How to use \textbf{sparse\_lib} in FEM package

October 18, 2005

Contents

1 Introduction 1

2 Three Basic Components in \textbf{sparse\_lib} 1

3 Parameters or Basic Information for \textbf{sparse\_lib} Library 2
   3.1 Input Parameters 2
   3.2 Parameters or Basic Information to Be Obtained 3

4 Flowchart of Using \textbf{sparse\_lib} Library 5

5 Demonstration of Using \textbf{sparse\_lib} in Program p92 6
   5.1 Input File \texttt{p92.dat} 6
   5.2 Main Program of P92 with Direct Solver Replaced By Sparse Preconditioned Iterative Method 6
   5.3 Output File \texttt{p92.res} 11

1 Introduction

\textbf{Sparse\_lib} library is built compatibly with the Fortran 90 programs given in the book “Programming the Finite Element Method” authored by Smith and Griffith (1997). 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.

2 Three Basic Components in \textbf{sparse\_lib}

The \textbf{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.

## 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.

### 3.1 Input Parameters

```
integer:: maxit,isolver,icho,ipsol,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.
ipsol: choose a preconditioner, ipsol = 1 for Jacobi preconditioner and ipsol = 2 for SSOR preconditioner. ipsol should be used in conjunction with icho with the following combinations:

- icho = 1 and ipsol = 1, choose standard Jacobi preconditioner;
- icho = 1 and ipsol = 2, choose standard SSOR preconditioner;
- icho = 2 and ipsol = 1, choose Generalized Jacobi preconditioner;
- icho = 2 and ipsol = 2, choose Modified SSOR preconditioner.

icc: choose convergence criterion, icc = 1 for relative ‘improvement’ norm criterion, that is,
\[ R_i = \frac{\|x_k - x_{k-1}\|_\infty}{\|x_k\|_\infty} \leq stop_{tol}, \quad k = 1, 2, \ldots, max\_it. \]
where \(\|\cdot\|_\infty\) represents the infinity norm. icc = 2 for relative residual norm criterion, that is,
\[ R_r = \frac{\|r_k\|}{\|r_0\|} = \frac{\|b - Ax_k\|}{\|b - Ax_0\|} \leq stop_{tol}, \quad k = 1, 2, \ldots, max\_it. \]
where \(x_0\) is the initial guess of the solution, and \(\|\cdot\|\) represents any vector norm, but 2-norm is commonly used.

iinc: check convergence every “iinc” step when ipsol = 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.

### 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.

iebe, jebe, ebea: global row index, global column index and correspondent value, re- 
   spectively, 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 conserv-
   ative estimation of uanz, i.e., uanz = int(neq - nband/2)*nband. Alternatively,
   the estimated formula is uanz = int(0.6*beanz) for large 3-D problems, here
   beanz is the returned true number of nonzero entries collected from element “stiff-
   ness” 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.

jcs, icsc, csca: subroutine sortadd sorts element based storage, iebe, jebe, ebea,
   and leads to compressed sparse column storage jcs, 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 algo-
   rithm 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.
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, ipsol, 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 iebe,jebe,ebea into sparse global stiffness
matrix storage, icsc,jcsc,csca with returned true uanz.

Form preconditioner (standard or modified diagonal):
call forma(neq,icsc,jcsc,csca,icho,ipsol,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,ipsol,icc,iinc,iters,resi).
End for
```
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.

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 ipsol 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 1 2 1 1 0 1 3 0 1 1 4 0 1 0 15 0 1 0
16 0 1 1 1 7 1 1 0 1 8 0 1 1 9 0 1 0 20 0 1 0
21 0 0 1 2 2 0 0 0 23 0 0 1
```

5.2 Main Program of P92 with Direct Solver Replaced By Sparse Preconditioned Iterative Method

