SPARSEIT++ User Guide

P.A. Forsyth*

May 21, 2010

Contents

*Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, *paforsyt@uwaterloo.ca*, tel: (519) 888-4567x34415, fax: (519) 885-1208

1 Introduction

The SPARSEIT++ package is a general purpose C++ library for iterative solution of nonsymmetric sparse matrices. There is no restriction on the structure of the sparse matrix. The solution method uses a PCG (preconditioned conjugate gradient-like) algorithm.

The speed and robustness of an iterative method depends critically on the preconditioning method employed. SPARSEIT++ uses an incomplete factorization (ILU) based on either level of fill or drop tolerance [1, 2] incomplete factorizations. This method has been applied successfully to problems in computational fluid dynamics [3, 4, 5], petroleum reservoir modelling [6], groundwater contamination [7], stress analysis [8, 9], semi-conductor device simulation [10] and computational finance [11, 12, 13, 14].

Note in general, it is advisable that there are non-zeros on the diagonals of the matrix. This is because the most efficient methods do not use pivoting. However, an option is available when doing a drop tolerance incomplete factorization, which carries out column pivoting.

The incomplete factorization is accelerated by CGSTAB acceleration [15] or ORTHOMIN [6]. Convergence of the algorithm cannot be guaranteed for arbitrary sparse matrices, but computational experience in many application areas has been good. In particular, if the the preconditioned system is *sufficiently* positive definite, then convergence can be expected.

Note that:

- A direct solver is always more efficient for problems which arise from one dimensional or near one dimensional PDE's.
- An iterative solver will usually be more efficient than a direct solver for a two dimensional PDE problem, for a sufficiently large problem, and if a good initial guess is available (a time dependent PDE).
- Unless the matrix is very badly conditioned, an iterative method will be superior to a direct method for a three dimensional problem.

2 Classes

There are only three classes that a user of the SPARSEIT++ package must be aware of: MatrixIter, Param, and MatrixStruc.

MatrixStruc This is a helper class which the user can use to specify the sparsity pattern (nonzero structure) of the sparse matrix. This class is used to initialize the matrix storage and solution class MatrixIter.

MatrixIter This class contains all information about the sparse matrix and the right hand side, and carries out the task of solving the matrix.

Param Contains public data members only. User controlled parameters for the solution algorithms.

3 Data Structures

3.1 Overview

The MatrixIter class stores the sparse matrix internally in the form of compressed row storage or form. You do not need to worry too much about this, since this data structure is set-up in a very transparent way using the class MatrixStruc. However, a few examples of the efficient method for accessing entries in the sparse matrix will be given in a later section. You may find that using this idea simplifies your code considerably.

3.2 Data Structure Set-up: Using the MatrixStruc Class

Perhaps the simplest way to specify the structure of a sparse square matrix A with nonzero entries in row i and column j, is simply to list those pairs (i, j) where A_{ij} are nonzero. These (i, j) pairs are passed to the MatrixStruc class in the following way:

```
// specify nonzero locations in matrix

    MatrixStruc matrix_struc(n); // A is an n x n matrix

    ..............

     i = 10; j = 22;

     matrix_struc.set_entry(i, j); // row i=10, column j=22
                           // has a nonzero entry

     i=0; j = 12;
     matrix_struc.set_entry(i,j); // row i=0, column j=12
                          // has a nonzero entry

    i = 10; j = 22;
    matrix_struc.set_entry(i,j); // repeats ok
```

The prototypes for the constructor and the *set_entry* functions are:

```
MatrixStruc( const int n   // number of unknowns
                          // i.e. A is an n x n matrix
      );

void MatrixStruc::
     set_entry (int row, // global row number, 0<= row <= n-1
                int col  // global col number, 0<= col <= n-1
```

```
);
```

An example of using the MatrixStruc class to specify the nonzero structure for a seven point three dimensional finite difference operator is given in the file *main.C*. You must ensure that there is a nonzero entry on the diagonal of each row.

