nn;write(11,'(a,i5,a,2e12.4)')"Node",k,"   ",g_coord(:,k);end do
  write(11,'(a)') "Global node numbers "
  do k = 1 , nels; write(11,'(a,i5,a,8i5)')                          &
                             "Element ",k,"        ",g_num(:,k); end do
  write(11,'(2(a,i5))')                                              &
     "There are  ",neq, "  equations and the half-bandwidth is   ",nband
  allocate(loads(0:neq),ans(0:neq),iebe(ebeanz),jebe(ebeanz),ebea(ebeanz))
                                                                !<-->(12)
             loads = .0 ; ebeanz = 0 ;                          !<-- (13)
```

```
!------------- element stiffness integration and assembly---------------
 elements_2:  do iel = 1 , nels
       num = g_num( : ,iel ); coord= transpose(g_coord(:,num))
       g = g_g ( : , iel )  ; coordf = coord(1 : 7 : 2, : )
       km = .0; c = .0; kp = .0
    gauss_points_1: do i = 1 , nip
       call shape_der(der,points,i);  jac = matmul(der,coord)
       det = determinant(jac); call invert(jac);deriv = matmul(jac,der)
       call beemat(bee,deriv); vol(:)=bee(1,:)+bee(2,:)
       km = km + matmul(matmul(transpose(bee),dee),bee) *det* weights(i)
!-----------------------now the fluid contribution----------------------
       call shape_fun(funf,points,i)
       call shape_der(derf,points,i)  ; derivf=matmul(jac,derf)
  kp=kp+matmul(matmul(transpose(derivf),kay),derivf)*det*weights(i)*dtim
       do l=1,nodf; volf(:,l)=vol(:)*funf(l); end do
       c= c+volf*det*weights(i)
    end do gauss_points_1
       !  store_kp( : , : , iel) = kp                          ! -->(14)
       call formke(km,kp,c,ke,theta);
       !---collect nonzero entries from element stiffness matrices---
       call formspars(ntot,g,ke,iebe,jebe,ebea,ebeanz)         !<-->(15)
                                               !@@ call formkv(bk,ke,g,neq)
 end do elements_2
!-----------------------factorise left hand side-----------------------
 uanz = int(neq - nband/2)*nband                               !<-- (16)
 allocate(icsc(uanz), jcsc(neq+1), csca(uanz), diag(neq), diaga(neq), &
          diaga1(neq))
 call sortadd(ebeanz, jebe(1:ebeanz), iebe(1:ebeanz), ebea(1:ebeanz), &
              neq+1,uanz,jcsc,icsc,csca)
    deallocate(iebe,jebe,ebea)
 !-form standard or modified diagonal for the chosen preconditioner-
 call formda(neq,icsc,jcsc,csca,icho,ipre,coef,omega,id,diag,diaga,diaga1)
! --------- enter the time-stepping loop--------------------------------
    time = .0
  time_steps:  do ns = 1 , nstep
     time = time +dtim    ;
     write(11,'(a,e12.4)') "The time is  ",time
     ans=.0; call kpu(icsc,jcsc,csca,theta,id,loads(1:),ans(1:));ans(0)=.0
                                                                !<-->(17)
!   ramp loading
     if(ns<=10) then
        ans(1)=ans(1)-.1/6.; ans(3)=ans(3)-.2/3.
        ans(4)=ans(4)-.1/6.
     end if
     call psolver(neq,icsc,jcsc,csca,diag,diaga,diaga1,ans(1:),maxit, &
               tol,isolver,icho,ipre,icc,iinc,iters,resi) ; ans(0)=.0 ;
     write(11,'(a,i5,a,e12.4)')" Iterative solver took", iters,"  iterations   &
                                   to converge to ",resi         !<-->(18)
     loads = loads + ans
     write(11,'(a)') " The nodal displacements and porepressures are    :"
   do k=1,23,22; write(11,'(i5,a,3e12.4)')k,"    ",loads(nf(:,k)) ; end do
!-------------------recover stresses at  Gauss-points-------------------
   elements_4 :  do iel = 1 , nels
        num = g_num(: , iel ); coord=transpose(g_coord(:,num))
        g = g_g( : , iel )   ;  eld = loads( g ( 1 : ndof ) )
!       print*,"The Gauss Point effective stresses for element",iel,"are"
     gauss_pts_2: do i = 1,nip
        call shape_der (der,points,i);  jac= matmul(der,coord)
        call invert ( jac );    deriv= matmul(jac,der)
        bee= 0.;call beemat(bee,deriv);sigma= matmul(dee,matmul(bee,eld))
!       print*,"Point    ",i       ;!  print*,sigma
     end do gauss_pts_2
   end do elements_4
  end do time_steps
 end program p92
```

The numbered arrow shown in the above program p92 can be explained as follows:

real(8): Replace real by real(8) for the main program and all subroutines.

<--(1): Include the module sparse_lib into the main program p92.