```fortran
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,ipsol,icc,iinc,iters,maxit !<--(2)
```
real(8):: permx, permy, e, v, det, dtim, theta, time, &
        coef, omega, resi, tol
character(len=15):: element = 'quadrilateral'
!----------------------------- dynamic arrays---------------------------
real(8), allocatable :: dee(:, :], points(:, :], coord(:, :], derivf(:, :], &
        jac(:, :], kay(:, :], der(:, :], derivf(:, :], weights(:, :], &
        derf(:, :], funf(:, :], coordf(:, :], bee(:, :], km(:, :], &
        eld(:, :], sigma(:, :], kp(:, :], ke(:, :], g_coord(:, :], &
        fun(:, :], c(:, :], width(:, :], depth(:, :], vol(:, :], loads(:, :], &
        ans(:, :], volf(:, :)), &
        !<--(3)
character(len=15):: element = 'quadrilateral'
!----------------------------- dynamic arrays---------------------------
real(8), allocatable :: dee(:, :], points(:, :], coord(:, :], derivf(:, :], &
        jac(:, :], kay(:, :], der(:, :], derivf(:, :], weights(:, :], &
        derf(:, :], funf(:, :], coordf(:, :], bee(:, :], km(:, :], &
        eld(:, :], sigma(:, :], kp(:, :], ke(:, :], g_coord(:, :], &
        fun(:, :], c(:, :], width(:, :], depth(:, :], vol(:, :], loads(:, :], &
        ans(:, :], volf(:, :)), &
        !<--(4)
integer, allocatable :: nf(:, :], g(:, :], num(:, :], g_num(:, :], g_g(:, :), &
        id(:, :], ibe(:, :], ibe(:, :], icsc(:, :], jcs(:, :)), &
        !-->(5)
integer, allocatable :: nf(:, :], g(:, :], num(:, :], g_num(:, :], g_g(:, :), &
        id(:, :], ibe(:, :], ibe(:, :], icsc(:, :], jcs(:, :)), &
        !-->(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, &
        maxit, tol, coef, omega, isolver, icho, ipsol, icc, iinc
end do elements_1
!
end do elements_1
write(11,'(a)')  "Global coordinates "
do k=1,nn;write(11,'(a,i5,a2e12.4)')"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 &
   "There are ",neq, " equations and the half-bandwidth is ",nbnd
allocate(loads(0:neq),ans(0:neq),iebe(ebeanz),jebe(ebeanz),ebea(ebeanz))
!<-->(12)
loads = .0 ; ebeanz = 0 ; !<-- (13)

-------------- element stiffness integration and assembly----------------
elments_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(jac,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,ke,theta);
!----collect nonzero entries from element stiffness matrices---
call formspars(ntot,g,ke,iebe,jebe,ebea,ebeanz)
!<-->(15)
!@@ call formkv(bk,ke,ntot,neq)
end do elements_2

--------------factorise left hand side---------------------------
uanz = int(neq - nband/2)*nband
allocate(icsc(uanz), jcsc(neq+1), csca(uanz), diag(neq), diaga(neq), &
diaga1(neq))
call sortadd(iebeanz, 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 forma(neq,icsc,jcsc,csca,icho,ipsol,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
5 DEMONSTRATION OF USING SPARSE_LIB IN PROGRAM P92

! 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,csc,diag,diaga,diaga1,ans(1:),maxit, &
tol,isolver,icho,ipsol,icc,iinc,iter,resi); ans(0)=.0;
write(11,'(a,i5,a,e12.4)')" SQMR 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-----------------

! 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,ipsol, &
icc,iinc,iter maxi
<--(3): Include the new integer parameters, coef,omega,resi,tol .
<--(4): Delete the storages, bk(:,),storekp(:,,:),phi0(:,),phi1( : ) .
<--(5): Include the storages, ebea(:,),csc(:,),diag(:,),diaga(:,),diaga1( : ) .
<--(6): Include the storages, id(:,),iebe(:,),jebe(:,),icbe(:,),jcbe(:,).
<--(7): Include the new input parameters, maxit,tol,coef,omega,isolver,icho, &
ipsol,icc,iinc.
<--(8): Delete the storages, storekp(nodf,nodf,nels),phi0(nodf),phi1(nodf).
5 DEMONSTRATION OF USING SPARSELIB IN PROGRAM P92

<--(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,ns)` 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.