The data structure for A is temporarily stored in the form of row-linked lists. This is very inefficient for carrying out the actual computations, so after you have finished specifying all the non-zero locations using the *set_entry* function, you request that this data structure be converted to packed form:

```
MatrixStruc matrix_struc(n); // A is an n x n matrix

...........      // specify structure using set_entry

matrix_struc.pack();     // convert to packed storage format
```

Note that once you have requested that the data structures be packed, you cannot add new non-zero locations. You must delete the MatrixStruc class and start over again.

Note: by default, the matrix is intialized with nonzero entries on the diagonal. If this is not desired, a special constructor can be used.

4 Initializing the MatrixIter Class

Once the data structure for your sparse matrix has been set-up, the actual sparse matrix class can be initialized:

```
MatrixStruc matrix_struc(n);
..............

matrix_struc.pack();


MatrixIter matrix( matrix_struc); // initialize
                                  // MatrixIter class
                                  // with data structure
                                  // in  matrix_struc
```

5 Inserting and reading values for the entries in the sparse matrix

5.1 Use global (row, column) method

Assume that we are solving:

$$Ax = b \tag{1}$$

with sparse matrix A, right hand side b, and solution vector x.

Now suppose we want to set or access the entries in A or right hand side b. The simplest way to do this is to use the *aValue* and *bValue* functions. These functions return references to the values, and so may be used to read or set values. The sparse matrix entries can be accessed using $(row, column)$ notation:

```
// Example of setting values for matrix and right hand side


 MatrixIter matrix(  matrix_struc);

 ................

   i = 20; j = 15; // we know that there is a nonzero in these
                   // rows and columns

   matrix.aValue(i, j) = 20. ; // set value

   matrix.bValue( i ) = 100. //  set right hand side for row 20

   i = 14; j = 10; //

   cout << "Matrix entry: row " << i
             << "  column " << j
             << matrix.aValue(i,j) << endl; // read entry

   cout << "Right hand side for row " << i
        << matrix.bValue(i);                // read rhs
```

Note that if the requested (i, j) pair was not specified in MatrixStruc, then this will flag an error. For debugging purposes, before accessing an (i, j) pair, you can check to make sure it is in the data structure:

```
//   Check to make sure the requested (i,j) pair
//   is in the sparse matrix data structure

 MatrixIter matrix( matrix_struc);
```

```
 ..............


   i = 20; j = 15;

   if( matrix.check_entry( i,j) != 0){
         matrix.aValue(i,j) = 10.;
    }
    else{
      cout << "error: row " << i
           << " column: << j
           << " not in data structure " << endl;
      }
```

5.2 Full row entry

Sometimes, it is convenient to load entries row by row. In this case, an efficient and easy way to insert entries in the $i'th$ row is:

```
// Use of full storage vector for entering
// values in a row

 MatrixIter matrix(  matrix_struc);

 ..............


 double* row = new double [n]; // temporary n-length
                               // array (A is an nxn matrix)

  { int k;
    for(k=0; k<n; k++){ row[k] = 0.0;} // zero row
  }


  i = 10 ;  // insert entries in row 10


  row[10] = 20.; // entry in (row, column) = (10,10)


  row[9] = 10.; // entry in (row, column) = (10, 9)
```

```
  row[14] = 4.;// entry in (row, column) = (10, 14)

  matrix.set_row( i, row);  // insert entries in row i

  matrix.zero_row(i, row); // zero entries in row vector
                           // where nonzeros in row i of A
                           // exist

//  go on to another row
```

5.3 Use of Compressed Row Format

The matrix entries can be accessed more efficiently using the compressed row format directly. At first reading, this Section may be skipped.