<--(2): Include the new integer parameters, sn,fn,ebeanz,uanz,isolver,icho,ipre, icc,iinc,iters,maxit.

<--(3): Include the new integer parameters, coef,omega,resi,tol .

-->(4): Delete the storages, bk(:),store_kp(:,:,:),phi0(:),phi1(:) .

<--(5): Include the storages, ebea(:),csca(:),diag(:),diaga(:),diaga1(:) .

<--(6): Include the storages, id(:),iebe(:),jebe(:),icsc(:),jcsc(:).

<--(7): Include the new input parameters, maxit,tol,coef,omega,isolver,icho, ipre,icc,iinc.

-->(8): Delete the storages, store_kp(nodf,nodf,nels),phi0(nodf),phi1(nodf).

<--(9): Call subroutine, snfn(nf,nn,nodof,ndim,nels,ntot,sn,fn,ebeanz), which is given as

```
subroutine snfn(nf,nn,nodof,ndim,nels,ntot,sn,fn,ebenz)
 ! This subroutine computes the displacement DOFs (sn)
 !     and pore pressure DOFs (fn).
 !     Then it gives an estimation (ebenz).
   implicit none
       integer:: i,nf(:,:),nn,nodof,ndim,nels,sn,fn,ntot,ebenz
       sn=0 ; fn = 0;
       fn = sum(nf(nodof,:)) ;
       do i=1,ndim ; sn = sn + sum(nf(i,:)) ; end do
       ebeanz = ntot*int(ntot/2)*nels ;
        return
 end subroutine snfn
```

<--(10): Include the storage, allocate(id(neq)) , for the identifier array.

<--(11): To form id, call the subroutine, form_id(nf,nn,nodof,id) which is given as

```
subroutine form_id(nf,nn,nodof,id)
 ! This subroutine for the identifier array, "id".
 !     nf(:,:) is the original input node freedom array.
 !     nn - total node number.
 !     nodof - number of freedoms per node.
   implicit none
       integer:: i,nf(:,:),nn,nodof,id(:)
        id(:) = 1;
        do i = 1,nn ;
           if(nf(nodof,i)/=0)id(nf(nodof,i))=0;
        end do;
        !
        return
 end subroutine form_id
```

`<-->(12):` Delete `bk(neq*(nband+1))`, and allocating `loads(0:neq),ans(0:neq),` `iebe(ebeanz),jebe(ebeanz),ebea(ebeanz)`.

`<-- (13):` Set `ebeanz = 0` .

`--> (14):` Delete `store_kp( :, :, iel) = kp` .

`<-->(15):` Replace `call formkv(bk,ke,g,neq)` by `call formspars(ntot,g,ke,iebe,` `jebe,ebea,ebeanz)`.

`<-- (16):` Give an estimation in terms of `uanz = int(neq - nband/2)*nband`, which may be a conservative estimation. Call subroutine `sortadd`, and then call subroutine `formda`.

`<-->(17):` Compute right hand side by `ans=.0; call kpu(icsc,jcsc,csca,theta,id,` `loads(1:),ans(1:)); ans(0)=.0` in terms of sparse storage instead of the element-by-element implementation.

`<-->(18):` Replace the direct solver with sparse preconditioned iterative method by calling subroutine `psolver`.

C.4.5.3 Output File `p92.res`

Numerical results for the simple consolidation problem are generated by three different solution methods: original direct solution method, GJ preconditioned SQMR method and MSSOR preconditioned SQMR method, respectively. The input has been given in section C.4.5.1, but for direct solution method, the underline part is not necessary. For standard Jacobi preconditioned SQMR method, it is to set ‘`isolver =1, icho = 1,` ipre = 1', while for standard SSOR preconditioned SQMR method, we set isolver = 1, icho = 1, ipre = 2. For GJ preconditioned SQMR method, it is just required to set 'isolver = 1, icho = 2, ipre = 1' and for MSSOR preconditioned SQMR method, it is to set 'isolver = 1, icho = 2, ipre = 2'. It is noteworthy that for such a small problem, MSSOR preconditioned SQMR may need more iterations than GJ preconditioned counterpart to converge to a preset tolerance because convergence is checked every fifth iteration for MSSOR preconditioned SQMR method. But for large problems, MSSOR is expected to more effective. On the other hand, standard Jacobi and SSOR preconditioners are still very effective in this example, but their performance can be seriously influenced when highly varied soil properties are involved.