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 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, ipsol = 1’, while for standard SSOR preconditioned SQMR method, we set isolver = 1, icho = 1, ipsol = 2. For GJ preconditioned SQMR method, it is just required to set ‘isolver = 1, icho = 2, ipsol = 1’ and for MSSOR preconditioned SQMR method, it is to set ‘isolver = 1, icho = 2, ipsol = 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.

1. Output file generated by direct solution method

<table>
<thead>
<tr>
<th>Global coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node  1</td>
</tr>
<tr>
<td>Node  2</td>
</tr>
<tr>
<td>Node  3</td>
</tr>
<tr>
<td>Node  4</td>
</tr>
<tr>
<td>Node  5</td>
</tr>
<tr>
<td>Node  6</td>
</tr>
<tr>
<td>Node  7</td>
</tr>
<tr>
<td>Node  8</td>
</tr>
<tr>
<td>Node  9</td>
</tr>
<tr>
<td>Node 10</td>
</tr>
<tr>
<td>Node 11</td>
</tr>
<tr>
<td>Node 12</td>
</tr>
<tr>
<td>Node 13</td>
</tr>
<tr>
<td>Node 14</td>
</tr>
<tr>
<td>Node 15</td>
</tr>
<tr>
<td>Node 16</td>
</tr>
<tr>
<td>Node 17</td>
</tr>
<tr>
<td>Node 18</td>
</tr>
<tr>
<td>Node 19</td>
</tr>
<tr>
<td>Node 20</td>
</tr>
<tr>
<td>Node 21</td>
</tr>
<tr>
<td>Node 22</td>
</tr>
<tr>
<td>Node 23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Global node numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element 1</td>
</tr>
<tr>
<td>Element 2</td>
</tr>
<tr>
<td>Element 3</td>
</tr>
</tbody>
</table>

<--> (18): Replace the direct solver with sparse preconditioned iterative method by calling subroutine psolver.
5 DEMONSTRATION OF USING SPARSELIB IN PROGRAM P92

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

2. Output file generated by SJ preconditioned SQMR method (isolver=1, icho = 1, ipsol = 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
  ---> 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

3. Output file generated by SSOR preconditioned SQMR method (isolver=1, icho = 1, ipsol = 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
5. Output file generated by MSSOR preconditioned SQMR method (isolver=1, icho = 2, ipsol = 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 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

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 0.3000E+01
The time is 0.1900E+02
  --> GJ preconditioned SQMR solver
  with relative residual norm criterion!
******************************************************************************
SQMR converges to user-defined tolerance.
SQMR took 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
  --> 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 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 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
  --> 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 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

5. Output file generated by GJ preconditioned SQMR method (isolver=1, icho = 2, ipsol = 1, icc = 2)
SQMR took 25 iterations to converge to $0.3545 \times 10^{-7}$
The nodal displacements and porepressures are:
\[
\begin{array}{ccc}
1 & 0.0000E+00 & -0.1233E+00 \\
23 & 0.0000E+00 & 0.0000E+00 -0.1000E+00
\end{array}
\]
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 $0.3329 \times 10^{-7}$
The nodal displacements and porepressures are:
\[
\begin{array}{ccc}
1 & 0.0000E+00 & -0.2872E+00 \\
23 & 0.0000E+00 & 0.0000E+00 -0.2000E+00
\end{array}
\]
The time is $0.3000E+01$

--- MSSOR preconditioned SQMR solver
with relative residual norm criterion!

SQMR converges to user-defined tolerance.

SQMR took 25 iterations to converge to $0.6407 \times 10^{-9}$
The nodal displacements and porepressures are:
\[
\begin{array}{ccc}
1 & 0.0000E+00 & -0.4277E+01 \\
23 & 0.0000E+00 & 0.0000E+00 -0.8811E+00
\end{array}
\]
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 $0.4624 \times 10^{-9}$
The nodal displacements and porepressures are:
\[
\begin{array}{ccc}
1 & 0.0000E+00 & -0.4424E+01 \\
23 & 0.0000E+00 & 0.0000E+00 -0.8636E+00
\end{array}
\]