Suppose we want to access or set the entries for any row in the sparse matrix. If, for example, we wanted to print out each entry in the sparse matrix A (row by row) and the right hand side vector b:

```
   MatrixIter matrix(  matrix_struc );

   ..............

   for(i=0; i<n; i++){ // main loop over rows
     cout << "row " i" << endl;

     for( k = matrix.rowBegin(i);
          k < matrix.rowEndPlusOne(i);
          k++){ // loop over entries in row i

       cout << "column index: " << matrix.getColIndex( k )
            << " entry value: " << matrix.aValue( k )
            <<  endl;

      }// end loop over entries in row i

     cout << "right hand side for row "<< i
          << matrix.bValue( i ) << endl;

    }// end main loop  over rows
```

We can also use these functions to insert entries in the matrix A and right hand side b.

```
//  example of inserting values for entries in
//  sparse matrix A and right hand side b
//
   MatrixIter matrix(  matrix_struc );

   ..............

   for(i=0; i<n; i++){ // main loop over rows
     int  isave = -1;
     double temp = 0.0;
       for(k=matrix.rowBegin(i);
           k < matrix.rowEndPlusOne(i);
           k++){//  loop over nonzeros in row

            int col_index = matrix.getColIndex( k );

            if( col_index != i){ // not diagonal entry

               matrix.aValue(k) = -1.0; // set offdiags = -1
               temp += matrix_ptr->aValue(k);

            }// end not diagonal
            else{// diagonal entry
              isave = k;
            }// end diagonal entry

          }// end loop over nonzeros in row

          matrix.aValue(isave) = -temp + .1;
                // set diagonal == -(sum of offdiagonals) + .1

          matrix.bValue( i ) = 1.0;// set right hand side in
                                   // each row
       } // end main loop over rows
```

6 Solution of the System

6.1 General

The iterative algorithm consists of two steps:

1. Incompletely Factor the Matrix (ILU)
2. Solve (iteration)

There are two options available for available for carrying out the incomplete (ILU) factorization. These are:

Drop Tolerance Sparse gaussian elimination is carried out on the matrix. Fill-in entries which are less than a user specified tolerance are discarded. The dropping criteria is to discard a fill-in entry if

$$|fill - in| < (drop\ tolerance)|current\ diagonal\ in\ that\ row| \tag{2}$$

Level of Fill The sparsity pattern of the matrix is analyzed, and the pattern of non-zero entries in the ILU is determined (without knowledge of the numerical values of these entries) ahead of time. The criteria for allowing fill-in entries is based on the level of fill of an entry [1, 2]. Briefly, the original entries are labeled level 0, entries which appear due to elimination of level 0 entries are labeled level 1, and so on. A $level = 0$ ILU allows no fill-in, a $level = \infty$ ILU is a complete factorization. This stage is known as the symbolic ILU factorization. After the symbolic factorization has been carried out, a numeric ILU is then performed.

Note that the symbolic and numeric ILU are combined in a single step when the drop tolerance ILU is requested.

It is typical in time-dependent PDE problems to solve a matrix with the same sparsity pattern many times. As well, the actual numerical values of the entries in the sparse matrix may change only slightly from timestep to timestep. This suggests the following strategy:

- For a level of fill ILU, do the symbolic factorization once only at the start of a simulation, and then use this same sparsity pattern for subsequent solves (i.e. for each matrix solve, do only the numeric ILU).
- For a drop tolerance ILU, carry out the drop tolerance ILU only every few timesteps. After an initial sparsity pattern for the ILU is determined, use the same sparsity pattern for several timesteps (i.e. for each matrix solve, do only the numeric ILU, based on previously determined sparsity pattern).

SPARSEIT++ allows the user to use any of these strategies.

6.2 Convergence Criteria

Suppose we want to solve the system

$$Ax = b \tag{3}$$

where A is the sparse matrix, b is the right hand side, and x is the solution vector. At the k^{th} iteration the iterative solution method produces an estimate of the solution x^k with residual r^k given by:

$$r^k = b - Ax^k \tag{4}$$

The SPARSEIT++ solver will stop iterating and return a the current solution if *any* of the following criteria are met:

- The current rms residual has been reduced to less than a user-specified fraction of the initial residual:

$$\frac{\|r^k\|_2}{\|r^0\|_2} < tol \tag{5}$$

- Each solution update is less than the corresponding entry in a tolerance vector (the actual criteria is a bit more complicated than this, but the following gives you the basic idea):

$$|x_i^k - x_i^{k-1}| < toler_i \quad \forall i \tag{6}$$

- The current iteration exceeds the maximum number of iterations specified by the user.