(a) *Output file generated by direct solution method*

```
 Global coordinates
Node    1         0.0000E+00  0.0000E+00
Node    2         0.5000E+00  0.0000E+00
Node    3         0.1000E+01  0.0000E+00
Node    4         0.0000E+00 -0.1250E+01
Node    5         0.1000E+01 -0.1250E+01
Node    6         0.0000E+00 -0.2500E+01
Node    7         0.5000E+00 -0.2500E+01
Node    8         0.1000E+01 -0.2500E+01
Node    9         0.0000E+00 -0.3750E+01
Node   10         0.1000E+01 -0.3750E+01
Node   11         0.0000E+00 -0.5000E+01
Node   12         0.5000E+00 -0.5000E+01
Node   13         0.1000E+01 -0.5000E+01
Node   14         0.0000E+00 -0.6250E+01
Node   15         0.1000E+01 -0.6250E+01
Node   16         0.0000E+00 -0.7500E+01
Node   17         0.5000E+00 -0.7500E+01
Node   18         0.1000E+01 -0.7500E+01
Node   19         0.0000E+00 -0.8750E+01
Node   20         0.1000E+01 -0.8750E+01
Node   21         0.0000E+00 -0.1000E+02
Node   22         0.5000E+00 -0.1000E+02
Node   23         0.1000E+01 -0.1000E+02
Global node numbers
Element     1            6    4    1    2    3    5    8    7
Element     2           11    9    6    7    8   10   13   12
Element     3           16   14   11   12   13   15   18   17
Element     4           21   19   16   17   18   20   23   22
There are     32  equations and the half-bandwidth is      13
The time is    0.1000E+01
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.1233E+00  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.1000E+00
The time is    0.2000E+01
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.2872E+00  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.2000E+00
The time is    0.3000E+01
      . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
The time is    0.1900E+02
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.4277E+01  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.8811E+00
```

```
The time is    0.2000E+02
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.4424E+01  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.8636E+00
```

(b) *Output file generated by SJ preconditioned SQMR method* (`isolver=1, icho = 1, ipre = 1, icc = 2`)

```
 There are     32  equations and the half-bandwidth is      13
The time is    0.1000E+01
   ---> SJ preconditioned SQMR solver
    with relative residual norm criterion!
 **********************************************
 Psolver converges to user-defined tolerance.
 SQMR took   18  iterations to   converge to   0.1718E-06
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.1233E+00  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.1000E+00
The time is    0.2000E+01
   ---> SJ preconditioned SQMR solver
    with relative residual norm criterion!
 **********************************************
 Psolver converges to user-defined tolerance.
 SQMR took   20  iterations to   converge to   0.5179E-07
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.2872E+00  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.2000E+00
The time is    0.3000E+01
      . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
The time is    0.1900E+02
   ---> SJ preconditioned SQMR solver
    with relative residual norm criterion!
 **********************************************
 Psolver converges to user-defined tolerance.
 SQMR took   24  iterations to   converge to   0.4783E-08
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.4277E+01  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.8811E+00
The time is    0.2000E+02
   ---> SJ preconditioned SQMR solver
    with relative residual norm criterion!
 **********************************************
 Psolver converges to user-defined tolerance.
 SQMR took   21  iterations to   converge to   0.3795E-06
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.4424E+01  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.8636E+00
```

(c) *Output file generated by SSOR preconditioned SQMR method* (`isolver=1, icho = 1, ipre = 2, icc = 2, iinc = 5`)

```
There are     32  equations and the half-bandwidth is      13
The time is    0.1000E+01
  ---> SSOR preconditioned SQMR solver
    with relative residual norm criterion!
 **********************************************
 SQMR converges to user-defined tolerance.
 SQMR took   20  iterations to   converge to   0.4189E-08
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.1233E+00  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.1000E+00
The time is    0.2000E+01
```

```
  ---> SSOR preconditioned SQMR solver
    with relative residual norm criterion!
 **********************************************
 SQMR converges to user-defined tolerance.
 SQMR took   20  iterations to   converge to   0.1259E-08
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.2872E+00  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.2000E+00
The time is    0.3000E+01
      . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
The time is    0.1900E+02
  ---> SSOR preconditioned SQMR solver
    with relative residual norm criterion!
 **********************************************
 SQMR converges to user-defined tolerance.
 SQMR took   20  iterations to   converge to   0.8254E-09
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.4277E+01  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.8811E+00
The time is    0.2000E+02
  ---> SSOR preconditioned SQMR solver
    with relative residual norm criterion!
 **********************************************
 SQMR converges to user-defined tolerance.
 SQMR took   20  iterations to   converge to   0.4956E-09
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.4424E+01  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.8636E+00
```

(d) *Output file generated by GJ preconditioned SQMR method* (isolver=1, icho = 2, ipre = 1, icc = 2)

```
There are     32  equations and the half-bandwidth is      13
The time is    0.1000E+01
   ---> GJ preconditioned SQMR solver
   with relative residual norm criterion!
 **********************************************
 SQMR converges to user-defined tolerance.
 SQMR took   18  iterations to   converge to   0.2636E-07
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.1233E+00  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.1000E+00
The time is    0.2000E+01
   ---> GJ preconditioned SQMR solver
   with relative residual norm criterion!
 **********************************************
 SQMR converges to user-defined tolerance.
 SQMR took   19  iterations to   converge to   0.9199E-08
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.2872E+00  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.2000E+00
The time is    0.3000E+01
      . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
The time is    0.1900E+02
   ---> GJ preconditioned SQMR solver
   with relative residual norm criterion!
 **********************************************
 SQMR converges to user-defined tolerance.
 SQMR took   24  iterations to   converge to   0.1061E-06
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.4277E+01  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.8811E+00
The time is    0.2000E+02
   ---> GJ preconditioned SQMR solver
   with relative residual norm criterion!
 **********************************************
 SQMR converges to user-defined tolerance.
 SQMR took   23  iterations to   converge to   0.4996E-06
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.4424E+01  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.8636E+00
```

(e) *Output file generated by MSSOR preconditioned SQMR method* (`isolver=1, icho = 2, ipre = 2, icc = 2, iinc = 5`)

```
There are     32  equations and the half-bandwidth is      13
The time is    0.1000E+01
   ---> MSSOR preconditioned SQMR solver
   with relative residual norm criterion!
 **********************************************
 SQMR converges to user-defined tolerance.
 SQMR took   25  iterations to   converge to   0.3545E-07
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.1233E+00  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.1000E+00
The time is    0.2000E+01
   ---> MSSOR preconditioned SQMR solver
   with relative residual norm criterion!
 **********************************************
 SQMR converges to user-defined tolerance.
 SQMR took   25  iterations to   converge to   0.3329E-07
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.2872E+00  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.2000E+00
The time is    0.3000E+01
      . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
The time is    0.1900E+02
   ---> MSSOR preconditioned SQMR solver
   with relative residual norm criterion!
 **********************************************
 SQMR converges to user-defined tolerance.
 SQMR took   25  iterations to   converge to   0.6407E-09
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.4277E+01  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.8811E+00
The time is    0.2000E+02
   ---> MSSOR preconditioned SQMR solver
   with relative residual norm criterion!
 **********************************************
 SQMR converges to user-defined tolerance.
 SQMR took   25  iterations to   converge to   0.4624E-09
 The nodal displacements and porepressures are    :
    1      0.0000E+00 -0.4424E+01  0.0000E+00
   23      0.0000E+00  0.0000E+00 -0.8636E+00
```

C.4.6 An Improved Version of sparse_lib