6.3 Solution Parameters

Parameters are passed to the solver functions via the Param class. The Param class contains the following public data members:

int order Specify ordering for the ILU.

=0 Original ordering.

=1 RCM ordering (default). Note that the RCM ordering algorithm requires that the symmetrized graph of the matrix be irreducible.

int level Level of fill for ILU. $0 < level < \infty$. (default = 1.)

int drop_ilu =0 Use level of fill ILU (default).

=1 Use drop tolerance ILU.

double drop_tol Drop threshold. Used only if *drop_ilu = 1.* (Default = 10^{-3}). If a_{ij}^k represents the entries in A after k steps of incomplete elimination (using the diagonals as pivots), then a fill-in entry a_{ij}^k is dropped if

$$|a_{ij}^k| < drop\_tol|a_{ii}^k| \tag{7}$$

int ipiv If drop_ilu = 1, then column pivoting is requested if $ipiv = 1$. (default = 0).

int iscal =0 Do not perform scaling.

=1 Scale matrix and right hand side by inverse of the diagonal (default). If $ipiv = 1$, and $drop\_ilu = 1$, the maximum entry in each row is used as the scaling factor. If scaling used, then matrix and right hand side altered in each call to the solver.

int nitmax Maximum number of iterations. (Default = 30).

double resid_reduc Convergence criteria based on l_2 residual reduction (see equation (5)). (default = 10^{-6}).

int info =0 No detailed information printed.

=1 Detailed solver information printed to ofstream oFile (Default).

int iaccel =0 CGSTAB

=1 ORTHOMIN

int new_rhat The CGSTAB acceleration method produces residual vectors which are orthogonal to the Krylov subspace generated by $(A^t)^k\hat{r}$. Typically, $\hat{r} = r^0$. However, in rare circumstances, this can result in a zero divide. This may happen if the initial guess x^0 generates r^0 which has mostly zero entries. Another possibility for $\hat{r}$ is $(LU)^{-1}r^0$, where LU is the incomplete factorization of A.

=0 Use $\hat{r} = r^0$ (default).

=1 Use $\hat{r} = (LU)^{-1}r^0$.

This is ignored if ORTHOMIN acceleration requested.

int north Number orthogonal vectors kept before restart (Default = 10). Used for ORTHOMIN option only.

7 Passing the Tolerance Vector

For both level of fill ILU and drop tolerance ILU, the update solution tolerance should be specified (see equation(6)). This tolerance vector is specified in the following way

```
//  Set convergence tolerance vector
//
   MatrixIter matrix(  matrix_struc );

     ........

    double* toler = new double [n]; // A is nxn

    int i;
    for(i=0; i<n ; i++){
       toler[i] = .... // tolerance for each variable
     }

    matrix.set_toler( toler );
```

Note that if $toler_i = 0; i = 0, ..., n-1$, then the iteration will terminate on the residual reduction criteria equation (5).

8 Parameters for the solve function

We list here details of the MatrixIter::solve function:

```
ofstream oFile ;
oFile.open("info_iteration.dat",ios::out); // output file for
                                // diagnostics

   void MatrixIter::solve( ParamIter& param,
              double* x, int& nitr,
               ofstream& oFile,
               const int initial_guess=0);
```

Param& param Structure with ILU and solution parameters.

double* x Array of length n. Storage must be allocated in caller. If *initial_guess* $= 0$, x is zeroed in *solve*. If *initial_guess* $= 1$, then on entry, x contains the initial guess for the solution vector. On exit, in all cases, x contains the solution vector.

int nitr On exit, contains the number of iterations required for the solve. If $nitr = -1$, then the number of iterations exceeded *nitmax* (see description of *Param* class data members) before the solution tolerance was reached.

oFile output for diagnostics.

int initial_guess

=0 Assume that the initial guess is $x = 0$. (This is a good initial guess for time dependent problems if x is the *change* in the solution from one time level to the next.) Note that in this case, any information in x is destroyed on entry. (Default = 0.)