In Section C.3, the three compressed sparse storage, icsc,jcsc,csca, is required apart from the temporary storage, iebe,jebe,ebea. In fact, the storage for icsc,jcsc,csca can be avoided by overwriting the temporary storage iebe,jebe,ebea with assembled compressed sparse storage. By using this strategy, the memory storage requirement can be kept almost as same as that required by EBE technique for symmetric linear system. It can be predicted that by using the same global matrix assembly strategy, for nonsymmetric linear system but with symmetric nonzero structure, the memory requirement is about 1.5 times of that required by EBE technique, and for nonsymmetric linear with nonsymmetric nonzero structure, the memory requirement is about 2.0 times of that required by EBE technique.

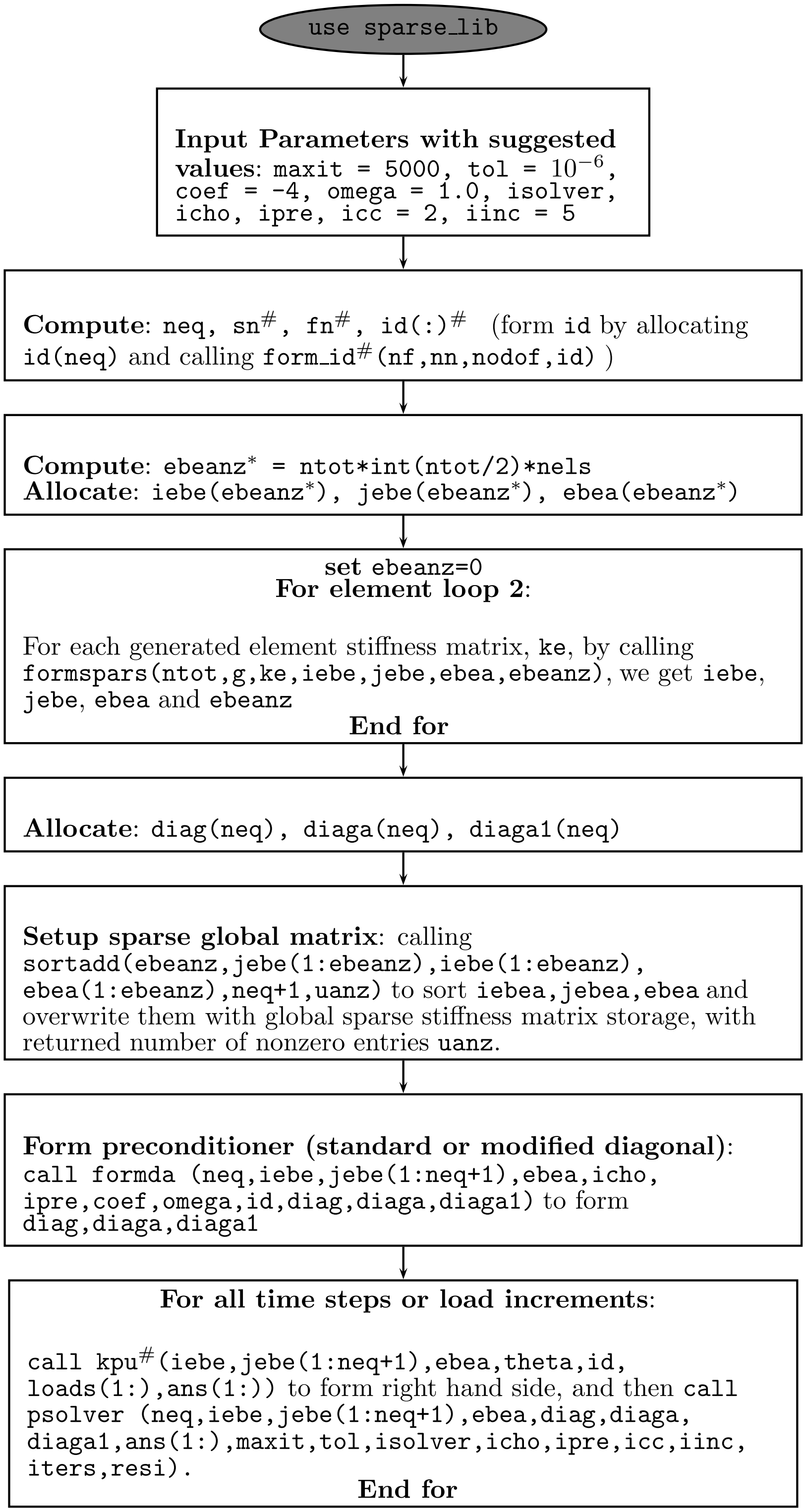
This improvement comes from a minor modification of subroutine sortadd in sparse_lib. This improved subroutine sortadd is given as follows:

```
subroutine sortadd(uanz,arr,brr,crr,ni,nnz)
  ! For the same arr index, subsort brr, and at the same time, crr
  ! changes correspondingly with brr. After this work, adding up all crr
  ! components with the same (arr, brr) or (brr, arr) index, and the
  ! zero-value crr entry will be removed. Finally forming the Compressed
  ! Sparse Row (CSR) format or Compressed Sparse Column (CSC) format
  ! to overwrite arr,brr,crr.
  !         uanz: the nonzero number of arr (or brr, crr).
  !  arr,brr,crr: three vectors required to be sorted.
  !           ni: = n + 1 (n is dimension of A)
  !          nnz: the nonzero number of crr.
    integer:: i,j,k,k1,k2,m,arr(:),brr(:),uanz,nnz,ni
    integer, allocatable:: itep(:)
    real(8):: crr(:), aa
    allocate (itep(ni))
    call quicksort(uanz,arr,brr,crr) ; ! sorting three vectors
    k=1;  itep(1)=1
    do i=2, uanz
       if(arr(i)/=arr(i-1)) then
        k=k+1 ;  itep(k)=i
       end if
    end do
    itep(k+1)=uanz+1
  !----------------------------
   do i=1, k
       k1=itep(i);  k2=itep(i+1)-1
       j=k2-k1+1
       if(j<=16) then    ! sub-brr sorting by Insertion sort if j <= 16.
         call subbrr2(brr(k1:k2),crr(k1:k2),j)
       else               ! quick sorting when j is larger (>16).
          call quicksort2(j,brr(k1:k2),crr(k1:k2))
       end if
   end do
  !----------------------------
   m = 0 ;
   do i=1, k
     k1=itep(i);  k2=itep(i+1)-1  ;  m=m+1;
     arr(i) = m ;  brr(m) = brr(k1) ; aa = .0
     do j=k1, k2-1
         aa = aa + crr(j) ;
       if(brr(j+1)/=brr(j) ) then
          if(aa /=.0) then
           crr(m) = aa
           m=m+1 ;
           brr(m)= brr(j+1)
           aa = .0
         else            !  aa is removed when it is zero.
           brr(m)= brr(j+1)
         end if
       end if
     end do
       crr(m) = aa + crr(k2)
       if(crr(m)==.0) m=m-1
   end do
   arr(k+1)=m+1;  nnz=m
   !
   return
end subroutine sortadd
```

Replacing the old subroutine with the above new one, the storage requirement for assembled (or compressed) sparse storage is removed, and the final CSC or CSR storage can overwrite this three vectors (`arr`, `brr` and `crr`). That is to say, before calling subroutine `sortadd`, the three vectors denote global row index, global column index and correspondent value, respectively, collected from element matrices. After calling subroutine `sortadd`, it is the compressed sparse storage in CSC format or CSR format.