=1 On entry, use x as an initial guess, i.e. compute

$$r^0 = b - Ax^0 \tag{8}$$

and return $x^k \simeq x^0 + A^{-1}r^0$.

9 Level of Fill ILU and Solve

If a level of fill ILU has been selected, then the symbolic factorization must be carried out. As mentioned above, if the sparsity pattern of the matrix does not change over the course of a simulation, then this need be done only once. The following is the prototype for the symbolic factorization:

```
//   prototype for symbolic ILU


   void MatrixIter::sfac(ParamIter& param , // ILU parameters
                         ofstream& oFile
                         );
```

9.1 Calling Sequence for Level of Fill ILU and Solve

Now, the complete sequence for solving a matrix with a level of fill ILU is:

```
MatrixStruc matrix_struc(n);

 ............. // set up sparsity pattern

   matrix_struc.pack(); // pack the structure

MatrixIter matrix(  matrix_struc );
............. // insert entries in sparse matrix
              // and rhs

Param param;  // set up parameters for ILU and solve

   param.drop_ilu = 0; // flag level of fill ilu

ofstream oFile ;
oFile.open("info_iteration.dat",ios::out); // output file for
                              // diagnostics

.............

matrix.sfac( param, oFile); // symbolic ILU for level of fill
                     // usually only once per simulation

..............

double* x = new double [n];  // allocate solution
                             // vector and initial guess

double* toler = new double [n]; // allocate tolerance
                                // vector

...............

matrix.set_toler( toler ); // pass tolerance vector to solver

matrix.solve( param, x, nitr, oFile,  initial_guess); // solve
```

9.2 Calling Sequence for Drop Tolerance ILU and Solve

If a drop tolerance ILU is requested, then the complete sequence of steps for a drop tolerance ILU is

```
MatrixStruc matrix_struc(n);

 ............. // set up sparsity pattern

  matrix_struc.pack(); // pack the structure

MatrixIter matrix( matrix_struc );
............. // insert entries in sparse matrix
              // and rhs

Param param;  // set up parameters for ILU and solve

  param.drop_ilu = 1; // flag drop tolerance ilu

.............

double* x = new double [n];  // allocate solution
                             // vector and initial guess

double* toler = new double [n]; // allocate tolerance
                                // vector

..............

matrix.set_toler( toler ); // pass tolerance vector to solver

matrix.solve( param, x, nitr, oFile,  initial_guess); // solve
```

Note that there is no symbolic factorization for the drop tolerance ILU since the numeric factor and solve are combined with the drop tolerance ILU.

10 Sequence of Solves

For time dependent PDE simulations, or nonlinear problems, it is typical to carry out the symbolic factorization only once for a level of fill ILU. For a drop tolerance ILU, the drop tolerance numerical factorization is carried out infrequently, and the same sparsity pattern is used for a number of solves.

10.1 Sequence of solves: level of fill ILU

For example, for a level of fill ILU, we would typically see the following use:

```
// Example of sequence of solves using a level of fill
// ILU.  Same sparsity pattern for the matrix at each
// solve.


   MatrixIter matrix(  matrix_struc );
   ...........


   ofstream oFile ;
   oFile.open("info_iteration.dat",ios::out); // output file for
                                 // diagnostics

   param.drop_ilu = 0; // level of fill ilu

   matrix.sfac( param, oFile ); // once only at start
   ...........

   for(timestep = 1; timestep <= max_steps; timestep++){
   .........

         matrix.solve(param, x, nitr, oFile,  initial_guess);
                // soln for each timestep

    }// end timestep loop
```