Here, we give the revised flowchart and demonstration example when using the improved `sparse_lib`.

The flowchart of calling the improved `sparse_lib` is given as following.



`use sparse_lib`

Input Parameters with suggested values: `maxit = 5000, tol =` 10^{-6}`, coef = -4, omega = 1.0, isolver, icho, ipre, icc = 2, iinc = 5`

Compute: `neq, sn`$^{\#}$`, fn`$^{\#}$`, id(:)`$^{\#}$ (form `id` by allocating `id(neq)` and calling `form_id`$^{\#}$`(nf,nn,nodof,id)`)

Compute: `ebeanz`* `= ntot*int(ntot/2)*nels`
Allocate: `iebe(ebeanz`*`), jebe(ebeanz`*`), ebea(ebeanz`*`)`

set `ebeanz=0`
For element loop 2:

For each generated element stiffness matrix, `ke`, by calling `formspars(ntot,g,ke,iebe,jebe,ebea,ebeanz)`, we get `iebe`, `jebe`, `ebea` and `ebeanz`
End for

Allocate: `diag(neq), diaga(neq), diaga1(neq)`

Setup sparse global matrix: calling `sortadd(ebeanz,jebe(1:ebeanz),iebe(1:ebeanz), ebea(1:ebeanz),neq+1,uanz)` to sort `iebea,jebea,ebea` and overwrite them with global sparse stiffness matrix storage, with returned number of nonzero entries `uanz`.

Form preconditioner (standard or modified diagonal): `call formda (neq,iebe,jebe(1:neq+1),ebea,icho, ipre,coef,omega,id,diag,diaga,diaga1)` to form `diag,diaga,diaga1`

For all time steps or load increments:

`call kpu`$^{\#}$`(iebe,jebe(1:neq+1),ebea,theta,id, loads(1:),ans(1:))` to form right hand side, and then `call psolver (neq,iebe,jebe(1:neq+1),ebea,diag,diaga, diaga1,ans(1:),maxit,tol,isolver,icho,ipre,icc,iinc, iters,resi).`
End for

The demonstration example of using the improved `sparse_lib` is given as following.

```
program p92
!-----------------------------------------------------------------------
!      program 9.2 plane strain consolidation of a Biot elastic !
solid using 8-node solid quadrilateral elements !      coupled to
4-node fluid elements : incremental version !      Linear systems
are solved by sparse iterative methods
!-----------------------------------------------------------------------
 use new_library; use geometry_lib; use sparse_lib; implicit none !<--(1)
 integer::nels,nxe,nye,neq,nband,nn,nr,nip,nodof=3,nod=8,nodf=4,nst=3,&
        ndim=2,ndof,i,k,l,iel,ns,nstep,ntot,nodofs=2,inc,             &
        sn,fn,ebeanz,uanz,isolver,icho,ipre,icc,iinc,iters,maxit  !<--(2)
 real(8):: permx,permy,e,v,det,dtim,theta,time,                       &
           coef,omega,resi,tol                                    !<--(3)
 character(len=15):: element = 'quadrilateral'
!---------------------------- dynamic
arrays--------------------------
 real(8),allocatable ::dee(:,:), points(:,:), coord(:,:), derivf(:,:),&
                       jac(:,:),kay(:,:),der(:,:),deriv(:,:),weights(:),&
                       derf(:,:),funf(:),coordf(:,:), bee(:,:), km(:,:),&
                       eld(:), sigma(:), kp(:,:), ke(:,:), g_coord(:,:),&
                       fun(:),c(:,:),width(:), depth(:),vol(:),loads(:),&
                       ans(:) ,volf(:,:),                               &
                       ! bk(:),store_kp(:,:,:),phi0(:),phi1(:)      !-->(4)
                       ebea(:),diag(:),diaga(:),diaga1(:)         !<--(5)
 integer, allocatable :: nf(:,:),g(:),num(:),g_num(:,:), g_g(:,:),    &
                           id(:),iebe(:),jebe(:)                   !<--(6)
!--------------------------input and
initialisation-------------------
  open (10,file='p92.dat',status='old',action='read')
  open (11,file='p92.res',status='replace',action='write')
  read (10,*) nels,nxe,nye,nn,nip,                                    &
              permx, permy, e,v, dtim, nstep, theta ,                 &
              maxit,tol,coef,omega,isolver,icho,ipre,icc,iinc    !<--(7)
  ndof=nod*2; ntot=ndof+nodf
  allocate ( dee(nst,nst), points(nip,ndim), coord(nod,ndim),         &
     derivf(ndim,nodf),jac(ndim,ndim),kay(ndim,ndim),der(ndim,nod),   &
     deriv(ndim,nod), derf(ndim,nodf),funf(nodf),coordf(nodf,ndim),   &
     bee(nst,ndof), km(ndof,ndof),eld(ndof),sigma(nst),kp(nodf,nodf), &
     g_g(ntot,nels),ke(ntot,ntot),fun(nod),c(ndof,nodf),width(nxe+1), &
     depth(nye+1),vol(ndof),nf(nodof,nn), g(ntot), volf(ndof,nodf),   &
     g_coord(ndim,nn),g_num(nod,nels),num(nod),weights(nip) )
     ! store_kp(nodf,nodf,nels),phi0(nodf),phi1(nodf) )         !-->(8)
     kay=0.0; kay(1,1)=permx; kay(2,2)=permy
     read (10,*)width , depth
  nf=1; read(10,*) nr ; if(nr>0) read(10,*)(k,nf(:,k),i=1,nr)
  call snfn(nf,nn,nodof,ndim,nels,ntot,sn,fn,ebeanz)             !<--(9)
  call formnf(nf); neq=maxval(nf)
  allocate(id(neq));                                             !<--(10)
  call form_id(nf,nn,nodof,id)                                   !<--(11)
  call deemat(dee,e,v); call sample(element,points,weights)
!--------- loop the elements to find nband and set up global
arrays-----
  nband = 0
 elements_1: do iel = 1 , nels
```

```
         call geometry_8qxv(iel,nxe,width,depth,coord,num)
         inc=0
         do i=1,8;do k=1,2;inc=inc+1;g(inc)=nf(k,num(i));end do;end do
         do i=1,7,2;inc=inc+1;g(inc)=nf(3,num(i)); end do
         g_num(:,iel)=num;g_coord(:,num)=transpose(coord);g_g(:,iel)= g
         if(nband<bandwidth(g))nband=bandwidth(g)
 end do elements_1
  write(11,'(a)') "Global coordinates "
  do k=1,nn;write(11,'(a,i5,a,2e12.4)')"Node",k,"   ",g_coord(:,k);end do
  write(11,'(a)') "Global node numbers "
  do k = 1 , nels; write(11,'(a,i5,a,8i5)')                            &
                              "Element ",k,"        ",g_num(:,k); end do
  write(11,'(2(a,i5))')                                                &
     "There are  ",neq, "  equations and the half-bandwidth is   ",nband
  allocate(loads(0:neq),ans(0:neq),iebe(ebeanz),jebe(ebeanz),ebea(ebeanz))
                                                                  !<-->(12)
               loads = .0 ; ebeanz = 0 ;                          !<-- (13)
!------------- element stiffness integration and
assembly---------------
 elements_2:  do iel = 1 , nels
       num = g_num( : ,iel ); coord= transpose(g_coord(:,num))
       g = g_g ( : , iel )  ; coordf = coord(1 : 7 : 2, : )
       km = .0; c = .0; kp = .0
    gauss_points_1: do i = 1 , nip
       call shape_der(der,points,i);  jac = matmul(der,coord)
       det = determinant(jac); call invert(jac);deriv = matmul(jac,der)
       call beemat(bee,deriv); vol(:)=bee(1,:)+bee(2,:)
       km = km + matmul(matmul(transpose(bee),dee),bee) *det* weights(i)
!----------------------now the fluid
contribution---------------------
       call shape_fun(funf,points,i)
       call shape_der(derf,points,i)  ; derivf=matmul(jac,derf)
  kp=kp+matmul(matmul(transpose(derivf),kay),derivf)*det*weights(i)*dtim
       do l=1,nodf; volf(:,l)=vol(:)*funf(l); end do
       c= c+volf*det*weights(i)
    end do gauss_points_1
       ! store_kp( : , : , iel) = kp                                 ! -->(14)
       call formke(km,kp,c,ke,theta);
       !---collect nonzero entries from element stiffness matrices---
       call formspars(ntot,g,ke,iebe,jebe,ebea,ebeanz)              !<-->(15)
                                               !@@ call formkv(bk,ke,g,neq)
 end do elements_2
!-------------------------factorise left hand
side-----------------------
 allocate( diag(neq), diaga(neq), diaga1(neq)  ) ;                  !<-- (16)
 call sortadd(ebeanz,jebe(1:ebeanz),iebe(1:ebeanz),ebea(1:ebeanz),     &
              neq+1,uanz)                                           !<-- (17)
 !-form standard or modified diagonal for the chosen preconditioner-
 call formda(neq,iebe,jebe(1:neq+1),ebea,icho,ipre,coef,omega,id,diag,&
             diaga,diaga1)                                          !<-- (18)
! --------- enter the time-stepping
loop-------------------------------
    time = .0
  time_steps:  do ns = 1 , nstep
     time = time +dtim    ;
     write(11,'(a,e12.4)') "The time is  ",time
     ans=.0;
     call kpu(iebe,jebe(1:neq+1),ebea,theta,id,loads(1:),ans(1:));!<-->(19)
     ans(0)=.0
!   ramp loading
     if(ns<=10) then
        ans(1)=ans(1)-.1/6.; ans(3)=ans(3)-.2/3.