10.2 Sequence of solves: drop tolerance ILU

For a drop tolerance ILU, we might see the following:

```
// Example of sequence of solves using a drop tolerance
// ILU.


   MatrixIter matrix( matrix_struc );
   ...........


   param.drop_ilu = 1; // flag drop tolerance ilu

   for(timestep = 1; timestep <= max_steps; timestep++){
   .........
```

```
if( (timestep/10)*10 == timestep){
   matrix.set_sym_unfactored(); // force new drop tol
                           // ILU every tenth timestep
                           // otherwise, carry out numeric
                           // ILU based on sparsity pattern
                           // from previous solve
 }

      matrix.solve(param, x, nitr, oFile, initial_guess);
             // soln for each timestep

 }// end timestep loop
```

Note: If a drop tolerance ILU is used, then the default for a sequence of solves is:

1. A full drop tolerance ILU is carried out on the first call to *solve*. This ILU dynamically determines the sparsity pattern of the factors as elimination proceeds.
2. On subsequent calls to *solve*, the numeric ILU is carried out using the static sparsity pattern for the factors determined from the first call to *solve*.
3. A full drop tolerance ILU (dynamically determine the sparsity pattern of the ILU based on current matrix entries) can be forced by a call to *set_sym_unfactored* before a call to *solve*.

11 Multiple Right Hand Sides

In some circumstances, you may want to solve the same matrix with different right hand sides. For example, this may be useful for a time dependent linear PDE problem with a constant timestep.

In this case, it is not necessary to redo the numerical incomplete factorization, and work can be saved by reusing the ILU factors. In this situation, the *matrix.solve* function can be replaced by:

```
matrix.solveWithOldFactors( param, x, nitr, oFile,  initial_guess);
```

However, the usual *MatrixIter::solve* function must have been called at least once (in order to factor the original matrix) before *MatrixIter::solveWithOldFactors* is called. After the MatrixIter::solve function has been called, the right hand side can be changed, but the the matrix values must not be altered. If you selected scaling for the original call to *MatrixIter::solve* , then the right hand side is scaled in *MatrixIter::solveWithOldFactors*, in order to be consistent with the original scaled matrix and factors.

As an example, if we are solving a timedependent PDE with constant timesteps, so that the same matrix is being solved at each timestep, then the following code fragment would be appropriate:

```
//   Use of same factors for each solve
//   Assumes that the same matrix is being
//   solved at each timestep


   MatrixIter matrix(  matrix_struc  );
   ...........

   ofstream oFile ;
   oFile.open("info_iteration.dat",ios::out); // output file for
                                 // diagnostics


   param.drop_ilu = 0; // level of fill ilu

   matrix.sfac( param, oFile); // once only at start
   ...........

     matrix.aValue(...); // insert matrix values once only

   for(timestep = 1; timestep <= max_steps; timestep++){
   .........

      matrix.bValue(...); // set rhs vector each timestep

     if( timestep == 1){ // factor once only
        matrix.solve(param, x, nitr, oFile,  initial_guess);
                // soln for first timestep
      }
      else{ // resuse old factors
       matrix.solveWithOldFactors(param, x, nitr, oFile, initial_guess);
       }    // soln for subsequent timesteps

    }// end timestep loop
```

12 Use of SPARSEIT++

To use the SPARSEIT++ package, simply include the following header files:

```
     #include "def_compiler.h"
     #include "Standard.h"

     #include "SparseItObj.h"
```

```
using namespace SparseItObj;
using namespace std;
```

The header *def_compiler.h* contains a #define statement to indicate if the compiler is Visual C++ or not. Visual C++ is non-conforming in terms of the Ansi C++ standard, so we have to set some preprocessor directives.

Similarly, the standard library headers required are in the header *Standard.h*, which have to be adjusted for Visual C++.