        ans(4)=ans(4)-.1/6.
     end if
     call psolver(neq,iebe,jebe(1:neq+1),ebea,diag,diaga,diaga1,ans(1:),&
                maxit,tol,isolver,icho,ipre,icc,iinc,iters,resi); !<-->(20)
                 ans(0)=.0 ;
     write(11,'(a,i5,a,e12.4)')" SQMR took", iters,"  iterations to   &
                                   converge to ",resi
     loads = loads + ans
     write(11,'(a)') " The nodal displacements and porepressures are    :"
   do k=1,23,22; write(11,'(i5,a,3e12.4)')k,"    ",loads(nf(:,k)) ; end do
!-------------------recover stresses at
Gauss-points-------------------
   elements_4 :  do iel = 1 , nels
        num = g_num(: , iel ); coord=transpose(g_coord(:,num))
        g = g_g( : , iel )   ;  eld = loads( g ( 1 : ndof ) )
!       print*,"The Gauss Point effective stresses for
element",iel,"are"
     gauss_pts_2: do i = 1,nip
        call shape_der (der,points,i);  jac= matmul(der,coord)
        call invert ( jac );    deriv= matmul(jac,der)
        bee= 0.;call beemat(bee,deriv);sigma= matmul(dee,matmul(bee,eld))
!       print*,"Point    ",i       ;!  print*,sigma
     end do gauss_pts_2
   end do elements_4
  end do time_steps
 end program p92
```

Aside from replacing `real` by `real(8)` for the main program and all subroutines. The numbered arrow shown in the above program `p92` can be explained as follows:

`<--(1)`: Include the module `sparse_lib` into the main program `p92`.

`<--(2)`: Include the new integer parameters, `sn,fn,ebeanz,uanz,isolver,icho,ipre,icc,iinc,iters,maxit`.

`<--(3)`: Include the new integer parameters, `coef,omega,resi,tol` .

`-->(4)`: Delete the storages, `bk(:),store_kp(:,:,:),phi0(:),phi1(:)` .

`<--(5)`: Include the storages, `ebea(:),diag(:),diaga(:),diaga1(:)`.

`<--(6)`: Include the storages, `id(:),iebe(:),jebe(:)`.

`<--(7)`: Include the new input parameters, `maxit,tol,coef,omega,isolver,icho,ipre,icc,iinc`.

`-->(8)`: Delete the storages, `store_kp(nodf,nodf,nels),phi0(nodf),phi1(nodf)`.

`<--(9):` Call subroutine, `snfn(nf,nn,nodof,ndim,nels,ntot,sn,fn,ebeanz)`.

`<--(10):` Include the storage, `allocate(id(neq))` , for the identifier array.

`<--(11):` To form `id`, call the subroutine, `call form_id(nf,nn,nodof,id)`.

`<-->(12):` Delete `bk(neq*(nband+1))`, and allocating `loads(0:neq),ans(0:neq), iebe(ebeanz),jebe(ebeanz),ebea(ebeanz)`.

`<-- (13):` Set `ebeanz = 0` .

`--> (14):` Delete `store_kp( :, :, iel) = kp` .

`<-->(15):` Replace `call formkv(bk,ke,g,neq)` by `call formspars(ntot,g,ke,iebe, jebe,ebea,ebeanz)`.

`<-- (16):` Allocate storage `allocate( diag(neq), diaga(neq), diaga1(neq) )`.

`<-- (17):` By using the collected nonzero entries in element stiffness matrices, then `call sortadd(ebeanz,jebe(1:ebeanz),iebe(1:ebeanz),ebea(1:ebeanz),neq+1,uanz` to sort the three vectors, `iebe(1:ebeanz),jebe(1:ebeanz),ebea(1:ebeanz)`, and form the compressed sparse data which is still stored in this three vectors. It should be noticed that whether the final compressed sparse storage is CSC or CSR depends on the relative positions between `jebe(1:ebeanz)` and `iebe(1:ebeanz)`. When `iebe(1:ebeanz)` is located behind `jebe(1:ebeanz)`, it is CSC storage, otherwise, it is CSR storage.

`<-- (18):` `call formda(neq,iebe,jebe(1:neq+1),ebea,icho,ipre,coef,omega,id, diag,diaga,diaga1)` to form the standard or modified diagonal for Jacobi or SSOR preconditioner. It should be noticed that the data in the range `1:neq+1` in `jebe` should be called because it is CSC storage as shown in item `<-- (17)`.

`<-->(19):` Compute right hand side by `call kpu(iebe,jebe(1:neq+1),ebea,theta, id,loads(1:),ans(1:))` in terms of sparse storage instead of the element-by-element implementation.

`<-->(20):` Replace the direct solver with sparse preconditioned iterative method by calling subroutine `psolver`.