The actual class defininitions for the Sparseit++ library are in *SparseItObj.h*. These declarations are all in the namespace *SparseItObj*. The *using* directive will bring these names into the global scope for this file. If this is not desired, then all the Sparseit++ classes should be prefaced with *SparseItObj::*, i.e.

```
matrix.solve(param, x, nitr, oFile, initial_guess);
      // if using namespace SparseItObj;

SparseItObj::matrix.solve(param, x, nitr, oFile, initial_guess);
      // without using directive
```

Then, link the SPARSEIT++ object modules in with your software. The name of the object modules or library files will be given in the README file in the SPARSEIT directory.

An example of the calling sequence is given the file *main.C*. The README file will also give you instructions on compiling and linking the example in *main.C*.

12.1 Exception Handling

Note: you should surround all declarations if SPARSEIT objects and use of SPARSEIT objects by *try...catch* clauses.

```
try{

  // SPARSEIT objects

 }
 catch(std::bad_alloc){
     // not enough memory
     // do something
 }
 catch( General_Exception excep){
        // General_Exception is defined in "util.h"
        // something bad has happened
     oFile << excep.p << "\n" ;
```

```
            // now, do something
    }
```

13 Prototypes of Functions Referred to In This Guide

We give a list of the function prototypes of the functions described in this guide in the order we have described them.

```
// MatrixStruc constructor

 MatrixStruc( const int n_in // number of unknowns
                         // i.e. A is an n x n matrix
        );

 void MatrixStruc::set_entry (
                int row, // global row number, 0<= row <= n-1
                int col  // global col number, 0<= col <= n-1
                 );
               // set (i,j) as nonzero entry in matrix


 void MatrixStruc::pack(void); // convert from linked list ->
                               // compressed row


 MatrixIter::MatrixIter(
             MatrixStruc& matrix_struc // data structure
           );


   void MatrixIter::zeroa(void); // set all entries in A to zero

   void MatrixIter::zerob(void); // set all entries in rhs to zero

 double& MatrixIter::aValue( const row ,// global row number of entry
                           const col //  global col number of entry
                ); // return reference to  A_{i,j}


 double& MatrixIter::bValue(const int i);
                       // reference to right hand side vector
                       // i'th row 0 < i < n-1
```

```
int MatrixIter::check_entry( int row , // row index
                             int col   // col index
                           );
       // returns 0 if requested (i,j) pair not in
       // data structure
       // returns !0 if ok


  void MatrixIter::set_row( const int i, // set values of i'th row
           double* row // row is full storage mode
                       // for i'th row (i.e. an n=length
                       // vector)
         );

  void MatrixIter::zero_row( const int i, // i'th row
                 double* row // full storage n-length
                             // array.  Zero those entries
                             // in row corresponding to nonzeros
                             // in row i of matrix
               );



double& MatrixIter::aValue( const int k );
            // efficient access to the sparse matrix entries
            // based on compressed row storage
            // uses routines below
            // returns reference to entry

 // the following functions are used to access and set
 // sparse matrix entries;
 // In row i, entries  are stored in aValue[k], such
 //  that rowBegin(i) < = k < rowEndPlusOne( i );
 //  So, for MatrixIter matrix, if A_{i,j} corresponds
 //     to a nonzero in row i and column j, then:
 //
 //       matrix.aValue(k) = A_{i,j}, with j = matrix.getColIndex(k)
 //
 //        matrix.rowBegin(i) < = k < matrix.rowEndPlusOne( i )


  int MatrixIter::rowBegin( const int row );

  int MatrixIter::rowEndPlusOne( const int row);
```

```
   int MatrixIter::getColIndex (const int k );



   void MatrixIter::set_toler( const double* toler // tolerance vector
                           // n-length array allocated in caller
             );

void MatrixIter::set_user_ordering_vector( int* lorder);
             // user ordering vector lorder[new_order] = old_order
             // lorder - n-length array allocated and deallocated
             // by user. Overrides any other ordering option
             // if set != NULL


     void MatrixIter::solve( ParamIter& param, // solution parameters
                                              // allocated in caller
               double* x, // on exit solution (n-length array)
                          // allocated in caller
               int &nitr, // return number of iterations
                          // return -1 if not converged
               ofstream& oFile, // output file
               const int initial_guess = 0
                  // = 0 assume x = init guess = 0
                  //    NOTE: x is zeroed in solve for this option
                  //    returns x = A^{-1}b
                  // !=0 assume x is initial guess
                  //    returns x = A^{-1}res^0 + x^0
                  //    res^0 = b - A x^0
              );

       void MatrixIter::solveWithOldFactors(
                     ParamIter& param, double* x, int &nitr,
                     ofstream& oFile,
                     const int initial_guess = 0);
           // use old factors for solve
           // used for solution of same matrix with
           // different right hand sides
           // void MatrixIter::solve must have been
           // called at least once previously

     void MatrixIter::set_sym_unfactored(void);
      // for drop tol ilu, force new drop tol ilu,
      // otherwise, uses previous sparsity pattern
```

References

[1] E.F. D'Azevedo, P.A. Forsyth, and W.P. Tang. Ordering methods for preconditioned conjugate gradient methods applied to unstructured grid problems. *SIAM J. Matrix Anal. Applic.*, 13:944–961, 1992.

[2] E.F. D'Azevedo, P.A. Forsyth, and W.P. Tang. Towards a cost effective ILU preconditioner with high level fill. *BIT*, 32:442–463, 1992.

[3] H. Jiang and P.A. Forsyth. Robust linear and nonlinear strategies for solution of the transonic Euler equations. *Computers & Fluids*, 24:753–770, 1995.

[4] P.A. Forsyth and H. Jiang. Nonlinear iteration methods for high speed laminar compressible Navier-Stokes equations. *Computers & Fluids*, 26:249–268, 1997.

[5] P.A. Forsyth and H. Jiang. Robust numerical methods for transonic flows. *Int. J. Num. Meth. Fluids*, 24:457–476, 1997.

[6] G.A. Behie and P.A. Forsyth. Incomplete factorization methods for fully implicit simulation of enhanced oil recovery. *SIAM J. Sci. Stat. Comp.*, 5:543–561, 1984.

[7] P.A. Forsyth, Y.S. Wu, and K. Pruess. Robust numerical methods for saturated-unsaturated flow with dry initial conditions in heterogeneous media. *Adv. Water Res.*, 18:25–38, 1995.

[8] J.K. Dickinson and P.A. Forsyth. Preconditioned conjugate gradient methods for three-dimensional linear elasticity. *Int. J. Num. Meth. Eng.*, 37:2211–2234, 1994.

[9] E. Graham and P.A. Forsyth. Preconditioned conjugate gradient methods for very ill-conditioned three dimensional linear elasticity problems. *Int. J. Num. Meth. Eng.*, 44:77–98, 1999.

[10] Q. Fan, P.A. Forsyth andJ. McMacken, and W.P. Tang. Performance issues for iterative solvers in device simulation. *SIAM J. Sci. Comp.*, 17:100–117, 1996.

[11] R. Zvan, P.A. Forsyth, and K.R. Vetzal. Robust numerical methods for PDE models of Asian options. *J. Comp. Fin.*, 1:39–78, Winter 1998.

[12] R. Zvan, K.R. Vetzal, and P.A. Forsyth. PDE methods for barrier options. 1998. Accepted in J. Econ. Dyn. Control, 1997, 25 pages, CS Tech. Report CS-97-27, ftp://cs-archive.uwaterloo.ca/cs-archive/CS-97-27/CS-97-27.ps.Z.

[13] R. Zvan amd P.A. Forsyth and K. Vetzal. Penalty methods for american options with stochastic volatility. *Comp. Appl. Math.*, 91:199–218, 1998.

[14] P.A. Forsyth, K.R. Vetzal, and R. Zvan. A finite element approach to the pricing of discrete lookbacks with stochastic volatility. to appear in Appl. Math. Fin., 1999.

[15] H.A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comp.*, 13:631–645, 1992.